Handbook of Magma Functions

John Cannon Wieb Bosma
Claus Fieker Allan Steel
Editors

Version 2.19
Sydney
April 24, 2013

MAGMA
COMPUTER ALGEBRA

Handbook of Magma Functions

Editors:

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors:

Geoff Bailey, Wieb Bosma, Gavin Brown, Nils Bruin, John Cannon, Jon Carlson, Scott Contini, Bruce Cox, Brendan Creutz, Steve Donnelly, Tim Dokchitser, Willem de Graaf, Andreas-Stephan Elsenhans, Claus Fieker, Damien Fisher, Volker Gebhardt, Sergei Haller, Michael Harrison, Florian Hess, Derek Holt, David Howden, Al Kasprzyk, Markus Kirschmer, David Kohel, Axel Kohnert, Dimitri Leemans, Paulette Lieby, Graham Matthews, Scott Murray, Eamonn O'Brien, Dan Roozemond, Ben Smith, Bernd Souvignier, William Stein, Allan Steel, Damien Stehlé, Nicole Sutherland, Don Taylor, Bill Unger, Alexa van der Waall, Paul van Wamelen, Helena Verrill, John Voight, Mark Watkins, Greg White

Production Editors:

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production:

Claus Fieker *Allan Steel*

PREFACE

The computer algebra system MAGMA is designed to provide a software environment for computing with the structures which arise in areas such as algebra, number theory, algebraic geometry and (algebraic) combinatorics. MAGMA enables users to define and to compute with structures such as groups, rings, fields, modules, algebras, schemes, curves, graphs, designs, codes and many others. The main features of MAGMA include:

- *Algebraic Design Philosophy:* The design principles underpinning both the user language and system architecture are based on ideas from universal algebra and category theory. The language attempts to approximate as closely as possible the usual mathematical modes of thought and notation. In particular, the principal constructs in the user language are set, (algebraic) structure and morphism.

- *Explicit Typing:* The user is required to explicitly define most of the algebraic structures in which calculations are to take place. Each object arising in the computation is then defined in terms of these structures.

- *Integration:* The facilities for each area are designed in a similar manner using generic constructors wherever possible. The uniform design makes it a simple matter to program calculations that span different classes of mathematical structures or which involve the interaction of structures.

- *Relationships:* MAGMA provides a mechanism that manages "relationships" between complex bodies of information. For example, when substructures and quotient structures are created by the system, the natural homomorphisms that arise are always stored. These are then used to support automatic coercion between parent and child structures.
- *Mathematical Databases:* MAGMA has access to a large number of databases containing information that may be used in searches for interesting examples or which form an integral part of certain algorithms. Examples of current databases include factorizations of integers of the form $p^n \pm 1$, p a prime; modular equations; strongly regular graphs; maximal subgroups of simple groups; integral lattices; $K3$ surfaces; best known linear codes and many others.

- *Performance:* The intention is that MAGMA provide the best possible performance both in terms of the algorithms used and their implementation. The design philosophy permits the kernel implementor to choose optimal data structures at the machine level. Most of the major algorithms currently installed in the MAGMA kernel are state-of-the-art and give performance similar to, or better than, specialized programs.

The theoretical basis for the design of MAGMA is founded on the concepts and methodology of modern algebra. The central notion is that of an *algebraic structure*. Every object created during the course of a computation is associated with a unique parent algebraic structure. The *type* of an object is then simply its parent structure.

Algebraic structures are first classified by *variety*: a variety being a class of structures having the same set of defining operators and satisfying a common set of axioms. Thus, the collection of all rings forms a variety. Within a variety, structures are partitioned into *categories*. Informally, a family of algebraic structures forms a category if its members all share a common *representation*. All varieties possess an *abstract* category of structures (the finitely presented structures). However, categories based on a concrete representation are as least as important as the abstract category in most varieties. For example, within the variety of algebras, the family of finitely presented algebras constitutes an abstract category, while the family of matrix algebras constitutes a concrete category.

MAGMA comprises a novel user programming language based on the principles outlined above together with program code and databases designed to support computational research in those areas of mathematics which are algebraic in nature. The major areas represented in MAGMA V2.19 include group theory, ring theory, commutative algebra, arithmetic fields and their completions, module theory and lattice theory, finite dimensional algebras, Lie theory, representation theory, homological algebra, general schemes and curve schemes, modular forms and modular curves, L-functions, finite incidence structures, linear codes and much else.

This set of volumes (known as the Handbook) constitutes the main reference work on MAGMA. It aims to provide a comprehensive description of the MAGMA language and the mathematical facilities of the system, In particular, it documents every function and operator available to the user. Our aim (not yet achieved) is to list not only the functionality of the MAGMA system but also to show how the tools may be used to solve problems in the various areas that fall within the scope of the system. This is attempted through the inclusion of tutorials and sophisticated examples. Finally, starting with the edition corresponding to release V2.8, this work aims to provide some information about the algorithms and techniques employed in performing sophisticated or time-consuming operations. It will take some time before this goal is fully realised.

We give a brief overview of the organization of the Handbook.

- Volume 1 contains a terse summary of the language together with a description of the central datatypes: sets, sequences, tuples, mappings, etc. An index of all intrinsics appears at the end of the volume.

- Volume 2 deals with basic rings and linear algebra. The rings include the integers, the rationals, finite fields, univariate and multivariate polynomial rings as well as real and complex fields. The linear algebra section covers matrices and vector spaces.

- Volume 3 covers global arithmetic fields. The major topics are number fields, their orders and function fields. More specialised topics include quadratic fields , cyclotomic fields and algebraically closed fields.

- Volume 4 is concerned with local arithmetic fields. This covers p-adic rings and their extension and power series rings including Laurent and Puiseux series rings,

- Volume 5 describes the facilities for finite groups and, in particular, discusses permutation groups, matrix groups and finite soluble groups defined by a power-conjugate presentation. A chapter is devoted to databases of groups.

- Volume 6 describes the machinery provided for finitely presented groups. Included are abelian groups, general finitely presented groups, polycyclic groups, braid groups and automatic groups. This volume gives a description of the machinery provided for computing with finitely presented semigroups and monoids.

- Volume 7 is devoted to aspects of Lie theory and module theory. The Lie theory includes root systems, root data, Coxeter groups, reflection groups and Lie groups.

- Volume 8 covers algebras and representation theory. Associative algebras include structure-constant algebras, matrix algebras, basic algebras and quaternion algebras. Following an account of Lie algebras there is a chapter on quantum groups and another on universal enveloping algebras. The representation theory includes group algebras, $K[G]$-modules, character theory, representations of the symmetric group and representations of Lie groups.

- Volume 9 covers commutative algebra and algebraic geometry. The commutative algebra material includes constructive ideal theory, affine algebras and their modules, invariant rings and differential rings. In algebraic geometry the main topics are schemes, sheaves and toric varieties. Also included are chapters describing specialised machinery for curves and surfaces.

- Volume 10 describes the machinery pertaining to arithmetic geometry. The main topics include the arithmetic properties of low genus curves such as conics, elliptic curves and hyperelliptic curves. The volume concludes with a chapter on L-series.

- Volume 11 is concerned with modular forms.

- Volume 12 covers various aspects of geometry and combinatorial theory. The geometry section includes finite planes, finite incidence geometry and convex polytopes. The combinatorial theory topics comprise enumeration, designs, Hadamard matrices, graphs and networks.

- Volume 13 is primarily concerned with coding theory. Linear codes over both fields and finite rings are considered at length. Further chapters discuss machinery for AG-codes, LDPC codes, additive codes and quantum error-correcting codes. The volume concludes with short chapters on pseudo-random sequences and on linear programming.

Although the Handbook has been compiled with care, it is possible that the semantics of some facilities have not been described adequately. We regret any inconvenience that this may cause, and we would be most grateful for any comments and suggestions for improvement. We would like to thank users for numerous helpful suggestions for improvement and for pointing out misprints in previous versions.

The development of MAGMA has only been possible through the dedication and enthusiasm of a group of very talented mathematicians and computer scientists. Since 1990, the principal members of the MAGMA group have included: Geoff Bailey, Mark Bofinger, Wieb Bosma, Gavin Brown, John Brownie, Herbert Brückner, Nils Bruin, Steve Collins, Scott Contini, Bruce Cox, Brendan Creutz, Steve Donnelly, Willem de Graaf, Andreas-Stephan Elsenhans, Claus Fieker, Damien Fisher, Alexandra Flynn, Volker Gebhardt, Katharina Geißler, Sergei Haller, Michael Harrison, Emanuel Herrmann, Florian Heß, David Howden, Al Kasprzyk, David Kohel, Paulette Lieby, Graham Matthews, Scott Murray, Anne O'Kane, Catherine Playoust, Richard Rannard, Colva Roney-Dougal, Dan Roozemond, Andrew Solomon, Bernd Souvignier, Ben Smith, Allan Steel, Damien Stehlé, Nicole Sutherland, Don Taylor, Bill Unger, John Voight, Alexa van der Waall, Mark Watkins and Greg White.

John Cannon
Sydney, December 2012

ACKNOWLEDGEMENTS

The Magma Development Team

Current Members

Geoff Bailey, BSc (Hons) (Sydney), [1995-]: Main interests include elliptic curves (especially those defined over the rationals), virtual machines and computer language design. Has implemented part of the elliptic curve facilities especially the calculation of Mordell-Weil groups. Other main areas of contribution include combinatorics, local fields and the Magma system internals.

John Cannon, Ph.D. (Sydney), [1971-]: Research interests include computational methods in algebra, geometry, number theory and combinatorics; the design of mathematical programming languages and the integration of databases with Computer Algebra systems. Contributions include overall concept and planning, language design, specific design for many categories, numerous algorithms (especially in group theory) and general management.

Brendan Creutz, Ph.D. (Jacobs University Bremen) [2011-]: Primary research interests are in arithmetic geometry. Main contributions focus on descent obstructions to the existence of rational points on curves and torsors under their Jacobians. Currently developing a package for cyclic covers of the projective line.

Steve Donnelly, Ph.D. (Athens, Ga) [2005-]: Research interests are in arithmetic geometry, particularly elliptic curves and modular forms. Major contributions include descent methods for elliptic curves (including over function fields) and Cassels-Tate pairings, classical modular forms of half-integral weight, Hilbert modular forms and fast algorithms for definite quaternion algebras. Currently working on Hilbert modular forms, and elliptic curves over number fields.

Andreas-Stephan Elsenhans, Ph.D. (Göttingen) [2012-]: Main research interests are in the areas of arithmetic and algebraic geometry, particularly cubic and K3 surfaces. Main contributions focus on cubic surfaces from the arithmetic and algebraic points of view. Currently working on the computation of invariants.

Michael Harrison, Ph.D. (Cambridge) [2003-]: Research interests are in number theory, arithmetic and algebraic geometry. Implemented the p-adic methods for counting points on hyperelliptic curves and their Jacobians over finite fields including Kedlaya's algorithm and the modular parameter method of Mestre. Currently working on machinery for general surfaces and cohomology for projective varieties.

David Howden, Ph.D. (Warwick) [2012-]: Primary research interests are in computational group theory. Main contributions focus on computing automorphism groups and isomorphism testing for soluble groups.

Allan Steel, Ph.D. (Sydney), [1989-]: Has developed many of the fundamental data structures and algorithms in MAGMA for multiprecision integers, finite fields, matrices and modules, polynomials and Gröbner bases, aggregates, memory management, environmental features, and the package system, and has also worked on the MAGMA language interpreter. In collaboration, he has developed the code for lattice theory (with Bernd Souvignier), invariant theory (with Gregor Kemper) and module theory (with Jon Carlson and Derek Holt).

Nicole Sutherland, BSc (Hons) (Macquarie), [1999-]: Works in the areas of number theory and algebraic geometry. Developed the machinery for Newton polygons and lazy power series and contributed to the code for local fields, number fields, modules over Dedekind domains, function fields, schemes and has worked on aspects of algebras.

Don Taylor, D.Phil. (Oxford), [2010-] Research interests are in reflection groups, finite group theory, and geometry. Implemented algorithms for complex reflection groups and complex root data. Contributed to the packages for Chevalley groups and groups of Lie type. Currently developing algorithms for classical groups of isometries, Clifford algebras and spin groups.

Bill Unger, Ph.D. (Sydney), [1998-]: Main area of interest is computational group theory, with particular emphasis on algorithms for permutation and matrix groups. Implemented many of the current permutation and matrix group algorithms for MAGMA, in particular BSGS verification, solvable radical and chief series algorithms. Recently discovered a new method for computing the character table of a finite group.

Mark Watkins, Ph.D. (Athens, Ga), [2003, 2004-2005, 2008-]: Works in the area of number theory, particularly analytic methods for arithmetic objects. Implemented a range of analytic tools for the study of elliptic curves including analytic rank, modular degree, Heegner points and (general) point searching methods. Also deals with conics, lattices, modular forms, and descent machinery over the rationals.

Former Members

Wieb Bosma, [1989-1996]: Responsible for the initial development of number theory in MAGMA and the coordination of work on commutative rings. Also has continuing involvement with the design of MAGMA.

Gavin Brown, [1998-2001]: Developed code in basic algebraic geometry, applications of Gröbner bases, number field and function field kernel operations; applications of Hilbert series to lists of varieties.

Herbert Brückner, [1998–1999]: Developed code for constructing the ordinary irreducible representations of a finite soluble group and the maximal finite soluble quotient of a finitely presented group.

Nils Bruin, [2002–2003]: Contributions include Selmer groups of elliptic curves and hyperelliptic Jacobians over arbitrary number fields, local solubility testing for arbitrary projective varieties and curves, Chabauty-type computations on Weil-restrictions of elliptic curves and some algorithms for, and partial design of, the differential rings module.

Bruce Cox, [1990–1998]: A member of the team that worked on the design of the MAGMA language. Responsible for implementing much of the first generation MAGMA machinery for permutation and matrix groups.

Claus Fieker, [2000-2011]: Formerly a member of the KANT project. Research interests are in constructive algebraic number theory and, especially, relative extensions and computational class field theory. Main contributions are the development of explicit algorithmic class field theory in the case of both number and function fields and the computation of Galois groups.

Damien Fisher, [2002-2006]: Implemented a package for p-adic rings and their extensions and undertook a number of extensions to the MAGMA language.

Alexandra Flynn, [1995–1998]: Incorporated various Pari modules into MAGMA, and developed much of the machinery for designs and finite planes.

Volker Gebhardt, [1999–2003]: Author of the MAGMA categories for infinite polycyclic groups and for braid groups. Other contributions include machinery for general finitely presented groups.

Katharina Geißler, [1999–2001]: Developed the code for computing Galois groups of number fields and function fields.

Willem de Graaf, [2004-2005]: Contributed functions for computing with finite-dimensional Lie algebras, finitely-presented Lie algebras, universal enveloping algebras and quantum groups.

Sergei Haller, [2004, 2006-2007]: Developed code for many aspects of Lie Theory. Of particular note was his work on the construction of twisted groups of Lie type and the determination of conjugacy classes of elements in the classical groups (jointly with Scott Murray (MAGMA)).

Emanuel Herrmann, [1999]: Contributed code for finding S-integral points on genus 1 curves (not elliptic curves).

Florian Heß, [1999–2001]: Developed a substantial part of the algebraic function field module in MAGMA including algorithms for the computation of Riemann-Roch spaces and class groups. His most recent contribution (2005) is a package for computing all isomorphisms between a pair of function fields.

Alexander Kasprzyk, [2009-2010]: Developed the toric geometry and polyhedra packages (along with Gavin Brown and Jaroslaw Buczynski).

David Kohel, [1999–2002]: Contributions include a model for schemes (with G Brown); algorithms for curves of low genus; implementation of elliptic curves, binary quadratic forms, quaternion algebras, Brandt modules, spinor genera and genera of lattices, modular curves, conics (with P Lieby), modules of supersingular points (with W Stein), Witt rings.

Paulette Lieby, [1999–2003]: Contributed to the development of algorithms for algebraic geometry, abelian groups and incidence structures. Developed datastructures for multigraphs and implemented algorithms for planarity, triconnectivity and network flows.

Graham Matthews, [1989–1993]: Involved in the design of the MAGMA semantics, user interface, and internal organisation.

Scott Murray, [2001-2002, 2004-2010]: Implemented algorithms for element operations in split groups of Lie type, representations of split groups of Lie type, split Cartan subalgebras of modular Lie algebras, and Lang's Theorem in finite reductive groups. More recently implemented solutions to conjugacy problems in the classical groups (with S. Haller and D. Taylor).

Catherine Playoust, [1989–1996]: Wrote extensive documentation and implemented an early help system. Contributed to system-wide consistency of design and functionality. Also pioneered the use of MAGMA for teaching undergraduates.

Richard Rannard, [1997–1998]: Contributed to the code for elliptic curves over finite fields including a first version of the SEA algorithm.

Colva M. Roney-Dougal, [2001–2003]: Completed the classification of primitive permutation groups up to degree 999 (with Bill Unger). Also undertook a constructive classification of the maximal subgroups of the classical simple groups.

Dan Roozemond, [2010-2012]: Research focused on the computational aspects of Lie theory. Ported algorithms for the Weight Multisets from LiE to Magma and developed a number of algorithms for reductive Lie algebras, particularly over fields of small characteristic.

Michael Slattery, [1987–2006]: Contributed a large part of the machinery for finite soluble groups including subgroup lattice and automorphism group.

Ben Smith, [2000–2003]: Contributed to an implementation of the Number Field Sieve and a package for integer linear programming.

Bernd Souvignier, [1996–1997]: Contributed to the development of algorithms and code for lattices, local fields, finite dimensional algebras and permutation groups.

Damien Stehlé, [2006, 2008-2010]: Implemented the proveably correct floating-point LLL algorithm together with a number of fast non-rigorous variants. Also developed a fast method for enumerating short vectors.

John Voight, [2005-2006]: Implemented algorithms for quaternion algebras over number fields, associative orders (with Nicole Sutherland), and Shimura curves.

Alexa van der Waall, [2003]: Implemented the module for differential Galois theory.

Paul B. van Wamelen, [2002–2003]: Implemented analytic Jacobians of hyperelliptic curves in MAGMA.

Greg White, [2000-2006]: Contributions include fast minimum weight determination, linear codes over Z/mZ, additive codes, LDPC codes, quantum error-correcting codes, and a database of best known linear codes (with Cannon and Grassl).

External Contributors

The MAGMA system has benefited enormously from contributions made by many members of the mathematical community. We list below those persons and research groups who have given the project substantial assistance either by allowing us to adapt their software for inclusion within MAGMA or through general advice and criticism. We wish to express our gratitude both to the people listed here and to all those others who participated in some aspect of the MAGMA development.

Algebraic Geometry

A major package for algebraic surfaces providing formal desingularization, the calculation of adjoints, and rational parameterization was developed by **Tobias Beck** (RICAM, Linz). He also implemented a package for computing with algebraic power series. This work was done while he was a student of **Josef Schicho**.

A package for working with divisors on varieties has been developed by **Martin Bright** (American University of Beirut), **Gavin Brown** (Loughborough), **Mike Harrison** (Magma) and **Andrew Wilson** (Edinburgh). The functionality includes decomposition into irreducible components, Riemann-Roch spaces, canonical divisors and (surface) intersection numbers.

Machinery for working with Hilbert series of polarised varieties and the associated databases of K3 surfaces and Fano 3-folds has been constructed by **Gavin Brown** (Warwick).

Jaroslaw Buczynski (Texas A&M), along with **Gavin Brown** (Loughborough) and **Alexander Kasprzyk** (Imperial College), developed the toric geometry and polyhedra packages.

Functions for computing Shioda invariants for genus 3 hyperelliptic curves, reconstructing models for a curve from such invariants and computing geometric automorphism groups have been contributed by **Reynald Lercier** (DGA, Rennes) and **Christophe Ritzenthaler** (Luminy).

Jana Pilnikova (Univerzita Komenskeho, Bratislava) (while a student of **Josef Schicho** in Linz) contributed code for the parameterization of degree 8 and 9 Del Pezzo surfaces, jointly written with **Willem de Graaf** (Trento).

Miles Reid (Warwick) has been heavily involved in the design and development of a database of $K3$ surfaces within MAGMA.

Josef Schicho (RICAM, Linz) has played a major role in the design and implementation of the algebraic surfaces package. In particular, Josef has also implemented several of the modules for rational surface parameterization.

A function that finds the intersection multiplicities for all intersection points of two plane curves was adapted into MAGMA from code provided by **Chris Smyth** (Edinburgh).

Andrew Wilson (Edinburgh) has contributed a package to compute the log canonical threshold for singular points on a curve.

Arithmetic Geometry Over Characteristic 0 Fields

The method of Chabauty for finding points on elliptic curves was originally implemented by **Nils Bruin** in 2003 while a member of the MAGMA group. In 2009 Nils improved it considerably by combining it with *Mordell-Weil sieving*.

Two-cover-descent has been implemented by **Nils Bruin** (Simon Fraser) for hyperelliptic curves. Given the Jacobian of a genus 2 curve, Nils has also provided code to compute all $(2,2)$-isogenous abelian surfaces.

The MAGMA facility for determining the Mordell-Weil group of an elliptic curve over the rational field is based on the MWRANK programs of **John Cremona** (Nottingham).

John Cremona (Nottingham) has contributed his code implementing Tate's algorithm for computing local minimal models for elliptic curves defined over number fields.

The widely-used database of all elliptic curves over Q having conductor up to 300,000 constructed by **John Cremona** (Warwick) is also included.

Tim Dokchitser (Durham) wrote code for computing root numbers of elliptic curves over number fields.

Andreas-Stephan Elsenhans (Bayreuth) has provided routines for performing minimisation and reduction for Del Pezzo surfaces of degrees 3 and 4.

Code for determining isomorphism of cubic surfaces has been contributed by **Andreas-Stephan Elsenhans** (Bayreuth).

A collection of tools that calculate information about the Picard rank of a surface has been developed by **Andreas-Stephan Elsenhans** (Bayreuth).

Code for calculating the invariants, covariants and contravariants of a cubic surface has been developed by **Andreas-Stephan Elsenhans** (Bayreuth).

A package contributed by **Tom Fisher** (Cambridge) deals with curves of genus 1 given by models of a special kind (genus one normal curves) having degree $2, 3, 4$ and 5.

The implementation of 3-descent on elliptic curves was mainly written by **Tom Fisher** (Cambridge). An earlier version as well as part of the current version were developed by **Michael Stoll** (Bremen).

The algorithms and implementations of $6-$ and 12-descent are due to **Tom Fisher** (Cambridge). The new alorithm/implementation of 8-descent is likewise by Tom Fisher; this partly incorporates and partly replaces the earlier one by **Sebastian Stamminger**.

Martine Girard (Sydney) has contributed her fast code for determining the heights of a point on an elliptic curve defined over a number field or a function field.

David Kohel (Singapore–NUS, MAGMA) has provided implementations of division polynomials and isogeny structures for elliptic curves.

Full and partial descents on cyclic covers of the projective line were implemented by **Michael Mourao** (Warwick).

A package for computing canonical heights on hyperelliptic curves has been contributed by **Steffan Müller** (Bayreuth).

David Roberts (Nottingham) contributed some descent machinery for elliptic curves over function fields.

David Roberts and **John Cremona** (Nottingham) implemented the Cremona-van Hoeij algorithm for parametrization of conics over rational function fields.

Jasper Scholten (Leuven) has developed much of the code for computing with elliptic curves over function fields.

Much of the initial development of the package for computing with hyperelliptic curves is due to **Michael Stoll** (Bayreuth). He also contributed many of the high level routines involving curves over the rationals and their Jacobians, such as Chabauty's method.

A database of $136,924,520$ elliptic curves with conductors up to 10^8 has been provided by **William Stein** (Harvard) and **Mark Watkins** (Penn State).

For elliptic curves defined over finite fields of characteristic 2, Kedlaya's algorithm for point counting has been implemented by **Frederick Vercauteren** (Leuven).

Tom Womack (Nottingham) contributed code for performing four-descent, from which the current implementation was adapted.

Arithmetic Geometry Over Finite Fields

Various point-counting algorithms for hyperelliptic curves have been implemented by **Pierrick Gaudry** (Ecole Polytechnique, Paris). These include an implementation of the Schoof algorithm for genus 2 curves.

An implementation of GHS Weil descent for ordinary elliptic curves in characteristic 2 has been provided by **Florian Heß** (TU, Berlin).

A MAGMA package for calculating Igusa and other invariants for genus 2 hyperelliptic curves was written by **Everett Howe** (CCR, San Diego) and is based on `gp` routines developed by **Fernando Rodriguez–Villegas** (Texas) as part of the Computational Number Theory project funded by a TARP grant.

Reynard Lercier (Rennes) provided much advice and assistance to the MAGMA group concerning the implementation of the SEA point counting algorithm for elliptic curves.

Reynard Lercier (Rennes) and **Christophe Ritzenthaler** provided extensions to the machinery for genus 2 curves defined over finite fields. These include the reconstruction of a curve from invariants which applies to every characteristic p (previously $p > 5$), the geometric automorphism group and the calculation of all twists (not just quadratic).

Frederik Vercauteren (Leuven) has produced efficient implementations of the Tate, Eta and Ate pairings in MAGMA.

Class fields over local fields and the multiplicative structure of local fields are computed using new algorithms and implementations due to **Sebastian Pauli** (TU Berlin).

The module for Lazy Power Series is based on the ideas of **Josef Schicho** (Linz).

Associative Algebras

Fast algorithms for computing the Jacobson radical and unit group of a matrix algebra over a finite field were designed and implemented by **Peter Brooksbank** (Bucknell) and **Eamonn O'Brien** (Auckland).

A package for computing with algebras equipped with an involution (*-algebras) has been contributed by **Peter Brooksbank** (Bucknell) and **James Wilson**.

An algorithm designed and implemented by **Jon Carlson** and **Graham Matthews** (Athens, Ga.) provides an efficient means for constructing presentations for matrix algebras.

For matrix algebras defined over a finite field, **Jon Carlson** (Athens, Ga.) designed and implemented algorithms for the Jacobson radical and unit group which are faster than the Brooksbank-O'Brien algorithms for larger examples.

A substantial package for working with substructures and homomorphisms of basic algebras, developed by **Jon Carlson** (Athens, Ga.), was released as part of V2.19. Among other things, the package can compute the automorphism group of a basic algebra and test pairs of basic algebras for isomorphism.

Markus Kirschmer (Aachen) has written a number of optimized routines for definite quaternion algebras over number fields.

Markus Kirschmer has also contributed a package for quaternion algebras defined over the function fields $F_q[t]$, for q odd. The package includes calculation of the normaliser of an order and an efficient algorithm for computing the two-sided ideal classes of an order in a definite quaternion algebra (over $\mathbf{Z}$ or $\mathbf{F}_q[t]$).

Quaternion algebras over the rational field Q were originally implemented by **David Kohel** (Singapore-NUS, MAGMA).

The vector enumeration program of **Steve Linton** (St. Andrews) provides an alternative to the use of Gröbner basis for constructing a matrix representation of a finitely presented associative algebra.

John Voight (Vermont) produced the package for quaternion algebras over number fields.

Coding Theory

A package for constructing linear codes associated with lattice points in a convex polytope has been contributed by **Gavin Brown** (Loughborough) and **Al Kasprzyk** (Imperial).

The PERM package developed by **Jeff Leon** (UIC) is used to determine automorphism groups of codes, designs and matrices.

The development of machinery for linear codes benefited greatly from the active involvement of **Markus Grassl** (Karlsruhe) over a long period. Of particular note is his contribution to the development of improved algorithms for computing the minimum weight and for the enumeration of codewords.

Routines implementing many different constructions for linear codes over finite fields were contributed by **Markus Grassl** (Karlsruhe).

Markus Grassl (Karlsruhe) played a key role in the design of MAGMA packages for Additive Codes and Quantum Error-Correcting Codes. The packages were implemented by Greg White (Magma).

The construction of a database of Best Known Linear Codes over GF(2) was a joint project with **Markus Grassl** (Karlsruhe, NUS). Other contributors to this project include: **Andries Brouwer**, **Zhi Chen**, **Stephan Grosse**, **Aaron Gulliver**, **Ray Hill**, **David Jaffe**, **Simon Litsyn**, **James B. Shearer** and **Henk van Tilborg**.

The databases of Best Known Linear Codes over GF(3), GF(4), GF(5), GF(7), GF(8) and GF(9) were constructed by **Markus Grassl** (IAKS, Karlsruhe).

A substantial collection of intrinsics for constructing and computing properties of Z_4 codes has been contributed by **Jaume Pernas**, **Jaume Pujol** and **Merc̀ Villanueva** (Universitat Autònoma de Barcelona).

Combinatorics

Michel Berkelaar (Eindhoven) gave us permission to incorporate his LP_SOLVE package for linear programming.

The first stage of the MAGMA database of Hadamard and skew-Hadamard matrices was prepared with the assistance of **Stelios Georgiou** (Athens), **Ilias Kotsireas** (Wilfrid Laurier) and **Christos Koukouvinos** (Athens). In particular, they made available their tables of Hadamard matrices of orders 32, 36, 44, 48 and 52. Further Hadamard matrices were contributed by Dragomir Djokovic.

The MAGMA machinery for symmetric functions is based on the Symmetrica package developed by **Abalbert Kerber** (Bayreuth) and colleagues. The MAGMA version was implemented by **Axel Kohnert** of the Bayreuth group.

The PERM package developed by **Jeff Leon** (UIC) is used to determine automorphism groups of designs and also to determine isomorphism of pairs of designs.

Automorphism groups and isomorphism of Hadamard matrices are determined by converting to a similar problem for graphs and then applying **Brendan McKay's** (ANU) program NAUTY. The adaption was undertaken by **Paulette Lieby** and **Geoff Bailey**.

The calculation of the automorphism groups of graphs and the determination of graph isomorphism is performed using **Brendan McKay's** (ANU) program NAUTY (version 2.2). Databases of graphs and machinery for generating such databases have also been made available by Brendan. He has also collaborated in the design of the sparse graph machinery.

The code to perform the regular expression matching in the `regexp` intrinsic function comes from the V8 regexp package written by **Henry Spencer** (Toronto).

Commutative Algebra

Gregor Kemper (TU München) has contributed most of the major algorithms of the Invariant Theory module of MAGMA, together with many other helpful suggestions in the area of Commutative Algebra.

Alexa van der Waall (Simon Fraser) has implemented the module for differential Galois theory.

Galois Groups

Jürgen Klüners (Kassel) has made major contributions to the Galois theory machinery for function fields and number fields. In particular, he implemented functions for constructing the subfield lattice and automorphism group of a field and also the subfield lattice of the normal closure of a field. In joint work with Claus Fieker (MAGMA), Jürgen has recently developed a new method for determining the Galois group of a polynomial of arbitary high degree.

Jürgen Klüners (Kassel) and **Gunter Malle** (Kassel) made available their extensive tables of polynomials realising all Galois groups over Q up to degree 15.

Galois Representations

Jeremy Le Borgne (Rennes) contributed his package for working with mod p Galois representations.

Code for constructing Artin representations of the Galois group of the absolute extension of a number field was developed by **Tim Dokchitser** (Cambridge).

Jared Weinstein (UCLA) wrote the package on admissible representations of $GL_2(\mathbf{Q}_p)$.

Geometry

The MAGMA code for computing with incidence geometries has been developed by **Dimitri Leemans** (Brussels).

Algorithms for testing whether two convex polytopes embedded in a lattice are isomorphic or equivalent have been implemented by **Al Kasprzyk** (Imperial College). Of particular note is Al's implementation of the PALP normal form algorithm.

Global Arithmetric Fields

Jean-Francois Biasse (Calgary) implemented a quadratic sieve for computing the class group of a quadratic field. He also developed a generalisation of the sieve for number fields having degree greater than 2.

Florian Heß (TU Berlin) has contributed a major package for determining all isomorphisms between a pair of algebraic function fields.

David Kohel (Singapore–NUS, MAGMA) has contributed to the machinery for binary quadratic forms and has implemented rings of Witt vectors.

Jürgen Klüners (Düsseldorf) and **Sebastian Pauli** (UNC Greensboro) have developed algorithms for computing the Picard group of non-maximal orders and for embedding the unit group of non-maximal orders into the unit group of the field.

The facilities for general number fields and global function fields in MAGMA are based on the KANT V4 package developed by **Michael Pohst** and collaborators, first at Düsseldorf and then at TU Berlin. This package provides extensive machinery for computing with maximal orders of number fields and their ideals, Galois groups and function fields. Particularly noteworthy are functions for computing the class and unit group, and for solving Diophantine equations.

The fast algorithm of Bosma and Stevenhagen for computing the 2-part of the ideal class group of a quadratic field has been implemented by **Mark Watkins** (Bristol).

Group Theory: Finitely-Presented Groups

See also the subsection *Group Theory: Soluble Groups*.

A new algorithm for computing all normal subgroups of a finitely presented group up to a specified index has been designed and implemented by **David Firth** and **Derek Holt** (Warwick).

The function for determining whether a given finite permutation group is a homomorphic image of a finitely presented group has been implemented in C by Volker Gebhardt (Magma) from a Magma language prototype developed by **Derek Holt** (Warwick). A variant developed by Derek allows one to determine whether a small soluble group is a homomorphic image.

A small package for working with subgroups of free groups has been developed by **Derek Holt** (Warwick). He has also provided code for computing the automorphism group of a free group.

Versions of MAGMA from V2.8 onwards employ the Advanced Coset Enumerator designed by **George Havas** (UQ) and implemented by **Colin Ramsay** (UQ). George has also contributed to the design of the machinery for finitely presented groups.

Derek Holt (Warwick) developed a modified version of his program, KBMAG, for inclusion within MAGMA. The MAGMA facilities for groups and monoids defined by confluent rewrite systems, as well as automatic groups, are supported by this code.

Derek Holt (Warwick) has provided a MAGMA implementation of his algorithm for testing whether two finitely presented groups are isomorphic.

An improved version of the Plesken-Fabianska algorithm for finding L2-quotients of a finitely presented group has been developed and implemented by **Sebastian Jambor** (Aachen).

The low index subgroup function is implemented by code that is based on a Pascal program written by **Charlie Sims** (Rutgers).

Group Theory: Finite Groups

A variation of the Product Replacement Algorithm for generating random elements of a group due to **Henrik Bäärnhielm** and **Charles Leedham-Green** has been coded with their assistance.

A Small Groups database containing all groups having order at most 2000, excluding order 1024 has been made available by **Hans Ulrich Besche** (Aachen), **Bettina Eick** (Braunschweig), and **Eamonn O'Brien** (Auckland). This library incorporates "directly" the libraries of 2-groups of order dividing 256 and the 3-groups of order dividing 729, which were prepared and distributed at various intervals by **Mike Newman** (ANU) and **Eamonn O'Brien** and various assistants, the first release dating from 1987.

Michael Downward and **Eamonn O'Brien** (Auckland) provided functions to access much of the data in the on-line Atlas of Finite Simple Groups for the sporadic groups. A function to select "good" base points for sporadic groups was provided by Eamonn and **Robert Wilson** (QMUL).

The Small Groups database was augmented in V2.14 by code that can enumerate all groups of any square-free order. This code was developed by **Bettina Eick** (Braunschweig) and **Eamonn O'Brien** (Auckland).

The calculation of automorphism groups (for permutation and matrix groups) and determining group isomorphism is performed by code written by **Derek Holt** (Warwick).

Lifting-style algorithms have been developed by **Derek Holt** (Warwick) for computing structural information in groups given in terms of the Composition Tree data structure. The operations include centralisers, conjugacy classes, normalizers, subgroup conjugacy and maximal subgroups.

Magma includes a database of almost-simple groups defined on standard generators. The database was originally conceived by **Derek Holt** (Warwick) with a major extension by **Volker Gebhardt** (Magma) and sporadic additions by **Bill Unger** (Magma).

The routine for computing the subgroup lattice of a group (as distinct from the list of all conjugacy classes of subgroups) is based on code written by **Dimitri Leemans** (Brussels).

Csaba Schneider (Lisbon) has implemented code which allows the user to write an arbitrary element of a classical group as an SLP in terms of its standard generators.

Robert Wilson (QMUL) has made available the data contained in the on-line *ATLAS of Finite Group Representations* for use in a MAGMA database of permutation and matrix representations for finite simple groups. See `http://brauer.maths.qmul.ac.uk/Atlas/`.

Group Theory: Matrix Groups

The Composition Tree (CT) package developed by **Henrik Bäärnhielm** (Auckland), **Derek Holt** (Warwick), **Charles Leedham-Green** (QMUL) and **Eamonn O'Brien** (Auckland), working with numerous collaborators, was first released in V2.17. This package is designed for computing structural information for large matrix groups defined over a finite field.

Constructive recognition of quasi-simple groups belonging to the Suzuki and two Ree families have been implemented by **Hendrik Bäärnhielm** (QMUL). The package includes code for constructing their Sylow p-subgroups and maximal subgroups.

The maximal subgroups of all classical groups having degree not exceeding 12 have been constructed and implemented in MAGMA by **John Bray** (QMUL), **Derek Holt** (Warwick) and **Colva Roney-Dougal** (St Andrews).

Peter Brooksbank (Bucknell) implemented a MAGMA version of his algorithm for performing constructive black-box recognition of low-dimensional symplectic and unitary groups. He also gave the MAGMA group permission to base its implementation of the Kantor-Seress algorithm for black-box recognition of linear groups on his GAP implementation.

Code which computes the normaliser of a linear group defined over a finite field, using a theorem of Aschbacher rather than backtrack search, has been provided by **Hannah Coutts** (St Andrews).

A package, "Infinite", has been developed by **Alla Detinko** (Galway), **Dane Flannery** (Galway) and **Eamonn O'Brien** (Auckland) for computing with groups defined over number fields, or (rational) function fields in zero or positive characteristic.

An algorithm for determining the conjugacy of any pair of matrices in $GL(2, Z)$ was developed and implemented by **D. Husert** (University of Paderborn). In particular, this allows the conjugacy of elements having infinite order to be determined.

Markus Kirschmer (RWTH, Aachen) has provided a package for computing with finite subgroups of $GL(n, \mathbf{Z})$. A MAGMA database of the maximal finite irreducible subgroups of $Sp_{2n}(\mathbf{Q})$ for $1 \leq i \leq 11$ has also been made available by Markus.

A much improved algorithm for computing the normaliser or centraliser of a finite subgroup of $GL(n, Z)$ has been implemented by **Markus Kirschmer** (Aachen). Markus has also implemented an algorithm that tests finite subgroups for conjugacy.

Procedures to list irreducible (soluble) subgroups of $GL(2, q)$ and $GL(3, q)$ for arbitrary q have been provided by **Dane Flannery** (Galway) and **Eamonn O'Brien** (Auckland).

A Monte-Carlo algorithm to determine the defining characteristic of a quasisimple group of Lie type has been contributed by **Martin Liebeck** (Imperial) and **Eamonn O'Brien** (Auckland).

A Monte-Carlo algorithm for non-constructive recognition of simple groups has been contributed by **Gunter Malle** (Kaiserslautern) and **Eamonn O'Brien** (Auckland). This

procedure includes an algorithm of Babai et al which identifies a quasisimple group of Lie type.

MAGMA incorporates a database of the maximal finite rational subgroups of $GL(n, \mathbf{Q})$ up to dimension 31. This database as constructed by **Gabriele Nebe** (Aachen) and **Wilhelm Plesken** (Aachen). A database of quaternionic matrix groups constructed by Gabriele is also included.

A function that determines whether a matrix group G (defined over a finite field) is the normaliser of an extraspecial group in the case where the degree of G is an odd prime uses the new Monte-Carlo algorithm of **Alice Niemeyer** (Perth) and has been implemented in MAGMA by **Eamonn O'Brien** (Auckland).

The package for recognizing large degree classical groups over finite fields was designed and implemented by **Alice Niemeyer** (Perth) and **Cheryl Praeger** (Perth). It has been extended to include 2-dimensional linear groups by **Eamonn O'Brien** (Auckland).

Eamonn O'Brien (Auckland) has contributed a MAGMA implementation of algorithms for determining the Aschbacher category of a subgroup of $\mathrm{GL}(n, q)$.

Eamonn O'Brien (Auckland) has provided implementations of constructive recognition algorithms for the matrix groups $(\mathrm{P})\mathrm{SL}(2, q)$ and $(\mathrm{P})\mathrm{SL}(3, q)$.

A fast algorithm for determining subgroup conjugacy based on Aschbacher's theorem classifying the maximal subgroups of a linear group has been designed and implemented by **Colva Roney-Dougal** (St Andrews).

A package for constructing the Sylow p-subgroups of the classical groups has been implemented by **Mark Stather** (Warwick).

Generators in the natural representation of a finite group of Lie type were constructed and implemented by **Don Taylor** (Sydney) with some assistance from **Leanne Rylands** (Western Sydney).

Group Theory: Soluble Groups

The soluble quotient algorithm in MAGMA was designed and implemented by **Herbert Brückner** (Aachen).

Code producing descriptions of the groups of order p^4, p^5, p^6, p^7 for $p > 3$ was contributed by **Boris Girnat, Robert McKibbin, Mike Newman, Eamonn O'Brien**, and **Mike Vaughan-Lee**.

A new approach to the more efficient calculation of the automorphism group of a finite soluble group has been developed and implemented **David Howden** (Warwick). A slight variation of the algorithm is used to test isomorphism.

Most of the algorithms for p-groups and many of the algorithms implemented in MAGMA for finite soluble groups are largely due to **Charles Leedham–Green** (QMUL, London).

The NQ program of **Werner Nickel** (Darmstadt) is used to compute nilpotent quotients of finitely presented groups. Version 2.2 of NQ was installed in MAGMA V2.14 by **Bill Unger** (Magma) and **Michael Vaughan-Lee** (Oxford).

The p-quotient program, developed by **Eamonn O'Brien** (Auckland) based on earlier work by **George Havas** and **Mike Newman** (ANU), provides a key facility for studying p-groups in MAGMA. Eamonn's extensions in MAGMA of this package for generating p-groups, computing automorphism groups of p-groups, and deciding isomorphism of p-groups are also included. He has contributed software to count certain classes of p-groups and to construct central extensions of soluble groups.

The package for classifying metacyclic p-groups has been developed by **Eamonn O'Brien** (Auckland) and **Mike Vaughan-Lee** (Oxford).

Group Theory: Permutation Groups

Derek Holt (Warwick) has implemented the MAGMA version of the Bratus/Pak algorithm for black-box recognition of the symmetric and alternating groups.

Alexander Hulpke (Colorado State) has made available his database of all transitive permutation groups of degree up to 30. This incorporates the earlier database of **Greg Butler** (Concordia) and **John McKay** (Concordia) containing all transitive groups of degree up to 15.

The PERM package developed by **Jeff Leon** (UIC) for efficient backtrack searching in permutation groups is used for most of the permutation group constructions that employ backtrack search.

A table containing all primitive groups having degree less than 2,500 has been provided by **Colva Roney-Dougal** (St Andrews). The groups of degree up to 1,000 were done jointly with **Bill Unger** (MAGMA).

A table containing all primitive groups having degrees in the range 2,500 to 4,095 has been provided by **Hannah Coutts**, **Martyn Quick** and **Colva Roney-Dougal** (all at St Andrews).

Colva Roney-Dougal (St Andrews) has implemented the Beals et al algorithm for performing black-box recognition on the symmetric and alternating groups.

Derek Holt (Warwick) has constructed a table of irreducible representations of quasisimple groups (up to degree 100). Some representations were contributed by **Allan Steel, Volker Gebhardt** and **Bill Unger** (all MAGMA).

A MAGMA database has been constructed from the permutation and matrix representations contained in the on-line Atlas of Finite Simple Groups with the assistance of its author **Robert Wilson** (QMUL).

Homological Algebra

The packages for chain complexes and basic algebras have been developed by **Jon F. Carlson** (Athens, GA).

Sergei Haller developed MAGMA code for computing the first cohomology group of a finite group with coefficients in a finite (not necessarily abelian) group. This formed the basis of a package for computing Galois cohomology of linear algebra groups.

Machinery for computing group cohomology and for producing group extensions has been developed by **Derek Holt** (Warwick). There are two parts to this machinery. The first part comprises Derek's older C-language package for permutation groups while the second part comprises a recent MAGMA language package for group cohomology.

In 2011, **Derek Holt** (Warwick) implemented an alternative algorithm for finding the dimension of the cohomology group $H^n(G, K)$, for G a finite group, and K a finite field. In this approach the dimension is found using projective covers and dimension shifting.

The code for computing A_∞-structures in group cohomology was developed by **Mikael Vejdemo Johansson** (Jena).

L-Functions

Tim Dokchitser (Cambridge) has implemented efficient computation of many kinds of L-functions, including those attached to Dirichlet characters, number fields, Artin representations, elliptic curves and hyperelliptic curves. **Vladimir Dokchitser** (Cambridge) has contributed theoretical ideas.

Anton Mellit has contributed code for computing symmetric powers and tensor products of L-functions.

Lattices and Quadratic Forms

The construction of the sublattice of an integral lattice is performed by code developed by **Markus Kirschmer** (Aachen).

A collection of lattices derived from the on-line tables of lattices prepared by **Neil Sloane** (AT&T Research) and **Gabriele Nebe** (Aachen) is included in MAGMA.

The original functions for computing automorphism groups and isometries of integral lattices are based on the AUTO and ISOM programs of **Bernd Souvignier** (Nijmegen). In V2.16 they are replaced by much faster versions developed by **Bill Unger** (MAGMA).

Coppersmith's method (based on LLL) for finding small roots of univariate polynomials modulo an integer has been implemented by **Damien Stehlé** (ENS Lyon).

Given a quadratic form F in an arbitrary number of variables, **Mark Watkins** (Bristol) has used Denis Simon's ideas as the basis of an algorithm he has implemented in MAGMA for finding a large (totally) isotropic subspace of F.

Lie Theory

The major structural machinery for Lie algebras has been implemented for MAGMA by **Willem de Graaf** (Utrecht) and is based on his ELIAS package written in GAP. He has also implemented a separate package for finitely presented Lie rings.

A database of soluble Lie algebras of dimensions 2, 3 and 4 over all fields has been implemented by **Willem de Graaf** (Trento). Willem has also provided a database of all nilpotent Lie algebras of dimension up to 6 over all base fields (except characteristic 2 when the dimension is 6).

More recent extensions to the Lie algebra package developed by **Willem de Graaf** (Trento) include quantum groups, universal enveloping algebras, the semisimple subalgebras of a simple Lie algebra and nilpotent orbits for simple Lie algebras.

A fast algorithm for multiplying the elements of Coxeter groups based on their automatic structure has been designed and implemented by **Bob Howlett** (Sydney). Bob has also contributed MAGMA code for computing the growth function of a Coxeter group.

Machinery for computing the W-graphs for Lie types A_n, E_6, E_7 and E_8 has been supplied by **Bob Howlett** (Sydney). Subsequently, Bob supplied code for working with directed W-graphs.

The original version of the code for root systems and permutation Coxeter groups was modelled, in part, on the Chevie package of GAP and implemented by **Don Taylor** (Sydney) with the assistance of **Frank Lübeck** (Aachen).

Functions that construct any finite irreducible unitary reflection group in C^n have been implemented by **Don Taylor** (Sydney). Extension to the infinite case was implemented by **Scott Murray** (Sydney).

The current version of Lie groups in MAGMA has been implemented by **Scott Murray** (Sydney) and **Sergei Haller** with some assistance from **Don Taylor** (Sydney).

An extensive package for computing the combinatorial properties of highest weight representations of a Lie algebra has been written by **Dan Roozemond** (Eindhoven). This code is based in the `LiE` package with permission of the authors.

Code has been contributed by **Robert Zeier** (Technical University of Munich) for determining the irreducible simple subalgebras of the Lie algebra $su(k)$.

Linear Algebra and Module Theory

Parts of the ATLAS (Automatically Tuned Linear Algebra Software) created by **R. Clint Whaley et al.** (UTSA) are used for some fundamental matrix algorithms over finite fields GF(p), where p is about the size of a machine integer.

Local Arithmetic Fields

Sebastian Pauli (TU Berlin) has implemented his algorithm for factoring polynomials over local fields within Magma. This algorithm may also be used for the factorization of ideals, the computation of completions of global fields, and for splitting extensions of local fields into towers of unramified and totally ramified extensions.

Modular Forms

Kevin Buzzard (Imperial College) made available his code for computing modular forms of weight one. The MAGMA implementation was developed using this as a starting point.

Lassina Dembélé (Warwick) wrote part of the code implementing his algorithm for computing Hilbert modular forms.

Enrique González-Jiménez (Madrid) contributed a package to compute curves over $\mathbf{Q}$, of genus at least 2, which are images of $X_1(N)$ for a given level N.

Matthew Greenberg (Calgary) and **John Voight** (Vermont) developed and implemented an algorithm for computing Hilbert modular forms using Shimura curves.

A new implementation (V2.19) of Brandt modules associated to definite quaternion orders, over $\mathbf{Z}$ and over function fields $\mathbf{F}_q[t]$, has been developed by **Markus Kirschmer** (Aachen) and **Steve Donnelly** (Magma).

David Kohel (Singapore-NUS, MAGMA) has provided implementations of division polynomials and isogeny structures for Brandt modules and modular curves. Jointly with **William Stein** (Harvard), he implemented the module of supersingular points.

Allan Lauder (Oxford) has contributed code for computing the characteristic polynomial of a Hecke operator acting on spaces of overconvergent modular forms.

MAGMA routines for constructing building blocks of modular abelian varieties were contributed by **Jordi Quer** (Cataluna).

A package for computing with modular symbols (known as HECKE) has been developed by **William Stein** (Harvard). William has also provided much of the package for modular forms.

In 2003–2004, **William Stein** (Harvard) developed extensive machinery for computing with modular abelian varieties within MAGMA.

A package for computing with congruence subgroups of the group $\mathrm{PSL}(2, \mathbf{R})$ has been developed by **Helena Verrill** (LSU).

John Voight (Vermont) produced the package for Shimura curves and arithmetic Fuchsian groups.

Dan Yasaki (UNC) provided the package for Bianchi modular forms.

Primality and Factorisation

The factorisation of integers of the form $p^n \pm 1$, for small primes p, makes use of tables compiled by **Richard Brent** that extend tables developed by the Cunningham project. In addition MAGMA uses Richard's intelligent factorization code FACTOR.

One of the main integer factorization tools available in MAGMA is due to **Arjen K. Lenstra** (EPFL) and his collaborators: a multiple polynomial quadratic sieve developed by Arjen from his "factoring by email" MPQS during visits to Sydney in 1995 and 1998.

The primality of integers is proven using the ECPP (Elliptic Curves and Primality Proving) package written by **François Morain** (Ecole Polytechnique and INRIA). The ECPP program in turn uses the BigNum package developed jointly by **INRIA** and **Digital PRL**.

MAGMA uses the **GMP-ECM** implementation of the Elliptic Curve Method (ECM) for integer factorisation. This was developed by **Pierrick Gaudry**, **Jim Fougeron**, **Laurent Fousse**, **Alexander Kruppa**, **Dave Newman**, and **Paul Zimmermann**. See `http://gforge.inria.fr/projects/ecm/`.

Real and Complex Arithmetic

The complex arithmetic in MAGMA uses the **MPC** package which is being developed by **Andreas Enge**, **Philippe Théveny** and **Paul Zimmermann**. (For more information see `www.multiprecision.org/mpc/`).

Xavier Gourdon (INRIA, Paris) made available his C implementation of A. Schönhage's splitting-circle algorithm for the fast computation of the roots of a polynomial to a specified precision. Xavier also assisted with the adaptation of his code for the MAGMA kernel.

Some portions of the **GNU GMP** multiprecision integer library (`http://gmplib.org`) are used for integer multiplication.

Most real arithmetic in MAGMA is based on the **MPFR** package which is developed by **Paul Zimmermann** (Nancy) and associates. (See `www.mpfr.org`).

Representation Theory

The algorithm of John Dixon for constructing the ordinary irreducible representation of a finite group from its character has been implemented by **Derek Holt** (Warwick).

Derek Holt (Warwick) has made a number of important contributions to the design of the module theory algorithms employed in MAGMA.

An algorithm of Sam Conlon for determining the degrees of the ordinary irreducible characters of a soluble group (without determining the full character table) has been implemented by **Derek Holt** (Warwick).

In 2011, **Derek Holt** (Warwick) and **John Cannon** (Magma) developed a package for computing the projective indecomposable KG-modules for a finite group G.

The algorithms used in MAGMA for finding the lattice of submodules and the endomorphism ring of a KG-module (K a finite field) were developed by **Charles Leedham-Green** (QMW, London) and **Allan Steel** (MAGMA).

Topology

A basic module for defining and computing with simplicial complexes was developed by **Mikael Johansson** (Jena).

Nathan Dunfield (Cornell) and **William Thurston** (Cornell) made available their database of the fundamental groups of the 10,986 small-volume closed hyperbolic manifolds in the Hodgson-Weeks census.

Handbook Contributors

Introduction

The Handbook of Magma Functions is the work of many individuals. It was based on a similar Handbook written for Cayley in 1990. Up until 1997 the Handbook was mainly written by Wieb Bosma, John Cannon and Allan Steel but in more recent times, as Magma expanded into new areas of mathematics, additional people became involved. It is not uncommon for some chapters to comprise contributions from 8 to 10 people. Because of the complexity and dynamic nature of chapter authorship, rather than ascribe chapter authors, in the table below we attempt to list those people who have made *significant* contributions to chapters.

We distinguish between:

• **Principal Author**, i.e. one who primarily conceived the core element(s) of a chapter and who was also responsible for the writing of a large part of its current content, and

◦ **Contributing Author**, i.e. one who has written a significant amount of content but who has not had primary responsibility for chapter design and overall content.

It should be noted that attribution of a person as an author of a chapter carries no implications about the authorship of the associated computer code: for some chapters it will be true that the author(s) listed for a chapter are also the authors of the corresponding code, but in many chapters this is either not the case or only partly true. Some information about code authorship may be found in the sections *Magma Development Team* and *External Contributors*.

The attributions given below reflect the authorship of the material comprising the V2.19 edition. Since many of the authors have since moved on to other careers, we have not been able to check that all of the attributions below are completely correct. We would appreciate hearing of any omissions.

In the chapter listing that follows, for each chapter the start of the list of principal authors (if any) is denoted by • while the start of the list of contributing authors is denoted by ◦.

People who have made minor contributions to one or more chapters are listed in a general acknowledgement following the chapter listing.

The Chapters

1 Statements and Expressions • *W. Bosma, A. Steel*
2 Functions, Procedures and Packages • *W. Bosma, A. Steel*
3 Input and Output • *W. Bosma, A. Steel*
4 Environment and Options • *A. Steel* ∘ *W. Bosma*
5 Magma Semantics • *G. Matthews*
6 The Magma Profiler • *D. Fisher*
7 Debugging Magma Code • *D. Fisher*
8 Introduction to Aggregates • *W. Bosma*
9 Sets • *W. Bosma, J. Cannon* ∘ *A. Steel*
10 Sequences • *W. Bosma, J. Cannon*
11 Tuples and Cartesian Products • *W. Bosma*
12 Lists • *W. Bosma*
13 Associative Arrays • *A. Steel*
14 Coproducts • *A. Steel*
15 Records • *W. Bosma*
16 Mappings • *W. Bosma*
17 Introduction to Rings • *W. Bosma*
18 Ring of Integers • *W. Bosma, A. Steel* ∘ *S. Contini, B. Smith*
19 Integer Residue Class Rings • *W. Bosma* ∘ *S. Donnelly, W. Stein*
20 Rational Field • *W. Bosma*
21 Finite Fields • *W. Bosma, A. Steel*
22 Nearfields • *D. Taylor*
23 Univariate Polynomial Rings • *A. Steel*
24 Multivariate Polynomial Rings • *A. Steel*
25 Real and Complex Fields • *W. Bosma*
26 Matrices • *A. Steel*
27 Sparse Matrices • *A. Steel*
28 Vector Spaces • *J. Cannon, A. Steel*
29 Polar Spaces • *D. Taylor*
30 Lattices • *A. Steel, D. Stehlé*
31 Lattices With Group Action • *B. Souvignier* ∘ *M. Kirschmer*
32 Quadratic Forms • *S. Donnelly*
33 Binary Quadratic Forms • *D. Kohel*
34 Number Fields • *C. Fieker* ∘ *W. Bosma, N. Sutherland*
35 Quadratic Fields • *W. Bosma*
36 Cyclotomic Fields • *W. Bosma, C. Fieker*
37 Orders and Algebraic Fields • *C. Fieker* ∘ *W. Bosma, N. Sutherland*
38 Galois Theory of Number Fields • *C. Fieker* ∘ *J. Klüners, K. Geißler*

39 Class Field Theory • *C. Fieker*
40 Algebraically Closed Fields • *A. Steel*
41 Rational Function Fields • *A. Steel* ◦ *A. van der Waall*
42 Algebraic Function Fields • *F. Heß* ◦ *C. Fieker, N. Sutherland*
43 Class Field Theory For Global Function Fields • *C. Fieker*
44 Artin Representations • *T. Dokchitser*
45 Valuation Rings • *W. Bosma*
46 Newton Polygons • *G. Brown, N. Sutherland*
47 p-adic Rings and their Extensions • *D. Fisher, B. Souvignier* ◦ *N. Sutherland*
48 Galois Rings • *A. Steel*
49 Power, Laurent and Puiseux Series • *A. Steel*
50 Lazy Power Series Rings • *N. Sutherland*
51 General Local Fields • *N. Sutherland*
52 Algebraic Power Series Rings • *T. Beck, M. Harrison*
53 Introduction to Modules • *J. Cannon*
54 Free Modules • *J. Cannon, A. Steel*
55 Modules over Dedekind Domains • *C. Fieker, N. Sutherland*
56 Chain Complexes • *J. Carlson*
57 Groups • *J. Cannon* ◦ *W. Unger*
58 Permutation Groups • *J. Cannon* ◦ *B. Cox, W. Unger*
59 Matrix Groups over General Rings • *J. Cannon* ◦ *B. Cox, E.A. O'Brien, A. Steel*
60 Matrix Groups over Finite Fields • *E.A. O'Brien*
61 Matrix Groups over Infinite Fields • *E.A. O'Brien*
62 Matrix Groups over Q and Z • *M. Kirschmer, B. Souvignier*
63 Finite Soluble Groups • *J. Cannon, M. Slattery* ◦ *E.A. O'Brien*
64 Black-box Groups • *W. Unger*
65 Almost Simple Groups ◦ *H. Bäärnhielm, J. Cannon, D. Holt, M. Stather*
66 Databases of Groups • *W. Unger* ◦ *V. Gebhardt*
67 Automorphism Groups • *D. Holt* ◦ *W. Unger*
68 Cohomology and Extensions • *D. Holt* ◦ *S. Haller*
69 Abelian Groups • *J. Cannon* ◦ *P. Lieby*
70 Finitely Presented Groups • *J. Cannon* ◦ *V. Gebhardt, E.A. O'Brien, M. Vaughan-Lee*
71 Finitely Presented Groups: Advanced • *H. Brückner, V. Gebhardt* ◦ *E.A. O'Brien*
72 Polycyclic Groups • *V. Gebhardt*
73 Braid Groups • *V. Gebhardt*
74 Groups Defined by Rewrite Systems • *D. Holt* ◦ *G. Matthews*
75 Automatic Groups • *D. Holt* ◦ *G. Matthews*
76 Groups of Straight-line Programs • *J. Cannon*

77 Finitely Presented Semigroups • *J. Cannon*
78 Monoids Given by Rewrite Systems • *D. Holt* ◦ *G. Matthews*
79 Algebras • *J. Cannon, B. Souvignier*
80 Structure Constant Algebras • *J. Cannon, B. Souvignier*
81 Associative Algebras ◦ *J. Cannon, S. Donnelly, N. Sutherland, B. Souvignier, J. Voight*
82 Finitely Presented Algebras • *A. Steel, S. Linton*
83 Matrix Algebras • *J. Cannon, A. Steel* ◦ *J. Carlson*
84 Group Algebras • *J. Cannon, B. Souvignier*
85 Basic Algebras • *J. Carlson* ◦ *M. Vejdemo-Johansson*
86 Quaternion Algebras • *D. Kohel, J. Voight* ◦ *S. Donnelly, M. Kirschmer*
87 Algebras With Involution • *P. Brooksbank, J. Wilson*
88 Clifford Algebras • *D. Taylor*
89 Modules over An Algebra • *J. Cannon, A. Steel*
90 $K[G]$-Modules and Group Representations • *J. Cannon, A. Steel*
91 Characters of Finite Groups • *W. Bosma, J. Cannon*
92 Representations of Symmetric Groups • *A. Kohnert*
93 Mod P Galois Representations • *J. Le Borgne*
94 Introduction to Lie Theory • *S. Murray* ◦ *D. Taylor*
95 Coxeter Systems • *S. Murray* ◦ *D. Taylor*
96 Root Systems • *S. Murray* ◦ *S. Haller, D. Taylor*
97 Root Data • *S. Haller, S. Murray* ◦ *D. Taylor*
98 Coxeter Groups • *S. Murray* ◦ *D. Taylor*
99 Reflection Groups • *S. Murray* ◦ *D. Taylor*
100 Lie Algebras • *W. de Graaf, D. Roozemond* ◦ *S. Haller, S. Murray*
101 Kac-moody Lie Algebras • *D. Roozemond*
102 Quantum Groups • *W. de Graaf*
103 Groups of Lie Type • *S. Murray* ◦ *S. Haller, D. Taylor*
104 Representations of Lie Groups and Algebras • *D. Roozemond* ◦ *S. Murray*
105 Gröbner Bases • *A. Steel* ◦ *M. Harrison*
106 Polynomial Ring Ideal Operations • *A. Steel* ◦ *M. Harrison*
107 Local Polynomial Rings • *A. Steel*
108 Affine Algebras • *A. Steel*
109 Modules over Multivariate Rings • *A. Steel* ◦ *M. Harrison*
110 Invariant Theory • *A. Steel*
111 Differential Rings • *A. van der Waall*
112 Schemes • *G. Brown* ◦ *J. Cannon, M. Harrison, N. Sutherland*
113 Coherent Sheaves • *M. Harrison*
114 Algebraic Curves • *G. Brown* ◦ *N. Bruin, J. Cannon, M. Harrison, A. Wilson*
115 Resolution Graphs and Splice Diagrams • *G. Brown*

116 Algebraic Surfaces • *T. Beck, M. Harrison*
117 Hilbert Series of Polarised Varieties • *G. Brown*
118 Toric Varieties • *G. Brown, A. Kasprzyk*
119 Rational Curves and Conics • *D. Kohel, P. Lieby* ◦ *S. Donnelly, M. Watkins*
120 Elliptic Curves • *G. Bailey* ◦ *S. Donnelly, D. Kohel*
121 Elliptic Curves over Finite Fields • *M. Harrison* ◦ *P. Lieby*
122 Elliptic Curves over $\mathbf{Q}$ and Number Fields ◦ *G. Bailey, N. Bruin, B. Creutz, S. Donnelly, D. Kohel, M. Watkins*
123 Elliptic Curves over Function Fields • *J. Scholten* ◦ *S. Donnelly*
124 Models of Genus One Curves • *T. Fisher, S. Donnelly*
125 Hyperelliptic Curves ◦ *N. Bruin, B. Creutz, S. Donnelly, M. Harrison, D. Kohel, P. van Wamelen*
126 Hypergeometric Motives • *M. Watkins*
127 L-functions • *T. Dokchitser* ◦ *M. Watkins*
128 Modular Curves • *D. Kohel* ◦ *M. Harrison, E. González-Jiménez*
129 Small Modular Curves • *M. Harrison*
130 Congruence Subgroups of $\mathrm{PSL}_2(\mathbf{R})$ • *H. Verrill*
131 Arithmetic Fuchsian Groups and Shimura Curves • *J. Voight*
132 Modular Forms • *W. Stein* ◦ *K. Buzzard, S. Donnelly*
133 Modular Symbols • *W. Stein* ◦ *K. Buzzard*
134 Brandt Modules • *D. Kohel*
135 Supersingular Divisors on Modular Curves • *D. Kohel, W. Stein*
136 Modular Abelian Varieties • *W. Stein* ◦ *J. Quer*
137 Hilbert Modular Forms • *S. Donnelly*
138 Modular Forms over Imaginary Quadratic Fields • *D. Yasaki* ◦ *S. Donnelly*
139 Admissible Representations of $\mathrm{GL}_2(\mathbf{Q}_p)$ • *J. Weinstein* ◦ *S. Donnelly*
140 Simplicial Homology • *M. Vejdemo-Johansson*
141 Finite Planes • *J. Cannon*
142 Incidence Geometry • *D. Leemans*
143 Convex Polytopes and Polyhedra • *G. Brown, A. Kasprzyk*
144 Enumerative Combinatorics • *G. Bailey* ◦ *G. White*
145 Partitions, Words and Young Tableaux • *G. White*
146 Symmetric Functions • *A. Kohnert*
147 Incidence Structures and Designs • *J. Cannon*
148 Hadamard Matrices • *G. Bailey*
149 Graphs • *J. Cannon, P. Lieby* ◦ *G. Bailey*
150 Multigraphs • *J. Cannon, P. Lieby*
151 Networks • *P. Lieby*
152 Linear Codes over Finite Fields • *J. Cannon, A. Steel* ◦ *G. White*

General Acknowledgements

In addition to the contributors listed above, we gratefully acknowledge the contributions to the Handbook made by the following people:

J. Brownie (group theory)
K. Geißler (Galois groups)
A. Flynn (algebras and designs)
E. Herrmann (elliptic curves)
E. Howe (Igusa invariants)
B. McKay (graph theory)
S. Pauli (local fields)
C. Playoust (data structures, rings)
C. Roney-Dougal (groups)
T. Womack (elliptic curves)

USING THE HANDBOOK

Most sections within a chapter of this Handbook consist of a brief introduction and explanation of the notation, followed by a list of MAGMA functions, procedures and operators.

Each entry in this list consists of an expression in a box, and an indented explanation of use and effects. The `typewriter` typefont is used for commands that can be used literally; however, one should be aware that most functions operate on variables that must have values assigned to them beforehand, and return values that should be assigned to variables (or the first value should be used in an expression). Thus the entry:

```
Xgcd(a, b)
```

The extended gcd; returns integers d, l and m such that d is the greatest common divisor of the integers a and b, and $d = l * a + m * b$.

indicates that this function could be called in MAGMA as follows:

```
g, a, b := Xgcd(23, 28);
```

If the function has optional named *parameters*, a line like the following will be found in the description:

`Proof` BOOLELT *Default :* `true`

The first word will be the name of the parameter, the second word will be the type which its value should have, and the rest of the line will indicate the default for the parameter, if there is one. Parameters for a function call are specified by appending a colon to the last argument, followed by a comma-separated list of assignments (using `:=`) for each parameter. For example, the function call `IsPrime(n: Proof := false)` calls the function `IsPrime` with argument n but also with the value for the parameter `Proof` set to `false`.

Whenever the symbol `#` precedes a function name in a box, it indicates that the particular function is not yet available but should be in the future.

An index is provided at the end of each volume which contains all the intrinsics in the Handbook.

Running the Examples

All examples presented in this Handbook are available to MAGMA users. If your MAGMA environment has been set up correctly, you can load the source for an example by using the name of the example as printed in boldface at the top (the name has the form HmEn, where m is the Chapter number and n is the Example number). So, to run the first example in the Chapter 28, type:

```
load "H28E1";
```

OVERVIEW

CONTENTS

PART I
THE MAGMA LANGUAGE

1 STATEMENTS AND EXPRESSIONS

Chapter 1
STATEMENTS AND EXPRESSIONS

1.1 Introduction

This chapter contains a very terse overview of the basic elements of the MAGMA language.

1.2 Starting, Interrupting and Terminating

If MAGMA has been installed correctly, it may be activated by typing 'magma'.

`<Ctrl>-C`

Interrupt MAGMA while it is performing some task (that is, while the user does not have a 'prompt') to obtain a new prompt. MAGMA will try to interrupt at a convenient point (this may take some time). If `<Ctrl>-C` is typed twice within half a second, MAGMA will exit completely immediately.

`quit;`

`<Ctrl>-D`

Terminate the current MAGMA-session.

`<Ctrl>-\`

Immediately quit MAGMA (send the signal SIGQUIT to the MAGMA process on Unix machines). This is occasionally useful when `<Ctrl>-C` does not seem to work.

1.3 Identifiers

Identifiers (names for user variables, functions etc.) must begin with a letter, and this letter may be followed by any combination of letters or digits, provided that the name is not a *reserved word* (see the chapter on reserved words a complete list). In this definition the underscore _ is treated as a letter; but note that a single underscore is a reserved word. Identifier names are case-sensitive; that is, they are distinguished from one another by lower and upper case.

Intrinsic MAGMA functions usually have names beginning with capital letters (current exceptions are `pCore, pQuotient` and the like, where the `p` indicates a prime). Note that these identifiers are *not* reserved words; that is, one may use names of intrinsic functions for variables.

1.4 Assignment

In this section the basic forms of assignment of values to identifiers are described.

1.4.1 Simple Assignment

`x := expression;`

Given an identifier `x` and an expression *expression*, assign the value of *expression* to `x`.

Example H1E1

```
> x := 13;
> y := x^2-2;
> x, y;
13 167
```

Intrinsic function names are identifiers just like the x and y above. Therefore it is possible to reassign them to your own variable.

```
> f := PreviousPrime;
> f(y);
163
```

In fact, the same can also be done with the infix operators, except that it is necessary to enclose their names in quotes. Thus it is possible to define your own function `Plus` to be the function taking the arguments of the intrinsic `+` operator.

```
> Plus := '+';
> Plus(1/2, 2);
5/2
```

Note that redefining the infix operator will *not* change the corresponding mutation assignment operator (in this case `+:=`).

$\texttt{x}_1$`, `$\texttt{x}_2$`, ..., `$\texttt{x}_n$` := expression;`

Assignment of $n \geq 1$ values, returned by the expression on the right hand side. Here the $\texttt{x}_i$ are identifiers, and the right hand side expression must return $m \geq n$ values; the first n of these will be assigned to $\texttt{x}_1$`, `$\texttt{x}_2$`, ..., `$\texttt{x}_n$ respectively.

`_ := expression;`

Ignore the value(s) returned by the expression on the right hand side.

`assigned x`

An expression which yields the value `true` if the 'local' identifier x has a value currently assigned to it and `false` otherwise. Note that the `assigned`-expression will return `false` for intrinsic function names, since they are not 'local' variables (the identifiers can be assigned to something else, hiding the intrinsic function).

Example H1E2

The extended greatest common divisor function `Xgcd` returns 3 values: the gcd d of the arguments m and n, as well as multipliers x and y such that $d = xm + yn$. If one is only interested in the gcd of the integers $m = 12$ and $n = 15$, say, one could use:

```
> d := Xgcd(12, 15);
```

To obtain the multipliers as well, type

```
> d, x, y := Xgcd(12, 15);
```

while the following offers ways to retrieve two of the three return values.

```
> d, x := Xgcd(12, 15);
> d, _, y := Xgcd(12, 15);
> _, x, y := Xgcd(12, 15);
```

1.4.2 Indexed Assignment

`x[`*expression*$_1$`][`*expression*$_2$`]...[`*expression*$_n$`] := `*expression*`;`

`x[`*expression*$_1$`,`*expression*$_2$`,...,`*expression*$_n$`] := `*expression*`;`

If the argument on the left hand side allows *indexing* at least n levels deep, and if this indexing can be used to modify the argument, this offers two equivalent ways of accessing and modifying the entry indicated by the expressions `expr`$_i$. The most important case is that of (nested) sequences.

Example H1E3

Left hand side indexing can be used (as is explained in more detail in the chapter on sequences) to modify existing entries.

```
> s := [ [1], [1, 2], [1, 2, 3] ];
> s;
[
       [ 1 ],
       [ 1, 2 ],
       [ 1, 2, 3 ]
]
> s[2, 2] := -1;
> s;
[
       [ 1 ],
       [ 1, -1 ],
       [ 1, 2, 3 ]
]
```

1.4.3 Generator Assignment

Because of the importance of naming the generators in the case of finitely presented magmas, special forms of assignment allow names to be assigned at the time the magma itself is assigned.

`E<`x_1`, `x_2`, ...`x_n`> :=` *expression*`;`

If the right hand side expression returns a structure that allows *naming* of 'generators', such as finitely generated groups or algebras, polynomial rings, this assigns the first n names to the variables x_1, x_2, ..., x_n. Naming of generators usually has two aspects; firstly, the *strings* x_1, x_2, ...x_n are used for printing of the generators, and secondly, to the *identifiers* x_1, x_2, ...x_n are assigned the values of the generators. Thus, except for this side effect regarding printing, the above assignment is equivalent to the $n + 1$ assignments:

`E :=` *expression*`;`
x_1 `:= E.1;` x_2 `:= E.2; ...` x_n `:= E.`n`;`

`E<[x]> :=` *expression*`;`

If the right hand side expression returns a structure S that allows *naming* of 'generators', this assigns the names of S to be those formed by appending the numbers 1, 2, etc. in order enclosed in square brackets to x (considered as a string) and assigns x to the sequence of the names of S.

Example H1E4

We demonstrate the sequence method of generator naming.

```
> P<[X]> := PolynomialRing(RationalField(), 5);
> P;
Polynomial ring of rank 5 over Rational Field
Lexicographical Order
Variables: X[1], X[2], X[3], X[4], X[5]
> X;
[
    X[1],
    X[2],
    X[3],
    X[4],
    X[5]
]
> &+X;
X[1] + X[2] + X[3] + X[4] + X[5]
> (&+X)^2;
X[1]^2 + 2*X[1]*X[2] + 2*X[1]*X[3] + 2*X[1]*X[4] +
    2*X[1]*X[5] + X[2]^2 + 2*X[2]*X[3] + 2*X[2]*X[4] +
    2*X[2]*X[5] + X[3]^2 + 2*X[3]*X[4] + 2*X[3]*X[5] +
    X[4]^2 + 2*X[4]*X[5] + X[5]^2
```

`AssignNames(~S, [s1, ... sn] )`

If S is a structure that allows *naming* of 'generators' (see the Index for a complete list), this procedure assigns the names specified by the strings to these generators. The number of generators has to match the length of the sequence. This will result in the creation of a new structure.

Example H1E5

```
> G<a, b> := Group<a, b | a^2 = b^3 = a^b*b^2>;
> w := a * b;
> w;
a * b
> AssignNames(~G, ["c", "d"]);
> G;
Finitely presented group G on 2 generators
Relations
    c^2 = d^-1 * c * d^3
    d^3 = d^-1 * c * d^3
> w;
a * b
> Parent(w);
Finitely presented group on 2 generators
Relations
    a^2 = b^-1 * a * b^3
    b^3 = b^-1 * a * b^3
> G eq Parent(w);
true
```

1.4.4 Mutation Assignment

`x o:=` *expression*`;`

This is the *mutation assignment*: the expression is evaluated and the operator `o` is applied on the result and the current value of x, and assigned to x again. Thus the result is equivalent to (but an optimized version of): `x := x o` *expression*`;`. The operator may be any of the operations `join`, `meet`, `diff`, `sdiff`, `cat`, `*`, `+`, `-`, `/`, `^`, `div`, `mod`, `and`, `or`, `xor` provided that the operation is legal on its arguments of course.

Example H1E6

The following simple program to produce a set consisting of the first 10 powers of 2 involves the use of two different mutation assignments.

```
> x := 1;
> S := { };
> for i := 1 to 10 do
>    S join:= { x };
>    x *:= 2;
> end for;
> S;
{ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 }
```

1.4.5 Deletion of Values

`delete x`

(Statement.) Delete the current value of the identifier x. The memory occupied is freed, unless other variables still refer to it. If x is the name of an intrinsic MAGMA function that has been reassigned to, the identifier will after deletion again refer to that intrinsic function. Intrinsic functions cannot be deleted.

1.5 Boolean values

This section deals with logical values ("Booleans").

Booleans are primarily of importance as (return) values for (intrinsic) predicates. It is important to know that the truth-value of the operators `and` and `or` is always evaluated *left to right*, that is, the left-most clause is evaluated first, and if that determines the value of the operator evaluation is aborted; if not, the next clause is evaluated, etc. So, for example, if x is a boolean, it is safe (albeit silly) to type:

```
> if x eq true or x eq false or x/0 eq 1 then
>    "fine";
> else
>    "error";
> end if;
```

even though $x/0$ would cause an error ("Bad arguments", not "Division by zero"!) upon evaluation, because the truth value will have been determined before the evaluation of `x/0` takes place.

1.5.1 Creation of Booleans

`Booleans()`

The Boolean structure.

`#B`

Cardinality of Boolean structure (2).

`true`

`false`

The Boolean elements.

`Random(B)`

Return a random Boolean.

1.5.2 Boolean Operators

`x and y`

Returns `true` if both x and y are `true`, `false` otherwise. If x is `false`, the expression for y is not evaluated.

`x or y`

Returns `true` if x or y is `true` (or both are `true`), `false` otherwise. If x is `true`, the expression for y is not evaluated.

`x xor y`

Returns `true` if either x or y is `true` (but not both), `false` otherwise.

`not x`

Negate the truth value of x.

1.5.3 Equality Operators

MAGMA provides two equality operators: `eq` for strong (comparable) equality testing, and `cmpeq` for weak equality testing. The operators depend on the concept of *comparability*. Objects x and y in MAGMA are said to be *comparable* if both of the following points hold:

(a) x and y are both elements of a structure S or there is a structure S such x and y will be coerced into S by automatic coercion;

(b) There is an equality test for elements of S defined within MAGMA.

The possible automatic coercions are listed in the descriptions of the various MAGMA modules. For instance, the table in the introductory chapter on rings shows that integers can be coerced automatically into the rational field so an integer and a rational are comparable.

`x eq y`

If x and y are comparable, return `true` if x equals y (which will always work by the second rule above). If x and y are not comparable, an error results.

`x ne y`

If x and y are comparable, return `true` if x does not equal y. If x and y are not comparable, an error results.

`x cmpeq y`

If x and y are comparable, return whether x equals y. Otherwise, return `false`. Thus this operator always returns a value and an error never results. It is useful when comparing two objects of completely different types where it is desired that no error can happen. However, it is strongly recommended that `eq` is usually used to allow MAGMA to pick up common unintentional type errors.

`x cmpne y`

If x and y are comparable, return whether x does not equal y. Otherwise, return `true`. Thus this operator always returns a value and an error never results. It is useful when comparing two objects of completely different types where it is desired that no error can happen. However, it is strongly recommended that `ne` is usually used to allow MAGMA to pick up common unintentional type errors.

Example H1E7

We illustrate the different semantics of `eq` and `cmpeq`.

```
> 1 eq 2/2;
true
> 1 cmpeq 2/2;
true
>  1 eq "x";
Runtime error in 'eq': Bad argument types
> 1 cmpeq "x";
false
>  [1] eq ["x"];
Runtime error in 'eq': Incompatible sequences
> [1] cmpeq ["x"];
false
```

1.5.4 Iteration

A Boolean structure B may be used for enumeration: `for x in B do`, and `x in B` in set and sequence constructors.

Example H1E8

The following program checks that the functions `ne` and `xor` coincide.

```
> P := Booleans();
> for x, y in P do
>      (x ne y) eq (x xor y);
> end for;
true
true
true
true
```

Similarly, we can test whether for any pair of Booleans x, y it is true that

$$x = y \quad \Longleftrightarrow \quad (x \wedge y) \vee (\neg x \wedge \neg y).$$

```
> equal := true;
> for x, y in P do
>     if (x eq y) and not ((x and y) or (not x and not y)) then
>         equal := false;
>     end if;
> end for;
> equal;
true
```

1.6 Coercion

Coercion is a fundamental concept in MAGMA. Given a structures A and B, there is often a natural mathematical mapping from A to B (e.g., embedding, projection), which allows one to transfer elements of A to corresponding elements of B. This is known as coercion. Natural and obvious coercions are supported in MAGMA as much as possible; see the relevant chapters for the coercions possible between various structures.

`S ! x`

Given a structure S and an object x, attempt to coerce x into S and return the result if successful. If the attempt fails, an error ensues.

`IsCoercible(S, x)`

Given a structure S and an object x, attempt to coerce x into S; if successful, return `true` and the result of the coercion, otherwise return `false`.

1.7 The `where ... is` Construction

By the use of the `where ... is` construction, one can within an expression temporarily assign an identifier to a sub-expression. This allows for compact code and efficient re-use of common sub-expressions.

expression$_1$ `where` *identifier* `is` *expression*$_2$
expression$_1$ `where` *identifier* `:=` *expression*$_2$

This construction is an expression that temporarily assigns the identifier to the second expression and then yields the value of the first expression. The identifier may be referred to in the first expression and it will equal the value of the second expression. The token `:=` can be used as a synonym for `is`. The scope of the identifier is the `where ... is` construction alone except for when the construction is part of an expression list — see below.

The `where` operator is left-associative. This means that there can be multiple uses of `where ... is` constructions and each expression can refer to variables bound in the enclosing constructions.

Another important feature is found in a set or sequence constructor. If there are `where ... is` constructions in the predicate, then any variables bound in them may be referred to in the expression at the beginning of the constructor. If the whole predicate is placed in parentheses, then any variables bound in the predicate do not extend to the expression at the beginning of the constructor.

The `where` operator also extends left in expression lists. That is, if there is an expression E in a expression list which is a `where` construction (or chain of where constructions), the identifiers bound in that where construction (or chain) will be defined in all expressions in the list which are to the left of E. Expression lists commonly arise as argument lists to functions or procedures, return arguments, print statements (with or without the word 'print') etc. A where construction also overrides (hides) any where construction to the right of it in the same list. Using parentheses around a where expression ensures that the identifiers bound within it are not seen outside it.

Example H1E9

The following examples illustrate simple uses of `where ... is`.

```
> x := 1;
> x where x is 10;
10
> x;
1
> Order(G) + Degree(G) where G is Sym(3);
9
```

Since `where` is left-associative we may have multiple uses of it. The use of parentheses, of course, can override the usual associativity.

```
> x := 1;
```

```
> y := 2;
> x + y where x is 5 where y is 6;
11
> (x + y where x is 5) where y is 6; // the same
11
> x + y where x is (5 where y is 6);
7
> x + y where x is y where y is 6;
12
> (x + y where x is y) where y is 6; // the same
12
> x + y where x is (y where y is 6);
8
```

We now illustrate how the left expression in a set or sequence constructor can reference the identifiers of `where` constructions in the predicate.

```
> { a: i in [1 .. 10] | IsPrime(a) where a is 3*i + 1 };
{  7, 13, 19, 31 }
> [<x, y>: i in [1 .. 10] | IsPrime(x) and IsPrime(y)
>    where x is y + 2 where y is 2 * i + 1];
[ <5, 3>, <7, 5>, <13, 11>, <19, 17> ]
```

We next demonstrate the semantics of `where` constructions inside expression lists.

```
> // A simple use:
> [a, a where a is 1];
[ 1, 1 ]
>  // An error: where does not extend right
>  print [a where a is 1, a];
User error: Identifier 'a' has not been declared
> // Use of parentheses:
> [a, (a where a is 1)] where a is 2;
[ 2, 1 ]
>  // Another use of parentheses:
>  print [a, (a where a is 1)];
User error: Identifier 'a' has not been declared
> // Use of a chain of where expressions:
> [<a, b>, <b, a> where a is 1 where b is 2];
[ <1, 2>, <2, 1> ]
> // One where overriding another to the right of it:
> [a, a where a is 2, a where a is 3];
[ 2, 2, 3 ]
```

1.8 Conditional Statements and Expressions

The conditional statement has the usual form `if ... then ... else ... end if;`. It has several variants. Within the statement, a special prompt will appear, indicating that the statement has yet to be closed. Conditional statements may be nested.

The conditional expression, `select ... else`, is used for in-line conditionals.

1.8.1 The Simple Conditional Statement

```
if Boolean expression then
   statements1
else
   statements2
end if;
```

```
if Boolean expression then
   statements
end if;
```

The standard conditional statement: the value of the Boolean expression is evaluated. If the result is `true`, the first block of statements is executed, if the result is `false` the second block of statements is executed. If no action is desired in the latter case, the construction may be abbreviated to the second form above.

```
if Boolean expression1 then
   statements1
elif Boolean expression2 then
   statements2
else
   statements3
end if;
```

Since nested conditions occur frequently, `elif` provides a convenient abbreviation for `else if`, which also restricts the 'level':

```
if Boolean expression then
   statements1
elif Boolean expression2 then
   statements2
else
   statements3
end if;
```

is equivalent to

```
if Boolean expression1 then
    statements1
else
    if Boolean expression2 then
```

```
        statements2
    else
        statements3
    end if;
end if;
```

Example H1E10

```
> m := Random(2, 10000);
> if IsPrime(m) then
>     m, "is prime";
> else
>     Factorization(m);
> end if;
[ <23, 1>, <37, 1> ]
```

1.8.2 The Simple Conditional Expression

Boolean expression `select` *expression*$_1$ `else` *expression*$_2$

This is an expression, of which the value is that of *expression*$_1$ or *expression*$_2$, depending on whether *Boolean expression* is `true` or `false`.

Example H1E11

Using the `select ... else` construction, we wish to assign the sign of y to the variable s.

```
> y := 11;
> s := (y gt 0) select 1 else -1;
> s;
1
```

This is not quite right (when $y = 0$), but fortunately we can nest `select ... else` constructions:

```
> y := -3;
> s := (y gt 0) select 1 else (y eq 0 select 0 else -1);
> s;
-1
> y := 0;
> s := (y gt 0) select 1 else (y eq 0 select 0 else -1);
> s;
0
```

The `select ... else` construction is particularly important in building sets and sequences, because it enables in-line `if` constructions. Here is a sequence containing the first 100 entries of the Fibonacci sequence:

```
>  f := [ i gt 2 select Self(i-1)+Self(i-2) else 1 : i in [1..100] ];
```

1.8.3 The Case Statement

```
case expression :
   when expression, ..., expression:
      statements
          ⋮
   when expression, ..., expression:
      statements
end case;
```

The expression following `case` is evaluated. The statements following the first expression whose value equals this value are executed, and then the `case` statement has finished. If none of the values of the expressions equal the value of the `case` expression, then the statements following `else` are executed. If no action is desired in the latter case, the construction may be abbreviated to the second form above.

Example H1E12

```
> x := 73;
> case Sign(x):
>    when 1:
>       x, "is positive";
>    when 0:
>       x, "is zero";
>    when -1:
>       x, "is negative";
> end case;
73 is positive
```

1.8.4 The Case Expression

```
case< expression |
      expression_left,1 : expression_right,1 ,
          ⋮
      expression_left,n : expression_right,n ,
      default : expression_def >
```

This is the expression form of `case`. The *expression* is evaluated to the value v. Then each of the left-hand expressions $expression_{\text{left},i}$ is evaluated until one is found whose value equals v; if this happens the value of the corresponding right-hand expression $expression_{\text{right},i}$ is returned. If no left-hand expression with value v is found the value of the default expression $expression_{\text{def}}$ is returned.

The default case cannot be omitted, and must come last.

1.9 Error Handling Statements

MAGMA has facilities for both reporting and handling errors. Errors can arise in a variety of circumstances within MAGMA's internal code (due to, for instance, incorrect usage of a function, or the unexpected failure of an algorithm). MAGMA allows the user to raise errors in their own code, as well as catch many kinds of errors.

1.9.1 The Error Objects

All errors in MAGMA are of type `Err`. Error objects not only include a description of the error, but also information relating to the location at which the error was raised, and whether the error was a user error, or a system error.

`Error(x)`

Constructs an error object with user information given by x, which can be of any type. The object x is stored in the `Object` attributed of the constructed error object, and the `Type` attribute of the object is set to "ErrUser". The remaining attributes are uninitialized until the error is raised by an `error` statement; at that point they are initialized with the appropriate positional information.

`e‘Position`

Stores the position at which the error object e was raised. If the error object has not yet been raised, the attribute is undefined.

`e‘Traceback`

Stores the stack traceback giving the position at which the error object e was raised. If the error object has not yet been raised, the attribute is undefined.

`e‘Object`

Stores the user defined error information for the error. If the error is a system error, then this will be a string giving a textual description of the error.

`e‘Type`

Stores the type of the error. Currently, there are only two types of errors in Magma: "Err" denotes a system error, and "ErrUser" denotes an error raised by the user.

1.9.2 Error Checking and Assertions

`error` *expression*`, ..., `*expression*`;`

Raises an error, with the error information being the printed value of the expressions. This statement is useful, for example, when an illegal value of an argument is passed to a function.

`error if` *Boolean expression*`, `*expression*`, ..., `*expression*`;`

If the given boolean expression evaluates to `true`, then raises an error, with the error information being the printed value of the expressions. This statement is designed for checking that certain conditions must be met, etc.

```
assert Boolean expression;
```

```
assert2 Boolean expression;
```

```
assert3 Boolean expression;
```

These assertion statements are useful to check that certain conditions are satisfied. There is an underlying `Assertions` flag, which is set to 1 by default.

For each statement, if the `Assertions` flag is less than the level specified by the statement (respectively 1, 2, 3 for the above statements), then nothing is done. Otherwise, the given boolean expression is evaluated and if the result is `false`, an error is raised, with the error information being an appropriate message.

It is recommended that when developing package code, `assert` is used for important tests (always to be tested in any mode), while `assert2` is used for more expensive tests, only to be checked in the debug mode, while `assert3` is be used for extremely stringent tests which are very expensive.

Thus the `Assertions` flag can be set to 0 for no checking at all, 1 for normal checks, 2 for debug checks and 3 for extremely stringent checking.

1.9.3 Catching Errors

```
try
   statements1
catch e
   statements2
end try;
```

The `try/catch` statement lets users handle raised errors. The semantics of a `try/catch` statement are as follows: the block of statements $statements_1$ is executed. If no error is raised during its execution, then the block of statements $statements_2$ is not executed; if an error is raised at any point in $statements_1$, execution *immediately* transfers to $statements_2$ (the remainder of $statements_1$ is not executed). When transfer is controlled to the `catch` block, the variable named e is initialized to the error that was raised by $statements_1$; this variable remains in scope until the end of the `catch` block, and can be both read from and written to. The catch block can, if necessary, reraise e, or any other error object, using an `error` statement.

Example H1E13

The following example demonstrates the use of error objects, and `try/catch` statements.

```
> procedure always_fails(x)
>     error Error(x);
> end procedure;
>
> try
>     always_fails(1);
```

```
>     always_fails(2);  // we never get here
> catch e
>     print "In catch handler";
>     error "Error calling procedure with parameter: ", e`Object;
> end try;
In catch handler
Error calling procedure with parameter:  1
```

1.10 Iterative Statements

Three types of iterative statement are provided in MAGMA: the `for`-statement providing definite iteration and the `while`- and `repeat`-statements providing indefinite iteration.

Iteration may be performed over an arithmetic progression of integers or over any finite enumerated structure. Iterative statements may be nested. If nested iterations occur over the same enumerated structure, abbreviations such as `for x, y in X do` may be used; the leftmost identifier will correspond to the outermost loop, etc. (For nested iteration in sequence constructors, see Chapter 10.)

Early termination of the body of loop may be specified through use of the 'jump' commands `break` and `continue`.

1.10.1 Definite Iteration

`for i :=` *expression*$_1$ `to` *expression*$_2$ `by` *expression*$_3$ `do`
statements
`end for;`

The expressions in this `for` loop must return integer values, say b, e and s (for 'begin', 'end' and 'step') respectively. The loop is ignored if either $s > 0$ and $b > e$, or $s < 0$ and $b < e$. If $s = 0$ an error occurs. In the remaining cases, the value $b + k \cdot s$ will be assigned to `i`, and the statements executed, for $k = 0, 1, 2, \ldots$ in succession, as long as $b + k \cdot s \leq e$ (for $e > 0$) or $b + k \cdot s \geq e$ (for $e < 0$).

If the required step size is 1, the above may be abbreviated to:

`for i :=` *expression*$_1$ `to` *expression*$_2$ `do`
statements
`end for;`

`for x in S do`
statements
`end for;`

Each of the elements of the finite enumerated structure S will be assigned to x in succession, and each time the statements will be executed. It is possible to nest several of these `for` loops compactly as follows.

`for` $\mathtt{x}_{11}$`, ...,` $\mathtt{x}_{1n_1}$ `in` $\mathtt{S}_1$`, ...,` $\mathtt{x}_{m1}$`, ...,` $\mathtt{x}_{mn_m}$ `in` $\mathtt{S}_m$ `do`
statements
`end for;`

1.10.2 Indefinite Iteration

```
while Boolean expression do
   statements
end while;
```

Check whether or not the Boolean expression has the value `true`; if it has, execute the statements. Repeat this until the expression assumes the value `false`, in which case statements following the `end while;` will be executed.

Example H1E14

The following short program implements a run of the famous $3x+1$ problem on a random integer between 1 and 100.

```
> x := Random(1, 100);
> while x gt 1 do
> x;
>     if IsEven(x) then
>       x div:= 2;
>     else
>        x := 3*x+1;
>     end if;
> end while;
13
40
20
10
5
16
8
4
2
```

```
repeat
   statements
until Boolean expression;
```

Execute the statements, then check whether or not the Boolean expression has the value `true`. Repeat this until the expression assumes the value `false`, in which case the loop is exited, and statements following it will be executed.

Example H1E15

This example is similar to the previous one, except that it only prints x and the number of steps taken before x becomes 1. We use a `repeat` loop, and show that the use of a `break` statement sometimes makes it unnecessary that the Boolean expression following the `until` ever evaluates to `true`. Similarly, a `while true` statement may be used if the user makes sure the loop will be exited using `break`.

```
> x := Random(1, 1000);
> x;
172
> i := 0;
> repeat
>     while IsEven(x) do
>         i +:= 1;
>         x div:= 2;
>     end while;
>     if x eq 1 then
>         break;
>     end if;
>     x := 3*x+1;
>     i +:= 1;
> until false;
> i;
31
```

1.10.3 Early Exit from Iterative Statements

`continue;`

The `continue` statement can be used to jump to the end of the innermost enclosing loop: the termination condition for the loop is checked immediately.

`continue` *identifier*`;`

As in the case of `break`, this allows jumps out of nested `for` loops: the termination condition of the loop with loop variable *identifier* is checked immediately after `continue` *identifier* is encountered.

`break;`

A `break` inside a loop causes immediate exit from the innermost enclosing loop.

`break` *identifier*`;`

In nested `for` loops, this allows breaking out of several loops at once: this will cause an immediate exit from the loop with loop variable *identifier*.

Example H1E16

```
> p := 10037;
> for x in [1 .. 100] do
>    for y in [1 .. 100] do
>       if x^2 + y^2 eq p then
>          x, y;
>          break x;
>       end if;
>    end for;
> end for;
46 89
```

Note that `break` instead of `break x` would have broken only out of the inner loop; the output in that case would have been:

```
46 89
89 46
```

1.11 Runtime Evaluation: the eval Expression

Sometimes it is convenient to able to evaluate expressions that are dynamically constructed at runtime. For instance, consider the problem of implementing a database of mathematical objects in MAGMA. Suppose that these mathematical objects are very large, but can be constructed in only a few lines of MAGMA code (a good example of this would be MAGMA's database of best known linear codes). It would be very inefficient to store these objects in a file for later retrieval; a better solution would be to instead store a string giving the code necessary to construct each object. MAGMA's `eval` feature can then be used to dynamically parse and execute this code on demand.

`eval` *expression*

The `eval` expression works as follows: first, it evaluates the given *expression*, which must evaluate to a string. This string is then treated as a piece of MAGMA code which yields a result (that is, the code must be an expression, not a statement), and this result becomes the result of the `eval` expression.

The string that is evaluated can be of two forms: it can be a MAGMA expression, e.g., "`1+2`", "`Random(x)`", or it can be a sequence of MAGMA statements. In the first case, the string does not have to be terminated with a semicolon, and the result of the expression given in the string will be the result of the `eval` expression. In the second case, the last statement given in the string should be a `return` statement; it is easiest to think of this case as defining the body of a function.

The string that is used in the `eval` expression can refer to any variable that is in scope during the evaluation of the `eval` expression. However, it is not possible for the expression to *modify* any of these variables.

Example H1E17

In this example we demonstrate the basic usage of the `eval` keyword.

```
> x := eval "1+1";  // OK
> x;
2
> eval "1+1;"; // not OK
2
>> eval "1+1;"; // not OK
   ^
Runtime error: eval must return a value
> eval "return 1+1;"; // OK
2
> eval "x + 1"; // OK
3
> eval "x := x + 1; return x";
>> eval "x := x + 1; return x";
   ^
In eval expression, line 1, column 1:
>> x := x + 1; return x;
   ^
    Located in:
    >> eval "x := x + 1; return x";
       ^
User error: Imported environment value 'x' cannot be used as a local
```

Example H1E18

In this example we demonstrate how `eval` can be used to construct MAGMA objects specified with code only available at runtime.

```
> M := Random(MatrixRing(GF(2), 5));
> M;
[1 1 1 1 1]
[0 0 1 0 1]
[0 0 1 0 1]
[1 0 1 1 1]
[1 1 0 1 1]
> Write("/tmp/test", M, "Magma");
> s := Read("/tmp/test");
> s;
MatrixAlgebra(GF(2), 5) ! [ GF(2) | 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1,
1, 0, 1, 1, 1, 1, 1, 0, 1, 1 ]
> M2 := eval s;
> assert M eq M2;
```

1.12 Comments and Continuation

`//`

One-line comment: any text following the double slash on the same line will be ignored by MAGMA.

`/* */`

Multi-line comment: any text between `/*` and `*/` is ignored by MAGMA.

`\`

Line continuation character: this symbol and the `<return>` immediately following is ignored by MAGMA. Evaluation will continue on the next line without interruption. This is useful for long input lines.

Example H1E19

```
>  // The following produces an error:
>  x := 12
>  34;
User error: bad syntax
> /* but this is correct
>       and reads two lines: */
> x := 12\
> 34;
> x;
1234
```

1.13 Timing

`Cputime()`

Return the CPU time (as a real number of default precision) used since the beginning of the MAGMA session. Note that for the MSDOS version, this is the real time used since the beginning of the session (necessarily, since process CPU time is not available).

`Cputime(t)`

Return the CPU time (as a real number of default precision) used since time t. Time starts at 0.0 at the beginning of a MAGMA session.

`Realtime()`

Return the absolute real time (as a real number of default precision), which is the number of seconds since 00:00:00 GMT, January 1, 1970. For the MSDOS version, this is the real time used since the beginning of the session.

`Realtime(t)`

Return the real time (as a real number of default precision) elapsed since time t.

`ClockCycles()`

Return the number of clock cycles of the CPU since Magma's startup. Note that this matches the real time (i.e., not process user/system time). If the operation is not supported on the current processor, zero is returned.

`time` *statement*;

Execute the statement and print the time taken when the statement is completed.

`vtime` *flag*: *statement*;

`vtime` *flag*, *n*: *statement*:

If the verbose flag *flag* (see the function `SetVerbose`) has a level greater than or equal to n, execute the statement and print the time taken when the statement is completed. If the flag has level 0 (i.e., is not turned on), still execute the statement, but do not print the timing. In the first form of this statement, where a specific level is not given, n is taken to be 1. This statement is useful in MAGMA code found in packages where one wants to print the timing of some sub-algorithm if and only if an appropriate verbose flag is turned on.

Example H1E20

The `time` command can be used to time a single statement.

```
> n := 2^109-1;
> time Factorization(n);
[<745988807, 1>, <870035986098720987332873, 1>]
Time: 0.149
```

Alternatively, we can extract the current time t and use `Cputime`. This method can be used to time the execution of several statements.

```
> m := 2^111-1;
> n := 2^113-1;
> t := Cputime();
> Factorization(m);
[<7, 1>, <223, 1>, <321679, 1>, <26295457, 1>, <319020217, 1>, <616318177, 1>]
> Factorization(n);
[<3391, 1>, <23279, 1>, <65993, 1>, <1868569, 1>, <1066818132868207, 1>]
> Cputime(t);
0.121
```

We illustrate a simple use of `vtime` with `vprint` within a function.

```
> function MyFunc(G)
>    vprint User1: "Computing order...";
>    vtime  User1: o := #G;
```

```
>    return o;
> end function;
> SetVerbose("User1", 0);
> MyFunc(Sym(4));
24
> SetVerbose("User1", 1);
> MyFunc(Sym(4));
Computing order...
Time: 0.000
24
```

1.14 Types, Category Names, and Structures

The following functions deal with *types* or *category names* and general structures. MAGMA has two levels of granularity when referring to types. In most cases, the coarser grained types (of type `Cat`) are used. Examples of these kinds of types are "polynomial rings" (`RngUPol`) and "finite fields" (`FldFin`). However, sometimes more specific typing information is sometimes useful. For instance, the algorithm used to factorize polynomials differs significantly, depending on the coefficient ring. Hence, we might wish to implement a specialized factorization algorithm polynomials over some particular ring type. Due to this need, MAGMA also supports *extended types*.

An extended type (of type `ECat`) can be thought of as a type taking a parameter. Using extended types, we can talk about "polynomial rings over the integers" (`RngUPol[RngInt]`), or "maps from the integers to the rationals" (`Map[RngInt, FldRat]`). Extended types can interact with normal types in all ways, and thus generally only need to be used when the extra level of information is required.

`Type(x)`

`Category(x)`

Given any object x, return the type (or category name) of x.

`ExtendedType(x)`

`ExtendedCategory(x)`

Given any object x, return the extended type (or category name) of x.

`ISA(T, U)`

Given types (or extended types) T and U, return whether T ISA U, i.e., whether objects of type T inherit properties of type U. For example, `ISA(RngInt, Rng)` is true, because the ring of integers $\mathbf{Z}$ is a ring.

`MakeType(S)`

Given a string S specifying a type return the actual type corresponding to S. This is useful when some intrinsic name hides the symbol which normally refers to the actual type.

`ElementType(S)`

Given any structure S, return the type of the elements of S. For example, the element type of the ring of integers **Z** is `RngIntElt` since that is the type of the integers which lie in **Z**.

`CoveringStructure(S, T)`

Given structures S and T, return a covering structure C for S and T, so that S and T both embed into C. An error results if no such covering structure exists.

`ExistsCoveringStructure(S, T)`

Given structures S and T, return whether a covering structure C for S and T exists, and if so, return such a C, so that S and T both embed into C.

Example H1E21

We demonstrate the type and structure functions.

```
> Type(3);
RngIntElt
> t := MakeType("RngIntElt");
> t;
RngIntElt
> Type(3) eq t;
true
> Z := IntegerRing();
> Type(Z);
RngInt
> ElementType(Z);
RngIntElt
> ISA(RngIntElt, RngElt);
true
> ISA(RngIntElt, GrpElt);
false
> ISA(FldRat, Fld);
true
```

The following give examples of when covering structures exist or do not exist.

```
> Q := RationalField();
> CoveringStructure(Z, Q);
Rational Field
> ExistsCoveringStructure(Z, DihedralGroup(3));
false
```

```
> ExistsCoveringStructure(Z, CyclotomicField(5));
true Cyclotomic Field of order 5 and degree 4
> ExistsCoveringStructure(CyclotomicField(3), CyclotomicField(5));
true Cyclotomic Field of order 15 and degree 8
> ExistsCoveringStructure(GF(2), GF(3));
false
> ExistsCoveringStructure(GF(2^6), GF(2, 15));
true Finite field of size 2^30
```

Our last example demonstrates the use of extended types:

```
> R<x> := PolynomialRing(Integers());
> ExtendedType(R);
RngUPol[RngInt]
> ISA(RngUPol[RngInt], RngUPol);
true
> f := x + 1;
> ExtendedType(f);
RngUPolElt[RngInt]
> ISA(RngUPolElt[RngInt], RngUPolElt);
true
```

1.15 Random Object Generation

Pseudo-random quantities are used in several MAGMA algorithms, and may also be generated explicitly by some intrinsics. Throughout the Handbook, the word 'random' is used for 'pseudo-random'.

Since V2.7 (June 2000), MAGMA contains an implementation of the *Monster* random number generator of G. Marsaglia [Mar00]. The period of this generator is $2^{29430} - 2^{27382}$ (approximately 10^{8859}), and passes all of the stringent tests in Marsaglia's *Diehard* test suite [Mar95]. Since V2.13 (July 2006), this generator is combined with the MD5 hash function to produce a higher-quality result.

Because the generator uses an internal array of machine integers, one 'seed' variable does not express the whole state, so the method for setting or getting the generator state is by way of a pair of values: (1) the seed for initializing the array, and (2) the number of steps performed since the initialization.

`SetSeed(s, c)`

`SetSeed(s)`

(Procedure.) Reset the random number generator to have initial seed s ($0 \leq s < 2^{32}$), and advance to step c ($0 \leq c < 2^{64}$). If c is not given, it is taken to be 0. Passing `-S`n to MAGMA at startup is equivalent to typing `SetSeed(`n`);` after startup.

`GetSeed()`

Return the initial seed s used to initialize the random-number generator and also the current step c. This is the complement to the `SetSeed` function.

`Random(S)`

Given a finite set or structure S, return a random element of S.

`Random(a, b)`

Return a random integer lying in the interval $[a, b]$, where $a \leq b$.

`Random(b)`

Return a random integer lying in the interval $[0, b]$, where b is a non-negative integer. Because of the good properties of the underlying Monster generator, calling `Random(1)` is a good safe way to produce a sequence of random bits.

Example H1E22

We demonstrate how one can return to a previous random state by the use of `GetSeed` and `SetSeed`. We begin with initial seed 1 at step 0 and create a multi-set of 100,000 random integers in the range [1..4].

```
> SetSeed(1);
> GetSeed();
1 0
> time S := {* Random(1, 4): i in [1..100000] *};
Time: 0.490
> S;
{* 1^^24911, 2^^24893, 3^^25139, 4^^25057 *}
```

We note the current state by `GetSeed`, and then print 10 random integers in the range [1..100].

```
> GetSeed();
1 100000
> [Random(1, 100): i in [1 .. 10]];
[ 85, 41, 43, 69, 66, 61, 63, 31, 84, 11 ]
> GetSeed();
1 100014
```

We now restart with a different initial seed 23 (again at step 0), and do the same as before, noting the different random integers produced.

```
> SetSeed(23);
> GetSeed();
23 0
> time S := {* Random(1, 4): i in [1..100000] *};
Time: 0.500
> S;
{* 1^^24962, 2^^24923, 3^^24948, 4^^25167 *}
> GetSeed();
```

```
23 100000
> [Random(1, 100): i in [1 .. 10]];
[ 3, 93, 11, 62, 6, 73, 46, 52, 100, 30 ]
> GetSeed();
23 100013
```

Finally, we restore the random generator state to what it was after the creation of the multi-set for the first seed. We then print the 10 random integers in the range [1..100], and note that they are the same as before.

```
> SetSeed(1, 100000);
> [Random(1, 100): i in [1 .. 10]];
[ 85, 41, 43, 69, 66, 61, 63, 31, 84, 11 ]
> GetSeed();
1 100014
```

1.16 Miscellaneous

`IsIntrinsic(S)`

Given a string S, return `true` if and only an intrinsic with the name S exists in the current version of MAGMA. If the result is `true`, return also the actual intrinsic.

Example H1E23

We demonstrate the function `IsIntrinsic`.

```
> IsIntrinsic("ABCD");
false
> l, a := IsIntrinsic("Abs");
> l;
true
> a(-3);
3
```

1.17 Bibliography

[**Mar95**] G. Marsaglia. DIEHARD: a battery of tests of randomness. URL:http://stat.fsu.edu/pub/diehard/, 1995.

[**Mar00**] G. Marsaglia. The Monster, a random number generator with period 10^{2857} times as long as the previously touted longest-period one. Preprint, 2000.

2 FUNCTIONS, PROCEDURES AND PACKAGES

Chapter 2
FUNCTIONS, PROCEDURES AND PACKAGES

2.1 Introduction

Functions are one of the most fundamental elements of the MAGMA language. The first section describes the various ways in which a standard function may be defined while the second section describes the definition of a procedure (i.e. a function which doesn't return a value). The second half of the chapter is concerned with user-defined *intrinsic* functions and procedures.

2.2 Functions and Procedures

There are two slightly different syntactic forms provided for the definition of a user function (as opposed to an intrinsic function). For the case of a function whose definition can be expressed as a single expression, an abbreviated form is provided. The syntax for the definition of user procedures is similar. Names for functions and procedures are ordinary identifiers and so obey the rules as given in Chapter 1 for other variables.

2.2.1 Functions

```
f := function(x1, ..., xn: parameters)
   statements
end function;
```

```
function f(x1, ..., xn: parameters)
   statements
end function;
```

This creates a function taking $n \geq 0$ arguments, and assigns it to f. The statements may comprise any number of valid MAGMA statements, but at least one of them must be of the form `return` *expression*`;`. The value of that expression (possibly dependent on the values of the arguments $x_1, \ldots, x_n$) will be the return value for the function; failure to return a value will lead to a run-time error when the function is invoked. (In fact, a return statement is also required for every additional 'branch' of the function that has been created using an `if ... then ... else ...` construction.)

The function may return multiple values. Usually one uses the form `return` *expression*`, ...,` *expression*`;`. If one wishes to make the last return value(s) undefined (so that the number of return values for the function is the same in all 'branches' of

the function) the underscore symbol (_) may be used. (The undefined symbol may only be used for final values of the list.) This construct allows behaviour similar to the intrinsic function `IsSquare`, say, which returns `true` and the square root of its argument if that exists, and `false` and the undefined value otherwise. See also the example below.

If there are parameters given, they must consist of a comma-separated list of clauses each of the form `identifier := value`. The identifier gives the name of the parameter, which can then be treated as a normal value argument within the statements. The value gives a default value for the parameter, and may depend on any of the arguments or preceding parameters; if, when the function is called, the parameter is not assigned a value, this default value will be assigned to the parameter. Thus parameters are always initialized. If no parameters are desired, the colon following the last argument, together with *parameters*, may be omitted.

The only difference between the two forms of function declaration lies in recursion. Functions may invoke themselves recursively since their name is part of the syntax; if the first of the above declarations is used, the identifier f cannot be used inside the definition of f (and `$$` will have to be used to refer to f itself instead), while the second form makes it possible to refer to f within its definition.

An invocation of the user function f takes the form `f(`$\mathtt{m}_1$`, ..., `$\mathtt{m}_n$`)`, where $m_1, \ldots, m_n$ are the actual arguments.

```
f := function(x1, ..., xn, ...: parameters)
   statements
end function;
```

```
function f(x1, ..., xn, ...: parameters)
   statements
end function;
```

This creates a *variadic* function, which can take n or more arguments. The semantics are identical to the standard function definition described above, with the exception of function invocation. An invocation of a variadic function f takes the form `f(`$\mathtt{y}_1$`, ..., `$\mathtt{y}_m$`)`, where $y_1, \ldots, y_m$ are the arguments to the function, and $m \geq n$. These arguments get bound to the parameters as follows: for $i < n$, the argument y_i is bound to the parameter x_i. For $i \geq n$, the arguments y_i are bound to the last parameter x_n as a list $[*y_n, \ldots, y_m*]$.

```
f := func< x1, ..., xn: parameters | expression>;
```

This is a short form of the function constructor designed for the situation in which the value of the function can be defined by a single expression. A function f is created which returns the value of the expression (possibly involving the function arguments $x_1, \ldots, x_n$). Optional parameters are permitted as in the standard function constructor.

```
f := func< x1, ..., xn, ...: parameters | expression>;
```

This is a short form of the function constructor for *variadic functions*, otherwise identical to the short form describe above.

Example H2E1

This example illustrates recursive functions.

```
> fibonacci := function(n)
>    if n le 2 then
>       return 1;
>    else
>       return $$(n-1) + $$(n-2);
>    end if;
> end function;
>
> fibonacci(10)+fibonacci(12);
199
```

```
> function Lucas(n)
>    if n eq 1 then
>       return 1;
>    elif n eq 2 then
>       return 3;
>    else
>       return Lucas(n-1)+Lucas(n-2);
>    end if;
> end function;
>
> Lucas(11);
199
```

```
> fibo := func< n | n le 2 select 1 else $$(n-1) + $$(n-2) >;
> fibo(10)+fibo(12);
199
```

Example H2E2

This example illustrates the use of parameters.

```
> f := function(x, y: Proof := true, Al := "Simple")
>    return <x, y, Proof, Al>;
> end function;
>
> f(1, 2);
<1, 2, true, Simple>
> f(1, 2: Proof := false);
<1, 2, false, Simple>
> f(1, 2: Al := "abc", Proof := false);
<1, 2, false, abc>
```

Example H2E3

This example illustrates the returning of undefined values.

```
> f := function(x)
>    if IsOdd(x) then
>        return true, x;
>    else
>        return false, _;
>    end if;
> end function;
>
> f(1);
true 1
> f(2);
false
> a, b := f(1);
> a;
true
> b;
1
> a, b := f(2);
> a;
false
>  // The following produces an error:
>  b;
>> b;
   ^
User error: Identifier 'b' has not been assigned
```

Example H2E4

This example illustrates the use of variadic functions.

```
> f := function(x, y, ...)
>     print "x: ", x;
>     print "y: ", y;
>     return [x + z : z in y];
> end function;
>
> f(1, 2);
x:  1
y:  [* 2*]
[ 3 ]
> f(1, 2, 3);
x:  1
y:  [* 2, 3*]
[ 3, 4 ]
> f(1, 2, 3, 4);
```

```
x:  1
y:  [* 2, 3, 4*]
[ 3, 4, 5 ]
```

2.2.2 Procedures

```
p := procedure(x1, ..., xn:  parameters)
   statements
end procedure;
```

```
procedure p(x1, ..., xn:  parameters)
   statements
end procedure;
```

The procedure, taking $n \geq 0$ arguments and defined by the statements is created and assigned to p. Each of the arguments may be either a variable (y_i) or a referenced variable ($\sim y_i$). Inside the procedure only referenced variables (and local variables) may be (re-)assigned to. The procedure p is invoked by typing `p(x1, ..., xn)`, where the same succession of variables and referenced variables is used (see the example below). Procedures cannot return values.

If there are parameters given, they must consist of a comma-separated list of clauses each of the form `identifier := value`. The identifier gives the name of the parameter, which can then be treated as a normal value argument within the statements. The value gives a default value for the parameter, and may depend on any of the arguments or preceding parameters; if, when the function is called, the parameter is not assigned a value, this default value will be assigned to the parameter. Thus parameters are always initialized. If no parameters are desired, the colon following the last argument, together with *parameters*, may be omitted.

As in the case of `function`, the only difference between the two declarations lies in the fact that the second version allows recursive calls to the procedure within itself using the identifier (p in this case).

```
p := procedure(x1, ..., xn, ...:  parameters)
   statements
end procedure;
```

```
procedure p(x1, ..., xn, ...:  parameters)
   statements
end procedure;
```

Creates and assigns a new *variadic* procedure to p. The use of a variadic procedure is identical to that of a variadic function, described previously.

`p := proc< x1, ..., xn: parameters | expression>;`

This is a short form of the procedure constructor designed for the situation in which the action of the procedure may be accomplished by a single statement. A procedure p is defined which calls the procedure given by the expression. This expression must be a simple procedure call (possibly involving the procedure arguments $x_1, \ldots, x_n$). Optional parameters are permitted as in the main procedure constructor.

`p := proc< x1, ..., xn, ...: parameters | expression>;`

This is a short form of the procedure constructor for variadic procedures.

Example H2E5

By way of simple example, the following (rather silly) procedure assigns a Boolean to the variable `holds`, according to whether or not the first three arguments x, y, z satisfy $x^2 + y^2 = z^2$. Note that the fourth argument is referenced, and hence can be assigned to; the first three arguments cannot be changed inside the procedure.

```
> procedure CheckPythagoras(x, y, z, ~h)
>     if x^2+y^2 eq z^2 then
>         h := true;
>     else
>         h := false;
>     end if;
> end procedure;
```

We use this to find some Pythagorean triples (in a particularly inefficient way):

```
> for x, y, z in { 1..15 } do
>     CheckPythagoras(x, y, z, ~h);
>     if h then
>       "Yes, Pythagorean triple!", x, y, z;
>     end if;
> end for;
Yes, Pythagorean triple! 3 4 5
Yes, Pythagorean triple! 4 3 5
Yes, Pythagorean triple! 5 12 13
Yes, Pythagorean triple! 6 8 10
Yes, Pythagorean triple! 8 6 10
Yes, Pythagorean triple! 9 12 15
Yes, Pythagorean triple! 12 5 13
Yes, Pythagorean triple! 12 9 15
```

2.2.3 The forward Declaration

```
forward f;
```

The forward declaration of a function or procedure f; although the assignment of a value to f is deferred, f may be called from within another function or procedure already.

The `forward` statement must occur on the 'main' level, that is, outside other functions or procedures. (See also Chapter 5.)

Example H2E6

We give an example of mutual recursion using the `forward` declaration. In this example we define a primality testing function which uses the factorization of $n-1$, where n is the number to be tested. To obtain the complete factorization we need to test whether or not factors found are prime. Thus the prime divisor function and the primality tester call each other.
First we define a simple function that proves primality of n by finding an integer of multiplicative order $n-1$ modulo n.

```
> function strongTest(primdiv, n)
>     return exists{ x : x in [2..n-1] | \
>       Modexp(x, n-1, n) eq 1 and
>       forall{ p : p in primdiv | Modexp(x, (n-1) div p, n) ne 1 }
>     };
> end function;
```

Next we define a rather crude `isPrime` function: for odd $n > 3$ it first checks for a few (3) random values of a that $a^{n-1} \equiv 1 \bmod n$, and if so, it applies the above primality prover. For that we need the not yet defined function for finding the prime divisors of an integer.

```
> forward primeDivisors;
> function isPrime(n)
>    if n in { 2, 3 } or
>       IsOdd(n) and
>       forall{ a : a in { Random(2, n-2): i in [1..3] } |
>          Modexp(a, n-1, n) eq 1 } and
>          strongTest( primeDivisors(n-1), n )
>    then
>       return true;
>    else
>       return false;
>    end if;
> end function;
```

Finally, we define a function that finds the prime divisors. Note that it calls the `isPrime` function. Note also that this function is recursive, and that it calls a function upon its definition, in the form `func< ..> ( .. )`.

```
> primeDivisors := function(n)
>    if isPrime(n) then
>       return { n };
```

```
>    else
>       return func< d | primeDivisors(d) join primeDivisors(n div d) >
>          ( rep{ d : d in [2..Isqrt(n)] | n mod d eq 0 });
>    end if;
> end function;
> isPrime(1087);
true;
```

2.3 Packages

2.3.1 Introduction

For brevity, in this section we shall use the term *function* to include both functions and procedures.

The term *intrinsic function* or *intrinsic* refers to a function whose signature is stored in the system table of signatures. In terms of their origin, there are two kinds of intrinsics, *system intrinsics* (or *standard functions*) and *user intrinsics*, but they are indistinguishable in their use. A *system intrinsic* is an intrinsic that is part of the definition of the MAGMA system, whereas a user intrinsic is an informal addition to MAGMA, created by a user of the system. While most of the standard functions in MAGMA are implemented in C, a growing number are implemented in the MAGMA language. User intrinsics are defined in the MAGMA language using a *package* mechanism (the same syntax, in fact, as that used by developers to write standard functions in the MAGMA language).

This section explains the construction of user intrinsics by means of packages. From now on, *intrinsic* will be used as an abbreviation for *user intrinsic*.

It is useful to summarize the properties possessed by an intrinsic function that are not possessed by an ordinary user-defined function. Firstly, the signature of every intrinsic function is stored in the system's table of signatures. In particular, such functions will appear when signatures are listed and printing the function's name will produce a summary of the behaviour of the function. Secondly, intrinsic functions are compiled into the MAGMA internal pseudo-code. Thus, once an intrinsic function has been debugged, it does not have to be compiled every time it is needed. If the definition of the function involves a large body of code, this can save a significant amount of time when the function definition has to be loaded.

An intrinsic function is defined in a special type of file known as a *package*. In general terms a package is a MAGMA source file that defines constants, one or more intrinsic functions, and optionally, some ordinary functions. The definition of an intrinsic function may involve MAGMA standard functions, functions imported from other packages and functions whose definition is part of the package. It should be noted that constants and functions (other than intrinsic functions) defined in a package will not be visible outside the package, unless they are explicitly imported.

The syntax for the definition of an intrinsic function is similar to that of an ordinary function except that the function header must define the function's signature together with

text summarizing the semantics of the function. As noted above, an intrinsic function definition must reside in a package file. It is necessary for MAGMA to know the location of all necessary package files. A package may be attached or detached through use of the `Attach` or `Detach` procedures. More generally, a family of packages residing in a directory tree may be specified through provision of a **spec** file which specifies the locations of a collection of packages relative to the position of the spec file. Automatic attaching of the packages in a spec file may be set by means of an environment variable (`MAGMA_SYSTEM_SPEC` for the MAGMA system packages and `MAGMA_USER_SPEC` for a users personal packages).

So that the user does not have to worry about explicitly compiling packages, MAGMA has an auto-compile facility that will automatically recompile and reload any package that has been modified since the last compilation. It does this by comparing the time stamp on the source file (as specified in an `Attach` procedure call or spec file) with the time stamp on the compiled code. To avoid the possible inefficiency caused by MAGMA checking whether the file is up to date every time an intrinsic function is referenced, the user can indicate that the package is stable by including the `freeze;` directive at the top of the package containing the function definition.

A constant value or function defined in the body of a package may be accessed in a context outside of its package through use of the `import` statement. The arguments for an intrinsic function may be checked through use of the `require` statement and its variants. These statements have the effect of generating an error message at the level of the caller rather than in the called intrinsic function.

See also the section on user-defined attributes for the `declare attributes` directive to declare user-defined attributes used by the package and related packages.

2.3.2 Intrinsics

Besides the definition of *constants* at the top, a package file just consists of *intrinsics*. There is only one way a intrinsic can be referred to (whether from within or without the package). When a package is *attached*, its intrinsics are incorporated into MAGMA. Thus intrinsics are 'global' — they affect the global MAGMA state and there is only one set of MAGMA intrinsics at any time. There are no 'local' intrinsics.

A package may contain undefined references to identifiers. These are presumed to be intrinsics from other packages which will be attached subsequent to the loading of this package.

```
intrinsic name(arg-list [, ...])  [ -> ret-list ]
{comment-text}
    statements
end intrinsic;
```

The syntax of a intrinsic declaration is as above, where *name* is the name of the intrinsic (any identifier; use single quotes for non-alphanumeric names like '+'); *arg-list* is the argument list (optionally including parameters preceded by a colon); optionally there is an arrow and return type list *ret-list*; the comment text is any text within the braces (use \} to get a right brace within the text, and use " to repeat the comment from the immediately preceding intrinsic); and *statements* is a list of

statements making up the body. *arg-list* is a list of comma-separated arguments of the form

name::*type*
∼*name*::*type*
∼*name*

where *name* is the name of the argument (any identifier), and *type* designates the type, which can be either a simple category name, an extended type, or one of the following:

`.`	Any type
`[ ]`	Sequence type
`{ }`	Set type
`{[ ]}`	Set or Sequence type
`{@ @}`	Iset type
`{* *}`	Multiset type
`< >`	Tuple type

or a *composite type*:

`[`*type*`]`	Sequences over *type*
{*type*}	Sets over *type*
{`[`*type*`]`}	Sets or sequences over *type*
{@*type*@}	Indexed sets over *type*
{**type***}	Multisets over *type*

where *type* is either a simple or extended type. The reference form *type* ∼*name* requires that the input argument must be initialized to an object of that type. The reference form ∼*name* is a plain reference argument — it need not be initialized. Parameters may also be specified—these are just as in functions and procedures (preceded by a colon). If *arg-list* is followed by "..." then the intrinsic is `variadic`, with semantics similar to that of a variadic function, described previously.

ret-list is a list of comma-separated simple types. If there is an arrow and the return list, the intrinsic is assumed to be functional; otherwise it is assumed to be procedural.

The body of *statements* should return the correct number and types of arguments if the intrinsic is functional, while the body should return nothing if the intrinsic is procedural.

Example H2E7

A functional intrinsic for greatest common divisors taking two integers and returning another:

```
intrinsic myGCD(x::RngIntElt, y::RngIntElt) -> RngIntElt
{ Return the GCD of x and y}
   return ...;
```

```
end intrinsic;
```

A procedural intrinsic for Append taking a reference to a sequence Q and any object then modifying Q:

```
intrinsic Append(~ Q::SeqEnum, . x)
{ Append x to Q }
    ...;
end intrinsic;
```

A functional intrinsic taking a sequence of sets as arguments 2 and 3:

```
intrinsic IsConjugate(G::GrpPerm, R::[ { } ], S::[ { } ]) -> BoolElt
{ True iff partitions R and S of the support of G are conjugate in G }
    return ...;
end intrinsic;
```

2.3.3 Resolving Calls to Intrinsics

It is often the case that many intrinsics share the same name. For instance, the intrinsic `Factorization` has many implementations for various object types. We will call such intrinsics *overloaded intrinsics*, or refer to each of the participating intrinsics as an *overload*. When the user calls such an overloaded intrinsic, MAGMA must choose the "best possible" overload.

MAGMA's overload resolution process is quite simple. Suppose the user is calling an intrinsic of arity r, with a list of parameters $\langle p_1, \ldots, p_r \rangle$. Let the tuple of the types of these parameters be $\langle t_1, \ldots, t_r \rangle$, and let S be the set of all relevant overloads (that is, overloads with the appropriate name and of arity r). We will represent overloads as r-tuples of types.

To pick the "best possible" overload, for each parameter $p \in \{p_1, \ldots, p_r\}$, MAGMA finds the set $S_i \subseteq S$ of participating intrinsics which are the best matches for that parameter. More specifically, an intrinsic $s = \langle u_1, \ldots, u_r \rangle$ is included in S_i if and only if t_i is a u_i, and no participating intrinsic $s' = \langle v_1, \ldots, v_r \rangle$ exists such that t_i is a v_i and v_i is a u_i. Once the sets S_i are computed, MAGMA finds their intersection. If this intersection is empty, then there is no match. If this intersection has cardinality greater than one, then the match is ambiguous. Otherwise, MAGMA calls the overload thus obtained.

An example at this point will make the above process clearer:

Example H2E8

We demonstrate MAGMA's lookup mechanism with the following example. Suppose we have the following overloaded intrinsics:

```
intrinsic overloaded(x::RngUPolElt, y::RngUPolElt) -> RngIntElt
{ Overload 1 }
   return 1;
end intrinsic;

intrinsic overloaded(x::RngUPolElt[RngInt], y::RngUPolElt) -> RngIntElt
```

```
{ Overload 2 }
   return 2;
end intrinsic;

intrinsic overloaded(x::RngUPolElt, y::RngUPolElt[RngInt]) -> RngIntElt
{ Overload 3 }
   return 3;
end intrinsic;

intrinsic overloaded(x::RngUPolElt[RngInt], y::RngUPolElt[RngInt]) -> RngIntElt
{ Overload 4 }
   return 4;
end intrinsic;
```

The following MAGMA session illustrates how the lookup mechanism operates for the intrinsic `overloaded`:

```
> R1<x> := PolynomialRing(Integers());
> R2<y> := PolynomialRing(Rationals());
> f1 := x + 1;
> f2 := y + 1;
> overloaded(f2, f2);
1
> overloaded(f1, f2);
2
> overloaded(f2, f1);
3
> overloaded(f1, f1);
4
```

2.3.4 Attaching and Detaching Package Files

The procedures `Attach` and `Detach` are provided to attach or detach package files. Once a file is attached, all intrinsics within it are included in MAGMA. If the file is modified, it is automatically recompiled just after the user hits return and just before the next statement is executed. So there is no need to re-attach the file (or 're-load' it). If the recompilation of a package file fails (syntax errors, etc.), all of the intrinsics of the package file are removed from the MAGMA session and none of the intrinsics of the package file are included again until the package file is successfully recompiled. When errors occur during compilation of a package, the appropriate messages are printed with the string '[PC]' at the beginning of the line, indicating that the errors are detected by the MAGMA package compiler.

If a package file contains the single directive `freeze;` at the top then the package file becomes `frozen` — it will not be automatically recompiled after each statement is entered into MAGMA. A frozen package is recompiled if need be, however, when it is attached (thus allowing fixes to be updated) — the main point of freezing a package which is 'stable' is to stop MAGMA looking at it between every statement entered into MAGMA interactively.

When a package file is complete and tested, it is usually installed in a spec file so it is automatically attached when the spec file is attached. Thus `Attach` and `Detach` are generally only used when one is developing a single package file containing new intrinsics.

`Attach(F)`

Procedure to attach the package file F.

`Detach(F)`

Procedure to detach the package file F.

`freeze;`

Freeze the package file in which this appears at the top.

2.3.5 Related Files

There are two files related to any package source file `file.m`:

`file.sig`	sig file containing signature information;
`file.lck`	lock file.

The lock file exists while a package file is being compiled. If someone else tries to compile the file, it will just sit there till the lock file disappears. In various circumstances (system down, MAGMA crash) `.lck` files may be left around; this will mean that the next time MAGMA attempts to compile the associated source file it will just sit there indefinitely waiting for the `.lck` file to disappear. In this case the user should search for `.lck` files that should be removed.

2.3.6 Importing Constants

`import "`*filename*`": `*ident_list*`;`

This is the general form of the import statement, where "*filename*" is a string and *ident_list* is a list of identifiers.

The import statement is a normal statement and can in fact be used anywhere in MAGMA, but it is recommended that it only be used to import common constants and functions/procedures shared between a collection of package files. It has the following semantics: for each identifier I in the list *ident_list*, that identifier is declared just like a normal identifier within MAGMA. Within the package file referenced by *filename*, there should be an assignment of the same identifier I to some object O. When the identifier I is then used as an expression after the import statement, the value yielded is the object O.

The file that is named in the import statement must already have been attached by the time the identifiers are needed. The best way to achieve this in practice is to place this file in the spec file, along with the package files, so that all the files can be attached together.

Thus the only way objects (whether they be normal objects, procedures or functions) assigned within packages can be referenced from outside the package is by an explicit import with the 'import' statement.

Example H2E9

Suppose we have a spec file that lists several package files. Included in the spec file is the file `defs.m` containing:

```
MY_LIMIT := 10000;
 function fred(x)
 return 1/x;
end function;
```

Then other package files (in the same directory) listed in the spec file which wish to use these definitions would have the line

```
import "defs.m": MY_LIMIT, fred;
```

at the top. These could then be used inside any intrinsics of such package files. (If the package files are not in the same directory, the pathname of `defs.m` will have to be given appropriately in the import statement.)

2.3.7 Argument Checking

Using 'require' etc. one can do argument checking easily within intrinsics. If a necessary condition on the argument fails to hold, then the relevant error message is printed and the error pointer refers to the caller of the intrinsic. This feature allows user-defined intrinsics to treat errors in actual arguments in exactly the same way as they are treated by the MAGMA standard functions.

`require` *condition*: *print_args*;

The expression *condition* may be any yielding a Boolean value. If the value is false, then *print_args* is printed and execution aborts with the error pointer pointing to the caller. The print arguments *print_args* can consist of any expressions (depending on arguments or variables already defined in the intrinsic).

`requirerange v, L, U;`

The argument variable v must be the name of one of the argument variables (including parameters) and must be of integer type. L and U may be any expressions each yielding an integer value. If v is not in the range $[L, \ldots, U]$, then an appropriate error message is printed and execution aborts with the error pointer pointing to the caller.

`requirege v, L;`

The argument variable v must be the name of one of the argument variables (including parameters) and must be of integer type. L must yield an integer value. If v is not greater than or equal to L, then an appropriate error message is printed and execution aborts with the error pointer pointing to the caller.

Example H2E10

A trivial version of `Binomial(n, k)` which checks that $n \geq 0$ and $0 \leq k \leq n$.

```
intrinsic Binomial(n::RngIntElt, k::RngIntElt) -> RngIntElt
{ Return n choose k }
    requirege n, 0;
    requirerange k, 0, n;
    return Factorial(n) div Factorial(n - k) div Factorial(k);
end intrinsic;
```

A simple function to find a random p-element of a group G.

```
intrinsic pElement(G::Grp, p::RngIntElt) -> GrpElt
{ Return p-element of group G }
   require IsPrime(p): "Argument 2 is not prime";
   x := random{x: x in G | Order(x) mod p eq 0};
   return x^(Order(x) div p);
end intrinsic;
```

2.3.8 Package Specification Files

A *spec file* (short for 'specification file') lists a complete tree of MAGMA package files. This makes it easy to collect many package files together and attach them simultaneously.

The specification file consists of a list of tokens which are just space-separated words. The tokens describe a list of package files and directories containing other packages. The list is described as follows. The files that are to be attached in the directory indicated by S are listed enclosed in { and } characters. A directory may be listed there as well, if it is followed by a list of files from that directory (enclosed in braces again); arbitrary nesting is allowed this way. A filename of the form `+`*spec* is interpreted as another specification file whose contents will be recursively attached when `AttachSpec` (below) is called. The files are taken relative to the directory that contains the specification file. See also the example below.

`AttachSpec(S)`

If S is a string indicating the name of a spec file, this command attaches all the files listed in S. The format of the spec file is given above.

`DetachSpec(S)`

If S is a string indicating the name of a spec file, this command detaches all the files listed in S. The format of the spec file is given above.

Example H2E11

Suppose we have a spec file /home/user/spec consisting of the following lines:

```
{
   Group
   {
      chiefseries.m
      socle.m
   }
   Ring
   {
      funcs.m
      Field
      {
            galois.m
      }
   }
}
```

Then there should be the files

```
/home/user/spec/Group/chiefseries.m
/home/user/spec/Group/socle.m
/home/user/spec/Ring/funcs.m
/home/user/spec/Ring/Field/galois.m
```

and if one typed within MAGMA

```
AttachSpec("/home/user/spec");
```

then each of the above files would be attached. If instead of the filename `galois.m` we have `+galspec`, then the file `/home/user/spec/Ring/Field/galspec` would be a specification file itself whose contents would be recursively attached.

2.3.9 User Startup Specification Files

The user may specify a list of spec files to be attached automatically when MAGMA starts up. This is done by setting the environment variable `MAGMA_USER_SPEC` to a colon separated list of spec files.

Example H2E12

One could have

```
setenv MAGMA_USER_SPEC "$HOME/Magma/spec:/home/friend/Magma/spec"
```

in one's `.cshrc` . Then when MAGMA starts up, it will attach all packages listed in the spec files `$HOME/Magma/spec` and `/home/friend/Magma/spec`.

2.4 Attributes

This section is placed beside the section on packages because the use of attributes is most common within packages.

For any structure within MAGMA, it is possible to have *attributes* associated with it. These are simply values stored within the structure and are referred to by named fields in exactly the same manner as MAGMA records.

There are two kinds of structure attributes: predefined system attributes and user-defined attributes. Both kinds are discussed in the following subsections. A description of how attributes are accessed and assigned then follows.

2.4.1 Predefined System Attributes

The valid fields of predefined system attributes are automatically defined at the startup of Magma. These fields now replace the old method of using the procedure `AssertAttribute` and the function `HasAttribute` (which will still work for some time to preserve backwards compatibility). For each name which is a valid first argument for `AssertAttribute` and `HasAttribute`, that name is a valid attribute field for structures of the appropriate category. Thus the backquote method for accessing attributes described in detail below should now be used instead of the old method. For such attributes, the code:

```
> S`Name := x;
```

is completely equivalent to the code:

```
> AssertAttribute(S, "Name", x);
```

(note that the function `AssertAttribute` takes a string for its second argument so the name must be enclosed in double quotes). Similarly, the code:

```
> if assigned S`Name then
>     x := S`Name;
>     // do something with x...
> end if;
```

is completely equivalent to the code:

```
> l, x := HasAttribute(S, "Name");
> if l then
>     // do something with x...
> end if;
```

(note again that the function `HasAttribute` takes a string for its second argument so the name must be enclosed in double quotes).

Note also that if a system attribute is not set, referring to it in an expression (using the backquote operator) will *not* trigger the calculation of it (while the corresponding intrinsic function will if it exists); rather an error will ensue. Use the `assigned` operator to test whether an attribute is actually set.

2.4.2 User-defined Attributes

For any category C, the user can stipulate valid attribute fields for structures of C. After this is done, any structure of category C may have attributes assigned to it and accessed from it.

There are two ways of adding new valid attributes to a category C: by the procedure `AddAttribute` or by the `declare attributes` package declaration. The former should be used outside of packages (e.g. in interactive usage), while the latter must be used within packages to declare attribute fields used by the package and related packages.

`AddAttribute(C, F)`

(Procedure.) Given a category C, and a string F, append the field name F to the list of valid attribute field names for structures belonging to category C. This procedure should not be used within packages but during interactive use. Previous fields for C are still valid – this just adds another valid one.

`declare attributes` C`:` $F_1, \ldots, F_n$`;`

Given a category C, and a comma-separated list of identifiers $F_1, \ldots, F_n$ append the field names specified by the identifiers to the list of valid attribute field names for structures belonging to category C. This declaration directive must be used within (and only within) packages to declare attribute fields used by the package and packages related to it which use the same fields. It is *not* a statement but a directive which is stored with the other information of the package when it is compiled and subsequently attached – *not* when any code is actually executed.

2.4.3 Accessing Attributes

Attributes of structures are accessed in the same way that records are: using the backquote (‘) operator. The double backquote operator (‘‘) can also be used if the field name is a string.

`S‘`*fieldname*

`S‘‘N`

Given a structure S and a field name, return the current value for the given field in S. If the value is not assigned, an error results. The field name must be valid for the category of S. In the `S‘‘N` form, N is a string giving the field name.

`assigned S‘`*fieldname*

`assigned S‘‘N`

Given a structure S and a field name, return whether the given field in S currently has a value. The field name must be valid for the category of S. In the `S‘‘N` form, N is a string giving the field name.

`S‘`*fieldname* `:=` *expression*`;`

`S‘‘N :=` *expression*`;`

Given a structure S and a field name, assign the given field of S to be the value of the expression (any old value is first discarded). The field name must be valid for the category of S. In the `S‘‘N` form, N is a string giving the field name.

`delete S‘`*fieldname*`;`

`delete S‘‘N;`

Given a structure S and a field name, delete the given field of S. The field then becomes unassigned in S. The field name must be valid for the category of S and the field must be currently assigned in S. This statement is not allowed for predefined system attributes. In the `S‘‘N` form, N is a string giving the field name.

`GetAttributes(C)`

Given a category C, return the valid attribute field names for structures belonging to category C as a sorted sequence of strings.

`ListAttributes(C)`

(Procedure.) Given a category C, list the valid attribute field names for structures belonging to category C.

2.5 User-defined Verbose Flags

Verbose flags may be defined by users within packages.

`declare verbose` F`,` m`;`

Given a verbose flag name F (without quotes), and a literal integer m, create the verbose flag F, with the maximal allowable level for the flag set to m. This directive may only be used within package files.

2.5.1 Examples

In this subsection we give examples which illustrate all of the above features.

Example H2E13

We illustrate how the predefined system attributes may be used. Note that the valid arguments for `AssertAttribute` and `HasAttribute` documented elsewhere now also work as system attributes so see the documentation for these functions for details as to the valid system attribute field names.

```
> // Create group G.
> G := PSL(3, 2);
> // Check whether order known.
> assigned G‘Order;
false
>  // Attempt to access order -- error since not assigned.
>  G‘Order;
```

```
>> G‘Order;
    ^
Runtime error in ‘: Attribute ’Order’ for this structure
is valid but not assigned
> // Force computation of order by intrinsic Order.
> Order(G);
168
> // Check Order field again.
> assigned G‘Order;
true
> G‘Order;
168
> G‘‘"Order"; // String form for field
168
> o := "Order";
> G‘‘o;
168
> // Create code C and set its minimum weight.
> C := QRCode(GF(2), 31);
> C‘MinimumWeight := 7;
> C;
[31, 16, 7] Quadratic Residue code over GF(2)
...
```

Example H2E14

We illustrate how user attributes may be defined and used in an interactive session. This situation would arise rarely – more commonly, attributes would be used within packages.

```
> // Add attribute field MyStuff for matrix groups.
> AddAttribute(GrpMat, "MyStuff");
> // Create group G.
> G := GL(2, 3);
> // Try illegal field.
>  G‘silly;
>> G‘silly;
    ^
Runtime error in ‘: Invalid attribute ’silly’ for this structure
> // Try legal but unassigned field.
>  G‘MyStuff;
>> G‘MyStuff;
    ^
Runtime error in ‘: Attribute ’MyStuff’ for this structure is valid but not
assigned
> // Assign field and notice value.
> G‘MyStuff := [1, 2];
> G‘MyStuff;
```

```
[ 1, 2 ]
```

Example H2E15

We illustrate how user attributes may be used in packages. This is the most common usage of such attributes. We first give some (rather naive) MAGMA code to compute and store a permutation representation of a matrix group. Suppose the following code is stored in the file `permrep.m`.

```
declare attributes GrpMat: PermRep, PermRepMap;
intrinsic PermutationRepresentation(G::GrpMat) -> GrpPerm
{A permutation group representation P of G, with homomorphism f: G -> P};
    // Only compute rep if not already stored.
    if not assigned G`PermRep then
        G`PermRepMap, G`PermRep := CosetAction(G, sub<G|>);
    end if;
    return G`PermRep, G`PermRepMap;
end intrinsic;
```

Note that the information stored will be reused in subsequent calls of the intrinsic. Then the package can be attached within a MAGMA session and the intrinsic `PermutationRepresentation` called like in the following code (assumed to be run in the same directory).

```
>  Attach("permrep.m");
>  G := GL(2, 2);
>  P, f := PermutationRepresentation(G);
>  P;
Permutation group P acting on a set of cardinality 6
    (1, 2)(3, 5)(4, 6)
    (1, 3)(2, 4)(5, 6)
>  f;
Mapping from: GrpMat: G to GrpPerm: P
```

Suppose the following line were also in the package file:

```
declare verbose MyAlgorithm, 3;
```

Then there would be a new verbose flag `MyAlgorithm` for use anywhere within MAGMA, with the maximum 3 for the level.

2.6 User-Defined Types

Since MAGMA V2.19, types may be defined by users within packages. This facility allows the user to declare new type names and create objects with such types and then supply some basic primitives and intrinsic functions for such objects.

The new types are known as *user-defined types*. The way these are typically used is that after declaring such a type T, the user supplies package intrinsics to: (1) create objects of type T and set relevant attributes to define the objects; (2) perform some basic primitives which are common to all objects in MAGMA; (3) perform non-trivial computations on objects of type T.

2.6.1 Declaring User-Defined Types

The following declarations are used to declare user-defined types. They **may only be placed in package files**, i.e., files that are included either by using `Attach` or a spec file (see above). Declarations may appear in any package file and at any place within the file at the top level (not in a function, etc.). In particular, it is not required that the declaration of a type appears before package code which refers to the type (as long as the type is declared before running the code). Examples below will illustrate how the basic declarations are used.

`declare type` T`;`

Declare the given type name T (without quotes) to be a user-defined type.

`declare type` $T : P_1, \ldots, P_n$`;`

Declare the given type name T (without quotes) to be a user-defined type, and also declare T to inherit from the user types $P_1, \ldots, P_n$ (which must be declared separately). As a result, `ISA(`T, P_i`)` will be true for each i and when intrinsic signatures are scanned at a function call, an object of type T will match an argument of a signature with type P_i for any i.

NB: currently one may not inherit from existing MAGMA internal types or virtual types (categories). It is hoped that this restriction will be removed in the future.

`declare type` $T[E]$`;`

Declare the given type names T and E (both without quotes) to be user-defined types. This form also specifies that E is the *element type* corresponding to T; i.e., if an object x has an element of type T for its parent, then x must have type E. This relationship is needed for the construction of sets and sequences which have objects of type T as a universe. The type E may also be declared separately, but this is not necessary.

`declare type` $T[E] : P_1, \ldots, P_n$`;`

This is a combination of the previous kinds two declarations: T and E are declared as user-defined types while E is also declared to be the element type of T, and T is declared to inherit from user-defined types $P_1, \ldots, P_n$.

2.6.2 Creating an Object

`New(T)`

Create an empty object of type T, where T is a user-defined type. Typically, after setting X to the result of this function, the user should set attributes in X to define relevant properties of the object which are characteristic of objects of type T.

2.6.3 Special Intrinsics Provided by the User

Let T be a user-defined type. Besides the declaration of T, the following special intrinsics are mostly required to be defined for type T (the requirements are specified for each kind of intrinsic). These intrinsics allow the internal MAGMA functions to perform some fundamental operations on objects of type T. Note that the special intrinsics need not be in one file or in the same file as the declaration.

```
intrinsic Print(X::T)
{Print X}
    // Code: Print X with no new line, via printf
end intrinsic;

intrinsic Print(X::T, L::MonStgElt)
{Print X at level L}
    // Code: Print X at level L with no new line, via printf
end intrinsic;
```

Exactly one of these intrinsics must be provided by the user for type T. Each is a procedure rather than a function (i.e., nothing is returned), and should contain one or more print statements. The procedure is called automatically by MAGMA whenever the object X of type T is to be printed. A new line should *not* occur at the end of the last (or only) line of printing: one should use `printf` (see examples below).

When the second form of the intrinsic is provided, it allows X to be printed differently depending on the print level L, which is a string equal to one of `"Default", "Minimal", "Maximal", "Magma"`.

```
intrinsic Parent(X::T) -> .
{Parent of X}
    // Code: Return the parent of X
end intrinsic;
```

This intrinsic is only needed when T is an element type, so objects of type T have parents. It should be a user-provided package function, which takes an object X of type T (user-defined), and returns the parent of X, assuming it has one. In such a case, typically the attribute `Parent` will be defined for X and so `X`Parent` should simply be returned.

```
intrinsic 'in'(e::., X::T) -> BoolElt
{Return whether e is in X}
    // Code: Return whether e is in X
end intrinsic;
```

This intrinsic is only needed when objects of type T (user-defined) have elements, and should be a user-provided package function, which takes any object e and an object X of type T (user-defined), and returns whether e is an element of X.

```
intrinsic IsCoercible(X::T, y::.) -> BoolElt, .
{Return whether y is coercible into X and the result if so}
    // Code: do tests on the type of y to see whether coercible
    // On failure, do:
    //    return false, "Illegal coercion"; // Or more particular message
    // Assumed coercible now; set x to result of coercion into X
    return true, x;
end intrinsic;
```

Assuming that objects of type T (user-defined) have elements (and so coercion into such objects makes sense), this must be a user-provided package function, which takes an object X of type T (user-defined) and an object Y of any type. If Y is coercible into X, the function should return `true` and the result of the coercion (whose parent should be X). Otherwise, the function should return `false` and a string giving the reason for failure. If this package intrinsic is provided, then the coercion operation `X!y` will also automatically work for an object X of type T (i.e., the internal coercion code in MAGMA will automatically call this function).

2.6.4 Examples

Some basic examples illustrating the general use of user-defined types are given here. Non-trivial examples can also be found in much of the standard MAGMA package code (one can search for `"declare type"` in the package `.m` files to see several typical uses).

Example H2E16

In this first simple example, we create a user-defined type `MyRat` which is used for a primitive representation of rational numbers. Of course, a serious version would keep the numerators & denominators always reduced, but for simplicity we skip such details. We define the operations `+` and `*` here; one would typically add other operations like `-`, `eq` and `IsZero`, etc.

```
declare type MyRat;
declare attributes MyRat: Numer, Denom;

intrinsic MyRational(n::RngIntElt, d::RngIntElt) -> MyRat
{Create n/d}
```

```
    require d ne 0: "Denominator must be non-zero";
    r := New(MyRat);
    r`Numer := n;
    r`Denom := d;
    return r;
end intrinsic;

intrinsic Print(r::MyRat)
{Print r}
    n := r`Numer;
    d := r`Denom;
    g := GCD(n, d);
    if d lt 0 then g := -g; end if;
    printf "%o/%o", n div g, d div g; // NOTE: no newline!
end intrinsic;

intrinsic '+'(r::MyRat, s::MyRat) -> MyRat
{Return r + s}
    rn := r`Numer;
    rd := r`Denom;
    sn := s`Numer;
    sd := s`Denom;
    return MyRational(rn*sd + sn*rd, rd*sd);
end intrinsic;

intrinsic '*'(r::MyRat, s::MyRat) -> MyRat
{Return r * s}
    rn := r`Numer;
    rd := r`Denom;
    sn := s`Numer;
    sd := s`Denom;
    return MyRational(rn*sn, rd*sd);
end intrinsic;
```

Assuming the above code is placed in a file `MyRat.m`, one could attach it in MAGMA and then do some simple operations, as follows.

```
> Attach("myrat.m");
> r := MyRational(3, -9);
> r;
-1/3
> s := MyRational(4, 7);
> s;
> r+s;
5/21
> r*s;
-4/21
```

Example H2E17

In this example, we define a type `DirProd` for direct products of rings, and a corresponding element type `DirProdElt` for their elements. Objects of type `DirProd` contain a tuple `Rings` with the rings making up the direct product, while objects of type `DirProdElt` contain a tuple `Element` with the elements of the corresponding rings, and also a reference to the parent direct product object.

```
/* Declare types and attributes */

// Note that we declare DirProdElt as element type of DirProd:
declare type DirProd[DirProdElt];
declare attributes DirProd: Rings;
declare attributes DirProdElt: Elements, Parent;

/* Special intrinsics for DirProd */

intrinsic DirectProduct(Rings::Tup) -> DirProd
{Create the direct product of given rings (a tuple)}
    require forall{R: R in Rings | ISA(Type(R), Rng)}:
        "Tuple entries are not all rings";
    D := New(DirProd);
    D`Rings := Rings;
    return D;
end intrinsic;

intrinsic Print(D::DirProd)
{Print D}
    Rings := D`Rings;
    printf "Direct product of %o", Rings; // NOTE: no newline!
end intrinsic;

function CreateElement(D, Elements)
    // Create DirProdElt with parent D and given Elements
    x := New(DirProdElt);
    x`Elements := Elements;
    x`Parent := D;
    return x;
end function;

intrinsic IsCoercible(D::DirProd, x::.) -> BoolElt, .
{Return whether x is coercible into D and the result if so}
    Rings := D`Rings;
    n := #Rings;
    if Type(x) ne Tup then
        return false, "Coercion RHS must be a tuple";
    end if;
    if #x ne n then
        return false, "Wrong length of tuple for coercion";
    end if;
```

```
    Elements := <>;
    for i := 1 to n do
        l, t := IsCoercible(Rings[i], x[i]);
        if not l then
            return false, Sprintf("Tuple entry %o not coercible", i);
        end if;
        Append(~Elements, t);
    end for;
    y := CreateElement(D, Elements);
    return true, y;
end intrinsic;

/* Special intrinsics for DirProdElt */

intrinsic Print(x::DirProdElt)
{Print x}
    printf "%o", x`Elements; // NOTE: no newline!
end intrinsic;

intrinsic Parent(x::DirProdElt) -> DirProd
{Parent of x}
    return x`Parent;
end intrinsic;

intrinsic '+'(x::DirProdElt, y::DirProdElt) -> DirProdElt
{Return x + y}
    D := Parent(x);
    require D cmpeq Parent(y): "Incompatible arguments";
    Ex := x`Elements;
    Ey := y`Elements;
    return CreateElement(D, <Ex[i] + Ey[i]: i in [1 .. #Ex]>);
end intrinsic;

intrinsic '*'(x::DirProdElt, y::DirProdElt) -> DirProdElt
{Return x * y}
    D := Parent(x);
    require D cmpeq Parent(y): "Incompatible arguments";
    Ex := x`Elements;
    Ey := y`Elements;
    return CreateElement(D, <Ex[i] * Ey[i]: i in [1 .. #Ex]>);
end intrinsic;
```

A sample MAGMA session using the above package is as follows. We create elements x, y of a direct product D and do simple operations on x, y. One would of course add other intrinsic functions for basic operations on the elements.

```
> Attach("DirProd.m");
> Z := IntegerRing();
```

```
> Q := RationalField();
> F8<a> := GF(2^3);
> F9<b> := GF(3^2);
> D := DirectProduct(<Z, Q, F8, F9>);
> x := D!<1, 2/3, a, b>;
> y := D!<2, 3/4, a+1, b+1>;
> x;
<1, 2/3, a, b>
> Parent(x);
Direct product of <Integer Ring, Rational Field, Finite field of
size 2^3, Finite field of size 3^2>
> y;
<2, 3/4, a^3, b^2>
> x+y;
<3, 17/12, 1, b^3>
> x*y;
<2, 1/2, a^4, b^3>
> D!x;
<1, 2/3, a, b>
> S := [x, y]; S;
[
    <1, 2/3, a, b>,
    <2, 3/4, a^3, b^2>
]
>
> &+S;
<3, 17/12, 1, b^3>
```

3 INPUT AND OUTPUT

Chapter 3
INPUT AND OUTPUT

3.1 Introduction

This chapter is concerned with the various facilities provided for communication between MAGMA and its environment. The first section describes character strings and their operations. Following this, the various forms of the `print`-statement are presented. Next the file type is introduced and its operations summarized. The chapter concludes with a section listing system calls. These include facilities that allow the user to execute an operating system command from within MAGMA or to run an external process.

3.2 Character Strings

Strings of characters play a central role in input/output so that the operations provided for strings to some extent reflect this. However, if one wishes, a more general set of operations are available if the string is first converted into a sequence. We will give some examples of this below.

MAGMA provides two kinds of strings: normal character strings, and *binary strings*. Character strings are an inappropriate choice for manipulating data that includes non-printable characters. If this is required, a better choice is the binary string type. This type is similar semantically to a sequence of integers, in which each character is represented by its ASCII value between 0 and 255. The difference between a binary string and a sequence of integers is that a binary string is stored internally as an array of bytes, which is a more space-efficient representation.

3.2.1 Representation of Strings

Character strings may consist of all ordinary characters appearing on your keyboard, including the blank (space). Two symbols have a special meaning: the double-quote " and the backslash \. The double-quote is used to delimit a character string, and hence cannot be used inside a string; to be able to use a double-quote in strings the backslash is designed to be an escape character and is used to indicate that the next symbol has to be taken literally; thus, by using \" inside a string one indicates that the symbol " has to be taken literally and is not to be interpreted as the end-of-string delimiter. Thus:

```
> "\"Print this line in quotes\"";
"Print this line in quotes"
```

To obtain a literal backslash, one simply types two backslashes; for characters other than double-quotes and backslash it does not make a difference when a backslash precedes them

inside a string, with the exception of `n`, `r` and `t`. Any occurrence of `\n` or `\r` inside a string is converted into a `<new-line>` while `\t` is converted into a `<tab>`. For example:

```
> "The first line,\nthe second line, and then\ran\tindented line";
The first line,
the second line, and then
an          indented line
```

Note that a backslash followed by a return allows one to conveniently continue the current construction on the next line; so `\<return>` inside a string will be ignored, except that input will continue on a new line on your screen.

Binary strings, on the hand, can consist of any character, whether printable or non-printable. Binary strings cannot be constructed using literals, but must be constructed either from a character string, or during a read operation from a file.

3.2.2 Creation of Strings

`"abc"`

Create a string from a succession of keyboard characters (a, b, c) enclosed in double quotes `" "`.

`BinaryString(s)`

`BString(s)`

Create a binary string from the character string s.

`s cat t`

`s * t`

Concatenate the strings s and t.

`s cat:= t`

`s *:= t`

Modification-concatenation of the string s with t: concatenate s and t and put the result in s.

`&cat s`

`&* s`

Given an enumerated sequence s of strings, return the concatenation of these strings.

`s ^ n`

Form the n-fold concatenation of the string s, for $n \geq 0$. If $n = 0$ this is the empty string, if $n = 1$ it equals s, etc.

`s[i]`

Returns the substring of s consisting of the i-th character.

`s[i]`

Returns the numeric value representing the i-th character of s.

`ElementToSequence(s)`
`Eltseq(s)`

Returns the sequence of characters of s (as length 1 strings).

`ElementToSequence(s)`
`Eltseq(s)`

Returns the sequence of numeric values representing the characters of s.

`Substring(s, n, k)`

Return the substring of s of length k starting at position n.

3.2.3 Integer-Valued Functions

`#s`

The length of the string s.

`Index(s, t)`
`Position(s, t)`

This function returns the position (an integer p with $0 < p \leq \#s$) in the string s where the beginning of a contiguous substring t occurs. It returns 0 if t is not a substring of s. (If t is the empty string, position 1 will always be returned, even if s is empty as well.)

3.2.4 Character Conversion

To perform more sophisticated operations, one may convert the string into a sequence and use the extensive facilities for sequences described in the next part of this manual; see the examples at the end of this chapter for details.

`StringToCode(s)`

Returns the code number of the first character of string s. This code depends on the computer system that is used; it is ASCII on most UNIX machines.

`CodeToString(n)`

Returns a character (string of length 1) corresponding to the code number n, where the code is system dependent (see previous entry).

`StringToInteger(s)`

Returns the integer corresponding to the string of decimal digits s. All non-space characters in the string s must be digits $(0, 1, \ldots, 9)$, except the first character, which is also allowed to be + or −. An error results if any other combination of characters occurs. Leading zeros are omitted.

`StringToInteger(s, b)`

Returns the integer corresponding to the string of digits s, all assumed to be written in base b. All non-space characters in the string s must be digits less than b (if b is greater than 10, 'A' is used for 10, 'B' for 11, etc.), except the first character, which is also allowed to be + or −. An error results if any other combination of characters occurs.

`StringToIntegerSequence(s)`

Returns the sequence of integers corresponding to the string s of space-separated decimal numbers. All non-space characters in the string s must be digits $(0, 1, \ldots, 9)$, except the first character after each space, which is also allowed to be + or −. An error results if any other combination of characters occurs. Leading zeros are omitted. Each number can begin with a sign (+ or −) without a space.

`IntegerToString(n)`

Convert the integer n into a string of decimal digits; if n is negative the first character of the string will be −. (Note that leading zeros and a + sign are ignored when MAGMA builds an integer, so the resulting string will never begin with + or 0 characters.)

`IntegerToString(n, b)`

Convert the integer n into a string of digits with the given base (which must be in the range $[2 \ldots 36]$); if n is negative the first character of the string will be −.

3.2.5 Boolean Functions

`s eq t`

Returns `true` if and only if the strings s and t are identical. Note that blanks are significant.

`s ne t`

Returns `true` if and only if the strings s and t are distinct. Note that blanks are significant.

`s in t`

Returns `true` if and only if s appears as a contiguous substring of t. Note that the empty string is contained in every string.

`s notin t`

Returns `true` if and only if s does not appear as a contiguous substring of t. Note that the empty string is contained in every string.

`s lt t`

Returns `true` if s is lexicographically less than t, `false` otherwise. Here the ordering on characters imposed by their ASCII code number is used.

`s le t`

Returns `true` if s is lexicographically less than or equal to t, `false` otherwise. Here the ordering on characters imposed by their ASCII code number is used.

`s gt t`

Returns `true` if s is lexicographically greater than t, `false` otherwise. Here the ordering on characters imposed by their ASCII code number is used.

`s ge t`

Returns `true` if s is lexicographically greater than or equal to t, false otherwise. Here the ordering on characters imposed by their ASCII code number is used.

Example H3E1

```
> "Mag" cat "ma";
Magma
```

Omitting double-quotes usually has undesired effects:

```
> "Mag cat ma";
Mag cat ma
```

And note that there are two different equalities involved in the following!

```
> "73" * "9" * "42" eq "7" * "3942";
true
> 73 * 9 * 42 eq 7 * 3942;
true
```

The next line shows how strings can be concatenated quickly, and also that strings of blanks can be used for formatting:

```
> s := ("Mag" cat "ma? ")^2;
> s, " "^30, s[4]^12, "!";
Magma? Magma?                              mmmmmmmmmmmm !
```

Here is a way to list (in a sequence) the first occurrence of each of the ten digits in the decimal expansion of π, using `IntegerToString` and `Position`.

```
> pi := Pi(RealField(1001));
> dec1000 := Round(10^1000*(pi-3));
> I := IntegerToString(dec1000);
> [ Position(I, IntegerToString(i)) : i in [0..9] ];
```

```
[ 32, 1, 6, 9, 2, 4, 7, 13, 11, 5 ]
```

Using the length `#` and string indexing `[ ]` it is also easy to count the number of occurrences of each digit in the string containing the first 1000 digits.

```
> [ #[i : i in [1..#I] | I[i] eq IntegerToString(j)] : j in [0..9] ];
[ 93, 116, 103, 102, 93, 97, 94, 95, 101, 106 ]
```

We would like to test if the ASCII-encoding of the string 'Magma' appears. This could be done as follows, using `StringToCode` and `in`, or alternatively, `Position`. To reduce the typing, we first abbreviate `IntegerToString` to `is` and `StringToCode` to `sc`.

```
> sc := StringToCode;
> its := IntegerToString;
> M := its(sc("M")) * its(sc("a")) * its(sc("g")) * its(sc("m")) * its(sc("a"));
> M;
779710310997
> M in I;
false
> Position(I, M);
0
```

So 'Magma' does not appear this way. However, we could be satisfied if the letters appear somewhere in the right order. To do more sophisticated operations (like this) on strings, it is necessary to convert the string into a sequence, because sequences constitute a more versatile data type, allowing many more advanced operations than strings.

```
> Iseq := [ I[i] : i in [1..#I] ];
> Mseq := [ M[i] : i in [1..#M] ];
> IsSubsequence(Mseq, Iseq);
false
> IsSubsequence(Mseq, Iseq: Kind := "Sequential");
true
```

Finally, we find that the string 'magma' lies in between 'Pi' and 'pi':

```
> "Pi" le "magma";
true
> "magma" lt "pi";
true
```

3.2.6 Parsing Strings

`Split(S, D)`

`Split(S)`

Given a string S, together with a string D describing a list of separator characters, return the sequence of strings obtained by splitting S at any of the characters contained in D. That is, S is considered as a sequence of fields, with any character in D taken to be a delimiter separating the fields. If D is omitted, it is taken to be the string consisting of the newline character alone (so S is split into the lines found in it). If S is desired to be split into space-separated words, the argument `" \t\n"` should be given for D.

Example H3E2

We demonstrate elementary uses of `Split`.

```
> Split("a b c d", " ");
[ a, b, c, d ]
> // Note that an empty field is included if the
> // string starts with the separator:
> Split(" a b c d", " ");
[ , a, b, c, d ]
> Split("abxcdyefzab", "xyz");
[ ab, cd, ef, ab ]
> // Note that no splitting happens if the delimiter
> // is empty:
> Split("abcd", "");
[ abcd ]
```

`Regexp(R, S)`

Given a string R specifying a regular expression, together with a string S, return whether S matches R. If so, return also the matched substring of S, together with the sequence of matched substrings of S corresponding to the parenthesized expressions of R. This function is based on the freely distributable reimplementation of the V8 regexp package by Henry Spencer. The syntax and interpretation of the characters `|`, `*`, `+`, `?`, `^`, `$`, `[]`, `\` is the same as in the UNIX command `egrep`. The parenthesized expressions are numbered in left-to-right order of their opening parentheses. Note that the parentheses should not have an initial backslash before them as the UNIX commands `grep` and `ed` require.

Example H3E3

We demonstrate some elementary uses of `Regexp`.

```
> Regexp("b.*d", "abcde");
true bcd []
> Regexp("b(.*)d", "abcde");
true bcd [ c ]
> Regexp("b.*d", "xyz");
false
> date := "Mon Jun 17 10:27:27 EST 1996";
> _, _, f := Regexp("([0-9][0-9]):([0-9][0-9]):([0-9][0-9])", date);
> f;
[ 10, 27, 27 ]
> h, m, s := Explode(f);
> h, m, s;
10 27 27
```

3.3 Printing

3.3.1 The `print`-Statement

`print` *expression*;

`print` *expression*`, ..., ` *expression*;

`print` *expression*`:` *parameters*;

Print the value of the expression. Some limited ways of formatting output are described in the section on strings. Four levels of printing (that may in specific cases coincide) exist, and may be indicated after the colon: `Default` (which is the same as the level obtained if no level is indicated), `Minimal`, `Maximal`, and `Magma`. The last of these produces output representing the value of the identifier as valid MAGMA-input (when possible).

3.3.2 The `printf` and `fprintf` Statements

`printf` *format*`,` *expression*`, ...,` *expression*`;`

Print values of the expressions under control of *format*. The first argument, the *format string*, must be a string which contains two types of objects: plain characters, which are simply printed, and conversion specifications (indicated by the `%` character), each of which causes conversion and printing of zero or more of the expressions. (Use `%%` to get a literal percent character.) Currently, the only conversion specifications allowed are: `%o` and `%O`, which stand for "object", `%m`, which stands for "magma", and `%h`, which stands for "hexadecimal".

The hexadecimal conversion specification will print its argument in hexadecimal; currently, it only supports integer arguments. The object and magma conversion specifications each print the corresponding argument; they differ only in the printing mode used. The `%o` form uses the default printing mode, while the `%O` form uses the printing mode specified by the next argument (as a string). The "magma" conversion specification uses a printing mode of `Magma`. It is thus equivalent to (but shorter than) using `%O` and an extra argument of `"Magma"`.

For each of these conversion specifications, the object can be printed in a field of a particular width by placing extra characters immediately after the `%` character: digits describing a positive integer, specifying a field with width equal to that number and with right-justification; digits describing a negative integer, specifying a field with width equal to the absolute value of the number and with left-justification; or the character `*` specifying a field width given by the next appropriate expression argument (with justification determined by the sign of the number). This statement is thus like the C language function `printf()`, except that `%o` (and `%O` and `%m`) covers all kinds of objects — it is not necessary to have different conversion specifications for the different types of MAGMA objects. Note also that this statement does *not* print a newline character after its arguments while the `print` statement does (a `\n` character should be placed in the format string if this is desired). A newline character will be printed just before the next prompt, though, if there is an incomplete line at that point.

Example H3E4

The following statements demonstrate simple uses of *printf*.

```
> for i := 1 to 150 by 33 do printf "[%3o]\n", i; end for;
[  1]
[ 34]
[ 67]
[100]
[133]
> for i := 1 to 150 by 33 do printf "[%-3o]\n", i; end for;
[1  ]
[34 ]
[67 ]
```

```
[100]
[133]
> for w := 1 to 5 do printf "[%*o]", w, 1; end for;
[1][ 1][  1][   1][    1]
```

Example H3E5

Some further uses of the `printf` statement are illustrated below.

```
> x := 3;
> y := 4;
> printf "x = %o, y = %o\n", x, y;
x = 3, y = 4
> printf "G'"; printf "day";
G'day
> p := 53.211;
> x := 123.2;
> printf "%.3o%% of %.2o is %.3o\n", p, x, p/100.0 * x;
53.211% of 123.20 is 65.556
> Zx<x> := PolynomialRing(Integers());
> printf "%0\n", x, "Magma";
Polynomial(\[0, 1])
```

`fprintf` *file*, *format*, *expression*, ..., *expression*;

Print values of the expressions under control of *format* into the file given by *file*. The first argument *file* must be either a string specifying a file which can be opened for appending (tilde expansion is performed on the filename), or an file object (see the section below on external files) opened for writing. The rest of the arguments are exactly as in the `printf` statement. In the string (filename) case, the file is opened for appending, the string obtained from the formatted printing of the other arguments is appended to the file, and the file is closed. In the file object case, the string obtained from the formatted printing of the other arguments is simply appended to the file. Note that this statement, like `printf`, does *not* print a newline character after its arguments (a `\n` character should be placed in the format string if this is desired).

Example H3E6

The following statements demonstrate a (rather contrived) use of *fprintf* with a file pipe.

```
> p := 10000000000000000000000000000057;
> F := POpen("sort -n", "w");
> for i := 100 to 110 do
>    fprintf F, "%30o (2^%o mod p)\n", 2^i mod p, i;
> end for;
> // Close F and then see output on standard output:
```

```
> delete F;
 37107316853453566312041115519 (2^109 mod p)
 70602400912917605986812821219 (2^102 mod p)
 74214633706907132624082231038 (2^110 mod p)
129638414606681695789005139447 (2^106 mod p)
141204801825835211973625642438 (2^103 mod p)
259276829213363391578010278894 (2^107 mod p)
267650600228229401496703205319 (2^100 mod p)
282409603651670423947251284876 (2^104 mod p)
518553658426726783156020557788 (2^108 mod p)
535301200456458802993406410638 (2^101 mod p)
564819207303340847894502569752 (2^105 mod p)
```

3.3.3 Verbose Printing (`vprint`, `vprintf`)

The following statements allow convenient printing of information conditioned by whether an appropriate verbose flag is turned on.

`vprint` *flag*: *expression*, ..., *expression*;

`vprint` *flag*, *n*: *expression*, ..., *expression*;

If the verbose flag *flag* (see the function `SetVerbose`) has a level greater than or equal to n, print the expressions to the right of the colon exactly as in the `print` statement. If the flag has level 0 (i.e. is not turned on), do nothing. In the first form of this statement, where a specific level is not given, n is taken to be 1. This statement is useful in MAGMA code found in packages where one wants to print verbose information if an appropriate verbose flag is turned on.

`vprintf` *flag*: *format*, *expression*, ..., *expression*;

`vprintf` *flag*, *n*: *format*, *expression*, ..., *expression*;

If the verbose flag *flag* (see the function `SetVerbose`) has a level greater than or equal to n, print using the format and the expressions to the right of the colon exactly as in the `printf` statement. If the flag has level 0 (i.e. is not turned on), do nothing. In the first form of this statement, where a specific level is not given, n is taken to be 1. This statement is useful in MAGMA code found in packages where one wants to print verbose information if an appropriate verbose flag is turned on.

3.3.4 Automatic Printing

MAGMA allows *automatic printing* of expressions: basically, a statement consisting of an expression (or list of expressions) alone is taken as a shorthand for the `print`-statement.

Some subtleties are involved in understanding the precise behaviour of MAGMA in interpreting lone expressions as statements. The rules MAGMA follows are outlined here. In the following, a *call-form* means any expression of the form $f(arguments)$; that is, anything which could be a procedure call *or* a function call.

(a) Any single expression followed by a semicolon which is not a call-form is printed, just as if you had 'print' in front of it.

(b) For a single call-form followed by a semicolon (which could be a function call or procedure call), the first signature which matches the input arguments is taken and if that is procedural, the whole call is taken as a procedure call, otherwise it is taken as function call and the results are printed.

(c) A comma-separated list of any expressions is printed, just as if you had 'print' in front of it. Here any call-form is taken as a function call only so procedure calls are impossible.

(d) A print level modifier is allowed after an expression list (whether the list has length 1 or more). Again any call-form is taken as a function call only so procedure calls are impossible.

(e) Any list of objects printed, whether by any of the above rules or by the 'print' statement, is placed in the previous value buffer. `$1` gives the last printed list, `$2` the one before, etc. Note that multi-return values stay as a list of values in the previous value buffer. The only way to get at the individual values of such a list is by assignment to a list of identifiers, or by `where` (this is of course the only way to get the second result out of `Quotrem`, etc.). In other places, a `$1` expression is evaluated with principal value semantics.

MAGMA also provides procedures to manipulate the previous value buffer in which $1, etc. are stored.

`ShowPrevious()`

Show all the previous values stored. This does *not* change the contents of the previous value buffer.

`ShowPrevious(i)`

Show the i-th previous value stored. This does *not* change the contents of the previous value buffer.

`ClearPrevious()`

Clear all the previous values stored. This is useful for ensuring that no more memory is used than that referred to by the current identifiers.

`SetPreviousSize(n)`

Set the size of the previous value buffer (this is not how many values are defined in it at the moment, but the maximum number that will be stored). The default size is 3.

`GetPreviousSize()`

Return the size of the previous value buffer.

Example H3E7

Examples which illustrate point (a):

```
> 1;
1
> x := 3;
> x;
3
```

Examples which illustrate point (b):

```
> 1 + 1;            // really function call '+'(1, 1)
2
> Q := [ 0 ];
> Append(~Q, 1);    // first (in fact only) match is procedure call
> Append(Q, 1);     // first (in fact only) match is function call
[ 0, 1, 1 ]
>  // Assuming fp is assigned to a procedure or function:
>  fp(x);            // whichever fp is at runtime
> SetVerbose("Meataxe", true);  // simple procedure call
```

Examples which illustrate point (c):

```
> 1, 2;
1 2
> // Assuming f assigned:
>  f(x), 1;                           // f only can be a function
>  SetVerbose("Meataxe", true), 1;    // type error in 'SetVerbose'
>                                     // (since no function form)
```

Examples which illustrate point (d):

```
> 1: Magma;
1
> Sym(3), []: Maximal;
Symmetric group acting on a set of cardinality 3
Order = 6 = 2 * 3
[]
>  SetVerbose("Meataxe", true): Magma; // type error as above
```

Examples which illustrate point (e):

```
> 1;
```

```
1
> $1;
1
> 2, 3;
2 3
> $1;
2 3
> Quotrem(124124, 123);
1009 17
> $1;
1009 17
> a, b := $1;
> a;
1009
```

3.3.5 Indentation

MAGMA has an indentation level which determines how many initial spaces should be printed before each line. The level can be increased or decreased. Each time the top level of Magma is reached (i.e. a prompt is printed), the level is reset to 0. The level is usually changed in verbose output of recursive functions and procedures. The functions `SetIndent` and `GetIndent` are used to control and examine the number of spaces used for each indentation level (default 4).

`IndentPush()`

Increase (push) the indentation level by 1. Thus the beginning of a line will have s more spaces than before, where s is the current number of indentation spaces.

`IndentPop()`

Decrease (pop) the indentation level by 1. Thus the beginning of a line will have s less spaces than before, where s is the current number of indentation spaces. If the current level is already 0, an error occurs.

3.3.6 Printing to a File

`PrintFile(F, x)`

`Write(F, x)`

`Overwrite` BOOLELT *Default :* `false`

Print x to the file specified by the string F. If this file already exists, the output will be appended, unless the optional parameter `Overwrite` is set to true, in which case the file is overwritten.

`WriteBinary(F, s)`

`Overwrite` BoolElt *Default :* `false`

Write the binary string s to the file specified by the string F. If this file already exists, the output will be appended, unless the optional parameter `Overwrite` is set to true, in which case the file is overwritten.

`PrintFile(F, x, L)`

`Write(F, x, L)`

`Overwrite` BoolElt *Default :* `false`

Print x in format defined by the string L to the file specified by the string F. If this file already exists, the output will be appended unless the optional parameter `Overwrite` is set to true, in which case the file is overwritten. The level L can be any of the print levels on the `print` command above (i.e., it must be one of the strings `"Default"`, `"Minimal"`, `"Maximal"`, or `"Magma"`).

`PrintFileMagma(F, x)`

`Overwrite` BoolElt *Default :* `false`

Print x in Magma format to the file specified by the string F. If this file already exists, the output will be appended, unless the optional parameter `Overwrite` is set to true, in which case the file is overwritten.

3.3.7 Printing to a String

Magma allows the user to obtain the string corresponding to the output obtained when printing an object by means of the `Sprint` function. The `Sprintf` function allows formatted printing like the `printf` statement.

`Sprint(x)`

`Sprint(x, L)`

Given any Magma object x, this function returns a string containing the output obtained when x is printed. If a print level L is given also (a string), the printing is done according to that level (see the `print` statement for the possible printing levels).

`Sprintf(F, ...)`

Given a format string F, together with appropriate extra arguments corresponding to F, return the string resulting from the formatted printing of F and the arguments. The format string F and arguments should be exactly as for the `printf` statement – see that statement for details.

Example H3E8

We demonstrate elementary uses of `Sprintf`.

```
> Q := [Sprintf("{%4o<->%-4o}", x, x): x in [1,10,100,1000]];
> Q;
[ {   1<->1   }, {  10<->10  }, { 100<->100 }, {1000<->1000} ]
```

3.3.8 Redirecting Output

`SetOutputFile(F)`

`Overwrite` BoolElt *Default :* `false`

Redirect all Magma output to the file specified by the string F. By using `SetOutputFile(F: Overwrite := true)` the file F is emptied before output is written onto it.

`UnsetOutputFile()`

Close the output file, so that output will be directed to standard output again.

`HasOutputFile()`

If Magma currently has an output or log file F, return `true` and F; otherwise return `false`.

3.4 External Files

Magma provides a special *file* type for the reading and writing of external files. Most of the standard C library functions can be applied to such files to manipulate them.

3.4.1 Opening Files

`Open(S, T)`

Given a filename (string) S, together with a type indicator T, open the file named by S and return a Magma file object associated with it. Tilde expansion is performed on S. The standard C library function `fopen()` is used, so the possible characters allowed in T are the same as those allowed for that function in the current operating system, and have the same interpretation. Thus one should give the value `"r"` for T to open the file for reading, and give the value `"w"` for T to open the file for writing, etc. (Note that in the PC version of Magma, the character `"b"` should also be included in T if the file is desired to be opened in binary mode.) Once a file object is created, various I/O operations can be performed on it — see below. A file is closed by deleting it (i.e. by use of the `delete` statement or by reassigning the variable associated with the file); there is no `Fclose` function. This ensures that the file is not closed while there are still multiple references to it. (The function is called `Open` instead of `Fopen` to follow Perl-style conventions. The following functions also follow such conventions where possible.)

3.4.2 Operations on File Objects

`Flush(F)`

Given a file F, flush the buffer of F.

`Tell(F)`

Given a file F, return the offset in bytes of the file pointer within F.

`Seek(F, o, p)`

Perform `fseek(F, o, p)`; i.e. move the file pointer of F to offset o (relative to p: 0 means beginning, 1 means current, 2 means end).

`Rewind(F)`

Perform `rewind(F)`; i.e. move the file pointer of F to the beginning.

`Put(F, S)`

Put (write) the characters of the string S to the file F.

`Puts(F, S)`

Put (write) the characters of the string S, followed by a newline character, to the file F.

`Getc(F)`

Given a file F, get and return one more character from file F as a string. If F is at end of file, a special EOF marker string is returned; the function `IsEof` should be applied to the character to test for end of file. (Thus the only way to loop over a file character by character is to get each character and test whether it is the EOF marker before processing it.)

`Gets(F)`

Given a file F, get and return one more line from file F as a string. The newline character is removed before the string is returned. If F is at end of file, a special EOF marker string is returned; the function `IsEof` should be applied to the string to test for end of file.

`IsEof(S)`

Given a string S, return whether S is the special EOF marker.

`Ungetc(F, c)`

Given a character (length one string) C, together with a file F, perform `ungetc(C, F)`; i.e. push the character C back into the input buffer of F.

Example H3E9

We write a function to count the number of lines in a file. Note the method of looping over the characters of the file: we must get the line and then test whether it is the special EOF marker.

```
> function LineCount(F)
>     FP := Open(F, "r");
>     c := 0;
>     while true do
>         s := Gets(FP);
>         if IsEof(s) then
>             break;
>         end if;
>         c +:= 1;
>     end while;
>     return c;
> end function;
> LineCount("/etc/passwd");
59
```

3.4.3 Reading a Complete File

`Read(F)`

Function that returns the contents of the text-file with name indicated by the string F. Here F may be an expression returning a string.

`ReadBinary(F)`

Function that returns the contents of the text-file with name indicated by the string F as a binary string.

Example H3E10

In this example we show how `Read` can be used to import the complete output from a separate C program into a MAGMA session. We assume that a file `mystery.c` (of which the contents are shown below) is present in the current directory. We first compile it, from within MAGMA, and then use it to produce output for the MAGMA version of our `mystery` function.

```
> Read("mystery.c");
#include <stdio.h>
main(argc, argv)
int     argc;
char    **argv;
{
    int n, i;
    n = atoi(argv[1]);
    for (i = 1; i <= n; i++)
        printf("%d\n", i * i);
```

```
    return 0;
}
> System("cc mystery.c -o mystery");
> mysteryMagma := function(n)
>    System("./mystery " cat IntegerToString(n) cat " >outfile");
>    output := Read("outfile");
>    return StringToIntegerSequence(output);
> end function;
> mysteryMagma(5);
[ 1, 4, 9, 16, 25 ]
```

3.5 Pipes

Pipes are used to communicate with newly-created processes. Currently pipes are only available on UNIX systems.

The MAGMA I/O module is currently undergoing revision, and the current pipe facilities are a mix of the old and new methods. A more uniform model will be available in future releases.

3.5.1 Pipe Creation

`POpen(C, T)`

Given a shell command line C, together with a type indicator T, open a pipe between the MAGMA process and the command to be executed. The standard C library function `popen()` is used, so the possible characters allowed in T are the same as those allowed for that function in the current operating system, and have the same interpretation. Thus one should give the value `"r"` for T so that MAGMA can read the output from the command, and give the value `"w"` for T so that MAGMA can write into the input of the command. See the `Pipe` intrinsic for a method for sending input to, and receiving output from, a single command.

Important: this function returns a `File` object, and the I/O functions for files described previously must be used rather then those described in the following.

`Pipe(C, S)`

Given a shell command C and an input string S, create a pipe to the command C, send S into the standard input of C, and return the output of C as a string. Note that for many commands, S should finish with a new line character if it consists of only one line.

Example H3E11

We write a function which returns the current time as 3 values: hour, minutes, seconds. The function opens a pipe to the UNIX command "date" and applies regular expression matching to the output to extract the relevant fields.

```
> function GetTime()
>     D := POpen("date", "r");
>     date := Gets(D);
>     _, _, f := Regexp("([0-9][0-9]):([0-9][0-9]):([0-9][0-9])", date);
>     h, m, s := Explode(f);
>     return h, m, s;
> end function;
> h, m, s := GetTime();
> h, m, s;
14 30 01
> h, m, s := GetTime();
> h, m, s;
14 30 04
```

3.5.2 Operations on Pipes

When a read request is made on a pipe, the available data is returned. If no data is currently available, then the process waits until some does becomes available, and returns that. (It will also return if the pipe has been closed and hence no more data can be transmitted.) It does not continue trying to read more data, as it cannot tell whether or not there is some "on the way".

The upshot of all this is that care must be exercised as reads may return less data than is expected.

`Read(P : parameters)`

`Max` RngIntElt *Default :* 0

Waits for data to become available for reading from P and then returns it as a string. If the parameter `Max` is set to a positive value then at most that many characters will be read. Note that less than `Max` characters may be returned, depending on the amount of currently available data.

If the pipe has been closed then the special EOF marker string is returned.

`ReadBytes(P : parameters)`

`Max` RngIntElt *Default :* 0

Waits for data to become available for reading from P and then returns it as a sequence of bytes (integers in the range 0..255). If the parameter `Max` is set to a positive value then at most that many bytes will be read. Note that less than `Max` bytes may be returned, depending on the amount of currently available data.

If the pipe has been closed then the empty sequence is returned.

`Write(P, s)`

Writes the characters of the string s to the pipe P.

`WriteBytes(P, Q)`

Writes the bytes in the byte sequence Q to the pipe P. Each byte must be an integer in the range 0..255.

3.6 Sockets

Sockets may be used to establish communication channels between machines on the same network. Once established, they can be read from or written to in much the same ways as more familiar I/O constructs like files. One major difference is that the data is not instantly available, so the I/O operations take much longer than with files. Currently sockets are only available on UNIX systems.

Strictly speaking, a *socket* is a communication endpoint whose defining information consists of a network address and a port number. (Even more strictly speaking, the communication protocol is also part of the socket. MAGMA only uses TCP sockets, however, so we ignore this point from now on.)

The network address selects on which of the available network interfaces communication will take place; it is a string identifying the machine on that network, in either domain name or dotted-decimal format. For example, both `"localhost"` and `"127.0.0.1"` identify the machine on the loopback interface (which is only accessible from the machine itself), whereas `"foo.bar.com"` or `"10.0.0.3"` might identify the machine in a local network, accessible from other machines on that network.

The port number is just an integer that identifies the socket on a particular network interface. It must be less than 65 536. A value of 0 will indicate that the port number should be chosen by the operating system.

There are two types of sockets, which we will call client sockets and server sockets. The purpose of a client socket is to initiate a connection to a server socket, and the purpose of a server socket is to wait for clients to initiate connections to it. (Thus the server socket needs to be created before the client can connect to it.) Once a server socket accepts a connection from a client socket, a communication channel is established and the distinction between the two becomes irrelevant, as they are merely each side of a communication channel.

In the following descriptions, the network address will often be referred to as the *host*. So a socket is identified by a *(host, port)* pair, and an established communication channel consists of two of these pairs: *(local-host, local-port), (remote-host, remote-port)*.

3.6.1 Socket Creation

`Socket(H, P : parameters)`

`LocalHost`	MONSTGELT	*Default : none*
`LocalPort`	RNGINTELT	*Default :* 0

Attempts to create a (client) socket connected to port P of host H. Note: these are the *remote* values; usually it does not matter which local values are used for client

sockets, but for those rare occasions where it does they may be specified using the parameters `LocalHost` and `LocalPort`. If these parameters are not set then suitable values will be chosen by the operating system. Also note that port numbers below 1 024 are usually reserved for system use, and may require special privileges to be used as the local port number.

`Socket( : parameters)`

`LocalHost`	MonStgElt	*Default : none*
`LocalPort`	RngIntElt	*Default :* 0

Attempts to create a server socket on the current machine, that can be used to accept connections. The parameters `LocalHost` and `LocalPort` may be used to specify which network interface and port the socket will accept connections on; if either of these are not set then their values will be determined by the operating system. Note that port numbers below 1 024 are usually reserved for system use, and may require special privileges to be used as the local port number.

`WaitForConnection(S)`

This may only be used on server sockets. It waits for a connection attempt to be made, and then creates a new socket to handle the resulting communication channel. Thus S may continue to be used to accept connection attempts, while the new socket is used for communication with whatever entity just connected. Note: this new socket is *not* a server socket.

3.6.2 Socket Properties

`SocketInformation(S)`

This routine returns the identifying information for the socket as a pair of tuples. Each tuple is a <*host, port*> pair — the first tuple gives the local information and the second gives the remote information. Note that this second tuple will be undefined for server sockets.

3.6.3 Socket Predicates

`IsServerSocket(S)`

Returns whether S is a server socket or not.

3.6.4 Socket I/O

Due to the nature of the network, it takes significant time to transmit data from one machine to another. Thus when a read request is begun it may take some time to complete, usually because the data to be read has not yet arrived. Also, data written to a socket may be broken up into smaller pieces for transmission, each of which may take different amounts of time to arrive. Thus, unlike files, there is no easy way to tell if there is still more data to be read; the current lack of data is no indicator as to whether more might arrive.

When a read request is made on a socket, the available data is returned. If no data is currently available, then the process waits until some does becomes available, and returns that. (It will also return if the socket has been closed and hence no more data can be transmitted.) It does not continue trying to read more data, as it cannot tell whether or not there is some "on the way".

The upshot of all this is that care must be exercised as reads may return less data than is expected.

`Read(S : parameters)`

`Max` RngIntElt *Default :* 0

Waits for data to become available for reading from S and then returns it as a string. If the parameter `Max` is set to a positive value then at most that many characters will be read. Note that less than `Max` characters may be returned, depending on the amount of currently available data.

If the socket has been closed then the special EOF marker string is returned.

`ReadBytes(S : parameters)`

`Max` RngIntElt *Default :* 0

Waits for data to become available for reading from S and then returns it as a sequence of bytes (integers in the range 0..255). If the parameter `Max` is set to a positive value then at most that many bytes will be read. Note that less than `Max` bytes may be returned, depending on the amount of currently available data.

If the socket has been closed then the empty sequence is returned.

`Write(S, s)`

Writes the characters of the string s to the socket S.

`WriteBytes(S, Q)`

Writes the bytes in the byte sequence Q to the socket S. Each byte must be an integer in the range 0..255.

Example H3E12

Here is a trivial use of sockets to send a message from one MAGMA process to another running on the same machine. The first MAGMA process sets up a server socket and waits for another MAGMA to contact it.

```
> // First Magma process
> server := Socket(: LocalHost := "localhost");
> SocketInformation(server);
<localhost, 32794>
> S1 := WaitForConnection(server);
```

The second MAGMA process establishes a client socket connection to the first, writes a greeting message to it, and closes the socket.

```
> // Second Magma process
> S2 := Socket("localhost", 32794);
> SocketInformation(S2);
<localhost, 32795> <localhost, 32794>
> Write(S2, "Hello, other world!");
> delete S2;
```

The first MAGMA process is now able to continue; it reads and displays all data sent to it until the socket is closed.

```
> // First Magma process
> SocketInformation(S1);
<localhost, 32794> <localhost, 32795>
> repeat
>     msg := Read(S1);
>     msg;
> until IsEof(msg);
Hello, other world!
EOF
```

3.7 Interactive Input

`read` *identifier*;

`read` *identifier*, *prompt*;

This statement will cause MAGMA to assign to the given identifier the string of characters appearing (at run-time) on the following line. This allows the user to provide an input string at run-time. If the optional prompt is given (a string), that is printed first.

`readi` *identifier*;

`readi` *identifier*, *prompt*;

This statement will cause MAGMA to assign to the given identifier the literal integer appearing (at run-time) on the following line. This allows the user to specify integer input at run-time. If the optional prompt is given (a string), that is printed first.

3.8 Loading a Program File

`load "`*filename*`";`

Input the file with the name specified by the string. The file will be read in, and the text will be treated as MAGMA input. Tilde expansion of file names is allowed.

`iload "`*filename*`";`

(Interactive load.) Input the file with the name specified by the string. The file will be read in, and the text will be treated as MAGMA input. Tilde expansion of file names is allowed. In contrast to `load`, the user has the chance to interact as each line is read in:

As the line is read in, it is displayed and the system waits for user response. At this point, the user can skip the line (by moving "down"), edit the line (using the normal editing keys) or execute it (by pressing "enter"). If the line is edited, the new line is executed and the original line is presented again.

3.9 Saving and Restoring Workspaces

`save "`*filename*`";`

Copy all information present in the current MAGMA workspace onto a file specified by the string "*filename*". The workspace is left intact, so executing this command does not interfere with the current computation.

`restore "`*filename*`";`

Copy a previously stored MAGMA workspace from the file specified by the string "*filename*" into central memory. Information present in the current workspace prior to the execution of this command will be lost. The computation can now proceed from the point it was at when the corresponding `save`-command was executed.

3.10 Logging a Session

`SetLogFile(F)`

`Overwrite` BoolElt *Default :* `false`

Set the log file to be the file specified by the string F: all input and output will be sent to this log file as well as to the terminal. If a log file is already in use, it is closed and F is used instead. By using `SetLogFile(F: Overwrite := true)` the file F is emptied before input and output are written onto it. See also `HasOutputFile`.

`UnsetLogFile()`

Stop logging Magma's output.

`SetEchoInput(b)`

Set to true or false according to whether or not input from external files should also be sent to standard output.

3.11 Memory Usage

`GetMemoryUsage()`

Return the current memory usage of Magma (in bytes as an integer). This is the process data size, which does not include the executable code.

`GetMaximumMemoryUsage()`

Return the maximum memory usage of Magma (in bytes as an integer) which has been attained since last reset (see `ResetMaximumMemoryUsage`). This is the maximum process data size, which does not include the executable code.

`ResetMaximumMemoryUsage()`

Reset the value of the maximum memory usage of Magma to be the current memory usage of Magma (see `GetMaximumMemoryUsage`).

3.12 System Calls

`Alarm(s)`

A procedure which when used on UNIX systems, sends the signal SIGALRM to the Magma process after s seconds. This allows the user to specify that a Magma-process should self-destruct after a certain period.

`ChangeDirectory(s)`

Change to the directory specified by the string s. Tilde expansion is allowed.

`GetCurrentDirectory()`

Returns the current directory as a string.

`Getpid()`

Returns Magma's process ID (value of the Unix C system call `getpid()`).

`Getuid()`

Returns the user ID (value of `the Unix C system call getuid()`).

`System(C)`

Execute the system command specified by the string C. This is done by calling the C function `system()`.

This also returns the system command's return value as an integer. On most Unix systems, the lower 8 bits of this value give the process status while the next 8 bits give the value given by the command to the C function `exit()` (see the Unix manual entries for `system(3)` or `wait(2)`, for example). Thus one should normally divide the result by 256 to get the exit value of the program on success.

See also the `Pipe` intrinsic function.

`%!` *shell-command*

Execute the given command in the Unix shell then return to Magma. Note that this type of shell escape (contrary to the one using a `System` call) takes place entirely outside MAGMA and does not show up in MAGMA's history.

3.13 Creating Names

Sometimes it is necessary to create names for files from within MAGMA that will not clash with the names of existing files.

`Tempname(P)`

Given a prefix string P, return a unique temporary name derived from P (by use of the C library function `mktemp()`).

4 ENVIRONMENT AND OPTIONS

Chapter 4
ENVIRONMENT AND OPTIONS

4.1 Introduction

This chapter describes the environmental features of MAGMA, together with options which can be specified at start-up on the command line, or within MAGMA by the `Set-` procedures. The history and line-editor features of MAGMA are also described.

4.2 Command Line Options

When starting up MAGMA, various command-line options can be supplied, and a list of files to be automatically loaded can also be specified. These files may be specified by simply listing their names as normal arguments (i.e., without a `-` option) following the MAGMA command. For each such file name, a search for the specified file is conducted, starting in the current directory, and in directories specified by the environment variable `MAGMA_PATH` after that if necessary. It is also possible to have a *startup file*, in which one would usually store personal settings of parameters and variables. The startup file is specified by the `MAGMA_STARTUP_FILE` environment variable which should be set in the user's `.cshrc` file or similar. This environment variable can be overridden by the `-s` option, or cancelled by the `-n` option. The files specified by the arguments to MAGMA are loaded *after* the startup file. Thus the startup file is not cancelled by giving extra file arguments, which is what is usually desired.

MAGMA also allows one to set variables from the command line — if one of the arguments is of the form *var*`:=`*val*, where *var* is a valid identifier (consisting of letters, underscores, or non-initial digits) and there is no space between *var* and the `:=`, then the variable *var* is assigned within MAGMA to the *string* value *val* at the point where that argument is processed. (Functions like `StringToInteger` should be used to convert the value to an object of another type once inside MAGMA.)

`magma -b`

If the `-b` argument is given to MAGMA, the opening banner and all other introductory messages are suppressed. The final "total time" message is also suppressed. This is useful when sending the whole output of a MAGMA process to a file so that extra removing of unwanted output is not needed.

`magma -c` *filename*

If the `-c` argument is given to MAGMA, followed by a filename, the filename is assumed to refer to a package source file and the package is compiled and MAGMA then exits straight away. This option is rarely needed since packages are automatically compiled when attached.

`magma -d`

If the `-d` option is supplied to MAGMA, the licence for the current `magmapassfile` is dumped. That is, the expiry date and the valid hostids are displayed. MAGMA then exits.

`magma -n`

If the `-n` option is supplied to MAGMA, any startup file specified by the environment variable `MAGMA_STARTUP_FILE` or by the `-s` option is cancelled.

`magma -q name`

If the `-q` option is supplied to MAGMA, then MAGMA operates in a special manner as a slave (with the given name) for the `MPQS` integer factorisation algorithm. Please see that function for more details.

`magma -r workspace`

If the `-r` option is supplied to MAGMA, together with a workspace file, that workspace is automatically restored by MAGMA when it starts up.

`magma -s` *filename*

If the `-s` option is supplied to MAGMA, the given filename is used for the startup file for MAGMA. This overrides the variable of the environment variable `MAGMA_STARTUP_FILE` if it has been set. This option should not be used (as it was before), for automatically loading files since that can be done by just listing them as arguments to the MAGMA process.

`magma -S integer`

When starting up MAGMA, it is possible to specify a seed for the generation of pseudo-random numbers. (Pseudo-random quantities are used in several MAGMA algorithms, and may also be generated explicitly by some intrinsics.) The seed should be in the range 0 to $(2^{32} - 1)$ inclusive. If `-S` is not followed by any number, or if the `-S` option is not used, MAGMA selects the seed itself.

Example H4E1

By typing the command

```
magma file1 x:=abc file2
```

MAGMA would start up, read the user's startup file specified by `MAGMA_STARTUP_FILE` if existent, then read the file `file1`, then assign the variable `x` to the string value `"abc"`, then read the file `file2`, then give the prompt.

4.3 Environment Variables

This section lists some environment variables used by MAGMA. These variables are set by an appropriate operating system command and are used to define various search paths and other run-time options.

`MAGMA_STARTUP_FILE`

The name of the default start-up file. It can be overridden by the `magma -s` command.

`MAGMA_PATH`

Search path for files that are loaded (a colon separated list of directories). It need not include directories for the libraries, just personal directories. This path is searched before the library directories.

`MAGMA_MEMORY_LIMIT`

Limit on the size of the memory that may be used by a MAGMA-session (in bytes).

`MAGMA_LIBRARY_ROOT`

The root directory for the MAGMA libraries (by supplying an absolute path name). From within MAGMA `SetLibraryRoot` and `GetLibraryRoot` can be used to change and view the value.

`MAGMA_LIBRARIES`

Give a list of MAGMA libraries (as a colon separated list of sub-directories of the library root directory). From within MAGMA `SetLibraries` and `GetLibraries` can be used to change and view the value.

`MAGMA_SYSTEM_SPEC`

The MAGMA system spec file containing the system packages automatically attached at start-up.

`MAGMA_USER_SPEC`

The personal user spec file containing the user packages automatically attached at start-up.

`MAGMA_HELP_DIR`

The root directory for the MAGMA help files.

`MAGMA_TEMP_DIR`

Optional variable containing the directory MAGMA is to use for temporary files. If not specified, this defaults to `/tmp` (on Unix-like systems) or the system-wide temporary directory (on Windows systems).

4.4 Set and Get

The `Set-` procedures allow the user to attach values to certain environment variables. The `Get-` functions enable one to obtain the current values of these variables.

`SetAssertions(b)`
`GetAssertions()`

Controls the checking of assertions (see the `assert` statement and related statements in the chapter on the language). Default is `SetAssertions(1)`. The relevant values are 0 for no checking at all, 1 for normal checks, 2 for debug checks and 3 for extremely stringent checking.

`SetAutoColumns(b)`
`GetAutoColumns()`

If enabled, the IO system will try to determine the number of columns in the window by using `ioctl()`; when a window change or a stop/cont occurs, the `Columns` variable (below) will be automatically updated. If disabled, the `Columns` variable will only be changed when explicitly done so by `SetColumns`. Default is `SetAutoColumns(true)`.

`SetAutoCompact(b)`
`GetAutoCompact()`

Control whether automatic compaction is performed. Normally the memory manager of MAGMA will compact all of its memory between each statement at the top level. This removes fragmentation and reduces excessive memory usage. In some very rare situations, the compactions may become very slow (one symptom is that an inordinate pause occurs between prompts when only a trivial operation or nothing is done). In such cases, turning the automatic compaction off may help (at the cost of possibly more use of memory). Default is `SetAutoCompact(true)`.

`SetBeep(b)`
`GetBeep()`

Controls 'beeps'. Default is `SetBeep(true)`.

`SetColumns(n)`
`GetColumns()`

Controls the number of columns used by the IO system. This affects the line editor and the output system. (As explained above, if AutoColumns is on, this variable will be automatically determined.) The number of columns will determine how words are wrapped. If set to 0, word wrap is not performed. The default value is `SetColumns(80)` (unless `SetAutoColumns(true)`).

`GetCurrentDirectory()`

Returns the current directory as a string. (Use `ChangeDirectory(s)` to change the working directory.)

`SetEchoInput(b)`

`GetEchoInput()`

Set to `true` or `false` according to whether or not input from external files should also be sent to standard output.

`GetEnvironmentValue(s)`

`GetEnv(s)`

Returns the value of the external environment variable *s* as a string.

`SetHistorySize(n)`

`GetHistorySize()`

Controls the number of lines saved in the history. If the number is set to 0, no history is preserved.

`SetIgnorePrompt(b)`

`GetIgnorePrompt()`

Controls the option to ignore the prompt to allow the pasting of input lines back in. If enabled, any leading '>' characters (possibly separated by white space) are ignored by the history system when the input file is a terminal, *unless* the line consists of the '>' character alone (without a following space), which could not come from a prompt since in a prompt a space or another character follows a '>'. Default is `SetIgnorePrompt(false)`.

`SetIgnoreSpaces(b)`

`GetIgnoreSpaces()`

Controls the option to ignore spaces when searching in the line editor. If the user moves up or down in the line editor using `<Ctrl>`-P or `<Ctrl>`-N (see the line editor key descriptions) and if the cursor is not at the beginning of the line, a search is made forwards or backwards, respectively, to the first line which starts with the same string as the string consisting of all the characters before the cursor. While doing the search, spaces are ignored if and only if this option is on (value `true`). Default is `SetIgnoreSpaces(true)`.

`SetIndent(n)`

`GetIndent()`

Controls the indentation level for formatting output. The default is `SetIndent(4)`.

`SetLibraries(s)`
`GetLibraries()`

Controls the MAGMA library directories via environment variable `MAGMA_LIBRARIES`. The procedure `SetLibraries` takes a string, which will be taken as the (colon-separated) list of sub-directories in the library root directory for the libraries; the function `GetLibraryRoot` returns the current value as a string. These directories will be searched when you try to `load` a file; note however that first the directories indicated by the current value of your path environment variable `MAGMA_PATH` will be searched. See `SetLibraryRoot` for the root directory.

`SetLibraryRoot(s)`
`GetLibraryRoot()`

Controls the root directory for the MAGMA libraries, via the environment variable `MAGMA_LIBRARY_ROOT`. The procedure `SetLibraryRoot` takes a string, which will be the absolute pathname for the root of the libraries; the function `GetLibraryRoot` returns the current value as a string. See also `SetLibraries`

`SetLineEditor(b)`
`GetLineEditor()`

Controls the line editor. Default is `SetLineEditor(true)`.

`SetLogFile(F)`

`Overwrite` BOOLELT *Default :* `false`

`UnsetLogFile()`

Procedure. Set the log file to be the file specified by the string F: all input and output will be sent to this log file as well as to the terminal. If a log file is already in use, it is closed and F is used instead. The parameter `Overwrite` can be used to indicate that the file should be truncated before writing input and output on it; by default the file is appended.

`SetMemoryLimit(n)`
`GetMemoryLimit()`

Set the limit (in bytes) of the memory which the memory manager will allocate (no limit if 0). Default is `SetMemoryLimit(0)`.

`SetNthreads(n)`
`GetNthreads()`

Set the number of threads to be used in multi-threaded algorithms to be n, if POSIX threads are enabled in this version of Magma. Currently, this affects the coding theory minimum weight algorithm (`MinimumWeight`) and the F_4 Gröbner basis algorithm for medium-sized primes (`Groebner`).

`SetOutputFile(F)`

`Overwrite` BoolElt *Default :* `false`

`UnsetOutputFile()`

Start/stop redirecting all Magma output to a file (specified by the string F). The parameter `Overwrite` can be used to indicate that the file should be truncated before writing output on it.

`SetPath(s)`

`GetPath()`

Controls the path by which the searching of files is done. The path consists of a colon separated list of directories which are searched in order (“.” implicitly assumed at the front). Tilde expansion is done on each directory. (May be overridden by the environment variable MAGMA_PATH.)

`SetPrintLevel(l)`

`GetPrintLevel()`

Controls the global printing level, which is one of `"Minimal"`, `"Magma"`, `"Maximal"`, `"Default"`. Default is `SetPrintLevel("Default")`.

`SetPrompt(s)`

`GetPrompt()`

Controls the terminal prompt (a string). Expansion of the following % escapes occurs:

%% The character %

%h The current history line number.

%S The parser ‘state’: when a new line is about to be read while the parser has only seen incomplete statements, the state consists of a stack of words like “if”, “while”, indicating the incomplete statements.

%s Like %S except that only the topmost word is displayed.

Default is `SetPrompt("%S> ")`.

`SetQuitOnError(b)`

Set whether Magma should quit on any error to b. If b is `true`, Magma will completely quit when any error (syntax, runtime, etc.) occurs. Default is `SetQuitOnError(false)`.

`SetRows(n)`

`GetRows()`

Controls the number of rows in a page used by the IO system. This affects the output system. If set to 0, paging is not performed. Otherwise a prompt is given after the given number of rows for a new page. The default value is `SetRows(0)`.

`GetTempDir()`

Returns the directory MAGMA uses for storing temporary files. May be influenced on startup via the MAGMA_TEMP_DIR environment variable (see Section 4.3).

`SetTraceback(n)`
`GetTraceback()`

Controls whether MAGMA should produce a traceback of user function calls before each error message. The default value is `SetTraceback(true)`.

`SetSeed(s, c)`
`GetSeed()`

Controls the initialization seed and step number for pseudo-random number generation. For details, see the section on random object generation in the chapter on statements and expressions.

`GetVersion()`

Return integers x, y and z such the current version of MAGMA is Vx.y–z.

`SetViMode(b)`
`GetViMode()`

Controls the type of line editor used: Emacs (`false`) or VI style. Default is `SetViMode(false)`.

4.5 Verbose Levels

By turning verbose printing on for certain modules within MAGMA, some information on computations that are performed can be obtained. For each option, the verbosity may have different levels. The default is level 0 for each option.

There are also 5 slots available for user-defined verbose flags. The flags can be set in user programs by `SetVerbose("User`n`", true)` where n should be one of $1, 2, 3, 4, 5$, and the current setting is returned by `GetVerbose("User`n`")`.

`SetVerbose(s, i)`
`SetVerbose(s, b)`

Set verbose level for `s` to be level i or b. Here the argument `s` must be a string. The verbosity may have different levels. An integer i for the second argument selects the appropriate level. A second argument i of 0 or b of `false` means no verbosity. A boolean value for b of `true` for the second argument selects level 1. (See above for the valid values for the string s).

`GetVerbose(s)`

Return the value of verbose flag s as an integer. (See above for the valid values for the string s).

`IsVerbose(s)`

Return the whether the value of verbose flag s is non-zero. (See above for the valid values for the string s).

`IsVerbose(s, l)`

Return the whether the value of verbose flag s is greater than or equal to l. (See above for the valid values for the string s).

`ListVerbose()`

List all verbose flags. That is, print each verbose flag and its maximal level.

`ClearVerbose()`

Clear all verbose flags. That is, set the level for all verbose flags to 0.

4.6 Other Information Procedures

The following procedures print information about the current state of MAGMA.

`ShowMemoryUsage()`

(Procedure.) Show MAGMA's current memory usage.

`ShowIdentifiers()`

(Procedure.) List all identifiers that have been assigned to.

`ShowValues()`

(Procedure.) List all identifiers that have been assigned to with their values.

`Traceback()`

(Procedure.) Display a traceback of the current Magma function invocations.

`ListSignatures(C)`

`Isa`	BOOLELT	*Default :* `true`
`Search`	MONSTGELT	*Default :* "*Both*"
`ShowSrc`	BOOLELT	*Default :* `false`

List all intrinsic functions, procedures and operators having objects from category C among their arguments or return values. The parameter `Isa` may be set to false so that any categories which C inherit from are not considered. The parameter `Search`, with valid string values `Both`, `Arguments`, `ReturnValues`, may be used to specify whether the arguments, the return values, or both, are considered (default both). `ShowSrc` can be used to see where package intrinsics are defined. Use `ListCategories` for the names of the categories.

`ListSignatures(F, C)`

`Isa`	BoolElt	*Default :* `true`
`Search`	MonStgElt	*Default :* "*Both*"
`ShowSrc`	BoolElt	*Default :* `false`

Given an intrinsic F and category C, list all signatures of F which match the category C among their arguments or return values. The parameters are as for the previous procedure.

`ListCategories()`

`ListTypes()`

Procedure to list the (abbreviated) names for all available categories in Magma.

4.7 History

Magma provides a history system which allows the recall and editing of previous lines. The history system is invoked by typing commands which begin with the history character '%'. Currently, the following commands are available.

`%p`

List the contents of the history buffer. Each line is preceded by its history line number.

`%p`n

List the history line n in %p format.

`%p`n_1 n_2

List the history lines in the range n_1 to n_2 in %p format.

`%P`

List the contents of the history buffer. The initial numbers are *not* printed.

`%P`n

List the history line n in %P format.

`%P`n_1 n_2

List the history lines in the range n_1 to n_2 in %P format.

`%s`

List the contents of the history buffer with an initial statement for each line to reset the random number seed to the value it was just before the line was executed. This is useful when one wishes to redo a computation using exactly the same seed as before but does not know what the seed was at the time.

`%s`n

Print the history line n in %s format.

`%s`n_1 n_2

Print the history lines in the range n_1 to n_2 in %s format.

`%S`

As for %s except that the statement to set the seed is only printed if the seed has changed since the previous time it was printed. Also, it is not printed if it would appear in the middle of a statement (i.e., the last line did not end in a semicolon).

`%S`n

Print the history line n in %S format.

`%S`n_1 n_2

Print the history lines in the range n_1 to n_2 in %S format.

`%`

Reenter the last line into the input stream.

`%`n

Reenter the line specified by line number n into the input stream.

`%`n_1 n_2

Reenter the history lines in the range n_1 to n_2 into the input stream.

`%e`

Edit the last line. The editor is taken to be the value of the EDITOR environment variable if is set, otherwise "/bin/ed" is used. If after the editor has exited the file has not been changed then nothing is done. Otherwise the contents of the new file are reentered into the input stream.

`%e`n

Edit the line specified by line number n.

`%e`n_1 n_2

Edit the history lines in the range n_1 to n_2.

`%!` *shell-command*

Execute the given command in the Unix shell then return to Magma.

4.8 The Magma Line Editor

Magma provides a line editor with both Emacs and VI style key bindings. To enable the VI style of key bindings, type

```
SetViMode(true)
```

and type

```
SetViMode(false)
```

to revert to the Emacs style of key bindings. By default ViMode is `false`; that is, the Emacs style is in effect.

Many key bindings are the same in both Emacs and VI style. This is because some VI users like to be able to use some Emacs keys (like `<Ctrl>-P`) as well as the VI command keys. Thus key bindings in Emacs which are not used in VI insert mode can be made common to both.

4.8.1 Key Bindings (Emacs and VI mode)

`<Ctrl>-`*key* means hold down the Control key and press *key*.

`<Return>`

Accept the line and print a new line. This works in any mode.

`<Backspace>`

`<Delete>`

Delete the previous character.

`<Tab>`

Complete the word which the cursor is on or just after. If the word doesn't have a unique completion, it is first expanded up to the common prefix of all the possible completions. An immediately following Tab key will list all of the possible completions. Currently completion occurs for system functions and procedures, parameters, reserved words, and user identifiers.

`<Ctrl>-A`

Move to the beginning of the line ("alpha" = "beginning").

`<Ctrl>-B`

Move back a character ("back").

`<Ctrl>-C`

Abort the current line and start a new line.

`<Ctrl>-D`

On an empty line, send a EOF character (i.e., exit at the top level of the command interpreter). If at end of line, list the completions. Otherwise, delete the character under the cursor ("delete").

`<Ctrl>-E`

Move to the end of the line ("end").

`<Ctrl>-F`

Move forward a character ("forward").

`<Ctrl>-H`

Same as Backspace.

`<Ctrl>-I`

Same as Tab.

`<Ctrl>-J`

Same as Return.

`<Ctrl>-K`

Delete all characters from the cursor to the end of the line ("kill").

`<Ctrl>-L`

Redraw the line on a new line (helpful if the screen gets wrecked by programs like "write", etc.).

`<Ctrl>-M`

Same as `<Return>`.

`<Ctrl>-N`

Go forward a line in the history buffer ("next"). If the cursor is not at the beginning of the line, go forward to the first following line which starts with the same string (ignoring spaces iff the ignore spaces option is on — see `SetIgnoreSpaces`) as the string consisting of all the characters before the cursor. Also, if `<Ctrl>`-N is typed initially at a new line and the last line entered was actually a recall of a preceding line, then the next line after that is entered into the current buffer. Thus to repeat a sequence of lines (with minor modifications perhaps to each), then one only needs to go back to the first line with `<Ctrl>`-P (see below), press `<Return>`, then successively press `<Ctrl>`-N followed by `<Return>` for each line.

`<Ctrl>-P`

Go back a line in the history buffer ("previous"). If the cursor is not at the beginning of the line, go back to the first preceding line which starts with the same string (ignoring spaces iff the ignore spaces option is on — see `SetIgnoreSpaces`) as the string consisting of all the characters before the cursor. For example, typing at a

new line `x:=` and then `<Ctrl>`-P will go back to the last line which assigned `x` (if a line begins with, say, `x :=`, it will also be taken).

`<Ctrl>-U`

Clear the whole of the current line.

`<Ctrl>-V`*char*

Insert the following character literally.

`<Ctrl>-W`

Delete the previous word.

`<Ctrl>-X`

Same as `<Ctrl>`-U.

`<Ctrl>-Y`

Insert the contents of the yank-buffer before the character under the cursor.

`<Ctrl>-Z`

Stop MAGMA.

`<Ctrl>-_`

Undo the last change.

`<Ctrl>-\`

Immediately quit MAGMA.

On most systems the arrow keys also have the obvious meaning.

4.8.2 Key Bindings in Emacs mode only

M*key* means press the Meta key and then *key*. (At the moment, the Meta key is only the Esc key.)

`Mb`
`MB`

Move back a word ("Back").

`Mf`
`MF`

Move forward a word ("Forward").

4.8.3 Key Bindings in VI mode only

In the VI mode, the line editor can also be in two modes: the insert mode and the command mode. When in the insert mode, any non-control character is inserted at the current cursor position. The command mode is then entered by typing the Esc key. In the command mode, various commands are given a *range* giving the extent to which they are performed. The following ranges are available:

`0`

Move to the beginning of the line.

`$`

Move to the end of the line.

`<Ctrl>-`*space*

Move to the first non-space character of the line.

`%`

Move to the matching bracket. (Bracket characters are (,), [,], {, }, <, and >.)

`;`

Move to the next character. (See 'F', 'f', 'T', and 't'.)

`,`

Move to the previous character. (See 'F', 'f', 'T', and 't'.)

`B`

Move back a space-separated word ("Back").

`b`

Move back a word ("back").

`E`

Move forward to the end of the space-separated word ("End").

`e`

Move forward to the end of the word ("end").

`F`*char*

Move back to the first occurrence of *char*.

`f`*char*

Move forward to the first occurrence of *char*.

`h`
`H`

Move back a character (`<Ctrl>-`H = Backspace).

`l`

`L`

Move back a character (`<Ctrl>-L` = forward on some keyboards).

`T`*char*

Move back to just after the first occurrence of *char*.

`t`*char*

Move forward to just before the first occurrence of *char*.

`w`

Move forward a space-separated word ("Word").

`W`

Move forward a word ("word").

Any range may be preceded by a number to multiply to indicate how many times the operation is done. The VI-mode also provides the *yank-buffer*, which contains characters which are deleted or "yanked" – see below.

The following keys are also available in command mode:

`A`

Move to the end of the line and change to insert mode ("Append").

`a`

Move forward a character (if not already at the end of the line) and change to insert mode ("append").

`C`

Delete all the characters to the end of line and change to insert mode ("Change").

`c`*range*

Delete all the characters to the specified range and change to insert mode ("change").

`D`

Delete all the characters to the end of line ("Delete").

`d`*range*

Delete all the characters to the specified range ("delete").

`I`

Move to the first non-space character in the line and change to insert mode ("Insert").

`i`

Change to insert mode ("insert").

`j`

Go forward a line in the history buffer (same as `<Ctrl>`-N).

`k`

Go back a line in the history buffer (same as `<Ctrl>`-P).

`P`

Insert the contents of the yank-buffer before the character under the cursor.

`p`

Insert the contents of the yank-buffer before the character after the cursor.

`R`

Enter over-type mode: typed characters replace the old characters under the cursor without insertion. Pressing Esc returns to the command mode.

`r`*char*

Replace the character the cursor is over with *char*.

`S`

Delete the whole line and change to insert mode ("Substitute").

`s`

Delete the current character and change to insert mode ("substitute").

`U`
`u`

Undo the last change.

`X`

Delete the character to the left of the cursor.

`x`

Delete the character under the cursor.

`Y`

"Yank" the whole line - i.e., copy the whole line into the yank-buffer ("Yank").

`y`*range*

Copy all characters from the cursor to the specified range into the yank-buffer ("yank").

4.9 The Magma Help System

Magma provides extensive online help facilities that can be accessed in different ways. The easiest way to access the documentation is by typing:

```
magmahelp
```

Which should start some browser (usually netscape) on the main page of the MAGMA documentation.

The easiest way to get some information about any MAGMA intrinsic is by typing: (Here we assume you to be interested in `FundamentalUnit`)

```
> FundamentalUnit;
```

Which now will list all signatures for this intrinsic (i.e. all known ways to use this function):

```
> FundamentalUnit;
Intrinsic 'FundamentalUnit'
Signatures:
    (<FldQuad> K) -> FldQuadElt
    (<RngQuad> O) -> RngQuadElt
        The fundamental unit of K or O
    (<RngQuad> R) -> RngQuadElt
        Fundamental unit of the real quadratic order.
```

Next, to get more detailed information, try

```
> ?FundamentalUnit
```

But now several things could happen depending on the installation. Using the default, you get

```
============================================================
PATH: /magma/ring-field-algebra/quadratic/operation/\
      class-group/FundamentalUnit
KIND: Intrinsic
============================================================
FundamentalUnit(K) : FldQuad -> FldQuadElt
FundamentalUnit(O) : RngQuad -> RngQuadElt
    A generator for the unit group of the order O or the
maximal order
    of the quadratic field K.
============================================================
```

Second, a WWW-browser could start on the part of the online help describing your function (or at least the index of the first character). Third, some arbitrary program could be called to provide you with the information.

If `SetVerbose("Help", true);` is set, MAGMA will show the exact command used and the return value obtained.

`SetHelpExternalBrowser(S, T)`

`SetHelpExternalBrowser(S)`

Defines the external browser to be used if `SetHelpUseExternalBrowser(true)` is in effect. The string has to be a valid command taking exactly one argument (`%s`) which will we replaced by a URL. In case two strings are provided, the second defines a fall-back system. Typical use for this is to first try to use an already running browser and if this fails, start a new one.

`SetHelpUseExternalBrowser(b)`

Tells MAGMA to actually use (or stop to use) the external browser. If both `SetHelpUseExternalSystem` and `SetHelpUseExternalBrowser` are set to `true`, the assignment made last will be effective.

`SetHelpExternalSystem(s)`

This will tell MAGMA to use a user defined external program to access the help. The string has to contain exactly one `%s` which will be replaced by the argument to `?`. The resulting string must be a valid command.

`SetHelpUseExternalSystem(b)`

Tells MAGMA to actually use (or stop to use) the external help system. If both `SetHelpUseExternalSystem` and `SetHelpUseExternalBrowser` are set to `true`, the assignment made last will be effective.

`GetHelpExternalBrowser()`

Returns the currently used command strings.

`GetHelpExternalSystem()`

Returns the currently used command string.

`GetHelpUseExternal()`

The first value is the currently used value from `SetHelpUseExternalBrowser`, the second reflects `SetHelpUseExternalSystem`.

4.9.1 Internal Help Browser

MAGMA has a very powerful internal help-browser that can be entered with

```
> ??
```

5 MAGMA SEMANTICS

Chapter 5
MAGMA SEMANTICS

5.1 Introduction

This chapter describes the semantics of MAGMA (how expressions are evaluated, how identifiers are treated, etc.) in a fairly informal way. Although some technical language is used (particularly in the opening few sections) the chapter should be easy and essential reading for the non-specialist. The chapter is descriptive in nature, describing how MAGMA works, with little attempt to justify why it works the way it does. As the chapter proceeds, it becomes more and more precise, so while early sections may gloss over or omit things for the sake of simplicity and learnability, full explanations are provided later.

It is assumed that the reader is familiar with basic notions like a function, an operator, an identifier, a type ...

And now for some buzzwords: MAGMA is an imperative, call by value, statically scoped, dynamically typed programming language, with an essentially functional subset. The remainder of the chapter explains what these terms mean, and why a user might want to know about such things.

5.2 Terminology

Some terminology will be useful. It is perhaps best to read this section only briefly, and to refer back to it when necessary.

The term *expression* will be used to refer to a textual entity. The term *value* will be used to refer to a run-time value denoted by an expression. To understand the difference between an expression and a value consider the expressions `1+2` and `3`. The expressions are textually different but they denote the same value, namely the integer 3.

A *function expression* is any expression of the form `function ... end function` or of the form `func< ... | ... >`. The former type of function expression will be said to be *in the statement form*, the latter *in the expression form*. A *function value* is the run-time value denoted by a function expression. As with integers, two function expressions can be textually different while denoting the same (i.e., extensionally equal) function value. To clearly distinguish function values from function expressions, the notation FUNC(... : ...) will be used to describe function *values*.

The *formal arguments* of a function in the statement form are the identifiers that appear between the brackets just after the `function` keyword, while for a function in the expression form they are the identifiers that appear before the `|`. The *arguments* to a function are the expressions between the brackets when a function is applied.

The *body* of a function in the statement form is the statements after the formal arguments. The body of a function in the expression form is the expression after the `|` symbol.

An identifier is said to occur *inside* a function expression when it is occurs textually anywhere in the body of a function.

5.3 Assignment

An assignment is an association of an identifier to a *value*. The statement,

```
> a := 6;
```

establishes an association between the identifier a and the value 6 (6 is said to be *the value of* a, or to be *assigned to* a). A collection of such assignments is called a *context*.

When a value V is assigned to an identifier I one of two things happens:

(1) if I has not been previously assigned to, it is added to the current context and associated with V. I is said to be *declared* when it is assigned to for the first time.

(2) if I has been previously assigned to, the value associated with I is changed to V. I is said to be *re-assigned*.

The ability to assign and re-assign to identifiers is why MAGMA is called an *imperative* language.

One very important point about assignment is illustrated by the following example. Say we type,

```
> a := 6;
> b := a+7;
```

After executing these two lines the context is `[ (a,6), (b,13) ]`. Now say we type,

```
> a := 0;
```

The context is now `[ (a,0), (b,13) ]`. Note that changing the value of a does *not* change the value of b because b's value is statically determined at the point where it is assigned. Changing a does *not* produce the context `[ (a,0), (b,7) ]`.

5.4 Uninitialized Identifiers

Before executing a piece of code MAGMA attempts to check that it is semantically well formed (i.e., that it will execute without crashing). One of the checks MAGMA makes is to check that an identifier is declared (and thus initialized) before it is used in an expression. So, for example assuming a had not been previously declared, then before executing either of the following lines MAGMA will raise an error:

```
> a;
> b := a;
```

MAGMA can determine that execution of either line will cause an error since a has no assigned value. The user should be aware that the checks made for semantic well-formedness are necessarily not exhaustive!

There is one important rule concerning uninitialized identifiers and assignment. Consider the line,

```
> a := a;
```

Now if a had been previously declared then this is re-assignment of a. If not then it is an error since a on the right hand side of the `:=` has no value. To catch this kind of error MAGMA checks the expression on the right hand side of the `:=` for semantic well formedness *before* it declares the identifiers on the left hand side of the `:=`. Put another way the identifiers on the left hand side are not considered to be declared in the right hand side, *unless* they were declared previously.

5.5 Evaluation in Magma

Evaluation is the process of computing (or constructing) a value from an expression. For example the value 3 can be computed from the expression `1+2`. Computing a value from an expression is also known as *evaluating an expression*.

There are two aspects to evaluation, namely *when* and *how* it is performed. This section discusses these two aspects.

5.5.1 Call by Value Evaluation

MAGMA employs call by value evaluation. This means that the arguments to a function are evaluated before the function is applied to those arguments. Assume f is a function value. Say we type,

```
> r := f( 6+7, true or false );
```

MAGMA evaluates the two arguments to 13 and true respectively, *before* applying f.

While knowing the exact point at which arguments are evaluated is not usually very important, there are cases where such knowledge is crucial. Say we type,

```
> f := function( n, b )
>        if b then return n else return 1;
> end function;
```

and we apply f as follows

```
> r := f( 4/0, false );
```

MAGMA treats this as an error since the $4/0$ is evaluated, and an error produced, *before* the function f is applied.

By contrast some languages evaluate the arguments to a function only if those arguments are encountered when executing the function. This evaluation process is known as call by name evaluation. In the above example r would be set to the value 1 and the expression $4/0$ would never be evaluated because b is false and hence the argument n would never be encountered.

Operators like + and ∗ are treated as infix functions. So

```
> r := 6+7;
```

is treated as the function application,

```
> r := '+'(6,7);
```

Accordingly all arguments to an operator are evaluated before the operator is applied.

There are three operators, 'select', 'and' and 'or' that are exceptions to this rule and are thus not treated as infix functions. These operators use call by name evaluation and only evaluate arguments as need be. For example if we type,

```
> false and (4/0 eq 6);
```

MAGMA will reply with the answer false since MAGMA knows that `false and X` for all X is false.

5.5.2 Magma's Evaluation Process

Let us examine more closely how MAGMA evaluates an expression as it will help later in understanding more complex examples, specifically those using functions and maps. To evaluate an expression MAGMA proceeds by a process of identifier substitution, followed by simplification to a canonical form. Specifically expression evaluation proceeds as follows,

(1) replace each identifier in the expression by its value in the current context.

(2) simplify the resultant *value* to its canonical form.

The key point here is that the replacement step takes an expression and yields an unsimplified *value*! A small technical note: to avoid the problem of having objects that are part expressions, part values, all substitutions in step 1 are assumed to be done simultaneously for all identifiers in the expression. The examples in this chapter will however show the substitutions being done in sequence and will therefore be somewhat vague about what exactly these hybrid objects are!

To clarify this process assume that we type,

```
> a := 6;
> b := 7;
```

producing the context `[ (a,6), (b,7) ]`. Now say we type,

```
> c := a+b;
```

This produces the context `[ (a,6), (b,7), (c,13) ]`. By following the process outlined above we can see how this context is calculated. The steps are,

(1) replace `a` in the expression `a+b` by its value in the current context giving `6+b`.

(2) replace `b` in `6+b` by its value in the current context giving `6+7`.

(3) simplify `6+7` to `13`

The result value of `13` is then assigned to `c` giving the previously stated context.

5.5.3 Function Expressions

MAGMA's evaluation process might appear to be an overly formal way of stating the obvious about calculating expression values. This formality is useful, however when it comes to function (and map) expressions.

Functions in MAGMA are first class values, meaning that MAGMA treats function values just like it treats any other type of value (e.g., integer values). A function value may be passed as an argument to another function, may be returned as the result of a function, and may be assigned to an identifier in the same way that any other type of value is. Most importantly however function expressions are evaluated *exactly* as are all other expressions. The fact that MAGMA treats functions as first class values is why MAGMA is said to have an essentially functional subset.

Take the preceding example. It was,

```
> a := 6;
> b := 7;
> c := a+b;
```

giving the context `[ (a,6),(b,7),(c,13) ]`. Now say I type,

```
> d := func< n | a+b+c+n >;
```

MAGMA uses the same process to evaluate the function expression `func< n | a+b+c+n >` on the right hand side of the assignment `d := ...` as it does to evaluate expression `a+b` on the right hand side of the assignment `c := ...`. So evaluation of this function expression proceeds as follows,

(1) replace `a` in the expression `func< n | a+b+c+n >` by its value in the current context giving `func< n | 6+b+c+n >`.

(2) replace `b` in `func< n | 6+b+c+n >` by its value in the current context giving `func< n | 6+7+c+n >`.

(3) replace c in `func< n | 6+7+c+n >` by its value in the current context giving `FUNC(n : 6+7+13+n)`

(4) simplify the resultant *value* `FUNC(n : 6+7+13+n)` to the *value* `FUNC(n : 26+n)`.

Note again that the process starts with an expression and ends with a value, and that throughout the function expression is evaluated just like any other expression. A small technical point: function simplification may not in fact occur but the user is guaranteed that the simplification process will at least produce a function extensionally equal to the function in its canonical form.

The resultant function value is now assigned to d just like any other type of value would be assigned to an identifier yielding the context `[ (a,6),(b,7), (c,8), (d,FUNC(n : 26+n)) ]`.

As a final point note that changing the value of any of a, b, and c, does *not* change the value of d!

5.5.4 Function Values Assigned to Identifiers

Say we type the following,

```
> a := 1;
> b := func< n | a >;
> c := func< n | b(6) >;
```

The first line leaves a context of the form `[ (a,1) ]`. The second line leaves a context of the form `[ (a,1), (b,FUNC(n : 1)) ]`.

The third line is evaluated as follows,

(1) replace the value of `b` in the expression `func< n | b(6) >` by its value in the current context giving `FUNC(n : (FUNC(n : 1))(6))`.

(2) simplify this value to `FUNC(n : 1)` since applying the function value `FUNC(n : 1)` to the argument 6 always yields 1.

The key point here is that identifiers whose assigned value is a function value (in this case *b*), are treated exactly like identifiers whose assigned value is any other type of value.

Now look back at the example at the end of the previous section. One step in the series of replacements was not mentioned. Remember that `+` is treated as a shorthand for an infix function. So `a+b` is equivalent to `'+'(a,b)`. `+` is an identifier (assigned a function value), and so in the replacement part of the evaluation process there should have been an extra step, namely,

(4) replace `+` in `func< n : 6+7+13+n >` by its value in the current context giving `FUNC(n : A( A( A(6,7), 13 ), n ))`.

(5) simplify the resultant value to `FUNC(n : A( 26, n ))`. where `A` is the (function) value that is the addition function.

5.5.5 Recursion and Mutual Recursion

How do we write recursive functions? Function expressions have no names so how can a function expression apply *itself* to do recursion?

It is tempting to say that the function expression could recurse by using the identifier that the corresponding function value is to be assigned to. But the function value may not be being assigned at all: it may simply be being passed as an actual argument to some other function value. Moreover even if the function value were being assigned to an identifier the function expression cannot use that identifier because the assignment rules say that the identifiers on the left hand side of the `:=` in an assignment statement are not considered declared on the right hand side, unless they were previously declared.

The solution to the problem is to use the `$$` pseudo-identifier. `$$` is a placeholder for the function value denoted by the function expression inside which the `$$` occurs. An example serves to illustrate the use of `$$`. A recursive factorial function can be defined as follows,

```
> factorial := function(n)
>     if n eq 1 then
>         return 1;
```

```
>      else
>          return n * $$(n-1);
>      end if;
>  end function;
```

Here `$$` is a placeholder for the function value that the function expression `function(n) if n eq ... end function` denotes (those worried that the denoted function value appears to be defined in terms of itself are referred to the fixed point semantics of recursive functions in any standard text on denotational semantics).

A similar problem arises with mutual recursion where a function value f applies another function value g, and g likewise applies f. For example,

```
> f := function(...) ... a := g(...); ... end function;
> g := function(...) ... b := f(...); ... end function;
```

Again MAGMA's evaluation process appears to make this impossible, since to construct f MAGMA requires a value for g, but to construct g MAGMA requires a value for f. Again there is a solution. An identifier can be declared 'forward' to inform MAGMA that a function expression for the forward identifier will be supplied later. The functions f and g above can therefore be declared as follows,

```
> forward f, g;
> f := function(...) ... a := g(...); ... end function;
> g := function(...) ... b := f(...); ... end function;
```

(strictly speaking it is only necessary to declare g forward as the value of f will be known by the time the function expression `function(...)  ... b := f(...); ... end function` is evaluated).

5.5.6 Function Application

It was previously stated that MAGMA employs call by value evaluation, meaning that the arguments to a function are evaluated before the function is applied. This subsection discusses how functions are applied once their arguments have been evaluated.

Say we type,

```
> f := func< a, b | a+b >;
```

producing the context `[ (f,FUNC(a,b :  a+b)) ]`.

Now say we apply f by typing,

```
> r := f( 1+2, 6+7 ).
```

How is the value to be assigned to r calculated? If we follow the evaluation process we will reach the final step which will say something like,

> "simplify `(FUNC(a, b :  A(a,b)))(3,13)` to its canonical form"

where as before A is the value that is the addition function. How is this simplification performed? How are function values applied to actual function arguments to yield result

values? Not unsurprisingly the answer is via a process of substitution. The evaluation of a function application proceeds as follows,

(1) replace each formal argument in the function body by the corresponding actual argument.

(2) simplify the function body to its canonical form.

Exactly what it means to "simplify the function body ..." is intentionally left vague as the key point here is the process of replacing formal arguments by values in the body of the function.

5.5.7 The Initial Context

The only thing that remains to consider with the evaluation semantics, is how to get the ball rolling. Where do the initial values for things like the addition function come from? The answer is that when MAGMA starts up it does so with an initial context defined. This initial context has assignments of all the built-in MAGMA function values to the appropriate identifiers. The initial context contains for example the assignment of the addition function to the identifier `+`, the multiplication function to the identifier `*`, etc.

If, for example, we start MAGMA and immediately type,

```
> 1+2;
```

then in evaluating the expression `1+2` MAGMA will replace `+` by its value in the initial context.

Users interact with this initial context by typing statements at the top level (i.e., statements not inside any function or procedure). A user can change the initial context through re-assignment or expand it through new assignments.

5.6 Scope

Say we type the following,

```
> temp := 7;
> f := function(a,b)
>    temp := a * b;
>    return temp^2;
> end function;
```

If the evaluation process is now followed verbatim, the resultant context will look like `[ (temp,7), (f,FUNC(a,b :  7 := a*b; return 7^2;)) ]`, which is quite clearly not what was intended!

5.6.1 Local Declarations

What is needed in the previous example is some way of declaring that an identifier, in this case `temp`, is a 'new' identifier (i.e., distinct from other identifiers with the same name) whose use is confined to the enclosing function. MAGMA provides such a mechanism, called a local declaration. The previous example could be written,

```
> temp := 7;
> f := function(a,b)
>    local temp;
>    temp := a * b;
>    return temp^2;
> end function;
```

The identifier `temp` inside the body of f is said to be '(declared) local' to the enclosing function. Evaluation of these two assignments would result in the context being `[ (temp, 7), (f, FUNC(a,b : local temp := a*b; return local temp^2;)) ]` as intended.

It is very important to remember that `temp` and `local temp` are *distinct*! Hence if we now type,

```
> r := f(3,4);
```

the resultant context would be `[ (temp,7), (f,FUNC(a,b : local temp := a*b; return local temp^2;)), (r,144) ]`. The assignment to `local temp` inside the body of f does *not* change the value of `temp` outside the function. The effect of an assignment to a local identifier is thus localized to the enclosing function.

5.6.2 The 'first use' Rule

It can become tedious to have to declare all the local variables used in a function body. Hence MAGMA adopts a convention whereby an identifier can be implicitly declared according to how it is first used in a function body. The convention is that if the first use of an identifier inside a function body is on the left hand side of a `:=`, then the identifier is considered to be local, and the function body is considered to have an implicit local declaration for this identifier at its beginning. There is in fact no need therefore to declare `temp` as local in the previous example as the first use of `temp` is on the left hand side of a `:=` and hence `temp` is implicitly declared local.

It is very important to note that the term 'first use' refers to the first *textual* use of an identifier. Consider the following example,

```
> temp := 7;
> f := function(a,b)
>    if false then
>       temp := a * b;
>       return temp;
>    else
>       temp;
>       return 1;
```

```
>     end if;
>  end function;
```

The first *textual* use of `temp` in this function body is in the line

```
>  temp := a * b;
```

Hence `temp` is considered as a local inside the function body. It is not relevant that the `if false ...` condition will never be true and so the first time `temp` will be encountered when f is applied to some arguments is in the line

```
>  temp;
```

'First use' means 'first textual use', modulo the rule about examining the right hand side of a `:=` before the left!

5.6.3 Identifier Classes

It is now necessary to be more precise about the treatment of identifiers in MAGMA. Every identifier in a MAGMA program is considered to belong to one of three possible classes, these being:

(a) the class of value identifiers

(b) the class of variable identifiers

(c) the class of reference identifiers

The class an identifier belongs to indicates how the identifier is used in a program.

The class of value identifiers includes all identifiers that stand as placeholders for values, namely:

(a) all loop identifiers;

(b) the $$ pseudo-identifier;

(c) all identifiers whose first use in a function expression is as a value (i.e., not on the left hand side of an `:=`, nor as an actual reference argument to a procedure).

Because value identifiers stand as placeholders for values to be substituted during the evaluation process, they are effectively constants, and hence they cannot be assigned to. Assigning to a value identifier would be akin to writing something like `7 := 8;`!

The class of variable identifiers includes all those identifiers which are declared as local, either implicitly by the first use rule, or explicitly through a local declaration. Identifiers in this class may be assigned to.

The class of reference identifiers will be discussed later.

5.6.4 The Evaluation Process Revisited

The reason it is important to know the class of an identifier is that the class of an identifier effects how it is treated during the evaluation process. Previously it was stated that the evaluation process was,

(1) replace each identifier in the expression by its value in the current context.

(2) simplify the resultant *value* to its canonical form.

Strictly speaking the first step of this process should read,

(1′) replace each *free* identifier in the expression by its value in the current context, where an identifier is said to be free if it is a value identifier which is not a formal argument, a loop identifier, or the $$ identifier.

This definition of the replacement step ensures for example that while computing the value of a function expression F, MAGMA does not attempt to replace F's formal arguments with values from the current context!

5.6.5 The 'single use' Rule

As a final point on identifier classes it should be noted that an identifier may belong to only *one* class within an expression. Specifically therefore an identifier can only be used in one way inside a function body. Consider the following function,

```
> a := 7;
> f := function(n) a := a; return a; end function;
```

It is *not* the case that a is considered as a variable identifier on the left hand side of the `:=`, and as a value identifier on the right hand side of the `:=`. Rather a is considered to be a value identifier as its first use is as a value on the right hand side of the `:=` (remember that MAGMA inspects the right hand side of an assignment, and hence sees a first as a value identifier, *before* it inspects the left hand side where it sees a being used as a variable identifier).

5.7 Procedure Expressions

To date we have only discussed function expressions, these being a mechanism for computing new values from the values of identifiers in the current context. Together with assignment this provides us with a means of changing the current context – to compute a new value for an identifier in the current context, we call a function and then re-assign the identifier with the result of this function. That is we do

```
> X := f(Y);
```

where Y is a list of arguments possibly including the current value of X.

At times however using re-assignment to change the value associated with an identifier can be both un-natural and inefficient. Take the problem of computing some reduced form of a matrix. We could write a function that looked something like this,

```
reduce :=
   function( m )
      local lm;
      ...
      lm := m;
      while not reduced do
```

```
        ...
        lm := some_reduction(m);
        ...
    end while;
...
end function;
```

Note that the local `lm` is necessary since we cannot assign to the function's formal argument `m` since it stands for a value (and values cannot be assigned to). Note also that the function is inefficient in its space usage since at any given point in the program there are at least two different copies of the matrix (if the function was recursive then there would be more than two copies!).

Finally the function is also un-natural. It is perhaps more natural to think of writing a program that takes a given matrix and *changes* that matrix into its reduced form (i.e., the original matrix is lost). To accommodate for this style of programming, Magma includes a mechanism, the *procedure expression* with its *reference arguments*, for changing an association of an identifier and a value *in place*.

Before examining procedure expressions further, it is useful to look at a simple example of a procedure expression. Say we type:

```
> a := 5; b := 6;
```

giving the context `[ (a,5), (b,6) ]`. Say we now type the following:

```
> p := procedure( x, ~y ) y := x; end procedure;
```

This gives us a context that looks like `[ (a,5), (b,6), (p, PROC(x,~y :  y := x;)) ]`, using a notation analogous to the `FUNC` notation.

Say we now type the following *statement*,

```
> p(a, ~b);
```

This is known as a *call of the procedure* p (strictly it should be known as a call to the *procedure value* associated with the identifier p, since like functions, procedures in Magma are first class values!). Its effect is to *change* the current context to `[ (a,5), (b,5), (p, PROC(a,~b :  b := a;)) ]`. a and x are called *actual* and *formal value arguments* respectively since they are not prefixed by a $\sim$, while b and y are called *actual* and *formal reference arguments* respectively because they are prefixed by a $\sim$.

This example illustrates the defining attribute of procedures, namely that rather than returning a value, a procedure changes the context in which it is called. In this case the value of b was changed by the call to p. Observe however that *only* b was changed by the call to p as *only* b in the call, and its corresponding formal argument y in the definition, are reference arguments (i.e., prefixed with a $\sim$). A procedure may therefore only change that part of the context associated with its reference arguments! All other parts of the context are left unchanged. In this case a and p were left unchanged!

Note that apart from reference arguments (and the corresponding fact that that procedures do not return values), procedures are exactly like functions. In particular:

a) procedures are first class values that can be assigned to identifiers, passed as arguments, returned from functions, etc.

b) procedure expressions are evaluated in the same way that function expressions are.

c) procedure value arguments (both formal and actual) behave exactly like function arguments (both formal and actual). Thus procedure value arguments obey the standard substitution semantics.

d) procedures employ the same notion of scope as functions.

e) procedure calling behaves like function application.

f) procedures may be declared 'forward' to allow for (mutual) recursion.

g) a procedure may be assigned to an identifier in the initial context.

The remainder of this section will thus restrict itself to looking at reference arguments, the point of difference between procedures and functions.

5.8 Reference Arguments

If we look at a context it consists of a set of pairs, each pair being a name (an identifier) and a value (that is said to be assigned to that identifier).

When a function is applied actual arguments are substituted for formal arguments, and the body of the function is evaluated. The process of evaluating an actual argument yields a value and any associated names are ignored. Magma's evaluation semantics treats identifiers as 'indexes' into the context – when Magma wants the value of say x it searches through the context looking for a pair whose name component is x. The corresponding value component is then used as the value of x and the name part is simply ignored thereafter.

When we call a procedure with a reference argument, however, the name components of the context become important. When, for example we pass x as an actual reference argument to a formal reference argument y in some procedure, Magma remembers the name x. Then if y is changed (e.g., by assignment) in the called procedure, Magma, knowing the name x, finds the appropriate pair in the calling context and updates it by changing its corresponding value component. To see how this works take the example in the previous section. It was,

```
> a := 5; b := 6;
> p := procedure( x, ~y ) y := x; end procedure;
> p(a, ~b);
```

In the call Magma remembers the name b. Then when y is assigned to in the body of p, Magma knows that y is really b in the calling context, and hence changes b in the calling context appropriately. This example shows that an alternate way of thinking of reference arguments is as synonyms for the same part of (or pair in) the calling context.

5.9 Dynamic Typing

MAGMA is a dynamically typed language. In practice this means that:

(a) there is no need to declare the type of identifiers (this is especially important for identifiers assigned function values!).

(b) type violations are only checked for when the code containing the type violation is actually executed.

To make these ideas clearer consider the following two functions,

```
> f := func< a, b | a+b >;
> g := func< a, b | a+true >;
```

First note that there are no declarations of the types of any of the identifiers.

Second consider the use of + in the definition of function f. Which addition function is meant by the + in a+b? Integer addition? Matrix addition? Group addition? ... Or in other words what is the type of the identifier + in function f? Is it integer addition, matrix addition, etc.? The answer to this question is that + here denotes all possible addition function values (+ is said to denote a *family* of function values), and MAGMA will automatically chose the appropriate function value to apply when it knows the type of a and b.

Say we now type,

```
> f(1,2);
```

MAGMA now knows that a and b in f are both integers and thus + in f should be taken to mean the integer addition function. Hence it will produce the desired answer of 3.

Finally consider the definition of the function g. It is clear X+true for all X is a type error, so it might be expected that MAGMA would raise an error as soon as the definition of g is typed in. MAGMA does not however raise an error at this point. Rather it is only when g is applied and the line `return a + true` is actually executed that an error is raised.

In general the exact point at which type checking is done is not important. Sometimes however it is. Say we had typed the following definition for g,

```
> g := function(a,b)
>    if false then
>       return a+true;
>    else
>       return a+b;
>    end if;
> end function;
```

Now because the `if false` condition will never be true, the line `return a+true` will *never* be executed, and hence the type violation of adding a to true will *never* be raised!

One closing point: it should be clear now that where it was previously stated that the initial context "contains assignments of all the built-in MAGMA function values to the appropriate identifiers", in fact the initial context contains assignments of all the built-in MAGMA function *families* to the appropriate identifiers.

5.10 Traps for Young Players

This section describes the two most common sources of confusion encountered when using MAGMA's evaluation strategy.

5.10.1 Trap 1

We boot MAGMA. It begins with an initial context something like `[ ..., ('+',A), ('-',S), ... ]` where A is the (function) value that is the addition function, and S is the (function) value that is the subtraction function.

Now say we type,

```
> '+' := '-';
> 1 + 2;
```

MAGMA will respond with the answer `-1`.

To see why this is so consider the effect of each line on the current context. After the first line the current context will be `[ ..., ('+',S), ('-',S), ... ]`, where S is as before. The identifier `+` has been re-assigned. Its new value is the value of the identifier '-' in the current context, and the value of '-' is the (function) value that is the subtraction function. Hence in the second line when MAGMA replaces the identifier `+` with its value in the current context, the value that is substituted is therefore S, the subtraction function!

5.10.2 Trap 2

Say we type,

```
> f := func< n | n + 1 >;
> g := func< m | m + f(m) >;
```

After the first line the current context is `[ (f,FUNC( n :  n+1)) ]`. After the second line the current context is `[ (f,FUNC( n :  n+1)), (g,FUNC(m :  m + FUNC(n : n+1)(m))) ]`.

If we now type,

```
> g(6);
```

MAGMA will respond with the answer 13.

Now say we decide that our definition of f is wrong. So we now type in a new definition for f as follows,

```
> f := func< n | n + 2 >;
```

If we again type,

```
> g(6);
```

MAGMA will again reply with the answer 13!

To see why this is so consider how the current context changes. After typing in the initial definitions of f and g the current context is `[ (f, FUNC(n :  n+1)), (g, FUNC(m :  m + FUNC(n :  n+1)(m))) ]`. After typing in the second definition of f the current

context is `[ (f, FUNC(n : n+2)), (g, FUNC(m : m + FUNC(n : n+1)(m)))]`. Remember that changing the *value* of one identifier, in this case f, does *not* change the value of any other identifiers, in this case g! In order to change the value of g to reflect the new value of f, g would have to be re-assigned.

5.11 Appendix A: Precedence

The table below defines the relative precedence of operators in MAGMA, with decreasing strength (so operators higher in the table bind more strongly). The column on the right indicates whether the operator is left-, right-, or non-associative.

‘ ‘‘	*left*
(	*left*
[	*left*
assigned	*right*
~	*non*
#	*non*
&* &+ &and &cat &join &meet &or	*non-associative*
$ $$	*non*
.	*left*
@ @@	*left*
! !!	*right*
^	*right*
unary-	*right*
cat	*left*
* / div mod	*left*
+ -	*left*
meet	*left*
sdiff	*left*
diff	*left*
join	*left*
adj in notadj notin notsubset subset	*non*
cmpeq cmpne eq ge gt le lt ne	*left*
not	*right*
and	*left*
or xor	*left*
^^	*non*
? else select	*right*
->	*left*
=	*left*
:= is where	*left*

5.12 Appendix B: Reserved Words

The list below contains all reserved words in the MAGMA language; these cannot be used as identifier names.

_	elif	is	require
adj	else	join	requirege
and	end	le	requirerange
assert	eq	load	restore
assert2	error	local	return
assert3	eval	lt	save
assigned	exists	meet	sdiff
break	exit	mod	select
by	false	ne	subset
case	for	not	then
cat	forall	notadj	time
catch	forward	notin	to
clear	fprintf	notsubset	true
cmpeq	freeze	or	try
cmpne	function	print	until
continue	ge	printf	vprint
declare	gt	procedure	vprintf
default	if	quit	vtime
delete	iload	random	when
diff	import	read	where
div	in	readi	while
do	intrinsic	repeat	xor

6 THE MAGMA PROFILER

Chapter 6
THE MAGMA PROFILER

6.1 Introduction

One of the most important aspects of the development cycle is optimization. It is often the case that during the implementation of an algorithm, a programmer makes erroneous assumptions about its run-time behavior. These errors can lead to performance which differs in surprising ways from the expected output. The unfortunate tendency of programmers to optimize code before establishing run-time bottlenecks tends to exacerbate the problem.

Experienced programmers will thus often be heard repeating the famous mantra "Premature optimization is the root of all evil", coined by Sir Charles A. R. Hoare, the inventor of the Quick sort algorithm. Instead of optimizing during the initial implementation, it is generally better to perform an analysis of the run-time behaviour of the complete program, to determine what are the actual bottlenecks. In order to assist in this task, MAGMA provides a *profiler*, which gives the programmer a detailed breakdown of the time spent in a program. In this chapter, we provide an overview of how to use the profiler.

6.2 Profiler Basics

The MAGMA profiler records timing information for each function, procedure, map, and intrinsic call made by your program. When the profiler is switched on, upon the entry and exit to each such call the current system clock time is recorded. This information is then stored in a call graph, which can be viewed in various ways.

`SetProfile(b)`

Turns profiling on (if b is `true`) or off (if b is `false`). Profiling information is stored cumulatively, which means that in the middle of a profiling run, the profiler can be switched off during sections for which profiling information is not wanted. At startup, the profiler is off. Turning the profiler on will slow down the execution of your program slightly.

`ProfileReset()`

Clear out all information currently recorded by the profiler. It is generally a good idea to do this after the call graph has been obtained, so that future profiling runs in the same MAGMA session begin with a clean slate.

`ProfileGraph()`

Get the call graph based upon the information recorded up to this point by the profiler. This function will return an error if the profiler has not yet been turned on.

The call graph is a directed graph, with the nodes representing the functions that were called during the program's execution. There is an edge in the call graph from a function x to a function y if y was called during the execution of x. Thus, recursive calls will result in cycles in the call graph.

Each node in the graph has an associated label, which is a record with the following fields:

(i) `Name`: the name of the function

(ii) `Time`: the total time spent in the function

(iii) `Count`: the number of times the function was called

Each edge $\langle x, y \rangle$ in the graph also has an associated label, which is a record with the following fields:

(i) `Time`: the total time spent in function y when it was called from function x

(ii) `Count`: the total number of times function y was called by function x

Example H6E1

We illustrate the basic use of the profiler in the following example. The code we test is a simple implementation of the Fibonacci sequence; this can be replaced by any MAGMA code that needs to be profiled.

```
> function fibonacci(n)
>     if n eq 1 or n eq 2 then
>       return 1;
>     else
>       return fibonacci(n - 1) + fibonacci(n - 2);
>     end if;
> end function;
>
> SetProfile(true);
> time assert fibonacci(27) eq Fibonacci(27);
Time: 10.940
> SetProfile(false);
> G := ProfileGraph();
> G;
Digraph
Vertex  Neighbours
1       2 3 6 7 ;
2       2 3 4 5 ;
3       ;
4       ;
5       ;
```

```
6       ;
7       ;
> V := Vertices(G);
> Label(V!1);
rec<recformat<Name: Strings(), Time: RealField(), Count: IntegerRing()> |
    Name := <main>,
    Time := 10.93999999999999950262,
    Count := 1
    >
> Label(V!2);
rec<recformat<Name: Strings(), Time: RealField(), Count: IntegerRing()> |
    Name := fibonacci,
    Time := 10.93999999999999950262,
    Count := 392835
    >
> E := Edges(G);
> Label(E![1,2]);
rec<recformat<Time: RealField(), Count: IntegerRing()> |
    Time := 10.93999999999999950262,
    Count := 1
    >
```

6.3 Exploring the Call Graph

6.3.1 Internal Reports

The above example demonstrates that while the call graph contains some useful information, it does not afford a particularly usable interface. The MAGMA profiler contains some profile report generators which can be used to study the call graph in a more intuitive way.

The reports are all tabular, and have a similar set of columns:

(i) `Index`: The numeric identifier for the function in the vertex list of the call graph.

(ii) `Name`: The name of the function. The function name will be followed by an asterisk if a recursive call was made through it.

(iii) `Time`: The time spent in the function; depending on the report, the meaning might vary slightly.

(iv) `Count`: The number of times the function was called; depending on the report, the meaning might vary slightly.

`ProfilePrintByTotalCount(G)`

`Percentage`	BoolElt	*Default :* `false`
`Max`	RngIntElt	*Default :* -1

Print the list of functions in the call graph, sorted in descending order by the total number of times they were called. The `Time` and `Count` fields of the report give the total time and total number of times the function was called. If `Percentage` is true, then the `Time` and `Count` fields represent their values as percentages of the total value. If `Max` is non-negative, then the report only displays the first `Max` entries.

`ProfilePrintByTotalTime(G)`

`Percentage`	BoolElt	*Default :* `false`
`Max`	RngIntElt	*Default :* -1

Print the list of functions in the call graph, sorted in descending order by the total time spent in them. Apart from the sort order, this function's behaviour is identical to that of `ProfilePrintByTotalCount`.

`ProfilePrintChildrenByCount(G, n)`

`Percentage`	BoolElt	*Default :* `false`
`Max`	RngIntElt	*Default :* -1

Given a vertex n in the call graph G, print the list of functions called by the function n, sorted in descending order by the number of times they were called by n. The `Time` and `Count` fields of the report give the time spent during calls by the function n and the number of times the function was called by the function n. If `Percentage` is true, then the `Time` and `Count` fields represent their values as percentages of the total value. If `Max` is non-negative, then the report only displays the first `Max` entries.

`ProfilePrintChildrenByTime(G, n)`

`Percentage`	BoolElt	*Default :* `false`
`Max`	RngIntElt	*Default :* -1

Given a vertex n in the call graph G, print the list of functions in the called by the function n, sorted in descending order by the time spent during calls by the function n. Apart from the sort order, this function's behaviour is identical to that of `ProfilePrintChildrenByCount`.

Example H6E2

Continuing with the previous example, we examine the call graph using profile reports.

```
> ProfilePrintByTotalTime(G);
Index Name                                                    Time    Count
1     <main>                                                  10.940  1
2     fibonacci                                               10.940  392835
3     eq(<RngIntElt> x, <RngIntElt> y) -> BoolElt             1.210   710646
```

```
4     -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.630   392834
5     +(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.250   196417
6     Fibonacci(<RngIntElt> n) -> RngIntElt                   0.000   1
7     SetProfile(<BoolElt> v)                                 0.000   1
> ProfilePrintChildrenByTime(G, 2);
Function: fibonacci
Function Time: 10.940
Function Count: 392835
Index Name                                                    Time    Count
2     fibonacci (*)                                           182.430 392834
3     eq(<RngIntElt> x, <RngIntElt> y) -> BoolElt             1.210   710645
4     -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.630   392834
5     +(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.250   196417
* A recursive call is made through this child
```

6.3.2 HTML Reports

While the internal reports are useful for casual inspection of a profile run, for detailed examination a text-based interface has serious limitations. MAGMA's profiler also supports the generation of HTML reports of the profile run. The HTML report can be loaded up in any web browser. If Javascript is enabled, then the tables in the report can be dynamically sorted by any field, by clicking on the column heading you wish to perform a sort with. Clicking the column heading multiple times will alternate between ascending and descending sorts.

`ProfileHTMLOutput(G, prefix)`

Given a call graph G, an HTML report is generated using the file prefix $prefix$. The index file of the report will be "$prefix$.html", and exactly n additional files will be generated with the given filename $prefix$, where n is the number of functions in the call graph.

6.4 Recursion and the Profiler

Recursive calls can cause some difficulty with profiler results. The profiler takes care to ensure that double-counting does not occur, but this can lead to unintuitive results, as the following example shows.

Example H6E3

In the following example, `recursive` is a recursive function which simply stays in a loop for half a second, and then recurses if not in the base case. Thus, the total running time should be approximately $(n+1)/2$ seconds, where n is the parameter to the function.

```
> procedure delay(s)
>     t := Cputime();
>     repeat
>       _ := 1+1;
>     until Cputime(t) gt s;
> end procedure;
>
> procedure recursive(n)
>     if n ne 0 then
>       recursive(n - 1);
>     end if;
>
>     delay(0.5);
> end procedure;
>
> SetProfile(true);
> recursive(1);
> SetProfile(false);
> G := ProfileGraph();
```

Printing the profile results by total time yield no surprises:

```
> ProfilePrintByTotalTime(G);
Index Name                                                    Time    Count
1     <main>                                                  1.020   1
2     recursive                                               1.020   2
5     delay                                                   1.020   2
8     Cputime(<FldReElt> T) -> FldReElt                       0.130   14880
7     +(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.020   14880
9     gt(<FldReElt> x, <FldReElt> y) -> BoolElt               0.020   14880
3     ne(<RngIntElt> x, <RngIntElt> y) -> BoolElt             0.000   2
4     -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.000   1
6     Cputime() -> FldReElt                                   0.000   2
10    SetProfile(<BoolElt> v)                                 0.000   1
```

However, printing the children of `recursive`, and displaying the results in percentages, does yield a surprise:

```
> ProfilePrintChildrenByTime(G, 2 : Percentage);
Function: recursive
Function Time: 1.020
Function Count: 2
Index Name                                                    Time    Count
5     delay                                                   100.00  33.33
2     recursive (*)                                           50.00   16.67
```

```
3     ne(<RngIntElt> x, <RngIntElt> y) -> BoolElt              0.00    33.33
4     -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt             0.00    16.67
* A recursive call is made through this child
```

At first glance, this doesn't appear to make sense, as the sum of the time column is 150%! The reason for this behavior is because some time is "double counted": the total time for the first call to `recursive` includes the time for the recursive call, which is also counted separately. In more detail:

```
> V := Vertices(G);
> E := Edges(G);
> Label(V!1)`Name;
<main>
> Label(V!2)`Name;
recursive
> Label(E![1,2])`Time;
1.01999999999999795718
> Label(E![2,2])`Time;
0.51000000000000000888
> Label(V!2)`Time;
1.01999999999999795718
```

As can seen in the above, the total time for `recursive` is approximately one second, as expected. The double-counting of the recursive call can be seen in the values of `Time` for the edges `[1,2]` and `[2,2]`.

7 DEBUGGING MAGMA CODE

Chapter 7
DEBUGGING MAGMA CODE

7.1 Introduction

In ordered to facilitate the debugging of complex pieces of MAGMA code, MAGMA includes a debugger. *This debugger is very much a prototype, and can cause* MAGMA *to crash.*

`SetDebugOnError(f)`

If f is `true`, then upon an error MAGMA will break into the debugger. The usage of the debugger is described in the next section.

7.2 Using the Debugger

When use of the debugger is enabled and an error occurs, MAGMA will break into the command-line debugger. The syntax of the debugger is modelled on the GNU GDB debugger for C programs, and supports the following commands (acceptable abbreviations for the commands are given in parentheses):

`backtrace (bt)` Print out the stack of function and procedure calls, from the top level to the point at which the error occurred. Each line i this trace gives a single *frame*, which consists of the function/procedure that was called, as well as all local variable definitions for that function. Each frame is numbered so that it can be referenced in other debugger commands.

`frame (f)` n Change the current frame to the frame numbered n (the list of frames can be obtained using the `backtrace` command). The current frame is used by other debugger commands, such as `print`, to determine the context within which expressions should be evaluated. The default current frame is the top-most frame.

`list (l)` [n] Print a source code listing for the current context (the context is set by the `frame` command). If n is specified, then the `list` command will print n lines of source code; the default value is 10.

`print (p)` *expr* Evaluate the expression *expr* in the current context (the context is set by the `frame` command). The `print` command has semantics identical to evaluating the expression `eval "expr"` at the current point in the program.

`help (h)` Print brief help on usage.

`quit (q)` Quit the debugger and return to the MAGMA session.

Example H7E1

We now give a sample session in the debugger. In the following, we have written a function to evaluate $f(n) = \Sigma_{i=1}^{n} 1/n$, but have in our implementation we have accidentally included the evaluation of the term at $n = 0$.

```
> function f(n)
>   if n ge 0 then
>      return 1.0 / n + f(n - 1);
>   else
>      return 1.0 / n;
>   end if;
> end function;
>
> SetDebugOnError(true);
> f(3);
f(
    n: 3
)
f(
    n: 2
)
f(
    n: 1
)
f(
    n: 0
)
>>      return 1.0 / n + f(n - 1);
                     ^
Runtime error in '/': Division by zero
debug> p n
0
debug> p 1.0 / (n + 1)
1.00000000000000000000000000000
debug> bt
#0 *f(
    n: 0
) at <main>:1
#1  f(
    n: 1
) at <main>:1
#2  f(
    n: 2
) at <main>:1
#3  f(
    n: 3
) at <main>:1
debug> f 1
```

```
debug> p n
1
debug> p 1.0 / n
1.00000000000000000000000000000000
```

PART II
SETS, SEQUENCES, AND MAPPINGS

8 INTRODUCTION TO AGGREGATES

Chapter 8
INTRODUCTION TO AGGREGATES

8.1 Introduction

This part of the Handbook comprises four chapters on aggregate objects in MAGMA as well as a chapter on maps.

Sets, sequences, tuples and lists are the four main types of aggregates, and each has its own specific purpose. *Sets* are used to collect objects that are elements of some common structure, and the most important operation is to test element membership. *Sequences* also contain objects of a common structure, but here the emphasis is on the ordering of the objects, and the most important operation is that of accessing (or modifying) elements at given positions. Sets will contain at most one copy of any element, whereas sequences may contain arbitrarily many copies of the same object. *Enumerated* sets and sequences are of arbitrary but finite length and will store all elements explicitly (with the exception of arithmetic progressions), while *formal* sets and sequences may be infinite, and use a Boolean function to test element membership. *Indexed* sets are a hybrid form of sets allowing indexing like sequences. Elements of *Cartesian products* of structures in MAGMA will be called *tuples*; they are of fixed length, and each coefficient must be in the corresponding structure of the defining Cartesian product. *Lists* are arbitrary finite ordered collections of objects of any type, and are mainly provided to the user to store assorted data to which access is not critical.

8.2 Restrictions on Sets and Sequences

Here we will explain the subtleties behind the mechanism dealing with sets and sequences and their universes and parents. Although the same principles apply to their formal counterparts, we will only talk about enumerated sets and sequences here, for two reasons: the enumerated versions are much more useful and common, and the very restricted number of operations on formal sets/sequences make issues of universe and overstructure of less importance for them.

In principle, every object e in MAGMA has some parent structure S such that $e \in S$; this structure can be used for type checking (are we allowed to apply function f to e?), algorithm look-up etc. To avoid storing the structure with every element of a set or sequence and having to look up the structure of every element separately, only elements of a *common structure* are allowed in sets or sequences, and that common parent will only be stored once.

8.2.1 Universe of a Set or Sequence

This common structure is called the *universe* of the set or sequence. In the general constructors it may be specified up front to make clear what the universe for the set or sequence will be; the difference between the sets i and s in

```
> i := { IntegerRing() | 1, 2, 3 };
> s := { RationalField() | 1, 2, 3 };
```

lies entirely in their universes. The specification of the universe may be omitted if there is an obvious common overstructure for the elements. Thus the following provides a shorter way to create the set containing 1, 2, 3 and having the ring of integers as universe:

```
> i := { 1, 2, 3 };
```

Only empty sets and sequences that have been obtained directly from the constructions

```
> S := { };
> T := [ ];
```

do not have their universe defined – we will call them the *null* set or sequence. (There are two other ways in which empty sets and sequences arise: it is possible to create empty sequences with a prescribed universe, using

```
> S := { U |  };
> T := [ U | ];
```

and it may happen that a non-empty set/sequence becomes empty in the course of a computation. In both cases these empty objects have their universe defined and will not be *null*).

Usually (but not always: the exception will be explained below) the universe of a set or sequence is the parent for all its elements; thus the ring of integers is the parent of 2 in the set $i = \{1, 2, 3\}$, rather than that set itself. The universe is not static, and it is not necessarily the same structure as the parent of the elements *before* they were put in the set or sequence. To illustrate this point, suppose that we try to create a set containing integers and rational numbers, say $T = \{1, 2, 1/3\}$; then we run into trouble with the rule that the universe must be common for all elements in T; the way this problem is solved in MAGMA is by automatic coercion: the obvious universe for T is the field of rational numbers of which $1/3$ is already an element and into which any integer can be coerced in an obvious way. Hence the assignment

```
> T := { 1, 2, 1/3 }
```

will result in a set with universe the field of rationals (which is also present when MAGMA is started up). Consequently, when we take the element 1 of the set T, it will have the rational field as its parent rather than the integer ring! It will now be clear that

```
> s := { 1/1, 2, 3 };
```

is a shorter way to specify the set of rational numbers 1,2, 3 than the way we saw before, but in general it is preferable to declare the universe beforehand using the `{ U | }` notation.

Of course

```
> T := { Integers() | 1, 2, 1/3 }
```

would result in an error because $1/3$ cannot be coerced into the ring of integers.

So, usually not every element of a given structure can be coerced into another structure, and even if it can, it will not always be done automatically. The possible (automatic) coercions are listed in the descriptions of the various MAGMA modules. For instance, the table in the introductory chapter on rings shows that integers can be coerced automatically into the rational field.

In general, every MAGMA structure is valid as a universe. This includes enumerated sets and sequences themselves, that is, it is possible to define a set or sequence whose elements are confined to be elements of a given set or sequence. So, for example,

```
> S := [ [ 1..10 ] | x^2+x+1 : x in { -3 .. 2 by 1 } ];
```

produces the sequence $[7, 3, 1, 1, 3, 7]$ of values of the polynomial $x^2 + x + 1$ for $x \in \mathbf{Z}$ with $-3 \leq x \leq 2$. However, an entry of S will in fact have the ring of integers as its parent (and *not* the sequence [1..10]), because the effect of the above assignment is that the values after the | are calculated and coerced into the universe, which is [1..10]; but coercing an element into a sequence or set means that it will in fact be coerced into the *universe* of that sequence/set, in this case the integers. So the main difference between the above assignment and

```
> T := [ Integers() | x^2+x+1 : x in { -3 .. 2 by 1} ];
```

is that in the first case it is checked that the resulting values y satisfy $1 \leq y \leq 10$, and an error would occur if this is violated:

```
> S := [ [ 1..10 ] | x^2+x+1 : x in { -3 .. 3 by 1} ];
```

leads to a run-time error.

In general then, the parent of an element of a set or sequence will be the universe of the set or sequence, unless that universe is itself a set or sequence, in which case the parent will be the universe of this universe, and so on, until a non-set or sequence is encountered.

8.2.2 Modifying the Universe of a Set or Sequence

Once a (non-null) set or sequence S has been created, the universe has been defined. If one attempts to *modify* S (that is, to add elements, change entries etc. using a procedure that will not reassign the result to a new set or sequence), the universe will not be changed, and the modification will only be successful if the new element can be coerced into the current universe. Thus,

```
> Z := Integers();
> T := [ Z | 1, 2, 3/3 ];
> T[2] := 3/4;
```

will result in an error, because $3/4$ cannot be coerced into Z.

The universe of a set or sequence S can be explicitly modified by creating a *parent* for S with the desired universe and using the `!` operator for the coercion; as we will see in the next subsection, such a parent can be created using the `PowerSet` and `PowerSequence` commands. Thus, for example, the set $\{1, 2\}$ can be made into a sequence of rationals as follows:

```
> I := { 1, 2 };
> P := PowerSet( RationalField() );
> J := P ! I;
```

The coercion will be successful if every element of the sequence can be coerced into the new universe, and it is *not* necessary that the old universe could be coerced completely into the new one: the set $\{3/3\}$ of rationals can be coerced into `PowerSet(Integers())`. As a consequence, the empty set (or sequence) with any universe can be coerced into the power set (power sequence) of any other universe.

Binary functions on sets or sequences (like `join` or `cat`) can only applied to sets and sequences that are *compatible*: the operation on S with universe A and T with universe B can only be performed if a common universe C can be found such that the elements of S and T are all elements of C. The compatibility conditions are dependent on the particular MAGMA module to which A and B belong (we refer to the corresponding chapters of this manual for further information) and do also apply to elements of $a \in A$ and $b \in B$ — that is, the compatibility conditions for S and T are the same as the ones that determine whether binary operations on $a \in A$ and $b \in B$ are allowed. For example, we are able to join a set of integers and a set of rationals:

```
> T := { 1, 2 } join { 1/3 };
```

for the same reason that we can do

```
> c := 1 + 1/3;
```

(automatic coercion for rings). The resulting set T will have the rationals as universe.
The basic rules for compatibility of two sets or sequences are then:

(1) every set/sequence is compatible with the null set/sequence (which has no universe defined (see above));

(2) two sets/sequences with the same universe are compatible;

(3) a set/sequence S with universe A is compatible with set/sequence T with universe B if the elements of A can be automatically coerced into B, or vice versa;

(4) more generally, a set/sequence S with universe A is also compatible with set/sequence T with universe B if MAGMA can automatically find an *over-structure* for the parents A and B (see below);

(5) nested sets and sequences are compatible only when they are of the same 'depth' and 'type' (that is, sets and sequences appear in exactly the same recursive order in both) and the universes are compatible.

The possibility of finding an overstructure C for the universe A and B of sets or sequences S and T (such that $A \subset C \supset B$), is again module-dependent. We refer the reader for

details to the Introductions of Parts III–VI, and we give some examples here; the next subsection contains the rules for parents of sets and sequences.

8.2.3 Parents of Sets and Sequences

The universe of a set or sequence S is the common parent for all its elements; but S itself is a MAGMA object as well, so it should have a parent too.

The parent of a set is a *power set*: the set of all subsets of the universe of S. It can be created using the `PowerSet` function. Similarly, `PowerSequence(A)` creates the parent structure for a sequence of elements from the structure A – that is, the elements of `PowerSequence(A)` are all sequences of elements of A.

The rules for finding a common overstructure for structures A and B, where either A or B is a set/sequence or the parent of a set/sequence, are as follows. (If neither A nor B is a set, sequence, or its parent we refer to the Part of this manual describing the operations on A and B.)

(1) The overstructure of A and B is the same as that of B and A.

(2) If A is the null set or sequence (empty, and no universe specified) the overstructure of A and B is B.

(3) If A is a set or sequence with universe U, the overstructure of A and B is the overstructure of U and B; in particular, the overstructure of A and A will be the universe U of A.

(4) If A is the parent of a set (a power set), then A and B can only have a common overstructure if B is also the parent of a set, in which case the overstructure is the power set of the overstructure of the universes U and V of A and B respectively. Likewise for sequences instead of sets.

We give two examples to illustrate rules (3) and (4). It is possible to create a set with a set as its universe:

```
> S := { { 1..100 } | x^3 : x in [ 0..3 ] };
```

If we wish to intersect this set with some set of integers, say the formal set of odd integers

```
> T := {! x : x in Integers() | IsOdd(x) !};
> W := S meet T;
```

then we can only do that if we can find a universe for W, which must be the common overstructure of the universe $U = \{1, 2, \ldots, 100\}$ of S and the universe 'ring of integers' of T. By rule (3) above, this overstructure of $U = \{1, 2, \ldots, 100\}$ will be the overstructure of the universe of U and the ring of integers; but the universe of U is the ring of integers (because it is the default for the set $\{1, 2, \ldots, 100\}$), and hence the overstructure we are looking for (and the universe for W) will be the ring of integers.

For the second example we look at sequences of sequences:

```
> a := [ [ 1 ], [ 1, 2, 3 ] ];
```

```
> b := [ [ 2/3 ] ];
```

so a is a sequence of sequences of integers, and b is a sequence of sequences of rationals. If we wish to concatenate a and b,

```
> c := a cat b;
```

we will only succeed if we find a universe for c. This universe must be the common overstructure of the universes of a and b, which are the 'power sequence of the integers' and the 'power sequence of the rationals' respectively. By rule (4), the overstructure of these two power sequences is the power sequence of the common overstructure of the rationals and the integers, which is the rationals themselves. Hence c will be a sequence of sequences of rationals, and the elements of a will have to be coerced.

8.3 Nested Aggregates

Enumerated sets and sequences can be arbitrarily nested (that is, one may create sets of sets, as well as sequences of sets etc.); tuples can also be nested and may be freely mixed with sets and sequences (as long as the proper Cartesian product parent can be created). Lists can be nested, and one may create lists of sets or sequences or tuples.

8.3.1 Multi-indexing

Since sequences (and lists) can be nested, assignment functions and mutation operators allow you to use *multi-indexing*, that is, one can use a multi-index $i_1, i_2, \ldots, i_r$ rather than a single i to reach r levels deep. Thus, for example, if $S = [\ [1,2],\ [2,3]\ ]$, instead of

```
> S[2][2] := 4;
```

one may use the multi-index $2, 2$ to obtain the same effect of changing the 3 into a 4:

```
> S[2,2] := 4;
```

All i_j in the multi-index $i_1, i_2, \ldots, i_r$ have to be greater than 0, and an error will also be flagged if any i_j indexes beyond the length at level j, that is, if $i_j > \#S[i_1, \ldots, i_{j-1}]$, (which means $i_1 > \#S$ for $j = 1$). There is one exception: the last index i_r is allowed to index beyond the current length of the sequence at level r if the multi-index is used on the left-hand side of an assignment, in which case any intermediate terms will be undefined. This generalizes the possibility to assign beyond the length of a 'flat' sequence. In the above example the following assignments are allowed:

```
> S[2,5] := 7;
```

(and the result will be $S = [\ [1,2],\ [2,3,\text{undef},\text{undef},7]\ ]$)

```
> S[4] := [7];
```

(and the result will be $S = [\ [1,2],\ [2,3],\ \text{undef},\ [7]\ ]$). But the following results in an error:

```
> S[4,1] := 7;
```

Finally we point out that multi-indexing should not be confused with the use of sequences as

indexes to create subsequences. For example, to create a subsequence of $S = [5, 13, 17, 29]$ consisting of the second and third terms, one may use

```
> S := [ 5, 13, 17, 29 ];
> T := S[ [2, 3] ];
```

To obtain the second term of this subsequence one could have done:

```
> x := S[ [2, 3] ][2];
```

(so x now has the value $S[3] = 17$), but it would have been more efficient to index the indexing sequence, since it is rather expensive to build the subsequence [$S[2], S[3]$] first, so:

```
> x :=  S[ [2, 3][2] ];
```

has the same effect but is better (of course `x := S[3]` would be even better in this simple example.) To add to the confusion, it is possible to mix the above constructions for indexing, since one can use lists of sequences and indices for indexing; continuing our example, there is now a third way to do the same as above, using an indexing list that first takes out the subsequence consisting of the second and third terms and then extracts the second term of that:

```
> x :=  S[ [2, 3], 2 ];
```

Similarly, the construction

```
> X :=  S[ [2, 3], [2] ];
```

pulls out the subsequence consisting of the second term of the subsequence of terms two and three of S, in other words, this assigns the *sequence* consisting of the element 17, not just the element itself!

9 SETS

Chapter 9
SETS

9.1 Introduction

A *set* in MAGMA is a (usually unordered) collection of objects belonging to some common structure (called the *universe* of the set). There are four basic types of sets: *enumerated sets*, whose elements are all stored explicitly (with one exception, see below); *formal sets*, whose elements are stored implicitly by means of a predicate that allows for testing membership; *indexed sets*, which are restricted enumerated sets having a numbering on elements; and *multisets*, which are enumerated sets with possible repetition of elements. In particular, enumerated and indexed sets and multisets are always finite, and formal sets are allowed to be infinite.

9.1.1 Enumerated Sets

Enumerated sets are finite, and can be specified in three basic ways (see also section 2 below): by listing all elements; by an expression involving elements of some finite structure; and by an arithmetic progression. If an arithmetic progression is specified, the elements are not calculated explicitly until a modification of the set necessitates it; in all other cases all elements of the enumerated set are stored explicitly.

9.1.2 Formal Sets

A formal set consists of the subset of elements of some carrier set (structure) on which a certain predicate assumes the value 'true'.

The only set-theoretic operations that can be performed on formal sets are union, intersection, difference and symmetric difference, and element membership testing.

9.1.3 Indexed Sets

For some purposes it is useful to be able to access elements of a set through an index map, which numbers the elements of the set. For that purpose MAGMA has indexed sets, on which a very few basic set operations are allowed (element membership testing) as well as some sequence-like operations (such as accessing the i-th term, getting the index of an element, appending and pruning).

9.1.4 Multisets

For some purposes it is useful to construct a set with some of its members repeated. For that purpose MAGMA has multisets, which take into account the repetition of members. The number of times an object x occurs in a multiset S is called the *multiplicity* of x in S. MAGMA has the `^^` operator to specify a multiplicity: the expression `x^^n` means the object x with multiplicity n. In the following, whenever any multiset constructor or function expects an element y, the expression x`^^`n may usually be used.

9.1.5 Compatibility

The binary operators for sets do not allow mixing of the four types of sets (so one cannot take the intersection of an enumerated set and a formal set, for example), but it is easy to convert an enumerated set into a formal set – see the section on binary operators below – and there are functions provided for making an enumerated set out of an indexed set or a multiset (and vice versa).

By the limitation on their construction formal sets can only contain elements from one structure in MAGMA. The elements of enumerated sets are also restricted, in the sense that either some universe must be specified upon creation, or MAGMA must be able to find such universe automatically. The rules for compatibility of elements and the way MAGMA deals with these universes are the same for sequences and sets, and are described in the previous chapter. The restrictions on indexed sets are the same as those for enumerated sets.

9.1.6 Notation

Certain expressions appearing in the sections below (possibly with subscripts) have a standard interpretation:

U the universe: any MAGMA structure;

E the carrier set for enumerated sets: any enumerated structure (it must be possible to loop over its elements – see the Introduction to this Part (Chapter 8));

F the carrier set for formal sets: any structure for which membership testing using `in` is defined – see the Introduction to this Part (Chapter 8));

x a free variable which successively takes the elements of E (or F in the formal case) as its values;

P a Boolean expression that usually involves the variable(s) x, $x_1, \ldots, x_k$;

e an expression that also usually involves the variable(s) x, $x_1, \ldots, x_k$.

9.2 Creating Sets

The customary braces { and } are used to define enumerated sets. Formal sets are delimited by the composite braces {`!` and `!`}. For indexed sets {`@` and `@`} are used. For multisets {`*` and `*`} are used.

9.2.1 The Formal Set Constructor

The formal set constructor has the following fixed format (the expressions appearing in the construct are defined above):

```
{!  x in F | P(x) !}
```

Form the formal set consisting of the subset of elements x of F for which $P(x)$ is true. If $P(x)$ is true for every element of F, the set constructor may be abbreviated to {`!  x in F !`}. Note that the universe of a formal set will always be equal to the carrier set F.

9.2.2 The Enumerated Set Constructor

Enumerated sets can be constructed by expressions enclosed in braces, provided that the values of all expressions can be automatically coerced into some common structure, as outlined in the Introduction, (Chapter 8). All general constructors have an optional universe (U in the list below) up front, that allows the user to specify into which structure all terms of the sets should be coerced.

`{ }`

The null set: an empty set that does not have its universe defined.

`{ U | }`

The empty set with universe U.

`{ e1, e2, ..., en }`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the set $\{ a_1, a_2, ..., a_n \}$ of elements of U.

Example H9E1

We create a set by listing its elements explicitly.

```
> S := { (7^2+1)/5, (8^2+1)/5, (9^2-1)/5 };
> S;
{ 10, 13, 16 }
> Parent(S);
Set of subsets of Rational Field
```

Thus S was created as a set of rationals, because `/` on integers has a rational result. If one wishes to obtain a set of integers, one could specify the universe (or one could use `div`, or one could use `!` on every element to coerce it into the ring of integers):

```
> T := { Integers() | (7^2+1)/5, (8^2+1)/5, (9^2-1)/5 };
> T;
{ 10, 13, 16 }
> Parent(T);
Set of subsets of Integer Ring
```

`{ U | e1, e2, ..., en }`

Given a list of expressions $e_1, \ldots, e_n$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the set $\{a_1, a_2, ..., a_n \}$ of elements of U.

```
{ e(x) :  x in E | P(x) }
```

Form the set of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8) (in particular, E must be a finite structure that can be enumerated).

If $P(x)$ is true for every value of x in E, then the set constructor may be abbreviated to { e(x) : x in E }.

```
{ U | e(x) :  x in E | P(x) }
```

Form the set of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as before.

If P is always true, it may be omitted (including the |).

```
{ e(x1,...,xk) :  x1 in E1, ..., xk in Ek | P(x1, ..., xk) }
```

The set consisting of those elements $e(x_1, \ldots, x_k)$, in some common structure, for which $x_1, \ldots, x_k$ in $E_1, \ldots, E_k$ have the property that $P(x_1, \ldots, x_k)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8).

Note that if two successive allowable structures E_i and E_{i+1} are identical, then the specification of the carrier sets for x_i and x_{i+1} may be abbreviated to `xi, xi+1 in Ei`.

Also, if $P(x_1, ..., x_k)$ is always true, it may be omitted (including the |).

```
{ U | e(x1,...,xk) :  x1 in E1, ..., xk in Ek | P(x1, ..., xk) }
```

As in the previous entry, the set consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true, is formed, as a set of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ are elements of or coercible into U).

Again, identical successive structures may be abbreviated, and a predicate that is always true may be omitted.

Example H9E2

Now that Fermat's last theorem may have been proven, it may be of interest to find integers that almost satisfy $x^n + y^n = z^n$. In this example we find all $2 < x, y, z < 1000$ such that $x^3 + y^3 = z^3 + 1$. First we build a set of cubes, then two sets of pairs for which the sum of cubes differs from a cube by 1. Note that we build a *set* rather than a sequence of cubes because we only need fast membership testing. Also note that the resulting sets of pairs do not have their elements in the order in which they were found.

```
> cubes := { Integers() | x^3 : x in [1..1000] };
> plus := { <a, b> : a in [2..1000], b in [2..1000] | \
>    b ge a and (a^3+b^3-1) in cubes };
> plus;
{
```

```
        < 9, 10 >,
        < 135, 235 >
        < 334, 438 >,
        < 73, 144 >,
        < 64, 94 >,
        < 244, 729 >
}
```

Note that we spend a lot of time cubing integers this way. For a more efficient approach, see a subsequent example.

9.2.3 The Indexed Set Constructor

The creation of indexed sets is similar to that of enumerated sets.

`{@ @}`

The null set: an empty indexed set that does not have its universe defined.

`{@ U | @}`

The empty indexed set with universe U.

`{@` e_1`,` e_2`, ...,` e_n `@}`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the indexed set $Q = \{\ a_1, a_2, ..., a_n\ \}$ of elements of U.

`{@ U |` e_1`,` e_2`, ...,` e_m `@}`

Given a list of expressions $e_1, \ldots, e_m$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the indexed set $Q = \{a_1, a_2, ..., a_n\ \}$ of elements of U.

`{@ e(x) : x in E | P(x) @}`

Form the indexed set of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8) (in particular, E must be a finite structure that can be enumerated).

If P is always true, it may be omitted (including the |).

`{@ U | e(x) : x in E | P(x) @}`

Form the indexed set of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as before.

If P is always true, it may be omitted (including the |).

{@ e(x$_1$,...,x$_k$) : x$_1$ in E$_1$, ..., x$_k$ in E$_k$ | P(x$_1$, ..., x$_k$) @}

The indexed set consisting of those elements $e(x_1, \ldots, x_k)$ (in some common structure), for which $x_1, \ldots, x_k$ in $E_1 \times \ldots \times E_k$ have the property that $P(x_1, \ldots, x_k)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8).

Note that if two successive allowable structures E_i and E_{i+1} are identical, then the specification of the carrier sets for x_i and x_{i+1} may be abbreviated to x$_i$, x$_{i+1}$ in E$_i$.

Also, if $P(x_1, ..., x_k)$ is always true, it may be omitted.

{@ U | e(x$_1$,...,x$_k$) : x$_1$ in E$_1$, ..., x$_k$ in E$_k$ | P(x$_1$, ..., x$_k$)@}

As in the previous entry, the indexed set consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true is formed, as an indexed set of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ are elements of or coercible into U).

Again, identical successive structures may be abbreviated, and a predicate that is always true may be omitted.

Example H9E3

In the previous example we found pairs x, y such that $x^3 + y^3$ differs by one from some cube z^3. Using indexed sets it is somewhat easier to retrieve the integer z as well. We give a small example. Note also that it is beneficial to know here that evaluation of expressions proceeds left to right.

```
> cubes := { @ Integers() | z^3 : z in [1..25] @};
> plus := { <x, y, z> : x in [-10..10], y in [-10..10], z in [1..25] |
>    y ge x and Abs(x) gt 1 and Abs(y) gt 1 and (x^3+y^3-1) in cubes
>    and (x^3+y^3-1) eq cubes[z] };
> plus;
{ <-6, 9, 8>, <9, 10, 12>, <-8, 9, 6> }
```

9.2.4 The Multiset Constructor

The creation of multisets is similar to that of enumerated sets. An important difference is that repetitions are significant and the operator ^^ (mentioned above) may be used to specify the multiplicity of an element.

{* *}

The null set: an empty multiset that does not have its universe defined.

{* U | *}

The empty multiset with universe U.

`{* e1, e2, ..., en *}`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the multiset $Q = \{* \ a_1, a_2, ..., a_n \ *\}$ of elements of U.

`{* U | e1, e2, ..., em *}`

Given a list of expressions $e_1, \ldots, e_m$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the multiset $Q = \{* \ a_1, a_2, ..., a_n \ *\}$ of elements of U.

`{* e(x) : x in E | P(x) *}`

Form the multiset of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8) (in particular, E must be a finite structure that can be enumerated).

If P is always true, it may be omitted (including the |).

`{* U | e(x) : x in E | P(x) *}`

Form the multiset of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as before.

If P is always true, it may be omitted (including the |).

`{* e(x1,...,xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) *}`

The multiset consisting of those elements $e(x_1, \ldots, x_k)$ (in some common structure), for which $x_1, \ldots, x_k$ in $E_1 \times \ldots \times E_k$ have the property that $P(x_1, \ldots, x_k)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8).

Note that if two successive allowable structures E_i and E_{i+1} are identical, then the specification of the carrier sets for x_i and x_{i+1} may be abbreviated to `xi, xi+1 in Ei`.

Also, if $P(x_1, ..., x_k)$ is always true, it may be omitted.

`{* U | e(x1,...,xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk)*}`

As in the previous entry, the multiset consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true is formed, as an multiset of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ are elements of or coercible into U).

Again, identical successive structures may be abbreviated, and a predicate that is always true may be omitted.

Example H9E4

Here we demonstrate the use of the multiset constructors.

```
> M := {* 1, 1, 1, 3, 5 *};
> M;
{* 1^^3, 3, 5 *}
> M := {* 1^^4, 2^^5, 1/2^^3 *};
> M;
> // Count frequency of digits in first 1000 digits of pi:
> pi := Pi(RealField(1001));
> dec1000 := Round(10^1000*(pi-3));
> I := IntegerToString(dec1000);
> F := {* I[i]: i in [1 .. #I] *};
> F;
{* 7^^95, 3^^102, 6^^94, 2^^103, 9^^106, 5^^97,
1^^116, 8^^101, 4^^93, 0^^93 *}
> for i := 0 to 9 do i, Multiplicity(F, IntegerToString(i)); end for;
0 93
1 116
2 103
3 102
4 93
5 97
6 94
7 95
8 101
9 106
```

9.2.5 The Arithmetic Progression Constructors

Some special constructors exist to create and store enumerated sets of integers in arithmetic progression efficiently. This only works for arithmetic progressions of elements of the ring of integers.

`{ i..j }`

`{ U | i..j }`

The enumerated set whose elements form the arithmetic progression $i, i+1, i+2, \ldots, j$, where i and j are (expressions defining) integers. If j is less than i then the empty set will be created.

The only universe U that is legal here is the ring of integers.

`{ i .. j by k }`

`{ U | i .. j by k }`

The enumerated set consisting of the integers forming the arithmetic progression $i, i+k, i+2*k, \ldots, j$, where i, j and k are (expressions defining) integers (but $k \neq 0$).

If k is positive then the last element in the progression will be the greatest integer of the form $i + n * k$ that is less than or equal to j. If j is less than i, the empty set will be constructed.

If k is negative then the last element in the progression will be the least integer of the form $i + n * k$ that is greater than or equal to j. If j is greater than i, the empty set will be constructed.

As for the previous constructor, only the ring of integers is allowed as a legal universe U.

Example H9E5

It is possible to use the arithmetic progression constructors to save typing in the creation of 'arithmetic progressions' of elements of other structures than the ring of integers, but it should be kept in mind that the result will not be treated especially efficiently like the integer case. Here is the 'wrong' way, as well as two correct ways to create a set of 10 finite field elements.

```
>  S := { FiniteField(13) | 1..10 };
Runtime error in { .. }: Invalid set universe
> S := { FiniteField(13) | x : x in { 1..10 } };
> S;
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
> G := PowerSet(FiniteField(13));
> S := G ! { 1..10 };
> S;
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

9.3 Power Sets

The `PowerSet` constructor returns a structure comprising the subsets of a given structure R; it is mainly useful as a parent for other set and sequence constructors. The only operations that are allowed on power sets are printing, testing element membership, and coercion into the power set (see the examples below).

`PowerSet(R)`

The structure comprising all enumerated subsets of structure R.

`PowerIndexedSet(R)`

The structure comprising all indexed subsets of structure R.

`PowerMultiset(R)`

The structure consisting of all submultisets of the structure R.

`S in P`

Returns `true` if enumerated set S is in the power set P, that is, if all elements of the set S are contained in or coercible into R, where P is the power set of R; `false` otherwise.

`PowerFormalSet(R)`

The structure comprising all formal subsets of structure R.

`S in P`

Returns `true` if indexed set S is in the power set P, that is, if all elements of the set S are contained in or coercible into R, where P is the power set of R; `false` otherwise.

`S in P`

Returns `true` if multiset S is in the power set P, that is, if all elements of the set S are contained in or coercible into R, where P is the power set of R; `false` otherwise.

`P ! S`

Return a set with universe R consisting of the elements of the set S, where P is the power set of R. An error results if not all elements of S can be coerced into R.

`P ! S`

Return an indexed set with universe R consisting of the elements of the set S, where P is the power set of R. An error results if not all elements of S can be coerced into R.

`P ! S`

Return an multiset with universe R consisting of the elements of the set S, where P is the power set of R. An error results if not all elements of S can be coerced into R.

Example H9E6

```
> S := { 1 .. 10 };
> P := PowerSet(S);
> P;
Set of subsets of { 1 .. 10 }
> F := { 6/3, 12/4 };
> F in P;
true
> G := P ! F;
> Parent(F);
Set of subsets of Rational Field
> Parent(G);
Set of subsets of { 1 .. 10 }
```

9.3.1 The Cartesian Product Constructors

Using `car< >` and `CartesianProduct( )`, it is possible to create the Cartesian product of sets (or, in fact, of any combination of structures), but the result will be of type 'Cartesian product' rather than set, and the elements are tuples – we refer the reader to Chapter 11 for details.

9.4 Sets from Structures

`Set(M)`

Given a finite structure that allows explicit enumeration of its elements, return the set containing its elements (having M as its universe).

`FormalSet(M)`

Given a structure M, return the formal set consisting of its elements.

9.5 Accessing and Modifying Sets

Enumerated sets can be modified by inserting or removing elements. Indexed sets allow some sequence-like operators for modification and access.

9.5.1 Accessing Sets and their Associated Structures

`#R`

Cardinality of the enumerated, indexed, or multi- set R. Note that for a multiset, repetitions are significant, so the result may be greater than the underlying set.

`Category(S)`

`Type(S)`

The category of the object S. For a set this will be one of `SetEnum`, `SetIndx`, `SetMulti`, or `SetFormal`. For a power set the type is one of `PowSetEnum`, `PowSetIndx`, `PowSetMulti`.

`Parent(R)`

Returns the parent structure of R, that is, the structure consisting of all (enumerated) sequences over the universe of R.

`Universe(R)`

Returns the 'universe' of the (enumerated or indexed or multi- or formal) set R, that is, the common structure to which all elements of the set belong. An error is signalled when R is the null set.

`Index(S, x)`

`Position(S, x)`

Given an indexed set S, and an element x, returns the index i such that $S[i] = x$ if such index exists, or return 0 if x is not in S. If x is not in the universe of S, an attempt will be made to coerce it; an error occurs if this fails.

`S[i]`

Return the i-th entry of indexed set S. If $i < 1$ or $i > \#S$ an error occurs. Note that indexing is *not* allowed on the left hand side.

`S[I]`

The indexed set $\{S[i_1], \ldots, S[i_r]\}$ consisting of terms selected from the indexed set S, according to the terms of the integer sequence I. If any term of I lies outside the range 1 to $\#S$, then an error results. If I is the empty sequence, then the empty set with universe the same as that of S is returned.

Example H9E7

We build an indexed set of sets to illustrate the use of the above functions.

```
> B := { @ { i : i in [1..k] } :  k in [1..5] @};
> B;
{ @
   { 1 },
   { 1, 2 },
   { 1, 2, 3 },
   { 1, 2, 3, 4 },
   { 1, 2, 3, 4, 5 },
@}
> #B;
5
> Universe(B);
Set of subsets of Integer Ring
> Parent(B);
Set of indexed subsets of Set of subsets of Integer Ring
> Category(B);
SetIndx
> Index(B, { 2, 1});
2
> #B[2];
2
> Universe(B[2]);
Integer Ring
```

9.5.2 Selecting Elements of Sets

Most finite structures in MAGMA, including enumerated sets, allow one to obtain a random element using `Random`. There is an alternative (and often preferable) option for enumerated sets in the `random{ }` constructor. This makes it possible to choose a random element of the set without generating the whole set first.

Likewise, `rep{ }` is an alternative to the general `Rep` function returning a representative element of a structure, having the advantage of aborting the construction of the set as soon as one element has been found.

Here, E will again be an enumerable structure, that is, a structure that allows enumeration of its elements (see the Appendix for an exhaustive list).

Note that `random{ e(x) :  x in E | P(x)}` does *not* return a random element of the set of values $e(x)$, but rather a value of $e(x)$ for a random x in E which satisfies P (and mutatis mutandis for `rep`).

See the subsection on Notation in the Introduction (Chapter 8) for conventions regarding e, x, E, P.

`Random(R)`

A random element chosen from the enumerated, indexed or multi- set R. Every element has an equal probability of being chosen for enumerated or indexed sets, and a weighted probability in proportion to its multiplicity for multisets. Successive invocations of the function will result in independently chosen elements being returned as the value of the function. If R is empty an error occurs.

`random{ e(x) : x in E | P(x) }`

Given an enumerated structure E and a Boolean expression P, return the value of the expression $e(y)$ for a randomly chosen element y of E for which $P(y)$ is true.

P may be omitted if it is always true.

`random{e(`x_1`, ..., `x_k`) : `x_1` in E`$_1$`, ..., `x_k` in E`$_k$` | P(`x_1`, ..., `x_k`)}`

Given enumerated structures $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, return the value of the expression $e(y_1, \cdots, y_k)$ for a randomly chosen element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, for which $P(y_1, \ldots, y_k)$ is true.

P may be omitted if it is always true.

If successive structures E_i and E_{i+1} are identical, then the abbreviation `x`$_i$`, x`$_{i+1}$ `in E`$_i$ may be used.

Example H9E8

Here are two ways to find a 'random' primitive element for a finite field.

```
> p := 10007;
> F := FiniteField(p);
> proots := {  z : z in F | IsPrimitive(z) };
> #proots;
5002
> Random(proots);
5279
```

This way, a set of 5002 elements is built (and primitivity is checked for all elements of F), and a random choice is made. Alternatively, we use `random`.

```
> random{ x : x in F | IsPrimitive(x) };
4263
```

In this case random elements in F are chosen until one is found that is primitive. Since almost half of F's elements are primitive, only very few primitivity tests will be done before success occurs.

`Representative(R)`

`Rep(R)`

An arbitrary element chosen from the enumerated, indexed, or multi- set R.

`ExtractRep(~R, ~r)`

Assigns an arbitrary element chosen from the enumerated set R to r, and removes it from R. Thus the set R is modified, as well as the element r. An error occurs if R is empty.

`rep{ e(x) : x in E | P(x) }`

Given an enumerated structure E and a Boolean expression P, return the value of the expression `e(y)` for the first element y of E for which $P(y)$ is true. If $P(x)$ is false for every element of E, an error will occur.

`rep{ e(x1, ..., xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) }`

Given enumerated structures $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, return the value of the expression $e(y_1, \cdots, y_k)$ for the first element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, for which $P(y_1, \ldots, y_k)$ is true. An error occurs if no element of $E_1 \times \cdots \times E_k$ satisfies P.

P may be omitted if it is always true.

If successive structures E_i and E_{i+1} are identical, then the abbreviation `xi, xi+1 in Ei` may be used.

Example H9E9

As an illustration of the use of `ExtractRep`, we modify an earlier example, and find cubes satisfying $x^3 + y^3 = z^3 - 1$ (with $x, y, z \leq 1000$).

```
> cubes := { Integers() | x^3 : x in [1..1000] };
> cc := cubes;
> min := { };
> while not IsEmpty(cc) do
>    ExtractRep(~cc, ~a);
>    for b in cc do
>       if a+b+1 in cubes then
>          min join:= { <a, b> };
>       end if;
>    end for;
> end while;
> { < Iroot(x[1], 3), Iroot(x[2], 3) > : x in min };
{ <138, 135>, <823, 566>, <426, 372>, <242, 720>,
      <138, 71>, <426, 486>, <6, 8> }
```

Note that instead of taking cubes over again, we only have to take cube roots in the last line (on the small resulting set) once.

`Minimum(S)`
`Min(S)`

Given a non-empty enumerated, indexed, or multi- set S, such that `lt` and `eq` are defined on the universe of S, this function returns the minimum of the elements of S. If S is an indexed set, the position of the minimum is also returned.

`Maximum(S)`
`Max(S)`

Given a non-empty enumerated, indexed, or multi- set S, such that `lt` and `eq` are defined on the universe of S, this function returns the maximum of the elements of S. If S is an indexed set, the position of the maximum is also returned.

`Hash(x)`

Given a Magma object x which can be placed in a set, return the hash value of x used by the set machinery. This is a fixed but arbitrary non-negative integer (whose maximum value is the maximum value of a C unsigned long on the particular machine). The crucial property is that if x and y are objects and x equals y then the hash values of x and y are equal (even if x and y have different internal structures). Thus one could implement sets manually if desired by the use of this function.

9.5.3 Modifying Sets

`Include(~S, x)`
`Include(S, x)`

Create the enumerated, indexed, or multi- set obtained by putting the element x in S (S is unchanged if S is not a multiset and x is already in S). If S is an indexed set, the element will be appended at the end. If S is a multiset, the multiplicity of x will be increased accordingly. If x is not in the universe of S, an attempt will be made to coerce it; an error occurs if this fails.

There are two versions of this: a procedure, where S is replaced by the new set, and a function, which returns the new set. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the set S will not be copied.

`Exclude(~S, x)`
`Exclude(S, x)`

Create a new set by removing the element x from S. If S is an enumerated set, nothing happens if x is not in S. If S is a multiset, the multiplicity of x will be decreased accordingly. If x is not in the universe of S, an attempt will be made to coerce it; an error occurs if this fails.

There are two versions of this: a procedure, where S is replaced by the new set, and a function, which returns the new set. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the set S will not be copied.

`ChangeUniverse(~S, V)`

`ChangeUniverse(S, V)`

Given an enumerated, indexed, or multi- set S with universe U and a structure V which contains U, construct a new set of the same type which consists of the elements of S coerced into V.

There are two versions of this: a procedure, where S is replaced by the new set, and a function, which returns the new set. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the set S will not be copied.

`CanChangeUniverse(S, V)`

Given an enumerated, indexed, or multi- set S with universe U and a structure V which contains U, attempt to construct a new set T of the same type which consists of the elements of S coerced into V; if successful, return `true` and T, otherwise return `false`.

Example H9E10

This example uses `Include` and `Exclude` to find a set (if it exists) of cubes of integers such that the elements of a given set R can be expressed as the sum of two of those.

```
> R := { 218, 271, 511 };
> x := 0;
> cubes := { 0 };
> while not IsEmpty(R) do
>    x +:= 1;
>    c := x^3;
>    Include(~cubes, c);
>    Include(~cubes, -c);
>    for z in cubes do
>        Exclude(~R, z+c);
>        Exclude(~R, z-c);
>    end for;
> end while;
```

We did not record how the elements of R were obtained as sums of a pair of cubes. For that, the following suffices.

```
> R := { 218, 271, 511 }; // it has been emptied !
> { { x, y } : x, y in cubes | x+y in R };
```

```
{
    { -729, 1000 },
    { -125, 343 },
    { -1, 512 },
}
```

`SetToIndexedSet(E)`

Given an enumerated set E, this function returns an indexed set with the same elements (and universe) as E.

`IndexedSetToSet(S)`

`Isetset(S)`

Given an indexed set S, this function returns an enumerated set with the same elements (and universe) as E.

`IndexedSetToSequence(S)`

`Isetseq(S)`

Given an indexed set S, this function returns a sequence with the same elements (and universe) as E.

`MultisetToSet(S)`

Given a multiset S, this function returns an enumerated set with the same elements (and universe) as S.

`SetToMultiset(E)`

Given an enumerated set E, this function returns a multiset with the same elements (and universe) as E.

`SequenceToMultiset(Q)`

Given an enumerated sequence E, this function returns a multiset with the same elements (and universe) as E.

9.6 Operations on Sets

9.6.1 Boolean Functions and Operators

As explained in the Introduction (Chapter 8), when elements are taken out of a set their parent will be the universe of the set (or, if the universe is itself a set, the universe of the universe, etc.); in particular, the set itself is not the parent. Hence equality testing on set elements is in fact equality testing between two elements of certain algebraic structures, and the sets are irrelevant. We only list the (in)equality operator for convenience here.

Element membership testing is of critical importance for all types of sets.

Testing whether or not R is a subset of S can be done if R is an enumerated or indexed set and S is any set; hence (in)equality testing is only possible between sets that are not formal sets.

`IsNull(R)`

Returns `true` if and only if the enumerated, indexed, or multi- set R is empty and does not have its universe defined.

`IsEmpty(R)`

Returns `true` if and only if the enumerated, indexed or multi- set R is empty.

`x eq y`

Given an element x of a set R with universe U and an element y of a set S with universe V, where a common overstructure W can be found with $U \subset W \supset V$ (see the Introduction (Chapter 8) for details on overstructures), return `true` if and only if x and y are equal as elements of W.

`x ne y`

Given an element x of a set R with universe U and an element y of a set S with universe V, where a common overstructure W can be found with $U \subset W \supset V$ (see the Introduction (Chapter 8) for details on overstructures), return `true` if and only if x and y are distinct as elements of W.

`x in R`

Returns `true` if and only if the element x is a member of the set R. If x is not an element of the universe U of R, it is attempted to coerce x into U; if this fails, an error occurs.

`x notin R`

Returns `true` if and only if the element x is not a member of the set R. If x is not an element of the parent structure U of R, it is attempted to coerce x into U; if this fails, an error occurs.

`R subset S`

Returns `true` if the enumerated, indexed or multi- set R is a subset of the set S, `false` otherwise. For multisets, if an element x of R has multiplicity n in R, the multiplicity of x in S must be at least n. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

`R notsubset S`

Returns `true` if the enumerated, indexed, or multi- set R is a not a subset of the set S, `false` otherwise. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

`R eq S`

Returns `true` if and only if R and S are identical sets, where R and S are enumerated, indexed or multi- sets For indexed sets, the index function is irrelevant for deciding equality. For multisets, matching multiplicities must also be equal. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

`R ne S`

Returns `true` if and only if R and S are distinct sets, where R and S are enumerated indexed, or multi- sets. For indexed sets, the index function is irrelevant for deciding equality. For multisets, matching multiplicities must also be equal. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

`IsDisjoint(R, S)`

Returns `true` iff the enumerated, indexed or multi- sets R and S are disjoint. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

9.6.2 Binary Set Operators

For each of the following operators, R and S are sets of the same type. If R and S are both formal sets, then an error will occur unless both have been constructed with the same carrier structure F in the definition. If R and S are both enumerated, indexed, or multi-sets, then an error occurs unless the universes of R and S are compatible, as defined in the Introduction to this Part (Chapter 8).
Note that

`Q := { ! x in R !}`

converts an enumerated set R into a formal set Q.

`R join S`

Union of the sets R and S (see above for the restrictions on R and S). For multisets, matching multiplicities are added in the union.

`R meet S`

Intersection of the sets R and S (see above for the restrictions on R and S). For multisets, the minimum of matching multiplicities is stored in the intersection.

`R diff S`

Difference of the sets R and S. i.e., the set consisting of those elements of R which are not members of S (see above for the restrictions on R and S). For multisets, the difference contains any elements of R remaining after removing the corresponding elements of S the appropriate number of times.

`R sdiff S`

Symmetric difference of the sets R and S. i.e., the set consisting of those elements which are members of either R or S but not both (see above for the restrictions on R and S). Alternatively, it is the union of the difference of R with S and the difference of S with R.

Example H9E11

```
> R := { 1, 2, 3 };
> S := { 1, 1/2, 1/3 };
> R join S;
{ 1/3, 1/2, 1, 2, 3 }
> R meet S;
{ 1 }
> R diff S;
{ 2, 3 }
> S diff R;
{ 1/3, 1/2 }
> R sdiff S;
{ 1/3, 1/2, 2, 3 }
```

9.6.3 Other Set Operations

`Multiplicity(S, x)`

Return the multiplicity in multiset S of element x. If x is not in S, zero is returned.

`Multiplicities(S)`

Returns the sequence of multiplicities of distinct elements in the multiset S. The order is the same as the internal enumeration order of the elements.

`Subsets(S)`

The set of all subsets of S.

`Subsets(S, k)`

The set of subsets of S of size k. If k is larger than the cardinality of S then the result will be empty.

`RandomSubset(S, k)`

A random subset of S of size k. It is an error if k is larger than the size of S.

`Multisets(S, k)`

The set of multisets consisting of k not necessarily distinct elements of S.

`Subsequences(S, k)`

The set of sequences of length k with elements from S.

`Permutations(S)`

The set of permutations (stored as sequences) of the elements of S.

`Permutations(S, k)`

The set of permutations (stored as sequences) of each of the subsets of S of cardinality k.

9.7 Quantifiers

To test whether some enumerated set is empty or not, one may use the `IsEmpty` function. However, to use `IsEmpty`, the set has to be created in full first. The existential quantifier `exists` enables one to do the test and abort the construction of the set as soon as an element is found; moreover, the element found will be assigned to a variable.

Likewise, `forall` enables one to abort the construction of the set as soon as an element not satisfying a certain property is encountered.

Note that `exists(t){ e(x) : x in E | P(x) }` is *not* designed to return true if an element of the set of values $e(x)$ satisfies P, but rather if there is an $x \in E$ satisfying $P(x)$ (in which case $e(x)$ is assigned to t).

For the notation used here, see the beginning of this chapter.

`exists(t){ e(x): x in E | P(x) }`

`exists(t1, ..., tr){ e(x) : x in E | P(x) }`

Given an enumerated structure E and a Boolean expression $P(x)$, the Boolean value true is returned if E contains at least one element x for which $P(x)$ is true. If $P(x)$ is not true for any element x of E, then the Boolean value false is returned.

Moreover, if $P(x)$ is found to be true for the element y, say, of E, then in the first form of the exists expression, variable t will be assigned the value of the expression $e(y)$. If $P(x)$ is never true for an element of E, t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y)$ should be a tuple of length r; each variable will then be assigned to the corresponding component of

the tuple. Similarly, all the variables will be left unassigned if $P(x)$ is never true. The clause `(t)` may be omitted entirely.

P may be omitted if it is always true.

`exists(t){e(`x_1`, ..., `x_k`): `x_1` in `E_1`, ..., `x_k` in `E_k` | P(`x_1`, ..., `x_k`)}`

`exists(`t_1`, ..., `t_r`){ e(`x_1`, ..., `x_k`) : `x_1` in `E_1`, ..., `x_k` in `E_k` | P }`

Given enumerated structures $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, the Boolean value true is returned if there is an element $< y_1, \ldots, y_k >$ in the Cartesian product $E_1 \times \cdots \times E_k$, such that $P(y_1, \ldots, y_k)$ is true. If $P(x_1, \ldots, x_k)$ is not true for any element $(y_1, \ldots, y_k)$ of $E_1 \times \cdots \times E_k$, then the Boolean value false is returned.

Moreover, if $P(x_1, \ldots, x_k)$ is found to be true for the element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, then in the first form of the exists expression, the variable t will be assigned the value of the expression $e(y_1, \cdots, y_k)$. If $P(x_1, \ldots, x_k)$ is never true for an element of $E_1 \times \cdots \times E_k$, then the variable t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y_1, \cdots, y_k)$ should be a tuple of length r; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x_1, \ldots, x_k)$ is never true. The clause `(t)` may be omitted entirely.

P may be omitted if it is always true.

If successive structures E_i and E_{i+1} are identical, then the abbreviation `x`$_i$`, x`$_{i+1}$ `in E`$_i$ may be used.

Example H9E12

As a variation on an earlier example, we check whether or not some integers can be written as sums of cubes (less than 10^3 in absolute value):

```
> exists(t){ <x, y> : x, y in [ t^3 : t in [-10..10] ] | x + y eq 218 };
true
> t;
<-125, 343>
> exists(t){ <x, y> : x, y in [ t^3 : t in [1..10] ] | x + y eq 218 };
false
>  t;
>> t;
          ^
User error: Identifier 't' has not been declared
```

`forall(t){ e(x) : x in E | P(x) }`

`forall(t1, ..., tr){ e(x) : x in E | P(x) }`

Given an enumerated structure E and a Boolean expression $P(x)$, the Boolean value true is returned if $P(x)$ is true for every element x of E.

If $P(x)$ is not true for at least one element x of E, then the Boolean value false is returned.

Moreover, if $P(x)$ is found to be false for the element y, say, of E, then in the first form of the exists expression, variable t will be assigned the value of the expression $e(y)$. If $P(x)$ is true for every element of E, t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y)$ should be a tuple of length r; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x)$ is always true. The clause `(t)` may be omitted entirely.

P may be omitted if it is always true.

`forall(t){e(x1, ..., xk): x1 in E1, ..., xk in Ek | P(x1, ..., xk)}`

`forall(t1, ..., tr){ e(x1, ..., xk) : x1 in E1, ..., xk in Ek | P }`

Given sets $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, the Boolean value true is returned if $P(x_1, \ldots, x_k)$ is true for every element $(x_1, \ldots, x_k)$ in the Cartesian product $E_1 \times \cdots \times E_k$.

If $P(x_1, \ldots, x_k)$ fails to be true for some element $(y_1, \ldots, y_k)$ of $E_1 \times \cdots \times E_k$, then the Boolean value false is returned.

Moreover, if $P(x_1, \ldots, x_k)$ is false for the element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, then in the first form of the exists expression, the variable t will be assigned the value of the expression $e(y_1, \cdots, y_k)$. If $P(x_1, \ldots, x_k)$ is true for every element of $E_1 \times \cdots \times E_k$, then the variable t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y_1, \cdots, y_k)$ should be a tuple of length r; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x_1, \ldots, x_k)$ is never true. The clause `(t)` may be omitted entirely.

P may be omitted if it is always true.

If successive structures E_i and E_{i+1} are identical, then the abbreviation `xi, xi+1 in Ei` may be used.

Example H9E13

This example shows that `forall` and `exists` may be nested.
It is well known that every prime that is 1 modulo 4 can be written as the sum of two squares, but not every integer m congruent to 1 modulo 4 can. In this example we explore for small m whether perhaps $m \pm \epsilon$ (with $|\epsilon| \leq 1$) is always a sum of squares.

```
> forall(u){ m : m in [5..1000 by 4] |
>       exists{ <x, y, z> : x, y in [0..30], z in [-1, 0, 1] |
>          x^2+y^2+z eq m } };
```

```
false
> u;
77
```

9.8 Reduction and Iteration over Sets

Both enumerated and indexed sets allow enumeration of their elements; formal sets do not. For indexed sets the enumeration will occur according to the order given by the indexing.

Instead of using a loop to apply the same binary associative operator to all elements of an enumerated or indexed set, it is in certain cases possible to use the *reduction operator* `&`.

`x in S`

Enumerate the elements of an enumerated or indexed set S. This can be used in *loops*, as well as in the set and sequence *constructors*.

`&o S`

Given an enumerated or indexed set $S = \{ a_1, a_2, \ldots, a_n\}$ of elements belonging to an algebraic structure U, and an (associative) operator $\circ : U \times U \rightarrow U$, form the element $a_{i_1} \circ a_{i_2} \circ a_{i_3} \circ \ldots \circ a_{i_n}$, for some permutation $i_1, \ldots, i_n$ of $1, \ldots, n$.

Currently, the following operators may be used to reduce enumerated sets: `+, *, and, or, join, meet` and `+, *, and, or` to reduce indexed sets. An error will occur if the operator is not defined on U.

If S contains a single element a, then the value returned is a. If S is the null set (empty and no universe specified) or S is empty with universe U (and the operation is defined in U), then the result (or error) depends on the operation and upon U. The following table defines the return value:

	empty	*null*
&+	U ! 0	error
&∗	U ! 1	error
&and	`true`	`true`
&or	`false`	`false`
&join	*empty*	*null*
&meet	error	error

Warning: since the reduction may take place in an arbitrary order on the arguments $a_1, \ldots, a_n$, the result is not unambiguously defined if the operation is not commutative on the arguments!

Example H9E14

The function `choose` defined below takes a set S and an integer k as input, and produces a set of all subsets of S with cardinality k.

```
> function choose(S, k)
>    if k eq 0 then
>       return { { } };
>    else
>       return &join{{  s join { x} : s in choose(S diff { x}, k-1) } : x in S};
>    end if;
> end function;
```

So, for example:

```
>  S := { 1, 2, 3, 4 };
> choose(S, 2);
{
       { 1, 3 },
       { 1, 4 },
       { 2, 4 },
       { 2, 3 },
       { 1, 2 },
       { 3, 4 }
}
```

Try to guess what happens if $k < 0$.

10 SEQUENCES

Chapter 10
SEQUENCES

10.1 Introduction

A *sequence* in MAGMA is a linearly ordered collection of objects belonging to some common structure (called the *universe* of the sequence).

There are two types of sequence: *enumerated sequences*, of which the elements are all stored explicitly (with one exception, see below); and *formal sequences*, of which elements are stored implicitly by means of a predicate that allows for testing membership. In particular, enumerated sequences are always finite, and formal sequences are allowed to be infinite. In this chapter a *sequence* will be either a formal or an enumerated sequence.

10.1.1 Enumerated Sequences

An *enumerated sequence of length* l is an array of indefinite length of which only finitely many terms – including the l-th term, but no term of bigger index — have been defined to be elements of some common structure. Such sequence is called *complete* if all of the terms (from index 1 up to the length l) are defined.

In practice the length of an enumerated sequence must be less than 2^{30}.

Incomplete enumerated sequences are allowed as a convenience for the programmer in building complete enumerated sequences. Some sequence functions require their arguments to be complete; if that is the case, it is mentioned explicitly in the description below. However, all functions using sequences in *other* MAGMA modules always assume that a sequence that is passed in as an argument is complete. Note that the following line converts a possibly incomplete sequence S into a complete sequence T:

```
T := [ s :  s in S ];
```

because the enumeration using the `in` operator simply ignores undefined terms.

Enumerated sequences of *Booleans* are highly optimized (stored as bit-vectors).

10.1.2 Formal Sequences

A formal sequence consists of elements of some range set on which a certain predicate assumes the value 'true'.

There is only a very limited number of operations that can be performed on them.

10.1.3 Compatibility

The binary operators for sequences do not allow mixing of the formal and enumerated sequence types (so one cannot take the concatenation of an enumerated sequence and a formal sequence, for example); but it is easy to convert an enumerated sequence into a formal sequence – see the section on binary operators below.

By the limitation on their construction formal sequences can only contain elements from one structure in MAGMA. The elements of enumerated sequences are also restricted, in the sense that either some common structure must be specified upon creation, or MAGMA must be able to find such universe automatically. The rules for compatibility of elements and the way MAGMA deals with these parents is the same for sequences and sets, and is outlined in Chapter 8.

10.2 Creating Sequences

Square brackets are used for the definition of enumerated sequences; formal sequences are delimited by the composite brackets `[!` and `!]`.

Certain expressions appearing below (possibly with subscripts) have the standard interpretation:

U the universe: any MAGMA structure;

E the range set for enumerated sequences: any enumerated structure (it must be possible to loop over its elements – see the Introduction to this Part);

F the range set for formal sequences: any structure for which membership testing using `in` is defined – see the Introduction to this Part);

x a free variable which successively takes the elements of E (or F in the formal case) as its values;

P a Boolean expression that usually involves the variable(s) x, $x_1, \ldots, x_k$;

e an expression that also usually involves the variable(s) x, $x_1, \ldots, x_k$.

10.2.1 The Formal Sequence Constructor

The formal sequence constructor has the following fixed format (the expressions appearing in the construct are defined above):

```
[!  x in F | P(x) !]
```

Create the formal sequence consisting of the subsequence of elements x of F for which $P(x)$ is true. If $P(x)$ is true for every element of F, the sequence constructor may be abbreviated to `[!  x in F !]`

10.2.2 The Enumerated Sequence Constructor

Sequences can be constructed by expressions enclosed in square brackets, provided that the values of all expressions can be automatically coerced into some common structure, as outlined in the Introduction. All general constructors have the universe U optionally up front, which allows the user to specify into which structure all terms of the sequences should be coerced.

`[ ]`

The null sequence (empty, and no universe specified).

`[ U | ]`

The empty sequence with universe U.

`[ e`$_1$`, e`$_2$`, ..., e`$_n$ `]`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the sequence $Q = [a_1, a_2, ..., a_n\ ]$ of elements of U.

As for multisets, one may use the expression `x^^n` to specify the object x with multiplicity n: this is simply interpreted to mean x repeated n times (i.e., no internal compaction of the repetition is done).

`[ U | e`$_1$`, e`$_2$`, ..., e`$_m$ `]`

Given a list of expressions $e_1, \ldots, e_m$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the sequence $Q = [a_1, a_2, ..., a_n\ ]$ of elements of U.

`[ e(x) :  x in E | P(x) ]`

Form the sequence of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given at the beginning of this section.

If $P(x)$ is true for every element of E, the sequence constructor may be abbreviated to `[ e(x) :  x in E ]` .

`[ U | e(x) :  x in E | P(x) ]`

Form the sequence of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as above.

`[ e(x`$_1$`,...,x`$_k$`) :  x`$_1$ `in E`$_1$`, ..., x`$_k$ `in E`$_k$ `| P(x`$_1$`, ..., x`$_k$`) ]`

The sequence consisting of those elements $e(x_1, \ldots, x_k)$, in some common structure, for which $x_1, \ldots, x_k$ in $E_1, \ldots, E_k$ have the property that $P(x_1, \ldots, x_k)$ is true.

The expressions appearing in this construct have the interpretation given at the beginning of this section.

Note that if two successive ranges E_i and E_{i+1} are identical, then the specification of the ranges for x_i and x_{i+1} may be abbreviated to `x`$_i$`, x`$_{i+1}$ `in E`$_i$.

Also, if $P(x_1, ..., x_k)$ is always true, it may be omitted.

```
[ U | e(x1,...,xk) :  x1 in E1, ..., xk in Ek | P(x1, ..., xk) ]
```

As in the previous entry, the sequence consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true is formed, as a sequence of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ are coercible into U).

10.2.3 The Arithmetic Progression Constructors

Since enumerated sequences of integers arise so often, there are a few special constructors to create and handle them efficiently in case the entries are in arithmetic progression. The universe must be the ring of integers. Some effort is made to preserve the special way of storing arithmetic progressions under sequence operations.

```
[ i..j ]
```

```
[ U | i..j ]
```

The enumerated sequence of integers whose elements form the arithmetic progression $i, i+1, i+2, \ldots, j$, where i and j are (expressions defining) arbitrary integers. If j is less than i then the empty sequence of integers will be created.

The universe U, if it is specified, has to be the ring of integers; any other universe will lead to an error.

```
[ i ..  j by k ]
```

```
[ U | i ..  j by k ]
```

The enumerated sequence consisting of the integers forming the arithmetic progression $i, i+k, i+2*k, \ldots, j$, where i, j and k are (expressions defining) arbitrary integers (but $k \neq 0$).

If k is positive then the last element in the progression will be the greatest integer of the form $i + n * k$ that is less than or equal to j; if j is less than i, the empty sequence of integers will be constructed.

If k is negative then the last element in the progression will be the least integer of the form $i + n * k$ that is greater than or equal to j; if j is greater than i, the empty sequence of integers will be constructed.

The universe U, if it is specified, has to be the ring of integers; any other universe will lead to an error.

Example H10E1

As in the case of sets, it is possible to use the arithmetic progression constructors to save some typing in the creation of sequences of elements of rings other than the ring of integers, but the result will not be treated especially efficiently.

```
> s := [ IntegerRing(200) | x : x in [ 25..125 ] ];
```

10.2.4 Literal Sequences

A literal sequence is an enumerated sequence all of whose terms are from the same structure and all of these are 'typed in' literally. The sole purpose of literal sequences is to load certain enumerated sequences very fast and very space-efficiently; this is only useful when reading in very large sequences (all of whose elements must have been specified literally, that is, not as some expression other than a literal), but then it may save a lot of time. The result will be an enumerated sequence, that is, not distinguished in any way from other such sequences.

At present, only literal sequences of integers are supported.

`\[ m1, ..., mn ]`

Given a succession of literal integers $m_1, \ldots, m_n$, build the enumerated sequence $[m_1, \ldots, m_n]$, in a time and space efficient way.

10.3 Power Sequences

The `PowerSequence` constructor returns a structure comprising the enumerated sequences of a given structure R; it is mainly useful as a parent for other set and sequence constructors. The only operations that are allowed on power sequences are printing, testing element membership, and coercion into the power sequence (see the examples below).

`PowerSequence(R)`

The structure comprising all enumerated sequences of elements of structure R. If R itself is a sequence (or set) then the power structure of its universe is returned.

`S in P`

Returns `true` if enumerated sequence S is in the power sequence P, that is, if all elements of the sequence S are contained in or coercible into R, where P is the power sequence of R; `false` otherwise.

`P ! S`

Return a sequence with universe R consisting of the entries of the enumerated sequence S, where P is the power sequence of R. An error results if not all elements of S can be coerced into R.

Example H10E2

```
> S := [ 1 .. 10 ];
> P := PowerSequence(S);
> P;
Set of sequences over [ 1 .. 10 ]
> F := [ 6/3, 12/4 ];
> F in P;
true
> G := P ! F;
```

```
> Parent(F);
Set of sequences over Rational Field
> Parent(G);
Set of sequences over [ 1 .. 10 ]
```

10.4 Operators on Sequences

This section lists functions for obtaining information about existing sequences, for modifying sequences and for creating sequences from others. Most of these operators only apply to enumerated sequences.

10.4.1 Access Functions

`#S`

Returns the length of the enumerated sequence S, which is the index of the last term of S whose value is defined. The length of the empty sequence is zero.

`Parent(S)`

Returns the parent structure for a sequence S, that is, the structure consisting of all (enumerated) sequences over the universe of S.

`Universe(S)`

Returns the 'universe' of the sequence S, that is, the common structure to which all elements of the sequence belong. This universe may itself be a set or sequence. An error is signalled when S is the null sequence.

`S[i]`

The i-th term s_i of the sequence S. If $i \leq 0$, or $i > \#S + 1$, or $S[i]$ is not defined, then an error results. Here i is allowed to be a multi-index (see Introduction for the interpretation). This can be used as the left hand side of an assignment: `S[i] := x` redefines the i-th term of the sequence S to be x. If $i \leq 0$, then an error results. If $i > n$, then the sequence $[s_1, \ldots, s_n, s_{n+1}, \ldots, s_{i-1}, x]$ replaces S, where $s_{n+1}, \ldots, s_{i-1}$ are all undefined. Here i is allowed to be a multi-index.

An error occurs if x cannot be coerced into the universe of S.

10.4.2 Selection Operators on Enumerated Sequences

Here, S denotes an enumerated sequence $[s_1, \ldots, s_n]$. Further, i and j are integers or multi-indices (see Introduction).

`S[I]`

The sequence $[s_{i_1}, \ldots, s_{i_r}]$ consisting of terms selected from the sequence S, according to the terms of the integer sequence I. If any term of I lies outside the range 1 to $\#S$, then an error results. If I is the empty sequence, then the empty set with universe the same as that of S is returned.

The effect of `T := S[I]` differs from that of `T := [ S[i] : i in I ]`: if in the first case an undefined entry occurs for $i \in I$ between 1 and $\#S$ it will be copied over; in the second such undefined entries will lead to an error.

`Minimum(S)`
`Min(S)`

Given a non-empty, complete enumerated sequence S such that `lt` and `eq` are defined on the universe of S, this function returns two values: a minimal element s in S, as well as the first position i such that $s = S[i]$.

`Maximum(S)`
`Max(S)`

Given a non-empty, complete enumerated sequence S such that `gt` and `eq` are defined on the universe of S, this function returns two values: a maximal element s in S, as well as the first position i such that $s = S[i]$.

`Index(S, x)`
`Index(S, x, f)`
`Position(S, x)`
`Position(S, x, f)`

Returns either the position of the first occurrence of x in the sequence S, or zero if S does not contain x. The second variants of each function starts the search at position f. This can save time in second (and subsequent) searches for the same entry further on. If no occurrence of x in S from position f onwards is found, then zero is returned.

`Representative(R)`
`Rep(R)`

An (arbitrary) element chosen from the enumerated sequence R

`Random(R)`

A random element chosen from the enumerated sequence R. Every element has an equal probability of being chosen. Successive invocations of the function will result in independently chosen elements being returned as the value of the function. If R is empty an error occurs.

`Explode(R)`

Given an enumerated sequence R of length r this function returns the r entries of the sequence (in order).

`Eltseq(R)`

The enumerated sequence R itself. This function is just included for completeness.

10.4.3 Modifying Enumerated Sequences

The operations given here are available as both procedures and functions. In the procedure version, the given sequence is destructively modified 'in place'. This is very efficient, since it is not necessary to make a copy of the sequence. In the function version, the given sequence is not changed, but a modified version of it is returned. This is more suitable if the old sequence is still required. Some of the functions also return useful but non-obvious values.

Here, S denotes an enumerated sequence, and x an element of some structure V. The modifications involving S and x will only be successful if x can be coerced into the universe of S; an error occurs if this fails. (See the Introduction to this Part).

`Append(~S, x)`

`Append(S, x)`

Create an enumerated sequence by adding the object x to the end of S, i.e., the enumerated sequence $[s_1, \ldots s_n, x]$.

There are two versions of this: a procedure, where S is replaced by the appended sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Exclude(~S, x)`

`Exclude(S, x)`

Create an enumerated sequence obtained by removing the first occurrence of the object x from S, i.e., the sequence $[s_1, \ldots s_{i-1}, s_{i+1}, \ldots, s_n]$, where s_i is the first term of S that is equal to x. If x is not in S then this is just S.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Include(~S, x)`

`Include(S, x)`

Create a sequence by adding the object x to the end of S, provided that no term of S is equal to x. Thus, if x does not occur in S, the enumerated sequence $[s_1, \ldots, s_n, x]$ is created.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Insert(~S, i, x)`

`Insert(S, i, x)`

Create the sequence formed by inserting the object x at position i in S and moving the terms $S[i], \ldots, S[n]$ down one place, i.e., the enumerated sequence [s_1,... s_{i-1}, x, s_i, ..., s_n]. Note that i may be bigger than the length n of S, in which case the new length of S will be i, and the entries $S[n+1], \ldots, S[i-1]$ will be undefined.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Insert(~S, k, m, T)`

`Insert(S, k, m, T)`

Create the sequence [s_1, ..., s_{k-1}, t_1, ..., t_l, s_{m+1}, ..., s_n]. If $k \leq 0$ or $k > m+1$, then an error results. If $k = m+1$ then the terms of T will be inserted into S immediately before the term s_k. If $k > n$, then the sequence $[s_1, \ldots, s_n, s_{n+1}, \ldots, s_{k-1}, t_1, \ldots, t_l]$ is created, where $s_{n+1}, \ldots, s_{k-1}$ are all undefined. In the case where T is the empty sequence, terms $s_k, \ldots, s_m$ are deleted from S.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Prune(~S)`

`Prune(S)`

Create the enumerated sequence formed by removing the last term of the sequence S, i.e., the sequence $[s_1, \ldots, s_{n-1}]$. An error occurs if S is empty.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Remove(~S, i)`

`Remove(S, i)`

Create the enumerated sequence formed by removing the i-th term from S, i.e., the sequence $[s_1, \ldots s_{i-1}, s_{i+1}, \ldots, s_n]$. An error occurs if $i < 1$ or $i > n$.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Reverse(~S)`

`Reverse(S)`

Create the enumerated sequence formed by reversing the order of the terms in the complete enumerated sequence S, i.e., the sequence $[s_n, \ldots, s_1]$.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Rotate(~S, p)`

`Rotate(S, p)`

Given a complete sequence S and an integer p, create the enumerated sequence formed by cyclically rotating the terms of the sequence p terms: if p is positive, rotation will be to the right; if p is negative, S is cyclically rotated $-p$ terms to the left; if p is zero nothing happens.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Sort(~S)`

`Sort(S)`

Given a complete enumerated sequence S whose terms belong to a structure on which `lt` and `eq` are defined, create the enumerated sequence formed by (quick-)sorting the terms of S into increasing order.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`Sort(~S, C)`

`Sort(~S, C, ~p)`

`Sort(S, C)`

Given a complete enumerated sequence S and a comparison function C which compares elements of S, create the enumerated sequence formed by sorting the terms of S into increasing order with respect to C. The comparison function C must take two arguments and return an integer less than, equal to, or greater than 0 according to whether the first argument is less than, equal to, or greater than the second argument (e.g.: `func<x, y | x - y>`).

There are three versions of this: a procedure, where S is replaced by the new sequence, a procedure, where S is replaced by the new sequence and the corresponding permutation p is set, and a function, which returns the new sequence and the corresponding permutation. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

`ParallelSort(~S, ~T)`

Given a complete enumerated sequence S, sorts it in place and simultaneously sorts T in the same order. That is, whenever the sorting process would swap the two elements `S[i]` and `S[j]` then the two elements `T[i]` and `T[j]` are also swapped.

`Undefine(~S, i)`

`Undefine(S, i)`

Create the sequence which is the same as the enumerated sequence S but with the i-th term of S undefined; i may be bigger than $\#S$, but $i \leq 0$ produces an error.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`ChangeUniverse(S, V)`

`ChangeUniverse(S, V)`

Given a sequence S with universe U and a structure V which contains U, construct a sequence which consists of the elements of S coerced into V.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

`CanChangeUniverse(S, V)`

Given a sequence S with universe U and a structure V which contains U, attempt to construct a sequence T which consists of the elements of S coerced into V; if successful, return `true` and T, otherwise return `false`.

Example H10E3

We present three ways to obtain the Farey series F_n of degree n.
The Farey series F_n of degree n consists of all rational numbers with denominator less than or equal to n, in order of magnitude. Since we will need numerator and denominator often, we first abbreviate those functions.

```
> D := Denominator;
> N := Numerator;
```

The first method calculates the entries in order. It uses the fact that for any three consecutive Farey fractions $\frac{p}{q}$, $\frac{p'}{q'}$, $\frac{p''}{q''}$ of degree n:

$$p'' = \lfloor \frac{q+n}{q'} \rfloor p' - p, \quad q'' = \lfloor \frac{q+n}{q'} \rfloor q' - q.$$

```
> farey := function(n)
>    f := [ RationalField() | 0, 1/n ];
>    p := 0;
>    q := 1;
>    while p/q lt 1 do
>       p := ( D(f[#f-1]) + n) div D(f[#f]) * N(f[#f])  - N(f[#f-1]);
>       q := ( D(f[#f-1]) + n) div D(f[#f]) * D(f[#f])  - D(f[#f-1]);
>       Append(~f, p/q);
>    end while;
>    return f;
> end function;
```

The second method calculates the Farey series recursively. It uses the property that F_n may be obtained from F_{n-1} by inserting a new fraction (namely $\frac{p+p'}{q+q'}$) between any two consecutive rationals $\frac{p}{q}$ and $\frac{p'}{q'}$ in F_{n-1} for which $q + q'$ equals n.

```
> function farey(n)
```

```
>    if n eq 1 then
>       return [RationalField() | 0, 1 ];
>    else
>       f := farey(n-1);
>       i := 0;
>       while i lt #f-1 do
>          i +:= 1;
>          if D(f[i]) + D(f[i+1]) eq n then
>             Insert( ~f, i+1, (N(f[i]) + N(f[i+1]))/(D(f[i]) + D(f[i+1])));
>          end if;
>       end while;
>       return f;
>    end if;
> end function;
```

The third method is very straightforward, and uses `Sort` and `Setseq` (defined above).

```
> farey := func< n |
>               Sort(Setseq({ a/b : a in { 0..n}, b in { 1..n} | a le b }))>;
> farey(6);
[ 0, 1/6, 1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 1 ]
```

10.4.4 Creating New Enumerated Sequences from Existing Ones

`S cat T`

The enumerated sequence formed by concatenating the terms of S with the terms of T, i.e. the sequence $[s_1, \ldots, s_n, t_1, \ldots, t_m]$.

If the universes of S and T are different, an attempt to find a common overstructure is made; if this fails an error results (see the Introduction).

`S cat:= T`

Mutation assignment: change S to be the concatenation of S and T. Functionally equivalent to `S := S cat T`.

If the universes of S and T are different, an attempt to find a common overstructure is made; if this fails an error results (see the Introduction).

`Partition(S, p)`

Given a complete non-empty sequence S as well as an integer p that divides the length n of S, construct the sequence whose terms are the sequences formed by taking p terms of S at a time.

`Partition(S, P)`

Given a complete non-empty sequence S as well as a complete sequence of positive integers P, such that the sum of the entries of P equals the length of S, construct the sequence whose terms are the sequences formed by taking $P[i]$ terms of S, for $i = 1, \ldots, \#P$.

`Setseq(S)`

`SetToSequence(S)`

Given a set S, construct a sequence whose terms are the elements of S taken in some arbitrary order.

`Seqset(S)`

`SequenceToSet(S)`

Given a sequence S, create a set whose elements are the distinct terms of S.

Example H10E4

The following example illustrates several of the access, creation and modification operations on sequences.
Given a rational number r, this function returns a sequence of different integers d_i such that $r = \sum 1/d_i$ [Bee93].

```
> egyptian := function(r)
>       n := Numerator(r);
>       d := Denominator(r);
>       s := [d : i in [1..n]];
>       t := { d};
>       i := 2;
>       while i le #s do
>              c := s[i];
>              if c in t then
>                     Remove(~s, i);
>                     s cat:= [c+1, c*(c+1)];
>              else
>                     t join:= { c};
>                     i := i+1;
>              end if;
>       end while;
>       return s;
> end function;
```

Note that the result may be rather larger than necessary:

```
> e := egyptian(11/13);
> // Check the result!
> &+[1/d : d in e];
11/13
```

```
> #e;
2047
> #IntegerToString(Maximum(e));
1158
```

while instead of this sequence of 2047 integers, the biggest of the entries having 1158 decimal digits, the following equation also holds:

$$\frac{1}{3}+\frac{1}{4}+\frac{1}{6}+\frac{1}{12}+\frac{1}{78}=\frac{11}{13}.$$

10.4.4.1 Operations on Sequences of Booleans

The following operation work pointwise on sequences of booleans of equal length.

`And(S, T)`
`And(~S, T)`

The sequence whose ith entry is the logical and of the ith entries of S and T. The result is placed in S if it is given by reference (~).

`Or(S, T)`
`Or(~S, T)`

The sequence whose ith entry is the logical or of the ith entries of S and T. The result is placed in S if it is given by reference.

`Xor(S, T)`
`Xor(~S, T)`

The sequence whose ith entry is the logical xor of the ith entries of S and T. The result is placed in S if it is given by reference.

`Not(S)`
`Not(~S)`

The sequence whose ith entry is the logical not of the ith entry of S. The result is placed in S if it is given by reference.

10.5 Predicates on Sequences

Boolean valued operators and functions on enumerated sequences exist to test whether entries are defined (see previous section), to test for membership and containment, and to compare sequences with respect to an ordering on its entries. On formal sequences, only element membership can be tested.

`IsComplete(S)`

Boolean valued function, returning `true` if and only if each of the terms $S[i]$ for $1 \leq i \leq \#S$ is defined, for an enumerated sequence S.

`IsDefined(S, i)`

Given an enumerated sequence S and an index i, this returns `true` if and only if $S[i]$ is defined. (Hence the result is `false` if $i > \#S$, but an error results if $i < 1$.) Note that the index i is allowed to be a multi-index; if $i = [i_1, \ldots, i_r]$ is a multi-index and $i_j > \#S[i_1, \ldots, i_{j-1}]$ the function returns false, but if S is s levels deep and $r > s$ while $i_j \leq \#S[i_1, \ldots, i_{j-1}]$ for $1 \leq j \leq s$, then an error occurs.

`IsEmpty(S)`

Boolean valued function, returning `true` if and only if the enumerated sequence S is empty.

`IsNull(S)`

Boolean valued function, returning `true` if and only if the enumerated sequence S is empty and its universe is undefined, `false` otherwise.

10.5.1 Membership Testing

Here, S and T denote sequences. The element x is always assumed to be compatible with S.

`x in S`

Returns `true` if the object x occurs as a term of the enumerated or formal sequence S, `false` otherwise. If x is not in the universe of S, coercion is attempted. If that fails, an error results.

`x notin S`

Returns `true` if the object x does not occur as a term of the enumerated or formal sequence S, `false` otherwise. If x is not in the universe of S, coercion is attempted. If that fails, an error results.

`IsSubsequence(S, T)`

`IsSubsequence(S, T: Kind := option)`

`Kind` MonStgElt *Default : "Consecutive"*

Returns `true` if the enumerated sequence S appears as a subsequence of consecutive elements of the enumerated sequence T, `false` otherwise.

By changing the default value `"Consecutive"` of the parameter `Kind` to `"Sequential"` or to `"Setwise"`, this returns `true` if and only if the elements of S appear in order (but not necessarily consecutively) in T, or if and only if all elements of S appear as elements of T; so in the latter case the test is merely whether the set of elements of S is contained in the set of elements of T.

If the universes of S and T are not the same, coercion is attempted.

`S eq T`

Returns `true` if the enumerated sequences S and T are equal, `false` otherwise. If the universes of S and T are not the same, coercion is attempted.

`S ne T`

Returns `true` if the enumerated sequences S and T are not equal, `false` otherwise. If the universes of S and T are not the same, coercion is attempted.

10.5.2 Testing Order Relations

Here, S and T denote complete enumerated sequences with universe U and V respectively, such that a common overstructure W for U and V can be found (as outlined in the Introduction), and such that on W an ordering on the elements is defined allowing the Magma operators `eq` ($=$), `le` ($\leq$), `lt` ($<$), `gt` ($>$), and `ge` ($\geq$) to be invoked on its elements.

With these comparison operators the *lexicographical* ordering is used to order complete enumerated sequences. Sequences S and T are equal (`S eq T`) if and only if they have the same length and all terms are the same. A sequence S precedes T (`S lt T`) in the ordering imposed by that of the terms if at the first index i where S and T differ then $S[i] < T[i]$. If the length of T exceeds that of S and S and T agree in all places where S until after the length of S, then `S lt T` is true also. In all other cases where $S \neq T$ one has `S gt T`.

`S lt T`

Returns `true` if the sequence S precedes the sequence T under the ordering induced from S, `false` otherwise. Thus, `true` is returned if and only if either $S[k] < T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#S$ and $\#S < \#T$.

`S le T`

Returns `true` if the sequence S either precedes the sequence T, under the ordering induced from S, or is equal to T, `false` otherwise. Thus, `true` is returned if and only if either $S[k] < T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#S$ and $\#S \leq \#T$.

`S ge T`

Returns `true` if the sequence S either comes after the sequence T, under the ordering induced from S, or is equal to T, `false` otherwise. Thus, `true` is returned if and only if either $S[k] > T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#T$ and $\#S \geq \#T$.

`S gt T`

Returns `true` if the sequence S comes after the sequence T under the ordering induced from S, `false` otherwise. Thus, `true` is returned if and only if either $S[k] > T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#T$ and $\#S > \#T$.

10.6 Recursion, Reduction, and Iteration

10.6.1 Recursion

It is often very useful to be able to refer to a sequence currently under construction, for example to define the sequence recursively. For this purpose the `Self` operator is available.

`Self(n)`
`Self()`

This operator enables the user to refer to an already defined previous entry $s[n]$ of the enumerated sequence s inside the sequence constructor, or the sequence s itself.

Example H10E5

The example below shows how the sequence of the first 100 Fibonacci numbers can be created recursively, using `Self`. Next it is shown how to use reduction on these 100 integers.

```
> s := [ i gt 2 select Self(i-2)+Self(i-1) else 1 : i in [1..100] ];
> &+s;
927372692193078999175
```

10.6.2 Reduction

Instead of using a loop to apply the same binary associative operator to all elements of a complete enumerated sequence, it is possible to use the *reduction operator* `&`.

`&○ S`

Given a complete enumerated sequence $S = [a_1, a_2, \ldots, a_n]$ of elements belonging to an algebraic structure U, and an (associative) operator $\circ : U \times U \to U$, form the element $a_1 \circ a_2 \circ a_3 \circ \ldots \circ a_n$.

Currently, the following operators may be used to reduce sequences: `+, *, and, or, join, meet, cat`. An error will occur if the operator is not defined on U.

If S contains a single element a, then the value returned is a. If S is the null sequence (empty and no universe specified), then reduction over S leads to an error; if S is empty with universe U in which the operation is defined, then the result (or error) depends on the operation and upon U. The following table defines the return value:

	empty	*null*
&+	U ! 0	error
&∗	U ! 1	error
&and	`true`	`true`
&or	`false`	`false`
&join	*empty*	*null*
&meet	error	error
&cat	*empty*	*null*

10.7 Iteration

Enumerated sequences allow iteration over their elements. In particular, they can be used as the range set in the sequence and set constructors, and as domains in `for` loops.

When multiple range sequences are used, it is important to know in which order the range are iterated over; the rule is that the repeated iteration takes place as nested loops where the first range forms the innermost loop, etc. See the examples below.

`for x in S do` *statements*`; end for;`

An enumerated sequence S may be the range for the `for`-statement. The iteration only enumerates the defined terms of the sequence.

Example H10E6

The first example shows how repeated iteration inside a sequence constructor corresponds to nesting of loops.

```
> [<number, letter> : number in [1..5], letter in ["a", "b", "c"]];
```

```
[ <1, a>, <2, a>, <3, a>, <4, a>, <5, a>, <1, b>, <2, b>, <3, b>, <4, b>, <5,
b>, <1, c>, <2, c>, <3, c>, <4, c>, <5, c> ]
> r := [];
> for letter in ["a", "b", "c"] do
>     for number in [1..5] do
>         Append(~r, <number, letter>);
>     end for;
> end for;
> r;
[ <1, a>, <2, a>, <3, a>, <4, a>, <5, a>, <1, b>, <2, b>, <3, b>, <4, b>, <5,
b>, <1, c>, <2, c>, <3, c>, <4, c>, <5, c> ]
```

This explains why the first construction below leads to an error, whereas the second leads to the desired sequence.

```
>  // The following produces an error:
>  [ <x, y> : x in [0..5], y in [0..x] | x^2+y^2 lt 16 ];
                                          ^
User error: Identifier 'x' has not been declared

> [ <x, y> : x in [0..y], y in [0..5] | x^2+y^2 lt 16 ];
[ <0, 0>, <0, 1>, <1, 1>, <0, 2>, <1, 2>, <2, 2>, <0, 3>, <1, 3>, <2, 3> ]
```

Note the following! In the last line below there are two different things with the name x. One is the (inner) loop variable, the other just an identifier with value 1000 that is used in the bound for the other (outer) loop variable y: the limited scope of the inner loop variable x makes it invisible to y, whence the error in the first case.

```
>  // The following produces an error:
>  #[ <x, y> : x in [0..5], y in [0..x] | x^2+y^2 lt 100 ];
                                           ^
User error: Identifier 'x' has not been declared

> x := 1000;
> #[ <x, y> : x in [0..5], y in [0..x] | x^2+y^2 lt 100 ];
59
```

10.8 Bibliography

[**Bee93**] L. Beeckmans. The splitting algorithm for Egyptian fractions. *J. Number Th.*, 43:173–185, 1993.

11 TUPLES AND CARTESIAN PRODUCTS

Chapter 11
TUPLES AND CARTESIAN PRODUCTS

11.1 Introduction

A cartesian product may be constructed from a finite number of factors, each of which may be a set or algebraic structure. The term *tuple* will refer to an element of a cartesian product.

Note that the rules for tuples are quite different to those for sequences. Sequences are elements of a cartesian product of n copies of a fixed set (or algebraic structure) while tuples are elements of cartesian products where the factors may be different sets (structures). The semantics for tuples are quite different to those for sequences. In particular, the parent cartesian product of a tuple is fixed once and for all. This is in contrast to a sequence, which may grow and shrink during its life (thus implying a varying parent cartesian product).

11.2 Cartesian Product Constructor and Functions

The special constructor `car< ... >` is used for the creation of cartesian products of structures.

`car< R1, ..., Rk >`

Given a list of sets or algebraic structures $R_1, \ldots, R_k$, construct the cartesian product set $R_1 \times \cdots \times R_k$.

`CartesianProduct(R, S)`

Given structures R and S, construct the cartesian product set $R \times S$. This is the same as calling the `car` constructor with the two arguments R and S.

`CartesianProduct(L)`

Given a sequence or tuple L of structures, construct the cartesian product of the elements of L.

`CartesianPower(R, k)`

Given a structure R and an integer k, construct the cartesian power set R^k.

`Flat(C)`

Given a cartesian product C of structures which may themselves be cartesian products, return the cartesian product of the base structures, considered in depth-first order (see `Flat` for the element version).

`NumberOfComponents(C)`

Given a cartesian product C, return the number of components of C.

`Component(C, i)`

`C[i]`

The i-th component of C.

`#C`

Given a cartesian product C, return the cardinality of C.

`Rep(C)`

Given a cartesian product C, return a representative of C.

`Random(C)`

Given a cartesian product C, return a random element of C.

Example H11E1

We create the product of **Q** and **Z**.

```
> C := car< RationalField(), Integers() >;
> C;
Cartesian Product<Rational Field, Ring of Integers>
```

11.3 Creating and Modifying Tuples

`elt< C | a1, a2, ..., ak >`

`C ! < a1, a2, ..., ak >`

Given a cartesian product $C = R_1 \times \cdots \times R_k$ and a sequence of elements $a_1, a_2, \ldots, a_k$, such that a_i belongs to the set R_i $(i = 1, \ldots, k)$, create the tuple $T =< a_1, a_2, ..., a_k >$ of C.

`< a1, a2, ..., ak >`

Given a cartesian product $C = R_1 \times \cdots \times R_k$ and a list of elements $a_1, a_2, \ldots, a_k$, such that a_i belongs to the set R_i, $(i = 1, \ldots, k)$, create the tuple $T =< a_1, a_2, ..., a_k >$ of C. Note that if C does not already exist, it will be created at the time this expression is evaluated.

`Append(T, x)`

Return the tuple formed by adding the object x to the end of the tuple T. Note that the result lies in a new cartesian product of course.

`Append(~T, x)`

(Procedure.) Destructively add the object x to the end of the tuple T. Note that the new T lies in a new cartesian product of course.

`Prune(T)`

Return the tuple formed by removing the last term of the tuple T. The length of T must be greater than 1. Note that the result lies in a new cartesian product of course.

`Prune(~T)`

(Procedure.) Destructively remove the last term of the tuple T. The length of T must be greater than 1. Note that the new T lies in a new cartesian product of course.

`Flat(T)`

Construct the flattened version of the tuple T. The flattening is done in the same way as `Flat`, namely depth-first.

Example H11E2

We build a set of pairs consisting of primes and their reciprocals.

```
> C := car< Integers(), RationalField() >;
> C ! < 26/13, 13/26 >;
<2, 1/2>
> S := { C | <p, 1/p> : p in [1..25] | IsPrime(p) };
> S;
{ <5, 1/5>, <7, 1/7>, <2, 1/2>, <19, 1/19>, <17, 1/17>, <23, 1/23>, <11, 1/11>,
<13, 1/13>, <3, 1/3> }
```

11.4 Tuple Access Functions

`Parent(T)`

The cartesian product to which the tuple T belongs.

`#T`

Number of components of the tuple T.

`T[i]`

Return the i-th component of tuple T. Note that this indexing can also be used on the left hand side for modification of T.

`Explode(T)`

Given a tuple T of length n, this function returns the n entries of T (in order).

`TupleToList(T)`

`Tuplist(T)`

Given a tuple T return a list containing the entries of T.

Example H11E3

```
>  f := < 11/2, 13/3, RootOfUnity(3, CyclotomicField(3)) >;
> f;
<11/2, 13/3, (zeta_3)>
> #f;
3
> Parent(f);
Cartesian Product<Rational Field, Rational Field, Cyclotomic field Q(zeta_3)>
> f[1]+f[2]+f[3];
(1/6) * (59 + 6*zeta_3)
> f[3] := 7;
> f;
<11/2, 13/3, 7>
```

11.5 Equality

`T eq U`

Return `true` if and only if the tuples T and U are equal.

`T ne U`

Return `true` if and only if the tuples T and U are distinct.

11.6 Other operations

`&*T`

For a tuple T where each component lies in a structure that supports multiplication and such there exists a common over structure, return the product of the entries.

12 LISTS

Chapter 12
LISTS

12.1 Introduction

A *list* in MAGMA is an ordered finite collection of objects. Unlike sequences, lists are not required to consist of objects that have some common parent. Lists are not stored compactly and the operations provided for them are not extensive. They are mainly provided to enable the user to gather assorted objects temporarily together.

12.2 Construction of Lists

Lists can be constructed by expressions enclosed in special brackets `[*` and `*]`.

`[* *]`

The empty list.

`[* e1, e2, ..., en *]`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$, create the list containing $a_1, a_2, \ldots, a_n$.

12.3 Creation of New Lists

Here, S denotes the list $[* s_1, \ldots, s_n *]$, while T denotes the list $[* t_1, \ldots, t_m *]$.

`S cat T`

The list formed by concatenating the terms of the list S with the terms of the list T, i.e. the list $[* s_1, \ldots, s_n, t_1, \ldots, t_m *]$.

`S cat:= T`

(Procedure.) Destructively concatenate the terms of the list T to S; i.e. so S becomes the list $[* s_1, \ldots, s_n, t_1, \ldots, t_m *]$.

`Append(S, x)`

The list formed by adding the object x to the end of the list S, i.e. the list $[* s_1, \ldots s_n, x *]$.

`Append(~S, x)`

(Procedure.) Destructively add the object x to the end of the list S; i.e. so S becomes the list $[* s_1, \ldots s_n, x *]$.

`Insert(~S, i, x)`

`Insert(S, i, x)`

Create the list formed by inserting the object x at position i in S and moving the terms $S[i], \ldots, S[n]$ down one place, i.e., the list $[* s_1, \ldots, s_{i-1}, x, s_i, \ldots, s_n *]$. Note that i must not be bigger than $n+1$ where n is the length of S.

There are two versions of this: a procedure, where S is replaced by the new list, and a function, which returns the new list. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the list S will not be copied.

`Prune(S)`

The list formed by removing the last term of the list S, i.e. the list $[* s_1, \ldots, s_{n-1} *]$.

`Prune(~S)`

(Procedure.) Destructively remove the last term of the list S; i.e. so S becomes the list $[* s_1, \ldots, s_{n-1} *]$.

`SequenceToList(Q)`

`Seqlist(Q)`

Given a sequence Q, construct a list whose terms are the elements of Q taken in the same order.

`TupleToList(T)`

`Tuplist(T)`

Given a tuple T, construct a list whose terms are the elements of T taken in the same order.

`Reverse(L)`

Given a list L return the same list, but in reverse order.

12.4 Access Functions

`#S`

The length of the list S.

`IsEmpty(S)`

Return whether S is empty (has zero length).

`S[i]`

Return the i-th term of the list S. If either $i \leq 0$ or $i > \#S+1$, then an error results. Here i is allowed to be a multi-index (see Section 8.3.1 for the interpretation).

`S[I]`

Return the sublist of S given by the indices in the sequence I. Each index in I must be in the range $[1..l]$, where l is the length of S.

`IsDefined(L, i)`

Checks whether the ith item in L is defined or not, that is it returns `true` if i is at most the length of L and `false` otherwise.

12.5 Assignment Operator

`S[i] := x`

Redefine the i-th term of the list S to be x. If $i \leq 0$, then an error results. If $i = \#S + 1$, then x is appended to S. Otherwise, if $i > \#S + 1$, an error results. Here i is allowed to be a multi-index.

13 ASSOCIATIVE ARRAYS

Chapter 13
ASSOCIATIVE ARRAYS

13.1 Introduction

An *associative array* in MAGMA is an array which may be indexed by arbitrary elements of an index structure I. The indexing may thus be by objects which are not integers. These objects are known as the *keys*. For each current key there is an associated value. The *values* associated with the keys need not lie in a fixed universe but may be of any type.

13.2 Operations

`AssociativeArray()`

Create the null associative array with no index universe. The first assignment to the array will determine its index universe.

`AssociativeArray(I)`

Create the empty associative array with index universe I.

`A[x] := y`

Set the value in A associated with index x to be y. If x is not coercible into the current index universe I of A, then an attempt is first made to lift the index universe of A to contain both I and x.

`A[x]`

Given an index x coercible into the index universe I of A, return the value associated with x. If x is not in the keys of A, then an error is raised.

`IsDefined(A, x)`

Given an index x coercible into the index universe I of A, return whether x is currently in the keys of A and if so, return also the value $A[x]$.

`Remove(~ A, x)`

(Procedure.) Destructively remove the value indexed by x from the array A. If x is not present as an index, then nothing happens (i.e., an error is not raised).

`Universe(A)`

Given an associative array A, return the index universe I of A, in which the keys of A currently lie.

Keys(A)

Given an associative array A, return the current keys of A as a set. Warning: this constructs a new copy of the set of keys, so should only be called when that is needed. It is not meant to be used as a quick access function.

Example H13E1

This example shows simple use of associative arrays. First we create an array indexed by rationals.

```
> A := AssociativeArray();
> A[1/2] := 7;
> A[3/8] := "abc";
> A[3] := 3/8;
> A[1/2];
7
> IsDefined(A, 3);
true 3/8
> IsDefined(A, 4);
false
> IsDefined(A, 3/8);
true abc
> Keys(A);
{ 3/8, 1/2, 3 }
> for x in Keys(A) do x, A[x]; end for;
1/2 7
3/8 abc
3 3/8
> Remove(~A, 3/8);
> IsDefined(A, 3/8);
false
> Keys(A);
{ 1/2, 3 }
> Universe(A);
Rational Field
```

We repeat that an associative array can be indexed by elements of any structure. We now index an array by elements of the symmetric group S_3.

```
> G := Sym(3);
> A := AssociativeArray(G);
> v := 1; for x in G do A[x] := v; v +:= 1; end for;
> A;
Associative Array with index universe GrpPerm: G, Degree 3, Order 2 * 3
> Keys(A);
{
    (1, 3, 2),
    (2, 3),
    (1, 3),
    (1, 2, 3),
```

```
    (1, 2),
    Id(G)
}
> A[G!(1,3,2)];
3
```

14 COPRODUCTS

Chapter 14
COPRODUCTS

14.1 Introduction

Coproducts can be useful in various situations, as they may contain objects of entirely different types. Although the coproduct structure will serve as a single parent for such diverse objects, the proper parents of the elements are recorded internally and restored whenever the element is retrieved from the coproduct.

14.2 Creation Functions

There are two versions of the coproduct constructor. Ordinarily, coproducts will be constructed from a list of structures. These structures are called the `constituents` of the coproduct. A single sequence argument is allowed as well to be able to create coproducts of parameterized families of structures conveniently.

14.2.1 Creation of Coproducts

`cop< S1, S2, ..., Sk >`

`cop< [ S1, S2, ..., Sk ] >`

Given a list or a sequence of two or more structures S_1, S_2, ..., S_k, this function creates and returns their coproduct C as well as a sequence of maps [m_1, m_2, ..., m_k] that provide the injections $m_i : S_i \to C$.

14.2.2 Creation of Coproduct Elements

Coproduct elements are usually created by the injections returned as the second return value from the `cop<>` constructor. The bang (!) operator may also be used but only if the type of the relevant constituent is unique for the particular coproduct.

`m(e)`

Given a coproduct injection map m and an element of one of the constituents of the coproduct C, create the coproduct element version of e.

`C ! e`

Given a coproduct C and an element e of one of the constituents of C such that the type of that constituent is unique within that coproduct, create the coproduct element version of e.

14.3 Accessing Functions

`Injections(C)`

Given a coproduct C, return the sequence of injection maps returned as the second argument from the `cop<>` constructor.

`#C`

Given a coproduct C, return the length (number of constituents) of C.

`Constituent(C, i)`

Given a coproduct C and an integer i between 1 and the length of C, return the i-th constituent of C.

`Index(x)`

Given an element x from a coproduct C, return the constituent number of C to which x belongs.

14.4 Retrieve

The function described here restores an element of a coproduct to its original state.

`Retrieve(x)`

Given an element x of some coproduct C, return the element as an element of the structure that formed its parent before it was mapped into C.

Example H14E1

We illustrate basic uses of the coproduct constructors and functions.

```
> C := cop<IntegerRing(), Strings()>;
> x := C ! 5;
> y := C ! "abc";
> x;
5
> y;
abc
> Parent(x);
Coproduct<Integer Ring, String structure>
> x eq 5;
true
> x eq y;
false
> Retrieve(x);
5
> Parent(Retrieve(x));
Integer Ring
```

14.5 Flattening

The function described here enables the 'concatenation' of coproducts into a single one.

`Flat(C)`

Given a coproduct C of structures which may themselves be coproducts, return the coproduct of the base structures, considered in depth-first order.

14.6 Universal Map

`UniversalMap(C, S, [ n`$_1$`, ..., n`$_m$` ])`

Given maps n_1, ..., n_m from structures S_1, ..., S_m that compose the coproduct C, to some structure S, this function returns the universal map $C \to S$.

15 RECORDS

Chapter 15
RECORDS

15.1 Introduction

In a *record* several objects can be collected. The objects in a record are stored in *record fields*, and are accessed by using *fieldnames*. Records are like tuples (and unlike sets or sequences) in that the objects need not all be of the same kind. Though records and tuples are somewhat similar, there are several differences too. The components of tuples are indexed by integers, and every component must be defined. The fields of records are indexed by fieldnames, and it is possible for some (or all) of the fields of a record not to be assigned; in fact, a field of a record may be assigned or deleted at any time. A record must be constructed according to a pre-defined *record format*, whereas a tuple may be constructed without first giving the Cartesian product that is its parent, since MAGMA can deduce the parent from the tuple.

In the definition of a record format, each field is given a fieldname. If the field is also given a parent magma or a category, then in any record created according to this format, that field must conform to this requirement. However, if the field is not given a parent magma or category, there is no restriction on the kinds of values stored in that field; different records in the format may contain disparate values in that field. By contrast, every component of a Cartesian product is a magma, and the components of all tuples in this product must be elements of the corresponding magma.

Because of the flexibility of records, with respect to whether a field is assigned and what kind of value is stored in it, Boolean operators are not available for comparing records.

15.2 The Record Format Constructor

The special constructor `recformat< ... >` is used for the creation of record formats. A record format must be created before records in that format are created.

`recformat< L >`

Construct the record format corresponding to the non-empty fieldname list L. Each term of L must be one of the following:

(a) *fieldname* in which case there is no restriction on values that may be stored in this field of records having this format;

(b) *fieldname*:*expression* where the expression evaluates to a magma which will be the parent of values stored in this field of records having this format; or

(c) *fieldname*:*expression* where the expression evaluates to a category which will be the category of values stored in this field of records having this format;

where *fieldname* consists of characters that would form a valid identifier name. Note that it is not a string.

Example H15E1

We create a record format with these fields: `n`, an integer; `misc`, which has no restrictions; and `seq`, a sequence (with any universe possible).

```
> RF := recformat< n : Integers(), misc, seq : SeqEnum >;
> RF;
recformat<n: IntegerRing(), misc, seq: SeqEnum>
> Names(RF);
[ n, misc, seq ]
```

15.3 Creating a Record

Before a record is created, its record format must be defined. A record may be created by assigning as few or as many of the record fields as desired.

`rec< F | L >`

Given a record format F, construct the record format corresponding to the field assignment list L. Each term of L must be of the form *fieldname* := *expression* where *fieldname* is in F and the value of the expression conforms (directly or by coercion) to any restriction on it. The list L may be empty, and there is no fixed order for the fieldnames.

Example H15E2

We build some records having the record format `RF`.

```
> RF := recformat< n : Integers(), misc, seq : SeqEnum >;
> r := rec< RF | >;
> r;
rec<RF | >
> s := rec< RF | misc := "adsifaj", n := 42, seq := [ GF(13) | 4, 8, 1 ]>;
> s;
rec<RF | n := 42, misc := adsifaj, seq := [ 4, 8, 1 ]>
> t := rec< RF | seq := [ 4.7, 1.9 ], n := 51/3 >;
> t;
rec<RF | n := 17, seq := [ 4.7, 1.9 ]>
> u := rec< RF | misc := RModule(PolynomialRing(Integers(7)), 4) >;
> u;
rec<RF | misc := RModule of dimension 4 with base ring Univariate Polynomial
Algebra over Integers(7)>
```

15.4 Access and Modification Functions

Fields of records may be inspected, assigned and deleted at any time.

`Format(r)`

The format of record r.

`Names(F)`

The fieldnames of the record format F returned as a sequence of strings.

`Names(r)`

The fieldnames of record r returned as a sequence of strings.

`r‘`*fieldname*

Return the field of record r with this fieldname. The format of r must include this fieldname, and the field must be assigned in r.

`r‘`*fieldname*`:=` *expression*`;`

Reassign the given field of r to be the value of the expression. The format of r must include this fieldname, and the expression's value must satisfy (directly or by coercion) any restriction on the field.

`delete r‘`*fieldname*

(Statement.) Delete the current value of the given field of record r.

`assigned r‘`*fieldname*

Returns true if and only if the given field of record r currently contains a value.

`r‘‘s`

Given an expression s that evaluates to a string, return the field of record r with the fieldname corresponding to this string. The format of r must include this fieldname, and the field must be assigned in r.

This syntax may be used anywhere that r‘*fieldname* may be used, including in left hand side assignment, `assigned` and `delete`.

Example H15E3

```
> RF := recformat< n : Integers(), misc, seq : SeqEnum >;
> r := rec< RF | >;
> s := rec< RF | misc := "adsifaj", n := 42, seq := [ GF(13) | 4, 8, 1 ]>;
> t := rec< RF | seq := [ 4.7, 1.9 ], n := 51/3 >;
> u := rec< RF | misc := RModule(PolynomialRing(Integers(7)), 4) >;
> V4 := u`misc;
> assigned r`seq;
false
> r`seq := Append(t`seq, t`n); assigned r`seq;
true
> r;
rec<RF | seq := [ 4.7, 1.9, 17 ]>
>  // The following produces an error:
>  t``(s`misc);
>> t``(s`misc);
          ^
Runtime error in `: Field 'adsifaj' does not exist in this record
> delete u``("m" cat "isc"); u;
rec<RF | >
```

16 MAPPINGS

Chapter 16
MAPPINGS

16.1 Introduction

Mappings play a fundamental role in algebra and, indeed, throughout mathematics. Reflecting this importance, mappings are one of the fundamental datatypes in our language. The most general way to define a mapping $f : A \to B$ in a programming language is to write a *function* which, given any element of A, will return its image under f in B. While this approach to the definition of mappings is completely general, it is desirable to have mappings as an independent datatype. It is then possible to provide a very compact notation for specifying important classes of mappings such as homomorphisms. Further, a range of operations peculiar to the mapping type can be provided.

Mappings are created either through use of *mapping constructors* as described in this Chapter, or through use of certain standard functions that return mappings as either primary or secondary values.

All mappings are objects in the MAGMA category `Map`.

16.1.1 The Map Constructors

There are three main mapping constructors: the general map constructor `map< >`, the homomorphism constructor `hom< >`, and the partial map constructor `pmap< >`. The general form of all constructors is the same: inside the angle brackets there are two components separated by a pipe `|`. To the left the user specifies a *domain* A and a *codomain* B, separated by `->`; to the right of the pipe the user specifies how images are obtained for elements of the domain. The latter can be done in one of several ways: one specifies either the *graph* of the map, or a *rule* describing how images are to be formed, or for homomorphisms, one specifies generator images. We will describe each in the next subsections. The result is something like `map< A -> B |` *expression*`>`.

The domain and codomain of the map can be arbitrary magmas. When a full map (as opposed to a partial map) is constructed by use of a graph, the domain is necessarily finite.

The main difference between maps and partial maps is that a partial map need not be defined for every element of the domain. The main difference between these two types of map and homomorphisms is that the latter are supposed to provide *structure-preserving* maps between algebraic structures. On the one hand this makes it possible to allow the specification of images for homomorphisms in a different fashion: homomorphism can be given via *images* for *generators* of the domain. On the other hand homomorphisms are restricted to cases where domain and (image in the) codomain have a similar structure. The generator image form only makes sense for domains that are *finitely presented*. Homomorphisms are described in more detail below.

16.1.2 The Graph of a Map

Let A and B be structures. A *subgraph* of the cartesian product $C = A \times B$ is a subset G of C such that each element of A appears at most once among the first components of the pairs $< a, b >$ of G. A subgraph having the additional property that every element of A appears as the first component of some pair $< a, b >$ of G is called a *graph* of $A \times B$.

A mapping between A and B can be identified with a graph G of $A \times B$, a partial map can be identified with a subgraph. We now describe how a graph may be represented in the context of the map constructor. An element of the graph of $A \times B$ can be given either as a *tuple* `<a, b>`, or as an *arrow pair* `a -> b`. The specification of a (sub)graph in a map constructor should then consist of either a (comma separated) list, a sequence, or a set of such tuples or arrow pairs (a mixture is permitted).

16.1.3 Rules for Maps

The specification of a rule in the map constructor involves a free variable and an expression, usually involving the free variable, separated by `:->`, for example `x :-> 3*x - 1`. The scope of the free variable is restricted to the map constructor (so the use of x does not interfere with values of x outside the constructor). A general expression is allowed in the rule, which may involve intrinsic or user functions, and even in-line definitions of such functions.

16.1.4 Homomorphisms

Probably the most useful form of the map-constructor is the version for homomorphisms. Most interesting mappings in algebra are homomorphisms, and if an algebraic structure A belongs to a family of algebraic structures which form a variety we have the fundamental result that a homomorphism is uniquely determined by the images of any generating set. This provides us with a particularly compact way of defining and representing homomorphisms. While the syntax of the homomorphism constructor is similar to that of the general mapping constructor, the semantics are sometimes different.

The kind of homomorphism built by the hom-constructor is determined entirely by the domain: thus, a *group* homomorphism results from applying `hom` to a domain A that is one of the types of group in Magma, a *ring* homomorphism results when A is a ring, etc. As a consequence, the requirements on the specification of homomorphisms are dependent on the category to which A belongs. Often, the codomain of a homomorphism is required to belong to the same variety. But even within a category the specification may depend on the type of structure; for details we refer the reader to the specific chapters.

A homomorphism can be specified using either a rule map or by generator images. In the latter case the processor will seek to express an element as a word in the generators of A when asked to compute its image. Thus A needs to be finitely presented.

16.1.5 Checking of Maps

It should be pointed out that checking the 'correctness' of mappings can be done to a limited extent only. If the mapping is given by means of a graph, Magma will check that no multiple images are specified, and that an image is given for every element of the

domain (unless a partial map is defined). If a rule is given, it cannot be checked that it is defined on all of the domain. Also, it is in general the responsibility of the user to ensure that the images provided for a `hom` constructor do indeed define a homomorphism.

16.2 Creation Functions

In this section we describe the creation of maps, partial maps, and homomorphisms via the various forms of the constructors, as well as maps that define coercions between algebraic structures.

16.2.1 Creation of Maps

Maps between structures A and B may be specified either by providing the full graph (as defined in the previous section) or by supplying an expression rule for finding images.

```
map<  A -> B | G  >
```

Given a finite structure A, a structure B and a graph G of $A \times B$, construct the mapping $f : A \to B$, as defined by G. The graph G may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter. Note that G must be a full graph, i.e., every element of A must occur exactly once as a first component.

```
map<  A -> B | x :-> e(x)  >
```

Given a set or structure A, a set or structure B, a variable x and an expression $e(x)$, usually involving x, construct the mapping $f : A \to B$, as defined by $e(x)$. It is the user's responsibility to ensure that a value is defined for every $x \in A$. The scope of the variable x is restricted to the map-constructor.

```
map<  A -> B | x :-> e(x), y :-> i(y)  >
```

Given a set or structure A, a set or structure B, a variable x, an expression $e(x)$, usually involving x, a variable y, and an expression $i(y)$, usually involving y, construct the mapping $f : A \to B$, as defined by $x \mapsto e(x)$, with corresponding inverse $f^{-1} : B \to A$, as defined by $y \mapsto i(y)$. It is the user's responsibility to ensure that a value $e(x)$ is defined for every $x \in A$, a value $i(y)$ is defined for every $y \in B$, and that $i(y)$ is the true inverse of $e(x)$. The scope of the variables x and y is restricted to the map-constructor.

16.2.2 Creation of Partial Maps

Partial mappings are quite different to both general mappings and homomorphisms, in that images need not be defined for every element of the domain.

`pmap< A -> B | G >`

Given a finite structure A of cardinality n, a structure B and a subgraph G of $A \times B$, construct the partial map $f : A \to B$, as defined by G. The subgraph G may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter.

`pmap< A -> B | x :-> e(x) >`

Given a set A, a set B, a variable x and an expression $e(x)$, construct the partial map $f : A \to B$, as defined by $e(x)$. This form of the map constructor is a special case of the previous one whereby the image of x can be defined using a single expression. Again the scope of x is restricted to the map-constructor.

`pmap< A -> B | x :-> e(x), y :-> i(y) >`

This constructor is the same as the map constructor above which allows the inverse map $i(y)$ to be specified, except that the result is marked to be a partial map.

16.2.3 Creation of Homomorphisms

The principal construction for homomorphisms consists of the generator image form, where the images of the generators of the domain are listed. Note that the kind of homomorphism and the kind and number of generators for which images are expected, depend entirely on the type of the domain. Moreover, some features of the created homomorphism, e.g. whether checking of the homomorphism is done during creation or whether computing preimages is possible, depend on the types of the domain and the codomain. We refer to the appropriate handbook chapters for further information.

`hom< A -> B | G >`

Given a finitely generated algebraic structure A and a structure B, as well as a graph G of $A \times B$, construct the homomorphism $f : A \to B$ defined by extending the map of the generators of A to all of A. The graph G may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter.

The detailed requirements on the specification are module-dependent, and can be found in the chapter describing the domain A.

`hom< A -> B | y`$_1$`, ..., y`$_n$ `>`

`hom< A -> B | x`$_1$ `-> y`$_1$`, ..., x`$_n$ `-> y`$_n$ `>`

This is a module-dependent constructor for homomorphisms between structures A and B; see the chapter describing the functions for A. In general after the bar the images for all generators of the structure A must be specified.

```
hom<  A -> B | x :-> e(x)  >
```

Given a structure A, a structure B, a variable x and an expression $e(x)$, construct the homomorphism $f : A \to B$, as defined by $e(x)$. This form of the map constructor is a special case of the previous one whereby the image of x can be defined using a single expression. Again the scope of x is restricted to the map-constructor.

```
hom<  A -> B | x :-> e(x), y :-> i(y)  >
```

This constructor is the same as the map constructor above which allows the inverse map $i(y)$ to be specified, except that the result is marked to be a homomorphism.

16.2.4 Coercion Maps

MAGMA has a sophisticated machinery for coercion of elements into structures other than the parent. Non-automatic coercion is usually performed via the `!` operator. To obtain the coercion map corresponding to `!` in a particular instance the `Coercion` function can be used.

```
Coercion(D, C)
```

```
Bang(D, C)
```

Given structures D and C such that elements from D can be coerced into C, return the map m that performs this coercion. Thus the domain of m will be D and the codomain will be C.

16.3 Operations on Mappings

16.3.1 Composition

Although compatible maps can be composed by repeated application, say $g(f(x))$, it is also possible to create a composite map.

```
f * g
```

Given a mapping $f : A \to B$, and a mapping $g : B \to C$, construct the composition h of the mappings f and g as the mapping $h = g \circ f : A \to C$.

```
Components(f)
```

Returns the maps which were composed to form f.

16.3.2 (Co)Domain and (Co)Kernel

The domain and codomain of any map can simply be accessed. Only for some intrinsic maps and for maps with certain domains and codomains, also the formation of image, kernel and cokernel is available.

`Domain(f)`

The domain of the mapping f.

`Codomain(f)`

The codomain of the mapping f.

`Image(f)`

Given a mapping f with domain A and codomain B, return the image of A in B as a substructure of B. This function is currently supported only for some intrinsic maps and for maps with certain domains and codomains.

`Kernel(f)`

Given the homomorphism f with domain A and codomain B, return the kernel of f as a substructure of A. This function is currently supported only for some intrinsic maps and for maps with certain domains and codomains.

16.3.3 Inverse

`Inverse(m)`

The inverse map of the map m.

16.3.4 Function

For a map given by a rule, it is possible to get access to the rule as a user defined function.

`Function(f)`

The function underlying the mapping f. Only available if f has been defined by the user by means of a rule map (i. e., an expression for the image under f of an arbitrary element of the domain).

16.4 Images and Preimages

The standard mathematical notation is used to denote the calculation of a map image. Some mappings defined by certain system intrinsics and constructors permit the taking of preimages. However, preimages are not available for any mapping defined by means of the mapping constructor.

`a @ f`
`f(a)`

Given a mapping f with domain A and codomain B, and an element a belonging to A, return the image of a under f as an element of B.

`S @ f`
`f(S)`

Given a mapping f with domain A and codomain B, and a finite enumerated set, indexed set, or sequence S of elements belonging to A, return the image of S under f as an enumerated set, indexed set, or sequence of elements of B.

`C @ f`
`f(C)`

Given a homomorphism f with domain A and codomain B, and a substructure C of A, return the image of C under f as a substructure of B.

`y @@ f`

Given a mapping f with domain A and codomain B, where f supports preimages, and an element y belonging to B, return the preimage of y under f as an element of A.

If the mapping f is a homomorphism, then a single element is returned as the preimage of y. In order to obtain the full preimage of y, it is necessary to form the coset $K * y@@f$, where K is the kernel of f.

`R @@ f`

Given a mapping f with domain A and codomain B, where f supports preimages, and a finite enumerated set, indexed set, or sequence of elements R belonging to B, return the preimage of R under f as an enumerated set, indexed set, or sequence of elements of A.

`D @@ f`

Given a mapping f with domain A and codomain B, where f supports preimages and the kernel of f is known or can be computed, and a substructure D of B, return the preimage of D under f as a substructure of A.

`HasPreimage(x, f)`

Return whether the preimage of x under f can be taken and the preimage as a second argument if it can.

16.5 Parents of Maps

Parents of maps are structures knowing a domain and a codomain. They are often used in automorphism group calculations where a map is returned from an automorphism group into the set of all automorphisms of some structure. Parents of maps all inherit from the type `PowMap`. The type `PowMapAut` which inherits from `PowMap` is type which the parents of automorphisms inherit from.

There is also a power structure of maps (of type `PowStr`, similar to that of other structures) which is used as a common overstructure of the different parents.

`Parent(m)`

The parent of m.

`Domain(P)`

`Codomain(P)`

The domain and codomain of the maps for which P is the parent.

`Maps(D, C)`

`Iso(D, C)`

The parent of maps (or isomorphisms) from D to C. `Iso` will only return a different structure to `Maps` if it has been specifically implemented for such maps.

`Aut(S)`

The parent of automorphisms of S.

PART III
BASIC RINGS

17 INTRODUCTION TO RINGS

Chapter 17
INTRODUCTION TO RINGS

17.1 Overview

Rings of various kinds form the richest source of algebraic structures in MAGMA. Tables 1 and 2 list the most important types.

symbol	*description*	`Category`	*Ch*
$\mathbf{Z}$	ring of integers	`RngInt`	18
$\mathbf{Z}/m\mathbf{Z}$	ring of residue classes	`RngIntRes`	18
$R[x]$	univariate polynomial ring	`RngUPol`	23
$F[x]/f(x)$	polynomial factor ring	`RngUPolRes`	23
$R[x_1, \cdots, x_m]$	multivariate polynomial ring	`RngMPol`	24
$R[x_1, \cdots, x_m]^G$	invariant ring	`RngInvar`	110
$R[[x]]$	power series ring	`RngSer`	49
O	order in a number field	`RngOrd`	37
O	order in a function field	`RngFunOrd`	42
$\mathbf{Z}_p$	p-adic ring	`RngPad`	47
R_m	local ring	`RngLoc`	47
V	valuation ring	`RngVal`	45

Table 1: The main types of Ring in MAGMA.

symbol	*description*	`Category`	*Ch*
$\mathbf{Q}$	rational field	`FldRat`	20
$\mathbf{F}_q$	finite field	`FldFin`	21
$F(x_1, \cdots, x_m)$	rational function field	`FldFunRat`	41
$F((x))$	field of Laurent series	`RngSerLaur`	49
$\mathbf{Q}(\sqrt{D})$	quadratic number field	`FldQuad`	35
$\mathbf{Q}(\zeta_n)$	cyclotomic number field	`FldCyc`	36
$\mathbf{Q}(\alpha)$	number field	`FldNum`	34
$F(x)(\alpha)$	function field	`FldFun`	42
$\mathbf{Q}_p$	p-adic field	`FldPad`	47
$\mathbf{Q}_p(\alpha)$	local field	`FldLoc`	47
$\mathbf{R}$	real field	`FldRe`	25
$\mathbf{C}$	complex field	`FldCom`	25

Table 2: The main types of Field in MAGMA.

The list of rings in Table 1 is not exhaustive, for two reasons. In the first place, some rings have been categorized differently, because their module structure or algebra structure seems pre-eminent; thus matrix rings and finitely presented algebras appear (more or less arbitrarily) in the Part on Algebras, and vector spaces and their generalizations appear in the Module Part. (Also, rings of class functions appear in the Part on Groups.) Furthermore, certain general constructions (such as `sub`) allow the user to define rings that do not appear in the above list, most notably subrings of **Z**.

Looking at the table it may seem that all rings in MAGMA are commutative and unital. This is not the case (even though it would have made life much easier); since polynomial rings and the like can be defined over any coefficient ring, the matrix rings and finitely presented algebras not listed here that are not generally commutative, allow the construction of non-commutative rings. Furthermore, the `sub` constructor allows the creation of rings without 1; certain functions for the construction of new rings from old ones do not allow such non-unital coefficient rings.

In this Chapter we give an overview of the various types and the relations between them. Moreover, we describe the important principles underlying the rules for coercion of elements of one ring into another. This Chapter also describes the common functions for all types of rings (and their elements), and subsequent Chapters deal with particular categories of rings, as indicated by the table.

17.2 The World of Rings

There are various ways in which to order the families of rings appearing in Table 1. We look at some in this section.

17.2.1 New Rings from Existing Ones

It is important to realize at the outset that to work comfortably with a ring, it should be finitely generated (over some subring); indeed, the only violations of this rule occur for real and complex fields, and p-adic and power series type structures, in which we necessarily have to cope with approximations. All other rings we will label as *exact*.

All rings in Table 1 can be obtained from the ring of rational integers **Z** by repeated application of a handful of fundamental mathematical constructions. The first such construction is *forming fractions*: the rational field **Q** can be obtained as the field of fractions of **Z**. The second construction is that of forming quotients: in this way the rings $\mathbf{Z}/m\mathbf{Z}$ are obtained from **Z**. The third important construction is that of *transcendental extension*: by adjoining an element that satisfies no relation over the coefficient ring, a polynomial ring is obtained. An *algebraic extension* can be obtained by a combination of a transcendental extension and a quotient. Finally, *completion* of a ring at a prime leads in general to the rings that were labelled above as not exact. Some other constructions are: tensoring, taking direct products (leading to tuple modules), and taking valuation rings (an operation inverse to taking fields of fractions).

Most of these constructions are supported by MAGMA. In many situations the `quo` and `ext` constructors will perform the quotient and algebraic extension operations, just like

`sub` creates sub-structures. Note an important distinction: usually `sub` creates structures of exactly the same type as the original structure—this is precisely why the construction of sub-object does not appear as an important construction for creating new objects in the previous paragraph.

Care should be taken not to confuse the mathematical properties of rings (or objects in MAGMA in general) and the properties of the object that MAGMA is aware of. For example, if one creates the ring of residue classes $\mathbf{Z}/p\mathbf{Z}$ for a prime number p, using the command `IntegerRing(p)`, the MAGMA object created is a residue class ring (whose modulus happens to be prime) and *not* a finite field; the functions applicable are the residue class ring functions, and it is, for instance, not possible to create a field extension over this object. If the intention was to create a finite field, the `FiniteField(p)` command should have been used, and for that object it is possible to create a field extension.

Similarly, a convenient way of thinking about a number field $K = \mathbf{Q}(\alpha)$ is to regard it as a quotient of the polynomial ring $\mathbf{Q}[X]$ and the ideal generated by the minimal polynomial f of the primitive element α:

$$K = \mathbf{Q}(\alpha) \cong \mathbf{Q}[X]/(f).$$

This is, however, not the way to create number fields in MAGMA. The quotient ring of a polynomial ring will be an object to which only the generic ring functions apply, whereas to obtain the number field with all the machinery to manipulate it one has to use a command like `NumberField(f)`.

17.2.2 Attributes

17.3 Coercion

A ring element can often be coerced into a ring other than its parent. The need for this occurs for example when one wants to perform a binary ring operation on elements of different structures, or when an intrinsic function is invoked on elements for which it has not been defined.

The basic principle is that such an operation may be performed whenever it makes sense mathematically. Before the operation can be performed however, an element may need to be coerced into some structure in which the operation can legally be performed. There are two types of coercion: *automatic* and *forced* coercion. Automatic coercion occurs when MAGMA can figure out for itself what the target structure should be, and how elements of the originating structure can be coerced into that structure. In other cases MAGMA may still be able to perform the coercion, provided the target structure has been specified; for this type of coercion `R ! x` instructs MAGMA to execute the coercion of element x into ring R.

The precise rules for automatic and forced coercion between rings are explained in the next two subsections. It is good to keep an important general distinction between automatic and forced coercion in mind: whether or not automatic coercion will succeed depends on the originating and the target structure alone, while for forced coercion success

may depend on the particular element as well. Thus, integers can be lifted automatically to the rationals if necessary, but conversely, only the integer elements of **Q** can be coerced into **Z** by using `!`.

The subsections below will describe for specific rings R and S in MAGMA whether or not an element r of R can be lifted automatically or by force into S. Suppose that the unary MAGMA function `Function` takes elements of the type of S as argument and one is interested in the result of that function when applied to r. If R can be coerced automatically into a unique structure S of the desired type, then `Function(r)` will produce the required result. If R cannot be coerced automatically into such S, but forced coercion on r is possible, then `Function(S ! r)` will yield the desired effect. If, finally, neither automatic nor forced coercion is possible, it may be possible to define a map m from R to S, and then `Function( m(r) )` will give the answer.

For example, the function `Order` is defined for elements of residue class rings (among others). But `Order(3)` has no obvious interpretation (and an error will arise) because there is not a unique residue class ring in which this should be evaluated.

If a binary operation $\circ : C \times C \to C$ on members C of a category of rings $\mathcal{C}$ is applied to elements r and s of members R and S from $\mathcal{C}$, the same rules for coercion will be used to determine the legality of $r \circ s$. If s can be coerced automatically into R, then this will take place and $r \circ s$ will be evaluated in R; otherwise, if r can be coerced automatically into S, then $r \circ s$ will be evaluated in S. If neither succeeds, then, in certain cases, MAGMA will try to find an existing *common overstructure* T for R and S, that is, an object T from $\mathcal{C}$ such that $\subset T \supset S$; then both r and s will be coerced into T and the result $t = r \circ s$ will be returned as an element of T. If none of these cases apply, an error results. It may still be possible to evaluate by forced coercion of r into S or s into R, using `(S ! r) o s` or using `r o (R ! s)`.

17.3.1 Automatic Coercion

We will first deal with the easier of the two cases: automatic coercion. A simple demonstration of the desirability of automatic coercion is given by the following example:

```
print 1 + (1/2);
```

It is obvious that one wants the result to be 3/2: we want to identify the integer 1 with the rational number 1 and perform the addition in **Q**, that is, we clearly wish to have automatic coercion from **Z** to **Q**.

The basic rule for automatic coercion is:

> *automatic coercion will only take place when there exists a unique target structure and an obvious homomorphism from the parent structure to the target structure*

In particular, if one structure is naturally contained in the other (and MAGMA knows about it!), automatic coercion will take place. (The provision that MAGMA must know about the embedding is in particular relevant for finite fields and number fields; in these cases it is possible to create subrings, or even isomorphic rings/fields, for which the embedding is not known.)

Also, for any ring R there is a natural ring homomorphism $\mathbf{Z} \to R$, hence any integer can be coerced automatically into any ring.

Table 3 gives a summary for all cases in which MAGMA will apply automatic coercion: if a ring operation is attempted on an element from one of the structures in the first row, and the specifications for the operation require that the argument is an element from a structure in the first column, the element will be coerced into the structure indicated by the table.

Automatic Coercion (Ring Elements)										
	$\mathbf{F}_{p^s}$	$\mathbf{Z}/n\mathbf{Z}$	$\mathbf{Z}$	$\mathbf{Q}$	$\mathbf{Q}(\sqrt{\Delta_2})$	$\mathbf{Q}(\zeta_n)$	L	O_L	$\mathbf{R}_n$	$\mathbf{C}_n$
$\mathbf{F}_{p^r}$	$\subset$	–	$\mathbf{F}_{p^r}$	–	–	–	–	–	–	–
$\mathbf{Z}/m\mathbf{Z}$	–	=	$\mathbf{Z}/m\mathbf{Z}$	–	–	–	–	–	–	–
$\mathbf{Z}$	$\mathbf{F}_{p^s}$	$\mathbf{Z}/n\mathbf{Z}$	$\mathbf{Z}$	$\mathbf{Q}$	$\mathbf{Q}(\sqrt{\Delta_2})$	$\mathbf{Q}(\zeta_n)$	L	O_L	$\mathbf{R}_n$	$\mathbf{C}_n$
$\mathbf{Q}$	–	–	$\mathbf{Q}$	$\mathbf{Q}$	$\mathbf{Q}(\sqrt{\Delta_2})$	$\mathbf{Q}(\zeta_n)$	L	–	$\mathbf{R}_n$	$\mathbf{C}_n$
$\mathbf{Q}(\sqrt{\Delta_1})$	–	–	$\mathbf{Q}(\sqrt{\Delta_1})$	$\mathbf{Q}(\sqrt{\Delta_1})$	=	–	–	–	–	–
$\mathbf{Q}(\zeta_m)$	–	–	$\mathbf{Q}(\zeta_m)$	$\mathbf{Q}(\zeta_m)$	–	$\mathbf{Q}(\zeta_{\mathrm{lcm}(m,n)})$	–	–	–	–
K	–	–	K	K	–	–	=	$K = L$	–	–
O_K	–	–	O_K	–	–	–	$K = L$	=	–	–
$\mathbf{Q}_p$	–	–	$\mathbf{Q}_p$	$\mathbf{Q}_p$	–	–	–	–	–	–
$\mathbf{Z}_p$	–	–	$\mathbf{Z}_p-$	–	–	–	–	–	–	–
$\mathbf{R}_m$	–	–	$\mathbf{R}_m$	$\mathbf{R}_m$	–	–	–	–	$\mathbf{R}_{\max(m,n)}$	$\mathbf{C}_{\max(m,n)}$
$\mathbf{C}_m$	–	–	$\mathbf{C}_m$	$\mathbf{C}_m$	–	–	–	–	$\mathbf{C}_{\max(m,n)}$	$\mathbf{C}_{\max(m,n)}$

Table 3.

The symbols in the table have the following meaning:

– indicates that automatic coercion will *not* take place; as the table shows for instance, automatic coercion will not take place when we try to add a finite field element and an element of $\mathbf{Z}/n\mathbf{Z}$ (not even when n is prime and of the same characteristic as the field).

$\subset$ indicates that automatic coercion will *only* take place if one structure is contained in the other, or MAGMA can find a common overstructure to both structures. Thus the top-left entry in the table indicates that two finite field elements can be added if they are members of finite fields F_1 and F_2 such that $F_1 \subset F_2$ or $F_2 \subset F_1$, or both fields have been created inside a field F, so $F_1 \subset F \supset F_2$.

= The = is used to denote that automatic coercion only takes place if both structures are the same. The entry $K = L$ is used to indicate that an element of an order O_K will only be coerced into the number field L if $K = L$.

In addition to these rules, general rules apply to polynomial and matrix algebras. The rules for polynomial rings are as follows. An element s from a ring S can be automatically coerced into $R[X_1, \ldots, X_n]$ if either S equals $R[X_1, \ldots, X_i]$ for some $1 \leq i \leq n$, or $S = R$. Note that in the latter case the element s must be an element of the coefficient ring R, and that it is not sufficient for it to be coercible into R.

So, for example, we can add an integer and a polynomial over the integers, we can add an element f of $\mathbf{Z}[X]$ and g of $\mathbf{Z}[X,Y]$, but *not* an integer and a polynomial over the rationals.

An element s can be coerced automatically in the matrix ring $M_{n,n}(R)$ if it is coercible automatically into the coefficient ring R, in which case s will be identified with the diagonal matrix that has each diagonal entry equal to s.

So we can add an integer and a matrix ring element over the rationals, but we cannot automatically add elements of $M_{n,n}(\mathbf{Z})$ and $M_{n,n}(\mathbf{Q})$, nor elements from $M_{2,2}(\mathbf{Z})$ and $M_{3,3}(\mathbf{Z})$.

17.3.2 Forced Coercion

In certain cases where automatic coercion will not take place, one can cast an element into the ring in which the operation should take place.

If, for example, one is working in a ring $\mathbf{Z}/p\mathbf{Z}$, and p happens to be prime, it may occur that one wishes to perform some finite field operations on an element in the ring; if F is a finite field of characteristic p an element x of $\mathbf{Z}/p\mathbf{Z}$ can be cast into an element of F using

```
F ! x;
```

Table 4 describes the possibilities for using `!` for coercion in rings. It shows when it is possible to coerce an element from a structure in the first row into a structure in the first column.

! Non-Automatic Coercion (Ring Elements)										
↙	$\mathbf{F}_{p^s}$	$\mathbf{Z}/n\mathbf{Z}$	$\mathbf{Z}$	$\mathbf{Q}$	$\mathbf{Q}(\sqrt{\Delta_2})$	$\mathbf{Q}(\zeta_n)$	L	O_L	$\mathbf{R}_n$	$\mathbf{C}_n$
$\mathbf{F}_{p^r}$	$s\|r$ or $\ni$	$n=p$	+	−	−	−	−	−	−	−
$\mathbf{Z}/m\mathbf{Z}$	$m=p, s=1$	$m\|n$	+	−	−	−	−	−	−	−
$\mathbf{Z}$	$s=1$ or $\ni$	+	+	$\ni$	$\ni$	$\ni$	$\ni$	$\ni$	−	−
$\mathbf{Q}$	−	−	+	+	$\ni$	$\ni$	$\ni$	$\ni$	−	−
$\mathbf{Q}(\sqrt{\Delta_1})$	−	−	+	+	$\Delta_1=\Delta_2$ or $\ni$	$\ni$	−	−	−	−
$\mathbf{Q}(\zeta_m)$	−	−	+	+	$\supset$ or $\ni$	$n\|m$ or $\ni$	−	−	−	−
K	−	−	+	+	−	−	$K=L$	$K=L$	−	−
O_K	−	−	+	$\ni$	−	−	$K=L, \ni$	$K=L$	−	−
$\mathbf{Q}_p$	$s=1$	$n=p$	+	$\ni$	−	−	−	−	−	−
$\mathbf{Z}_p$	$s=1$	$n=p$	+	−	−	−	−	−	−	−
$\mathbf{R}_m$	−	−	+	+	$\Delta_2>0$ or $\ni$	$\ni$	−	−	+	$\ni$
$\mathbf{C}_m$	−	−	+	+	+	+	−	−	+	+

Table 4.

The symbols are the same as those in Table 3 except:

+ indicates that coercion can take place without restrictions (sometimes it will be done automatically if necessary);

|, = (In)equalities on parameters of the structures indicate the restrictions for `!` to work; thus, an element from $\mathbf{F}_{p^s}$ can only be coerced into $\mathbf{Z}/m\mathbf{Z}$ if $s=1$ and $m=p$;

$\ni$ The $\ni$ symbols in this table indicates that coercion only applies to certain elements of the domain; thus only those elements of **Q** can be coerced into **Z** that are in fact integers.

or In some cases coercion may either take place if some condition on parameters is satisfied *or* on a subset of the domain; thus the entry $s|r$ or $\ni$ for coercion of $\mathbf{F}_{p^s}$ into $\mathbf{F}_{p^r}$ indicates that such coercion is always possible if s divides r, and only on a subset of $\mathbf{F}_{p^s}$ (like $\mathbf{F}_p$) in general (note that the characteristics have to be the same).

The rules for coercion from and to polynomial rings and matrix rings are as follows.

If an attempt is made to forcibly coerce s into $P = R[X_1, \ldots, X_n]$, the following steps are executed successively:

(a) if s is an element of P it remains unchanged;

(b) if s is a sequence, then the zero element of P is returned if s is empty, and if it is non-empty but the elements of the sequence can be coerced into $P[X_1, \ldots, X_{n-1}]$ then the polynomial $\sum_j s[j]X_n^{j-1}$ is returned;

(c) if s can be coerced into the coefficient ring R, then the constant polynomial s is returned;

(d) if s is a polynomial in $R[X_1, \ldots, X_k]$ for some $1 \leq k \leq n$, then it is lifted in the obvious way into P;

(e) if s is a polynomial in $R[X_1, \ldots, X_k]$ for some $k > n$, but constant in the indeterminates $X_{n+1}, \ldots, X_k$, then s is projected down in the obvious way to P.

If none of these steps successfully coerces s into P, an error occurs.

The ring element s can be coerced into $M_{n,n}(R)$ if either it can be coerced into R (in which case s will be identified with the diagonal matrix with s on the diagonal), or $s \in S = M_{n,n}(R')$, where R' can be coerced into R. Also a sequence of n^2 elements coercible into R can be coerced into the matrix ring $M_{n,n}(R)$.

Elements from a matrix ring $M_{n,n}(R)$ can only be coerced into rings other than a matrix ring if $n = 1$; in that case the usual rules for the coefficient ring R apply.

Note that in some cases it is possible to go from (a subset of) some structure to another in two steps, but not directly: it is possible to go

```
> y := L ! (Q ! x)
```

to coerce a rational element of one number field into another via the rationals.

Finally we note that the binary Boolean operator `in` returns true if and only if forced coercion will be successful.

17.4 Generic Ring Functions

The generic functions described in this Chapter apply in principle to every type of ring. For certain rings these are the only applicable functions. The qualification 'in principle' in the first sentence is made because for some classes of rings an algorithm to compute certain of these functions does not exist, or has not been implemented. In that case an error will result.

This general list is provided primarily to avoid duplication of common descriptions. In the following Chapters the generic functions will be listed merely without further description, and the emphasis can be on the functions specific to a particular type of ring.

17.4.1 Related Structures

`Parent(R)`

The parent of ring R. Currently this returns the *power structure* of the ring.

`Category(R)`

`Type(R)`

The 'type' of R, that is, the MAGMA category to which the ring R belongs. The procedure call `ListCategories()` gives a list of all the categories, as does the Appendix.

`PrimeField(F)`

For a field F, this returns either $\mathbf{F}_p$, if the characteristic p of F is positive, or $\mathbf{Q}$, if the characteristic of F is 0. If F is an extension field then it will return the field at the bottom of the extension tower.

`PrimeRing(R)`

For a unitary ring R, this returns either $\mathbf{Z}/n\mathbf{Z}$, if the characteristic n of R is positive, or $\mathbf{Z}$, if the characteristic of R is 0. If R is an extension ring then it will return the ring at the bottom of the extension tower.

`Centre(R)`

`Center(R)`

Given a ring R, return its centre, consisting of the subring of elements commuting with all other elements of R.

17.4.2 Numerical Invariants

`Characteristic(R)`

The characteristic of the ring R, which is the smallest positive integer m such that $m \cdot r = 0$ for every $r \in R$, or zero if such m does not exist.

`#R`

The cardinality of the ring R; here R must be finite.

17.4.3 Predicates and Boolean Operations

`IsCommutative(R)`

Returns `true` if it is known that the ring R is commutative, `false` if it is known that R is not commutative. An error results if the answer is not known.

`IsUnitary(R)`

Returns `true` if the ring R is known to be unitary (that is, if R has a multiplicative identity), `false` if R has no 1.

`IsFinite(R)`

Returns `true` if the ring R is known to be a finite ring, `false` if it is known to be infinite. An error results if the answer is not known.

`IsOrdered(R)`

Returns `true` if the ring R has a total ordering defined on the set of its elements, `false` otherwise.

`IsField(R)`

Returns `true` if the ring R is known to be a field, `false` if it is known to not be a field. An error results if the answer is not known.

`IsDivisionRing(R)`

Returns `true` if the ring R is known to be a division ring (that is, every non-zero element is invertible), `false` if it is known that R is not a division ring. An error results if the answer is not known.

`IsEuclideanDomain(R)`

Returns `true` if the ring R is known to be a euclidean domain, `false` if it is known that R is not a euclidean domain. An error results if the answer is not known.

`IsEuclideanRing(R)`

Returns `true` if the ring R is known to be euclidean, `false` if it is known that R is not euclidean. An error results if the answer is not known.

`IsMagmaEuclideanRing(R)`

Returns `true` iff the ring R is a computable euclidean ring within Magma (i.e., iff the necessary euclidean operations are defined for R so algorithms requiring a euclidean ring will work).

`IsPID(R)`

`IsPrincipalIdealDomain(R)`

Returns `true` if the ring R is known to be a principal ideal domain, `false` if it is known that R is not a principal ideal domain. An error results if the answer is not known.

`IsPIR(R)`

`IsPrincipalIdealRing(R)`

Returns `true` if the ring R is known to be a principal ideal ring, `false` if it is known that R has non-principal ideals. An error results if the answer is not known.

`IsUFD(R)`

`IsUniqueFactorizationDomain(R)`

Returns `true` if the ring R is known to be a unique factorization domain, `false` if it is known that R is not a unique factorization domain. An error results if the answer is not known.

`IsDomain(R)`

`IsIntegralDomain(R)`

Returns `true` if it is known that R is an integral domain (i. e., R has no zero divisors), `false` if R is known to have zero divisors. An error results if the answer is not known.

`HasGCD(R)`

Returns `true` iff there is a GCD algorithm for elements of ring R in MAGMA.

`R eq S`

For certain pairs R, S of rings, this returns `true` if R and S refer to the same ring, and `false` otherwise. However, if R and S belong to different categories an error may result.

`R ne S`

For certain pairs R, S of rings, this returns `true` if R and S refer to different rings, and `false` otherwise. However, if R and S belong to different categories an error may result.

17.5 Generic Element Functions

17.5.1 Parent and Category

`Parent(r)`

The (default) parent ring of ring element r. Usually the parent of r has been created explicitly before, but in certain cases, such as literal integers, rationals, reals, and values returned by certain functions a default parent is created in the background.

`Category(r)`

`Type(r)`

The 'type' of r, that is, the MAGMA category to which the ring element r belongs. The procedure call `ListCategories()` gives a list of all the categories, as does the Appendix.

17.5.2 Creation of Elements

`Zero(R)`

The zero element of ring R; this is equivalent to `R ! 0`.

`One(R)`

`Id(R)`

The multiplicative identity 1 of ring R; this is equivalent to `R ! 1`.

`R ! a`

Coerce the element a of some ring into the ring R. (The rules on coercion are explained earlier in this Chapter.) If a is an integer, the coercion will always succeed: the element $a \cdot 1_R$ will be returned, where 1_R is the unit element of R.

`Random(R)`

A random element of the finite ring R (every element of R has the same probability of being returned).

`Representative(R)`

`Rep(R)`

A representative element of the finite ring R.

17.5.3 Arithmetic Operations

`+a`

Element a.

`-a`

The negation (additive inverse) of element a.

`a + b`

The sum of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the sum can be taken.

`a - b`

The difference of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the difference can be taken.

`a * b`

The product of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the product can be taken.

`a ^ k`

Form the k-th power of the ring element a, for small, non-negative, k. If $a = 0$ then we must have $k > 0$.

`a ^ -k`

Form the k-th power of the multiplicative inverse of the unit a.

`a / b`

Given an element a of R and a unit b of R, form the quotient of the elements a and b. If b is not invertible in R, an error results, unless both a and b are integers, in which case `a / b` returns the rational number a/b. If a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the quotient can be taken.

`a +:= b`

Mutation assignment: change a into the sum of a and b.

`a -:= b`

Mutation assignment: change a into the difference of a and b.

`a *:= b`

Mutation assignment: change a into the product of a and b.

`a /:= b`

Mutation assignment: change a into the quotient of a and b.

`a ^:= k`

Mutation assignment: change a into the power a^k.

17.5.4 Equality and Membership

`a eq b`

Returns `true` if the elements a and b of R are the same, otherwise `false`.

`a ne b`

Returns `true` if the elements a and b of R are distinct, otherwise `false`.

`R eq S`

Returns `true` if the rings R and S are the same, otherwise `false`.

`R ne S`

Returns `true` if the rings R and S are distinct, otherwise `false`.

`a in R`

Returns `true` if and only if a is an element of R.

`a notin R`

Returns `true` if and only if a is not an element of R.

17.5.5 Predicates on Ring Elements

`IsZero(a)`

Returns `true` if and only if the element a of R equals 0_R.

`IsOne(a)`

Returns `true` if and only if the element a of R equals 1_R.

`IsMinusOne(a)`

Returns `true` if and only if the element a of R equals the element -1 of R.

`IsUnit(a)`

Returns `true` if a is a unit in its parent R, `false` otherwise.

`IsIdempotent(x)`

Returns `true` if and only if x^2 equals x.

`IsNilpotent(x)`

Returns `true` if and only if some integer power x^i of x is zero.

`IsZeroDivisor(x)`

Returns `true` if and only if x is a zero-divisor, that is, there exists an element y in the parent R of x such that $xy = 0$.

`IsIrreducible(x)`

Returns `true` if and only if the parent R of the element x is a domain and x is irreducible in R, that is, x is a non-unit of R and whenever a product ab of elements of R divides x then a or b is a unit of R.

`IsPrime(x)`

Returns `true` if and only if the parent R of the element x is a domain and x is a prime element of R, that is, x is neither 0 nor a unit and whenever x divides the product ab of two elements of R it divides a or b.

17.5.6 Comparison of Ring Elements

The comparison operations are only defined on types of ring that are ordered.

`a gt b`

Returns `true` if the ring element a is greater than the ring element b, otherwise `false`.

`a ge b`

Returns `true` if the ring element a is greater than or equal to the ring element b, otherwise `false`.

`a lt b`

Returns `true` if the ring element a is less than the ring element b, otherwise `false`.

`a le b`

Returns `true` if the ring element a is less than or equal to the ring element b, otherwise `false`.

`Maximum(a, b)`

The maximum of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the maximum can be taken.

`Maximum(Q)`

The maximum of the sequence Q of ring elements.

`Minimum(a, b)`

The minimum of the ring elements a and b; if a and b do not belong to the same ring R, an attempt will be made to find a common overstructure in which the minimum can be taken.

`Minimum(Q)`

The minimum of the sequence Q of ring elements.

17.6 Ideals and Quotient Rings

The following entries describe the operations on ideals in a *commutative ring* R. Certain operations on left and right ideals in non-commutative rings will be described in the Chapters for the corresponding rings.

17.6.1 Defining Ideals and Quotient Rings

`ideal< R | a1, ..., ar >`

Given a ring R and elements $a_1, \ldots, a_r$ of R, create the ideal I of R generated by $a_1, \ldots, a_r$.

`quo< R | ar, ..., ar >`

Given a ring R and elements $a_1, \ldots, a_r$ of R, construct the quotient ring $Q = R/I$, where I is the ideal of R generated by $a_1, \ldots, a_r$.

`R / I`

Given a ring R and an ideal I of R, construct the quotient ring $Q = R/I$, as well as the canonical map $R \to R/I$.

`PowerIdeal(R)`

The set of ideals of R. This is the parent of all ideals of R.

17.6.2 Arithmetic Operations on Ideals

`I + J`

The sum of the ideals I and J of the ring R. This ideal consists of elements $a + b$, with $a \in I$ and $b \in J$. If I is generated by $\{a_1, \ldots, a_k\}$ and J is generated by $\{b_1, \ldots, b_m\}$, then $I + J$ is generated by $\{a_1, \ldots, a_k, b_1, \ldots, b_m\}$.

`I * J`

The product of the ideals I and J of the ring R. This is the ideal generated by elements $a \cdot b$, with $a \in I$ and $b \in J$, and it consists of elements $a_1 b_1 + \cdots + a_n b_n$, with $a_i \in I$ and $b_j \in J$.

`I meet J`

The intersection of the ideals I and J of the ring R.

17.6.3 Boolean Operators on Ideals

Throughout this subsection I and J are ideals belonging to the same integer ring R, while a is an element of R.

`a in I`

Returns `true` if and only if the element a is a member of the ideal I.

`a notin I`

Returns `true` if and only if the element a is not a member of the ideal I.

`I eq J`

Returns `true` if and only if the ideals I and J are equal.

`I ne J`

Returns `true` if and only if the ideals I and J are distinct.

`I subset J`

Returns `true` if and only if the ideal I is contained in the ideal J.

`I notsubset J`

Returns `true` if and only if the ideal I is not contained in the ideal J.

17.7 Other Ring Constructions

MAGMA allows the construction of residue fields, localization of rings, and completion of rings. These constructions really just create appropriate rings of different categories within MAGMA.

17.7.1 Residue Class Fields

`ResidueClassField(I)`

Given a maximal ideal I of a ring R, create the residue class field K of the quotient ring R/I, together with a map sending an element of R to the corresponding element of K.

17.7.2 Localization

`loc< R | a`$_1$`, ..., a`$_r$ `>`

Given a ring R and elements $a_1, \ldots, a_r$ of R, which generate a prime ideal P of R, create the localization L of R at P, together with a map sending an element of R to the corresponding element of L.

`Localization(R, P)`

Given a ring R and a prime ideal P of R, create the localization L of R at P, together with a map sending an element of R to the corresponding element of L.

17.7.3 Completion

`comp< R | a1, ..., ar >`

Given a ring R and elements $a_1, \ldots, a_r$ of R, which generate a prime ideal or zero ideal P of R, create the completion C of R at P, together with a map sending an element of R to the corresponding element of C.

`Completion(R, P)`

Given a ring R and a prime ideal or zero ideal P of R, create the completion C of R at P, together with a map sending an element of R to the corresponding element of C.

17.7.4 Transcendental Extension

`ext< R | >`

Given a ring R create the univariate transcendental extension $R[x]$ of R. This is equivalent to `PolynomialRing(R)`.

`ext< R, n | >`

Given a ring R and an integer $n \geq 1$, create the multivariate transcendental extension $R[x_1, \ldots, x_n]$ of R. This is equivalent to `PolynomialRing(R, n)`.

18 RING OF INTEGERS

Chapter 18
RING OF INTEGERS

18.1 Introduction

This Chapter describes the operators and functions for working with the ring of rational integers **Z**.

Integers are the most commonly used objects in MAGMA. They can be created by just typing in the literal (decimal) digits. Integers thus created are elements of the ring of integers which is automatically created when MAGMA is started up. There is just one single object 'integer ring' around, but references to it (new 'names' for it) can be created using the `IntegerRing` function.

18.1.1 Representation

Since large integers occur so frequently, the first requirement for a computer algebra system is to support fast arithmetic for integers of arbitrary size. Indeed, within the bounds set by the available memory, it is possible to operate reasonably efficiently with integers of any number of decimal digits.

Although it is well possible to use the integer facilities without being aware of the internal representation of (large) integers, it is sometimes useful to know how integers are stored. The most important fact is that integers smaller than $2^{30} = 1073741824$ in absolute value are 'single precision', and in many circumstances such 'small integers' allow considerably faster arithmetic (they are treated slightly differently internally and escape the overhead of memory management used to deal with multi-precision integers).

18.1.2 Coercion

Integers will be automatically coerced into almost every unitary ring R using the identification of 1 and 1_R. This means that integer arguments are allowed for almost any ring element function, and that it is not necessary to convert an integer before applying binary operators (such as +) on a combination of arguments consisting of an integer and another ring element.

For more on coercion we refer to Chapter 17.

18.1.3 Homomorphisms

Ring homomorphisms are required to be unitary. Therefore, to specify a homomorphism with the integers as its domain requires merely the specification of the codomain.

```
hom<  Z -> R |  >
```

The natural homomorphism from **Z** to the ring R.

Example H18E1

```
> h := hom< Integers() -> MatrixRing(RealField(12), 3) | >;
> h(2)^-1;
[0.5   0   0]
[  0 0.5   0]
[  0   0 0.5]
```

18.2 Creation Functions

18.2.1 Creation of Structures

The ring of integers is automatically created when Magma is first loaded. The ring may be formally created (and, if desired, assigned to a variable) using the function `IntegerRing()`. Subrings of **Z** are always ideals; see the section on ideals for details.

`IntegerRing()`

`Integers()`

`IntegerRing(Q)`

`RingOfIntegers(Q)`

Create the ring of integers **Z**. Analogous to the creation of the ring of integers of any number field, there is a version of `IntegerRing` that creates **Z** as the ring of integers of **Q**.

18.2.2 Creation of Elements

Since the ring of integers is present when Magma is started up, integers typed into Magma without any explicit context will be regarded as elements of the ring of integers. Integers can be specified using both decimal and hexadecimal notation.

`a`$_1$`a`$_2$`...a`$_r$

Given a succession of decimal digits $a_1, \ldots, a_r$, create the corresponding integer. Leading zeros will be ignored.

`0xa`$_1$`a`$_2$`...a`$_r$

Given a succession of hexadecimal digits $a_1, \ldots, a_r$, create the corresponding integer. Leading zeros will be ignored.

`elt< Z | a`$_1$`a`$_2$`...a`$_r$ `>`

Given a succession of decimal digits $a_1, \ldots, a_r$, create the corresponding integer as an element of Z.

`elt< Z | 0xa1a2...ar >`

Given a succession of hexadecimal digits $a_1, \ldots, a_r$, create the corresponding integer as an element of Z.

`Z ! a`
`Z ! [a]`

Coerce the ring element a into the ring of integers Z. The element a is allowed to be an element of the ring of integers modulo m (in which case the result r satisfies $0 \leq r < m$), or an element of a finite field (in which case the result r satisfies $0 \leq r < p$ if a is in the prime field, of characteristic p, and an error otherwise), or an element of the integers, rationals, a quadratic field, a cyclotomic field or a number field (in which cases the result is the obvious integer if a is integral and an error otherwise).

Example H18E2

```
> Z := IntegerRing();
> n := 1234567890;
> n in Z;
true
> m := elt< Z | 1234567890 >;
> m eq n;
true
> l := Z ! elt< QuadraticField(3) | 1234567890, 0>;
> l;
1234567890
> k := elt< Z | 0x499602D2 >;
1234567890
```

`One(Z)` `Identity(Z)`
`Zero(Z)` `Representative(Z)`

These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively, in the integer ring **Z**.

18.2.3 Printing of Elements

Magma supports the printing of integers in both decimal and hexadecimal form. The default print method is to print integers in base 10; base 16 printing is performed using the `Hex` print level.

Example H18E3

```
> n := 1234567890;
> n;
1234567890
> n:Hex;
0x499602D2
```

18.2.4 Element Conversions

`FactorizationToInteger(s)`

`FactorisationToInteger(s)`

`Facint(s)`

Given a sequence of two-element tuples $s = [< p_1, k_1 >, ..., < p_r, k_r >]$ containing pairs of integers $< p_i, k_i >$, $1 \leq i \leq r$, with k_i non-negative, this function returns the integer $p_1^{k_1} \cdots p_r^{k_r}$. It is normally used for converting a factorization sequence to the corresponding integer.

`IntegerToSequence(n, b)`

`Intseq(n, b)`

Given a non-negative integer n and a positive integer $b \geq 2$, return the unique base b representation of n in the form of a sequence Q. That is, if $n = a_0 b^0 + a_1 b^1 + \ldots + a_{k-1} b^{k-1}$ with $0 \leq a_i < b$ and $a_{k-1} > 0$, then $Q = [a_0, a_1, \ldots, a_{k-1}]$. (If $n = 0$, then $Q = [\,]$.)

`SequenceToInteger(s, b)`

`Seqint(s, b)`

Given a positive integer $b \geq 2$ and a sequence $Q = [a_0, \ldots, a_{k-1}]$ of non-negative integers such that $0 \leq a_i < b$, return the integer $n = a_0 b^0 + a_1 b^1 + \ldots + a_{k-1} b^{k-1}$. If Q is the empty sequence, the integer zero is returned. This function performs the inverse operation of the base b representation.

`IntegerToString(n)`

Create the string consisting of the decimal digits of the integer n. In the case in which n is negative the first character will be the minus sign.

`IntegerToString(n, b)`

Create the string consisting of the digits of the integer n in base b. In the case in which n is negative the first character will be the minus sign. The base b can be between 2 and 36. For $b \leq 10$, the digits are represented numerically. For $b > 10$, the digits are represented both numerically and alphabetically, so that, 10 is 'A', 11 is 'B', et cetera.

`Eltseq(n)`

The sequence $[n]$ which can be coerced back into $\mathbf{Z}$.

`Denominator(n)`

The denominator of n, ie. 1.

18.3 Structure Operations

The following generic ring functions are applicable to the ring of integers and its elements.

18.3.1 Related Structures

`Category(Z)` `Parent(Z)` `PrimeRing(Z)` `Center(Z)`

`AdditiveGroup(Z)`

Create the abelian group of integers under addition. This returns an infinite (additive) abelian group A of rank 1 together with a map from A to the ring of integers Z, sending $A.1$ to 1.

`MultiplicativeGroup(Z)`
`UnitGroup(Z)`

Create the abelian group of invertible integers, that is, an abelian group isomorphic to the multiplicative subgroup $\langle -1 \rangle$. This returns an (additive) abelian group A of order 2 together with a map from A to the ring of integers Z, sending $A.1$ to -1.

`ClassGroup(Z)`

The class group of the ring of $\mathbf{Z}$ (which is trivial).

`FieldOfFractions(Z)`

Create the field of fractions $\mathbf{Q}$ of the ring of rational integers.

`sub< Z | n >`

Given Z, the ring of integers or an ideal of it, and an element n of Z, create the ideal $aZ \cap Z$ of the ring of integers. Note that this creates an ideal, not just a subring.

18.3.2 Numerical Invariants

```
Characteristic(Z)
```

```
Signature(Z)
```

The signature of **Z** as an order of **Q**, i.e. `1, 0`.

18.3.3 Ring Predicates and Booleans

```
IsCommutative(Z)   IsUnitary(Z)
IsFinite(Z)   IsOrdered(Z)
IsField(Z)   IsEuclideanDomain(Z)
IsPID(Z)   IsUFD(Z)
IsDivisionRing(Z)   IsEuclideanRing(Z)
IsPrincipalIdealRing(Z)   IsDomain(Z)
Z eq R   Z ne R
```

18.4 Element Operations

18.4.1 Arithmetic Operations

Magma includes both the Karatsuba algorithm and the Schönhage-Strassen FFT-based algorithm for the multiplication of integers ([AHU74, Chap. 7], [vzGG99, Sec. 8.3]). The crossover point (where the FFT method beats the Karatsuba method) is currently 2^{15} bits (approx. 10000 decimal digits) on Sun SPARC workstations and 2^{17} bits (approx. 40000 decimal digits) on Digital Alpha workstations. Assembler macros are used for critical operations and 64-bit operations are used on DEC-Alpha machines.

Magma also contains an asymptotically-fast integer (and polynomial) division algorithm which reduces division to multiplication with a constant scale factor that is in the practical range. Thus division of integers and polynomials are based on the Karatsuba and Schönhage-Strassen (FFT) methods when applicable. The crossover point for integer division (when the new method outperforms the classical method) is currently at the point of dividing a 2^{12} bit (approx. 1200 decimal digit) integer by a 2^{11} (approx. 600 decimal digit) integer on Sun SPARC workstations.

`+ n` `- n`

`m + n` `m - n` `m * n` `n ^ k` `m / n`

`m +:= n` `m -:= n` `m *:= n` `m /:= n` `m ^:= k`

`n div m`

The quotient q of the division with remainder $n = qm + r$, where $0 \leq r < m$ or $m < r \leq 0$ (depending on the sign of m), for integers n and $m \neq 0$.

`n mod m`

The remainder r of the division with remainder $n = qm + r$, where $0 \leq r < m$ or $m < r \leq 0$ (depending on the sign of m), for integers n and $m \neq 0$.

`ExactQuotient(n, d)`

Assuming that the integer n is exactly divisible by the integer d, return the exact quotient of n by d (as an integer). An error results if d does not divide n exactly.

`n div:= m` `n mod:= m`

18.4.2 Bit Operations

The following functions use bit operations on the internal representation, so are in general quicker than using the usual arithmetic operators.

`ShiftLeft(n, b)`

Given integers n and b, with $b \geq 0$, return $n \times 2^b$.

`ShiftRight(n, b)`

Given integers n and b, with $b \geq 0$, return n `div` 2^b.

`ModByPowerOf2(n, b)`

Given integers n and b, with $b \geq 0$, return $n \bmod 2^b$ (so the result is always non-negative).

18.4.3 Equality and Membership

`m eq n` `m ne n`

`n in R` `n notin R`

18.4.4 Parent and Category

`Parent(n)` `Category(n)`

18.4.5 Predicates on Ring Elements

`IsEven(n)`

Returns `true` if the integer n is even, otherwise `false`.

`IsOdd(n)`

Returns `true` if the integer n is odd, otherwise `false`.

`IsDivisibleBy(n, d)`

Returns `true` if and only if the integer n is divisible by the integer d; if `true`, the quotient of n by d is also returned.

`IsSquare(n)`

Returns `true` if the non-negative integer n is the square of an integer, `false` otherwise. If n is a square, its positive square root is also returned.

`IsSquarefree(n)`

Returns `true` if the non-zero integer n is not divisible by the square of any prime, `false` otherwise.

`IsPower(n)`

If the integer $n > 1$ is a power $n = b^k$ of an integer b, with $k > 1$, this function returns true, the minimal positive b and its associated k; if it is not such integer power the function returns false.

`IsPower(n, k)`

If the integer $n > 1$ is k-th power, with $k > 1$, of some integer b, so that $n = b^k$, this function returns true, and b; if it is not a k-th integer power the function returns false.

`IsPrime(n)`

`Proof` BoolElt *Default :* `true`

Returns `true` if and only if the integer n is a prime. A rigorous primality test which returns a proven result will be used unless the parameter `Proof` is `false`. The reader is referred to the section 18.9 for a complete description of this function.

Example H18E4

In this example we find some 10-digit primes that are congruent to 3 modulo 4 such that $(p-1)/2$ is also prime.

```
> { p : p in [10^10+3..10^10+1000 by 4] |
>       IsPrime(p) and IsPrime((p-1) div 2)};
{ 10000000259, 10000000643 }
```

`IsIntegral(n)`

Returns `true` if and only if a is integral, which is of course true for every integer n.

`IsSinglePrecision(n)`

Returns `true` if n fits in a single word in the internal representation of integers in MAGMA, that is, if $|n| < 2^{30}$, `false` otherwise.

`IsZero(n)` `IsOne(n)` `IsMinusOne(n)`
`IsNilpotent(n)` `IsIdempotent(n)`
`IsUnit(n)` `IsZeroDivisor(n)` `IsRegular(n)`
`IsIrreducible(n)` `IsPrime(n)`

18.4.6 Comparison of Ring Elements

`m gt n` `m ge n` `m lt n` `m le n`
`Maximum(m, n)` `Maximum(Q)`
`Minimum(m, n)` `Minimum(Q)`

18.4.7 Conjugates, Norm and Trace

`ComplexConjugate(n)`

The complex conjugate of n, which will be the integer n itself.

`Conjugate(n)`

The conjugate of n, which will be the integer n itself.

`Norm(n)`

The norm in $\mathbf{Q}$ of n, which will be the integer n itself.

`EuclideanNorm(n)`

The Euclidean norm (length) of n, which will equal the absolute value of n.

`Trace(n)`

The trace (in $\mathbf{Q}$) of n, which will be the integer n itself.

`MinimalPolynomial(n)`

Returns the minimal polynomial of the integer n, which is the monic linear polynomial with constant coefficient n in a univariate polynomial ring R over the integers.

18.4.8 Other Elementary Functions

`AbsoluteValue(n)`
`Abs(n)`

Absolute value of the integer n.

`Ilog2(n)`

The integral part of the logarithm to the base two of the positive integer n.

`Ilog(b, n)`

The integral part of the logarithm to the base b of the positive integer n i.e., the largest integer k such that $b^k \leq n$. The integer b must be greater than or equal to two.

`Quotrem(m, n)`

Returns both the quotient q and remainder r obtained upon dividing the integer m by the integer n, that is, $m = q \cdot n + r$, where $0 \leq r < n$ if $n > 0$ and $n < r \leq 0$ if $n < 0$.

`Valuation(x, p)`

The valuation of the integer x at the prime p. This is the largest integer v for which p^v divides x. If $x = 0$ then $v = \infty$. The optional second return value is the integer u such that $x = p^v u$.

`Iroot(a, n)`

Given a positive integer a, return the integer $b = \lfloor \sqrt[n]{a} \rfloor$, i.e. the integral part of the n-th root of a. To obtain the actual root (as a real number), a must e coerced into a real field and the function `Root` applied.

`Sign(n)`

Returns -1, 0 or 1 depending upon whether the integer n is negative, zero or positive, respectively.

`Ceiling(n)`

The ceiling of the integer n, that is, n itself.

`Floor(n)`

The floor of the integer n, that is, n itself.

`Round(n)`

This function rounds the integer n to itself.

`Truncate(n)`

This function returns the integer truncation of the integer n, that is, n itself.

`SquarefreeFactorization(n)`

Given a non-negative integer n, return a squarefree integer x as well as a positive integer y, such that $n = xy^2$.

`Isqrt(n)`

Given a positive integer n, return the integer $\lfloor\sqrt{n}\rfloor$, i.e., the integral part of the square root of the integer n.

18.5 Random Numbers

Pseudo-random integers in MAGMA are generated using the *Monster* random number generator of G. Marsaglia [Mar00]. The period of this generator is $2^{29430} - 2^{27382}$ (approximately 10^{8859}), and the generator passes all of the stringent tests in Marsaglia's *Diehard* test suite [Mar95]. Throughout the following text, the word 'random' is used to mean 'pseudo-random'.

`Random(a, b)`

A random integer lying in the interval $[a, b]$, where $a \leq b$.

`Random(b)`

A random integer lying in the interval $[0, b]$, where b is a non-negative integer. Because of the good properties of the underlying Monster generator, calling `Random(1)` is a good safe way of producing a sequence of random bits.

`RandomBits(n)`

A random integer m such that $0 \leq m < 2^n$, where n is a small non-negative integer. Thus, m has n random bits with a probability of $1/2$ for each bit. The function always returns 0 when $n = 0$.

`RandomPrime(n:` *parameter*`)`

`Proof` BOOLELT *Default :* `true`

A random prime integer m such that $0 < m < 2^n$, where n is a small non-negative integer. The function always returns 0 for $n = 0$ or $n = 1$. A rigorous method will be used to check primality, unless $m > 34 \cdot 10^{13}$ and the optional parameter `Proof` is set to `Proof := false`, in which case the result indicates that m is a probable prime (of order 20).

`RandomPrime(n, a, b, x:` *parameter*`)`

`Proof` BOOLELT *Default :* `true`

Tries up to x iterations to find a random prime integer m congruent to a modulo b such that $0 < m < 2^n$. If successful, the function returns true and the integer m, otherwise false. The integer n must be larger than 0, a must lie between 0 and $b-1$ and b must be larger than 0. A rigorous method will be used to establish primality, unless $m > 34 \cdot 10^{13}$ and the optional parameter `Proof` is set to `Proof := false`, in which case the result indicates that m is a probable prime (of order 20).

`RandomConsecutiveBits(n, a, b)`

A integer m such that $0 \leq m < 2^n$, and the binary expansion of n consists of consecutive strings of zeros or ones each of random length in the range $[a \dots b]$.

18.6 Common Divisors and Common Multiples

This section deals with computing greatest common divisors and related computations.

Within the classical range, MAGMA uses the fast classical Accelerated GCD algorithm of Kenneth Weber [Web95] to compute the GCD of two integers, and the fast classical Lehmer extended GCD ('XGCD') algorithm [Knu97, pp. 345–348] (which is about 5 times faster than the Euclidean XGCD algorithm) to compute the extended GCD of two integers.

For larger integers, MAGMA uses the asymptotically fast Schönhage recursive ("half-GCD") algorithm ([Sch71]; see also [Mon92, Sec. 3.8] for the basic idea, applied to polynomials). On a Sun SPARC workstation, the crossover point for the Schönhage GCD algorithm (where it beats the classical Accelerated GCD algorithm) is 32768 bits (about 10000 decimal digits), while the crossover point for the Schönhage XGCD algorithm (when it beats the Lehmer XGCD algorithm) is 6000 bits (about 2000 decimal digits).

`GreatestCommonDivisor(m, n)`
`Gcd(m, n)`
`GCD(m, n)`

The greatest common divisor of m and n, normalized to be non-negative. If either of the inputs is zero, then the result is the absolute value of the other input, while if m and n are both zero the result is zero.

`GreatestCommonDivisor(s)`
`Gcd(s)`
`GCD(s)`

The GCD of the entries of the sequence s. If all entries of the sequence are zero, the result is zero. An error results if the sequence is the null sequence.

`ExtendedGreatestCommonDivisor(m, n)`
`Xgcd(m, n)`
`XGCD(m, n)`

The extended GCD of m and n; returns integers g, x and y such that g is the greatest common divisor of the integers m and n, and $g = x \cdot m + y \cdot n$. If m and n are both zero, g is zero; otherwise g is always positive. If m and m are both non-zero, the multipliers x and y are unique.

`ExtendedGreatestCommonDivisor(s)`

`Xgcd(s)`

`XGCD(s)`

Given a sequence of integers $s = [s_1, \ldots, s_r]$, return the non-negative integer g and a sequence $X = (x_1, \ldots, x_r)$ such that g is the greatest common divisor of the integers s_i and $g = \sum_{i=1}^{r} x_i \cdot s_i$.

`LeastCommonMultiple(m, n)`

`Lcm(m, n)`

`LCM(m, n)`

The smallest non-negative integer divisible by both m and n. If m or n equals zero, the result is zero; this ensures that $\text{lcm}(m, n)\text{gcd}(m, n) = m \cdot n$.

`LeastCommonMultiple(s)`

`Lcm(s)`

`LCM(s)`

Least common multiple of the sequence of integers s.

18.7 Arithmetic Functions

Each of the functions in this section may take an integer or the factorization of that integer.

`CarmichaelLambda(n)`

`CarmichaelLambda(Q)`

`CarmichaelLambda(Q)`

The Carmichael function $\lambda(n)$; its value equals the exponent of $(\mathbf{Z}/n\mathbf{Z})^*$.

`DickmanRho(u)`

Computes $\rho(u)$ where ρ is Dickman's rho function.

`FactoredCarmichaelLambda(n)`

`FactoredCarmichaelLambda(Q)`

`FactoredCarmichaelLambda(Q)`

The Carmichael function $\lambda(n)$, returned as a factorization sequence.

`DivisorSigma(i, n)`

`DivisorSigma(i, Q)`

The divisor function $\sigma_i(n) = \sum_{d|n} d^i$ for integer n and small non-negative integer i.

`NumberOfDivisors(n)`
`NumberOfDivisors(Q)`

The number of divisors of the positive integer n. This is a special case of `DivisorSigma`.

`SumOfDivisors(n)`
`SumOfDivisors(Q)`

The sum of the divisors of the positive integer n. This is a special case of `DivisorSigma`.

`EulerPhi(n)`
`EulerPhi(Q)`
`EulerPhi(Q)`

The Euler totient function $\phi(n)$; its value equals the order of $(\mathbf{Z}/n\mathbf{Z})^*$.

`FactoredEulerPhi(n)`
`FactoredEulerPhi(Q)`
`FactoredEulerPhi(Q)`

The Euler totient function $\phi(n)$, returned as a factorization sequence.

`EulerPhiInverse(m)`
`EulerPhiInverse(Q)`

The inverse of the Euler totient function $\phi(n)$; that is, the sorted sequence of all integers n such that $\phi(n) = m$.

`FactoredEulerPhiInverse(n)`
`FactoredEulerPhiInverse(Q)`

The factored inverse of the Euler totient function $\phi(n)$; that is, the sorted sequence of the factorizations of all integers n such that $\phi(n) = m$.

`LegendreSymbol(n, m)`

The Legendre symbol $\left(\frac{n}{m}\right)$: for prime m this checks whether or not n is a quadratic residue modulo m. The function returns 0 if m divides n, -1 if n is not a quadratic residue, and 1 if n is a quadratic residue modulo m. A fast probabilistic primality test is performed on m. If m fails the test (and is therefore composite), an error results; if it passes the test the Jacobi symbol is computed.

`JacobiSymbol(n, m)`

The Jacobi symbol $\left(\frac{n}{m}\right)$. For odd $m > 1$ this is defined (but not calculated!) as the product of the Legendre symbols $\left(\frac{n}{p_i}\right)$, where the product is taken over all primes p_i dividing m including multiplicities. Quadratic reciprocity is used to calculate this symbol, which has the values -1, 0 or 1.

`KroneckerSymbol(n, m)`

The Kronecker symbol $\left(\frac{n}{m}\right)$. This is the extension of the Jacobi symbol to all integers m, by multiplicativity, and by defining $\left(\frac{n}{2}\right) = (-1)^{(n^2-1)/8}$ for odd n (and 0 for even n) and $\left(\frac{n}{-1}\right) = \pm 1$ according to the sign of n for $n \neq 0$ (and 1 for $n = 0$).

`MoebiusMu(n)`

`MoebiusMu(Q)`

The Möbius function $\mu(n)$. This is a multiplicative function characterized by $\mu(1) = 1$, $\mu(p) = -1$, and $\mu(p^k) = 0$ for $k \geq 2$, where p is a prime number.

Example H18E5

A pair of positive integers (m, n) is called *amicable* if the sum of the proper divisors (that is: excluding m itself) of m equals n, and vice versa. The following function finds such pairs. Note that it also finds perfect numbers: amicable pairs of the form (m, m).

```
> d := func< m | DivisorSigma(1, m)-m >;
> z := func< m | d(d(m)) eq m >;
> for m := 2 to 10000 do
>     if z(m) then
>         m, d(m);
>     end if;
> end for;
6 6
28 28
220 284
284 220
496 496
1184 1210
1210 1184
2620 2924
2924 2620
5020 5564
5564 5020
6232 6368
6368 6232
8128 8128
```

18.8 Combinatorial Functions

`Binomial(n, r)`

The binomial coefficient $\binom{n}{r}$.

`Multinomial(n, [a1, ... an])`

Given a sequence $Q = [r_1, \ldots, r_k]$ of positive integers such that $n = r_1 + ... + r_k$, return the multinomial coefficient $\binom{n}{r_1,\ldots,r_k}$.

`Factorial(n)`

The factorial $n!$ for positive small integer n.

`IsFactorial(n)`

Tests if $n = k!$ for some k. If so, return `true` and k, `false` otherwise.

`Partitions(n)`

The unrestricted partitions of the positive integer n. This function returns a sequence of integer sequences, each of which is a different sequence of positive integers (in descending order) adding up to n. The integer n must be small.

`NumberOfPartitions(n)`

The number of unrestricted partitions of the non-negative integer n. The integer n must be small.

`RestrictedPartitions(n, Q)`

The partitions of the positive integer n, restricted to elements of the positive integer sequence Q.

`RestrictedPartitions(n, k, M)`

The partitions of the positive integer n into k parts, restricted to elements of the positive integer sequence Q.

`StirlingFirst(n, k)`

The Stirling number of the first type, $\left[{n \atop k}\right]$, where n and k are non-negative integers.

`StirlingSecond(n, k)`

The Stirling number of the second type, $\left\{{n \atop k}\right\}$, where n and k are non-negative integers.

`Bell(n)`

The nth Bell number, giving the number of partitions of a set of size n. (Not to be confused with `NumberOfPartitions(n)`, which gives the number of partitions of the *integer* n.) This is equal to the sum of `StirlingSecond(n,k)` for k between 0 and n (inclusive).

`Fibonacci(n)`

Given an integer n, this function returns the n-th Fibonacci number F_n, which can be defined via the recursion $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for all integers n. Note that n is allowed to be negative, and that $F_{-n} = (-1)^{n+1}F_n$.

`Lucas(n)`

Given an integer n, this function returns the n-th Lucas number L_n, which can be defined via the recursion $L_0 = 2$, $L_1 = 1$ and $L_n = L_{n-1} + L_{n-2}$ for all integers n. Note that n is allowed to be negative, and that $L_{-n} = (-1)^n L_n$.

`GeneralizedFibonacciNumber(g0, g1, n)`

The nth member of the generalized Fibonacci sequence defined by $G_0 = g_0$, $G_1 = g_1$ and $G_n = G_{n-1} + G_{n-2}$ for all integers n. Note that n is allowed to be negative. The Fibonacci and Lucas numbers are special cases where $(g_0, g_1) = (0, 1)$ or $(2, 1)$ respectively.

18.9 Primes and Primality Testing

Primality testing algorithms enable the user to certify the primality of prime integers. Proving the primality of very big integers can be time consuming and therefore in some of the algorithms using primes and factorization of integers the user can speed up the algorithm by explicitly allowing MAGMA to use probable primes rather than certified primes.

A *probable prime* is an integer that has failed some compositeness test; if an integer passes a compositeness test it will be composite, but there is a (small) probability that a composite number will fail the test and is hence called a probable prime. Each Miller-Rabin test for instance, has a probability of less than 1/4 of declaring a composite number probably prime; in practice that means that numbers that fail several such cheap independent Miller-Rabin compositeness tests will be prime.

Unless specifically asked otherwise, MAGMA will use rigorous primality proofs.

18.9.1 Primality

If a positive integer n is composite, this can be shown quickly by exhibiting a *witness* to this fact. A witness for the compositeness of n is an integer $1 < a < n$ with the property that

$$a^r \not\equiv 1 \bmod n \quad \text{and} \quad a^{r2^i} \not\equiv -1 \bmod n \text{ for } i = 0, 1, \ldots, k-1$$

where r odd, and k are such that $n - 1 = r \cdot 2^k$. A witness never falsely claims that n is composite, because for prime n it must hold that $a^{n-1} \equiv 1 \bmod n$ and only ± 1 are square roots of 1 modulo prime n. Moreover, it has been shown that a fraction of at least 3/4 of all a in the range $2 \ldots n-1$ will witness the compositeness of n. Thus randomly choosing a will usually quickly expose compositeness. Unless more than 3/4 of all possibilities for a are checked though (which in practice will be impossible for reasonable n) the procedure of checking k bases a at random for being a witness (often referred to as 'Miller-Rabin') will not suffice to prove the primality of n; it does however lend credibility to the claim that n

is most likely prime if among k (say 20) random choices for a no witness for compositeness has been found. In such cases n is called *probably prime of order* k, and in some sense the probability that n is composite is less than 4^{-k}.

A slight adaptation of this compositeness test can be used for primality proofs in a bounded range. There are no composites smaller than $34 \cdot 10^{13}$ for which a witness does not exist among $a = 2, 3, 5, 7, 11, 13, 17$ ([Jae93]). Using these values of a for candidate witnesses it is certain that for any number n less than $34 \cdot 10^{13}$ the test will either find a witness or correctly declare n prime.

But even for large integers it is thus usually easy to identify composites without finding a factor; to be certain that a large probable prime is truly prime, a primality proving algorithm is invoked. MAGMA uses the ECPP (Elliptic Curve Primality Proving) method, as implemented by François Morain (Ecole Polytechnique and INRIA). The ECPP program in turn uses the BigNum package developed jointly by INRIA and Digital PRL. This ECPP method is both fast and rigorous, but for large integers (of say more than 100 decimal digits) it will be still be much slower than the Miller-Rabin compositeness test. The method is too involved to be explained here; we refer the reader to the literature ([AM93]).

The `IsPrime` function invokes ECPP, unless a Boolean flag is used to indicate that only 'probable primality' is required. The latter is equivalent to a call to `IsProbablePrime`.

`IsPrime(n)`

`IsPrime(n: parameter)`

`Proof`	BOOLELT	*Default :* `true`

Returns `true` iff the integer n is prime. A rigorous method will be used, unless $n > 34 \cdot 10^{13}$ and the optional parameter `Proof` is set to `Proof := false`, in which case the result indicates that n is a probable prime (a strong pseudoprime to 20 bases).

`SetVerbose("ECPP", v)`

Sets the verbose level for output when the ECPP algorithm is used in the above primality tests. The legal values are `true`, `false`, 0, 1 and 2 (`false` and `true` are the same as 0 and 1 respectively). Level 1 outputs only basic information about the times for the top-level stages (downrun and uprun). Level 2 outputs full information about every step : this level is very verbose!

`PrimalityCertificate(n)`

`IsPrimeCertificate(cert)`

`ShowCertificate`	BOOLELT	*Default :* `true`
`Trust`	RNGINTELT	*Default :* 0

`PrimalityCertificate` is a variant on `IsPrime` which uses ECPP and outputs a certificate of primality at the conclusion. If the number n is actually proven to be composite or the test fails, then a runtime error occurs. The certificate is a `Magma` list with data in the format described in [AM93].

To verify that a given number is prime from its primality certificate, the function `IsPrimeCertificate` is used. By default, this outputs only the result of the verification : `true` or `false`. If the user wishes to see the stages of the verification, the parameter `ShowCertificate` should be set to `true`. This is rather verbose as it shows the verification of primality of all small factors that need to be shown to be prime at each substage of the algorithm. It is usually more convenient to set the parameter `Trust` to a positive integer N which means that asserted primes less than N are not checked. This slightly reduces the time for the verification, but more importantly, it greatly reduces the output of `ShowCertificate`.

`IsProbablePrime(n:` *parameter*`)`

`IsProbablyPrime(n:` *parameter*`)`

`Bases` RngIntElt *Default :* 20

Returns `true` if and only if the integer n is a probable prime. More precisely, the function returns `true` if and only if either n is prime for $n < 34 \cdot 10^{13}$, or n is a strong pseudoprime for 20 random bases b with $1 < b < n$. By setting the optional parameter `Bases` to some value B, the number of random bases used is B instead of 20.

`IsPrimePower(n)`

Returns `true` if and only if the integer n is a prime power; that is, if n equals p^k for some prime p and exponent $k \geq 1$. If this is the case, the prime p and the exponent k are also returned, Note that the primality of p is rigorously proven.

Example H18E6

This piece of code uses 5 Miller-Rabin tests to find the next probable *repunit*-prime (consisting of all 1's as decimal digits), using the fact that primes of this form consist of a prime number of digits:

```
> NextPPRepunit := function(nn)
>    n := nn;
>    repeat
>       n := NextPrime(n);
>    until IsProbablePrime( (10^n-1) div 9 : Bases := 5);
>    return n;
> end function;
```

The first few cases are easy:

```
> NextPPRepunit(1);
2
> NextPPRepunit(2);
19
> NextPPRepunit(19);
23
> NextPPRepunit(23);
```

```
317
```

So we found a 317 digit prime (although we should check genuine primality, using `IsPrime`)! We leave it to the reader to find the next (it has more than 1000 decimal digits).

18.9.2 Other Functions Relating to Primes

The functions `NextPrime` and `PreviousPrime` can be used to find primes in the neighbourhood of a given integer. After sieving out only multiples of very small primes, the remaining integers are tested for primality in order. Again, a rigorous method is used unless the user flags that probable primes suffice.

The `PrimeDivisors` function is different from all other functions in this section since it requires the factorization of its argument.

`NextPrime(n)`

`NextPrime(n:` *parameter*`)`

`Proof` BoolElt *Default :* `true`

The least prime number greater than n, where n is a non-negative integer. The primality is proved. The optional boolean parameter 'Proof' (`Proof := true` by default) can be set to `Proof := false`, to indicate that the next probable prime (of order 20) may be returned.

`PreviousPrime(n)`

`PreviousPrime(n:` *parameter*`)`

`Proof` BoolElt *Default :* `true`

The greatest prime number less than n, where $n \geq 3$ is an integer. The primality is proved. The optional boolean parameter 'Proof' (`Proof := true` by default) can be set to `Proof := false`, to indicate that the previous probable prime (of order 20) may be returned.

`PrimesUpTo(B)`

This function lists the primes up to (and including) the (positive) bound B. The algorithm is not super-optimised, but is reasonable.

`PrimesInInterval(b, e)`

This function lists the primes in the interval from b to e, including the endpoints. The algorithm is not very optimised.

`NthPrime(n)`

Given a number n, this function returns the nth prime. This is implemented for primes up to 10^{10}.

`RandomPrime(n: parameter)`

`Proof` BoolElt *Default :* `true`

A random prime integer m such that $0 < m < 2^n$, where n is a small non-negative integer. The function always returns 0 for $n = 0$ or $n = 1$. A rigorous method will be used to check primality, unless $m > 34 \cdot 10^{13}$ and the optional parameter 'Proof' is set to `Proof := false`, in which case the result indicates that m is a probable prime (of order 20).

`RandomPrime(n, a, b, x: parameter)`

`Proof` BoolElt *Default :* `true`

Tries up to x iterations to find a random prime integer m congruent to a modulo b such that $0 < m < 2^n$. Returns true, m if found, or false if not found. n must be larger than 0. a must be between 0 and $b - 1$ and b must be larger than 0. A rigorous method will be used to check primality, unless $m > 34 \cdot 10^{13}$ and the optional parameter 'Proof' is set to `Proof := false`, in which case the result indicates that m is a probable prime (of order 20).

`PrimeBasis(n)`

`PrimeDivisors(n)`

A sequence containing the distinct prime divisors of the positive integer $|n|$.

18.10 Factorization

This section contains a description of most of the machinery provided in Magma for the factorization of integers. An account of the Number Field Sieve is deferred until later in the chapter.

In the first subsection the general-purpose `Factorization` function is described. It employs a combination of methods in an attempt to find the complete prime factorization of a given integer. Some control is possible over each of the methods, but in general default choices for the parameters would give good results for a wide range of arguments.

In the second subsection we describe functions that enable access to each of the factorization methods available in Magma. The user has control over parameters for these methods.

Factorization functions in Magma return a *factorization sequence*. This is a sequence of two-element tuples $[< p_1, k_1 >, \ldots, < p_r, k_r >]$, with $p_1 < p_2 < \cdots < p_r$ distinct prime numbers and k_i positive, which is used to represent integers in factored form: $n = \prod_{i=1}^{r} p_i^{k_i}$. Although such sequences are printed like ordinary sequences, they form a separate category `RngIntEltFact`. Operations on such factorization sequences are described in the next section.

18.10.1 General Factorization

The general `Factorization` function is designed to give close to optimal performance for the factorization of integers that may be encountered in the course of daily computations. The strategy employed is as follows (the next subsection gives a more detailed description of the individual methods). First of all a compositeness test is used to ensure that the argument is composite; if not the primality proving algorithm is invoked (unless a flag is set to avoid this — see below). See the previous section for compositeness testing and primality proving. This operation is repeated for any non-trivial factor (and cofactor) found along the way. Before any of the general factorization techniques is employed, it is checked whether $|n|$ is of the special form $b^k \pm 1$, in which case an intelligent database look-up is used which is likely to be successful if b and k are not too large. This is equivalent to the `Cunningham` function on $b, k, \pm 1$, described in the next subsection. In the first true stage of factorization *trial division* is used to find powers of 2 and other small primes (by default up to 10000). After this it is checked whether the remaining composite number is the power of a positive integer; if so the appropriate root is used henceforth. After this *Pollard's ρ method* is applied (using 8191 iterations by default). The bound on trial division factors and the number of iterations for ρ can be set by the optional parameters `TrialDivisionLimit` and `PollardRhoLimit`. It is possible, from this point on, that several composite factors still need factorization. The description below applies to each of these.

The final two algorithms deployed are usually indicated by ECM (for Elliptic Curve Method) and MPQS (for Multiple Polynomial Quadratic Sieve). By default, ECM (which is likely to find 'smaller' factors if they exist) is used with parameters that depend on the size of the remaining (composite) factors. After that, if a composite factor of at least 25 digits remains, MPQS is used; it is the best method available for factoring integers of more than about 40 decimal digits especially for products of two primes of roughly equal size. If the remaining composite is smaller than 25 digits, ECM is again invoked, now in an indefinite loop until a factor is found. The latter will also occur if the user, via a flag `MPQSLimit` indicates that MPQS should not be applied to numbers of the given size, and provided the user has not limited the number of ECM trials by setting the `ECMLimit`. Thus, unless both `MPQSLimit` and `ECMLimit` are set as optional parameters by the users, the algorithm will continue until the complete factorization has been completed.

Besides the limiting parameters just mentioned it also possible to avoid the use of primality proofs and receive probable primes, with a flag similar to that used on `IsPrime`; see the previous section.

A verbose flag can be set to obtain informative printing on progress in the various stages of factorization. Specific flags for ECM and MPQS may be used as well; they are described in the next subsection.

```
SetVerbose("Factorization", v)
```

(Procedure.) Set the verbose printing level for all of the factorization algorithms to be v. Currently the legal values for v are `true`, `false`, 0 or 1 (`false` is the same as 0, and `true` is the same as 1). If the level is 1, information is printed at each stage of the algorithm as a number is factored.

`Factorization(n)`
`Factorisation(n)`
`Factorization(n:` *parameters*`)`
`Factorisation(n:` *parameters*`)`

A combination of algorithms (Cunningham, trial division, SQUFOF, Pollard ρ, ECM and MPQS) is used to attempt to find the complete factorization of $|n|$, where n is a non-zero integer. A factorization sequence is returned, representing the completely factored part of $|n|$ (which is usually all of $|n|$). The second return value is 1 or -1, reflecting the sign of n. If the factorization could not be completed, a third sequence is returned, containing composite factors that could not be decomposed with the given values of the parameters; this can only happen if both `ECMLimit` and `MPQSLimit` have been set. (Note that the third variable will remain unassigned if the full factorization is found.)

When a very large prime (more than 200 decimal digits say), appears in the factorization, proving its primality may dominate the running time.

There are 6 optional parameters.

`Proof`	BoolElt	*Default :* `true`
`Bases`	RngIntElt	*Default :* 20

The parameter `Proof` (`Proof := true` by default) can be set to false to indicate that the first sequence may contain probable primes (see also the previous section), in which case the parameter `Bases` indicates the number of tests used by Miller-Rabin (`Bases := 20` by default).

`TrialDivisionLimit`	RngIntElt	*Default :* 10000

The parameter `TrialDivisionLimit` can be used to specify an upper bound for the primes used in the trial division stage (default `TrialDivisionLimit := 10000`).

`SQUFOFLimit`	RngIntElt	*Default :* 24

The parameter `SQUFOFLimit` can be used specify the maximum number of decimal digits for an integer to which SQUFOF should be applied; this is Shank's square form factorization method (default `SQUFOFLimit := 24`).

`PollardRhoLimit`	RngIntElt	*Default :* 8191

The parameter `PollardRhoLimit` can be used to specify an upper bound on the number of iterations in the ρ method (default `PollardRhoLimit := 8191`).

`ECMLimit`	RngIntElt	*Default :*

This optional parameter can be used to limit the number of curves used by the ECM part of the factorization attempt. Setting `ECMLimit := 0` prevents the use of ECM. The default value depends on the size of the input, and ranges from 2 for n with less than 37 digits to around 500 for n with 80 digits. The smoothness is incremented in each step to grow by default from 500 to 600 (for 37 digits and less), and from 500 to about 10000 for n having 80 digits. For the indefinite case of ECM (which applies

when MPQS is disallowed) the initial smoothness is 500, the number of curves is infinite and the smoothness is incremented by 100 in each step.

`MPQSLimit` RngIntElt *Default :* ∞

The parameter `MPQSLimit` can be used specify the maximum number of decimal digits for an integer to which MPQS should still be applied; MPQS will not be invoked on integers having less than (or sometimes equal) 25 decimal digits. Setting the parameter to anything less than 25 will therefore prevent MPQS from being used. Unless `ECMLimit` has been set, this will imply that ECM will be applied until the full factorization has been obtained.

Note that progress can be monitored by use of `Verbose("Factorization", true)`.

18.10.2 Storing Potential Factors

As of V2.14 (October 2007), Magma now internally stores a list of factors found by the ECM and MPQS algorithms. Subsequently, when either of those algorithms are to be invoked by the `Factorization` function, the integers in the list are first tried to see whether factors can be easily found. One may also give prime factors to Magma to store in this list via the following procedure.

`StoreFactor(n)`
`StoreFactor(S)`

(Procedure.) Store the single integer n or the integers in the set/sequence S in the list of factors to be tried by the `Factorization` function. Each integer must be a positive *prime*.

`GetStoredFactors()`

Return a sequence containing the currently stored integers.

18.10.3 Specific Factorization Algorithms

In this subsection we discuss how various factorization algorithms can be accessed individually. Generally these function should not be used for ordinary factorization (for that use `Factorization` discussed in the previous subsection), but they can be used for experimentation, or to build a personal factorization function with control over each of the methods used.

On some functions a little preprocessing is done to ensure that the argument is composite, that powers of 2 (and sometimes 3) are taken out and that the integer to be factored is not the power of an integer.

For each of these functions the `Proof` (default true) and `Bases` parameters can be used to indicate that primality of prime factors need not be rigorously proved, and how many bases should be used in the compositeness test, as discussed in the subsection on `IsPrime`.

```
SetVerbose("Cunningham", b)
```

```
SetVerbose("ECM", b)
```

```
SetVerbose("MPQS", b)
```

Using this procedure to set either of the verbose flags `"Cunningham"`, `"ECM"` or `"MPQS"`, (which are false by default) enables the user to obtain progress information on attempts to factor integers using the 'Cunningham' method, ECM or MPQS.

```
Cunningham(b, k, c)
```

This function attempts to factor $n = b^k + c$, where $c \in \{\pm 1\}$ and b and k are not too big. This function uses R. Brent's factor algorithm [BtR92], which employs a combination of table-lookups and attempts at 'algebraic' factorization (Aurifeuillian techniques). An error results if the tables, containing most of the known factors for numbers of this form (including the 'Cunningham tables'), cannot be located by the system. The function will always return the complete prime factorization (in the form of a factorization sequence) of the number n (but it may take very long before it completes); it should be pointed out, however, that the primes appearing in the factorization are only *probable primes* and a rigorous primality prover has not been applied.

```
AssertAttribute(RngInt, "CunninghamStorageLimit", l)
```

This attribute is used to change the number of Cunningham factorizations which are stored in MAGMA. Normally, MAGMA stores a certain number of factorizations computed by the `Cunningham` intrinsic function so that commonly needed factorizations can be recalled quickly. When the stored list fills up, the factorization least recently accessed is removed from the list. Setting this attribute to zero ensures that no storage is done. The default value is 20.

```
TrialDivision(n)
```

```
TrialDivision(n, B)
```

`Proof`	BOOLELT	*Default :* `true`
`Bases`	RNGINTELT	*Default :* 20

The integer $n \neq 0$ is subjected to trial division by primes up to a certain bound B (the sign of n is ignored). If only the argument n is given, B is taken to be 10000. The function returns a factorization sequence and a sequence containing an unfactored composite that remains.

`PollardRho(n)`

`PollardRho(n, c, s, k)`

`Proof`	BoolElt	*Default :* `true`
`Bases`	RngIntElt	*Default :* 20

The ρ-method of Pollard is invoked by this function to find the factorization of an integer $n > 1$. For this method a quadratic function $x^2 + c$ is iterated k times, with starting value $x = s$. If only n is used as argument to the function, the default values $c = 1$, $s = 1$, and $k = 8191$ are selected. A speed-up to the original algorithm, due to R. P. Brent [Bre80], is implemented. The function returns two values: a factorization sequence and a sequence containing unfactored composite factors.

`pMinus1(n, B1)`

`x0`	RngIntElt	*Default :*
`B2`	RngIntElt	*Default :*
`k`	RngIntElt	*Default :*

Given an integer $n > 1$, an attempt to find a factor is made using Paul Zimmermann's GMP-ECM implementation of Pollard's $p - 1$ method. If a factor f with $1 < f < n$ is found, then f is returned; otherwise 0 is returned.

The Step 1 bound B_1 is given as the second argument `B1`. By default, the Step 2 bound B_2 is optimally chosen, but may be given with the parameter `B2` instead. By default, an optimal number of blocks is chosen for Step 2, but this may be overridden via the parameter `k` (see the function `ECM`). The base x_0 is chosen randomly by default, but may instead be supplied via the parameter `x0`.

This method will return a prime factor p of n if $p - 1$ has all its prime factors less than or equal to the Step 1 bound B_1, except for one factor which may be less than or equal to the Step 2 bound B_2.

`pPlus1(n, B1)`

`x0`	RngIntElt	*Default :*
`B2`	RngIntElt	*Default :*
`k`	RngIntElt	*Default :*

Given an integer $n > 1$, an attempt to find a factor is made using Paul Zimmermann's GMP-ECM implementation of Williams' $p + 1$ method. If a factor f with $1 < f < n$ is found, then f is returned; otherwise 0 is returned.

The Step 1 bound B_1 is given as the second argument `B1`. By default, the Step 2 bound B_2 is optimally chosen, but may be given with the parameter `B2` instead. By default, an optimal number of blocks is chosen for Step 2, but this may be overridden via the parameter `k` (see the function `ECM`). The base x_0 is chosen randomly by default, but may instead be supplied via the parameter `x0`.

This method may return a prime factor p of n if $p + 1$ has all its prime factors less than or equal to the Step 1 bound B_1, except for one factor which may be

less than or equal to the Step 2 bound B_2. A base x_0 is used, and not all bases will succeed: only half of the bases work (namely those where the Jacobi symbol of $x_0^2 - 4$ and p is -1.) Unfortunately, since p is usually not known in advance, there is no way to ensure that this holds. However, if the base is chosen randomly, there is a probability of about 1/2 that it will give a Jacobi symbol of -1 (so that the factor p would be found assuming that $p+1$ is smooth enough). A rule of thumb is to run `pPlus1` three times with different random bases.

`SQUFOF(n)`

`SQUFOF(n, k)`

`Proof`	BoolElt	*Default :* `true`
`Bases`	RngIntElt	*Default :* 20

This is a fast implementation of Shanks's square form factorization method that will only work for integers $n > 1$ less than 2^{2b-2}, where b is the number of bits in a long (which is either 32 or 64). The argument k may be used to specify the maximum number of iterations used to find the square; by default it is 200000. The expected number of iterations is $O(N^{1/4})$.

`ECM(n, B1)`

`Sigma`	RngIntElt	*Default :*
`x0`	RngIntElt	*Default :*
`B2`	RngIntElt	*Default :*
`k`	RngIntElt	*Default :* 2

Given an integer $n > 1$, an attempt is made to find a factor using the GMP-ECM implementation of the Elliptic Curve Method (ECM). If a factor f with $1 < f < n$ is found, then f is returned together with the corresponding successful σ seed; otherwise 0 is returned.

The Step 1 bound B_1 is given as the second argument `B1`. By default, the Step 2 bound B_2 is optimally chosen, but may be given with the parameter `B2` instead.

The elliptic curve used is defined by Suyama's parametrization and is determined by a parameter σ. By default, σ is chosen randomly with $0 < \sigma < 2^{32}$, but an alternative positive integer may be supplied instead via the parameter `Sigma`. Let $u = \sigma^2 - 5$, $v = 4\sigma$ and $a = (v-u)^3(3u+v)/(4u^3v) - 2$. The starting point used is $(x_0 : 1)$, where by default $x_0 = u^3/v^3$, but x_0 may instead be supplied via the parameter `x0`. Finally, the curve used is $by^2 = x^3 + ax^2 + x$, where $b = x_0^3 + ax_0^2 + x_0$.

Step 1 uses very little memory, but Step 2 may use a large amount of memory, especially for large B_2, since its efficient algorithms use some large tables. To reduce the memory usage of Step 2, one may increase the parameter `k`, which controls the number of "blocks" used. Multiplying the default value of k by 4 will decrease the memory usage by a factor of 2. For example, with $B_2 = 10^{10}$ and a 155-digit number n, Step 2 requires about 96MB with the default $k = 2$, but only 42MB with $k = 8$. Increasing k does, however, slightly increase the time required for Step 2.

`ECMSteps(n, L, U)`

Given an integer $n > 1$, an attempt to find a factor of n is made by repeated calls to ECM. The initial B_1 bound is taken to be L, and subsequently B_1 is replaced with $B_1 + \lfloor \sqrt{B_1} \rfloor$ at each step. If a factor is found at any point, then this is returned with the corresponding successful σ seed; otherwise, if B_1 becomes greater than the upper bound U, then 0 is returned.

`MPQS(n)`

`MPQS(n, D)`

`Proof`	BoolElt	*Default :* `true`
`Bases`	RngIntElt	*Default :* 20

This function can be used to drive Arjen Lenstra's implementation of the multiple polynomial quadratic sieve MPQS. Given an integer $n > 5 \cdot 10^{24}$ an attempt is made to find the prime factorization of n using MPQS. The name of a directory (which should not yet exist) may be specified as a string D where files used by MPQS will be stored. By default, the directory indicated by the environment variable `MAGMA_QS_DIR` will be used, and if that has not been set, the directory `/tmp`. It is possible to assist the master running the main Magma job by generating relations on other machines (slaves), starting an auxiliary process on such machine, in the directory D, by typing `magma -q D machine` where `machine` is the name of the machine. The function returns two values: a factorization sequence and a sequence containing unfactored composite factors.

18.10.4 Factorization Related Functions

`ECMOrder(p, s)`

`ECMFactoredOrder(p, s)`

Suppose p is a prime factor found by the ECM algorithm and such that the σ value determining the successful curve was s. These functions compute the order of the corresponding elliptic curve. The first function returns the order as an integer, while the second function returns the factorization of the order. In general, this order will have been smooth with respect to the relevant bounds for the ECM algorithm to have worked, and these functions allow one to examine how small the prime divisors of the curve order really are.

`PrimeBasis(n)`

`PrimeDivisors(n)`

A sequence containing the distinct prime divisors of the positive integer $|n|$, given in increasing order.

`Divisors(n)`
`Divisors(f)`

Returns a sequence containing all divisors of the positive integer, including 1 and the integer itself, given in increasing order. The argument given must be either the integer n itself, or a factorization sequence f representing it.

`CoprimeBasis(S)`

Given a set or sequence S of integers, return a coprime basis of S in the form of a factorization sequence Q whose integer value is the same as the product of the elements of S but Q has coprime bases (i.e., the first components of tuples from Q are coprime).

Example H18E7

In this example we use the `Divisors` function together with the `&+` reduction of sequences to find the first few perfect numbers, that is, numbers n such that the sum of the divisors less than n equals n.

```
> {  x : x in [2..1000] | &+Divisors(x) eq 2*x };
{ 6, 28, 496 }
> f := Factorization(496);
> f;
[ <2, 4>, <31, 1> ]
> Divisors(f);
[ 1, 2, 4, 8, 16, 31, 62, 124, 248, 496 ]
```

`PartialFactorization(S)`

Given a sequence of non-zero integers S, return, for each integer $S[i]$, two factorization lists F_i and G_i, such that $S[i] =$ `Facint`$(F_i)*$`Facint`(G_i). All the divisors in F_i are square factors, and, for any i and j, the divisors in G_i and G_j are either equal or are pairwise coprime. In other terms, `PartialFactorization(S)` provides a partial decomposition of the integers in S in square and coprime factors. The interesting fact is that this factorization uses only gcd and exact integer division. This algorithm is due to J.E. Cremona.

Example H18E8

A partial factorization is shown.

```
> PartialFactorization([1380, 675, 3408, 654]);
[
  [
    [ <2, 2> ],
    [ <115, 1>, <3, 1> ]
  ],
  [
```

```
    [ <5, 2>, <3, 2> ],
    [ <3, 1> ]
  ],
  [
    [ <2, 4> ],
    [ <71, 1>, <3, 1> ]
  ],
  [
    [],
    [ <218, 1>, <3, 1> ]
  ]
]
```

18.11 Factorization Sequences

The factorization of integers results in a factorization sequence, consisting of a sequence of pairs of prime and exponent. It is sometimes convenient to perform operations on such sequences without converting back to the integers they represent — it would, for example, be very inefficient to factor the product of two integers that have both been factored already. In this section we briefly list the operations that are allowed on such factorization sequences — note that these factorization sequences now have their own special type: `RngIntEltFact`. Conversion functions are supplied as well.

18.11.1 Creation and Conversion

Factorization sequence usually arise as the result of the `Factorization` of an integer, possibly via functions like `FactoredOrder`. The functions below allow conversion from and to ordinary sequences, and the inverse operation to factorization, creating an integer from a factorization.

`Facint(f)`

`FactorizationToInteger(f)`

Create the integer corresponding to the factorization sequence f.

`SeqFact(s)`

`SequenceToFactorization(s)`

Given a sequence of tuples, each consisting of pairs of prime integers and positive integer exponents, create the corresponding factorization sequence. The pairs must be ordered with strictly increasing primes as first components.

`Eltseq(f)`

`ElementToSequence(f)`

Given a factorization sequence f, create the enumerated sequence containing the same pairs of primes and exponents.

18.11.2 Arithmetic

The difference of two factorization sequences is only permitted when the first integer represented is greater than the second integer represented. An error results from division when the quotient does not correspond to an integer.

`s + t` `s - t` `s * t` `s / t` `s ^ k`

18.11.3 Divisors

The functions listed below can be applied to factorization sequences; their behaviour will be clear, and all of them are documented elsewhere when the argument is the corresponding positive integer.

`Lcm(s, t)` `Gcd(s, t)` `SquarefreeFactorization(f)`

`MoebiusMu(f)` `Divisors(f)` `PrimeDivisors(f)`

`NumberOfDivisors(f)` `SumOfDivisors(f)`

18.11.4 Predicates

All predicates listed below are applicable both to factorization sequences and to the positive integers these represent, and have been documented for integer arguments elsewhere.

`IsOne(s)` `IsOdd(s)` `IsEven(s)` `IsUnit(s)`

`IsPrime(s)` `IsPrimePower(s)` `IsSquare(s)` `IsSquarefree(s)`

18.12 Modular Arithmetic

In this section we describe some functions that make it possible to perform modular arithmetic without conversions to residue class rings.

18.12.1 Arithmetic Operations

`Modexp(n, k, m)`

The modular power $n^k \bmod m$, where n is an integer, k is an integer and m is an integer greater than one. If k is negative, n must have an inverse i modulo m, and the result is then $i^{-k} \bmod m$. The result is always an integer r with $0 \leq r < m$.

`n mod m`

Remainder upon dividing the integer n by the integer m. The result always has the same sign as m. An error results if m is zero.

`Modinv(n, m)`

`InverseMod(n, m)`

Given an integer n and a positive integer m, such that n and m are coprime, return an inverse u of n modulo m, that is, return an integer $1 \leq u < m$ such that $u \cdot n \equiv 1 \bmod m$.

`Modsqrt(n, m)`

Given an integer n and an integer $m \geq 2$, this function returns an integer b such that $0 \leq b < m$ and $b^2 \equiv n \bmod m$ if such b exists; an error results if no such root exists.

`Modorder(n, m)`

For integers n and m, $m > 1$, the function returns the least integer $k \geq 1$ such that $n^k \equiv 1 \bmod m$, or zero if $\gcd(n, m) \neq 1$.

`IsPrimitive(n, m)`

Returns `true` if n is a primitive root for m, `false` otherwise $(0 < n < m)$.

`PrimitiveRoot(m)`

Given an integer $m > 1$, this function returns an integer value defined as follows: If $\mathbf{Z}/m\mathbf{Z}$ has a primitive root and the function is successful in finding it, the root a is returned. If $\mathbf{Z}/m\mathbf{Z}$ has a primitive root but the algorithm does not succeed in finding it, or $\mathbf{Z}/m\mathbf{Z}$ does not possess a primitive root, then zero is returned.

18.12.2 The Solution of Modular Equations

The functions described here can be used if an occasional modular operation is required; the results are integers again. For more extensive modular arithmetic it is preferable to convert to residue class ring arithmetic. See section 19.4 for details.

`Solution(a, b, m)`

If a solution exists to the linear congruence $ax \equiv b \bmod m$, then returns $x0, k$ such that $x = x0 + i * k$ represents the complete set of solutions, where i can be any integer. Otherwise, returns -1.

`ChineseRemainderTheorem(X, N)`

`CRT(X, N)`

Apply the Chinese Remainder Theorem to the integer sequences X and N. The sequences must have the same length, k say. The function returns the unique integer x in the range $0 \leq x < LCM(N[1] \cdot \ldots \cdot N[k])$ such that $x \equiv X[i] \bmod N[i]$. The elements of N must all be positive integers greater than one. If there is no solution, then -1 is returned.

`Solution(A, B, N)`

Return a solution x to the system of simultaneous linear congruences defined by the integer sequences A, B and N. Each of these sequences must have the same number of terms, k say. The elements of N must all be positive integers greater than one. The i-th congruence is $A[i] \cdot x \equiv B[i] \bmod N[i]$. The solution x will satisfy $0 \leq x < LCM(N[1] \cdot \ldots \cdot N[k])$. If no solution exists, -1 is returned.

`NormEquation(d, m)`

`NormEquation(d, m:` *parameters*`)`

`Factorization` [<RngIntElt, RngIntElt>]

Given a positive integer d and a non-negative integer m, return true and two non-negative integers x and y, such that $x^2 + y^2 d = m$, if such a solution exists. If such a solution does not exists only the value false is returned. If the factorization of m is known, it may be supplied as the value of the parameter `Factorization` to speed up the computation.

Example H18E9

```
> d := 95744000009574400000227774976O;
> m := 5102197760510219776012138128480644;
> time NormEquation(d, m);
true 98 73
Time: 2.990
> time f := Factorization(m);
Time: 4.670
> f;
[ <2, 2>, <19, 1>, <671341810593449970527912911642l9, 1> ]
> time NormEquation(d, m: Factorization := f);
true 98 73
Time: 0.420
```

18.13 Infinities

Occasionally it is convenient to work with infinite quantities (for example, when working with valuations or cardinalities). Magma provides two such objects, the positive and negative infinities. This section describes the Magma facilities for dealing with such objects.

The infinities are compatible with certain finite quantities: integers, rationals and real numbers. In contexts where a common universe is needed to contain both finite and infinite quantities (for example, if creating a sequence of valuations) the extended reals (type `ExtRe`) are used. The extended reals are a coproduct-like object that can contain both infinities and compatible finite objects. When viewed as members of the extended reals, the elements are of type `ExtReElt`.

18.13.1 Creation

Certain system intrinsics such as `Valuation` which normally return an integer may return an infinite object for appropriate exceptional cases. Two special intrinsics are also provided to create infinite objects.

`Infinity()`

The positive infinity object.

`MinusInfinity()`

The negative infinity object.

18.13.2 Arithmetic

Only basic arithmetic operations are provided for infinite objects. The operations described below may freely mix infinite and finite quantities, but note that certain forms (such as $\infty - \infty$ or $\infty * 0$) are not well defined and will cause an error.

`- x`

`x + y` `x - y` `x * y` `x / y` `x ^ n`

18.13.3 Comparison

Infinite objects may be compared with themselves and finite quantities.

`x eq y` `x ne y` `x lt y` `x le y` `x gt y` `x ge y`

`Maximum(x, y)` `Minimum(x, y)`

18.13.4 Miscellaneous

`Sign(x)`

Returns 1 if x is the positive infinite object, -1 if x is the negative infinite object.

`Abs(x)`

`AbsoluteValue(x)`

Returns the positive infinite object.

`Round(x)`

`Floor(x)`

`Ceiling(x)`

Returns the infinite object x again; these functions are for convenience when dealing with objects which could be either finite numeric types or infinite objects.

`IsFinite(x)`

Returns `true` if x is finite, otherwise `false`. This is more convenient than checking the type of x.

18.14 Advanced Factorization Techniques: The Number Field Sieve

MAGMA provides an experimental implementation of the fastest general purpose factoring algorithm known: the Number Field Sieve (NFS). The implementation may be used both as a General Number Field Sieve and a Special Number Field Sieve – the only difference is in the selection of a suitable polynomial.

18.14.1 The MAGMA Number Field Sieve Implementation

In order to make use of the MAGMA NFS, the user should have some knowledge of the algorithm. The MAGMA NFS implementation also requires a significant amount of memory and disk space to be available for the duration of the factorization. For example, factorization of an 80-digit number may require at least 64 megabytes of RAM and half a gigabyte of disk space.

MAGMA's NFS implementation uses one linear polynomial (the "rational side") and one polynomial of higher degree (the "algebraic side"). At the time of writing this is not a major restriction, since the best methods for selecting polynomials for factorization of numbers of more than 100 digits involve one linear and one non-linear polynomial. MAGMA provides a number of functions to assist in choosing a good algebraic-side polynomial for the factorization of a particular number, following the ideas of Montgomery and Murphy in [Mur99].

MAGMA provides two methods for using the NFS implementation. The first is the one-step function `NFS`, which provides a naive NFS factorization attempt using default algorithm parameters.

The second, more powerful method is to work with an NFS process object, splitting the algorithm into four stages: Sieving, Auxiliary data, Linear algebra and Final factorization. This approach allows greater control over the algorithm, as the user may supply their own algorithm parameter values. It also allows the user to distribute the computationally intensive sieving and final factorization stages over several machines or processors.

Some functions are included to allow MAGMA users to co-operate in factorization attempts using CWI tools.

A verbose flag may be set to obtain informative printing on progress in the various stages of the NFS algorithm.

```
SetVerbose("NFS", v)
```

Set the verbose printing level for the NFS algorithms to the integer v. Currently the legal values for v are 0, 1, 2 and 3.

If the level is 0, no verbose output is produced.

If the level is 1, NFS will produce basic information about its progress, and will also print information on NFS algorithm parameters.

If the level is 2, NFS will provide more detailed information about progress and parameters.

If the level is 3, NFS will print out extremely detailed information about progress and data. This level will only be useful for experts and developers.

18.14.2 Naive NFS

MAGMA's Number Field Sieve implementation provides a one-step black-box function NFS. Here, the user provides the integer n to be factored, a homogeneous bivariate integer polynomial F and integers $m1$ and $m2$ such that $F(m1, m2) \equiv 0 \bmod n$. MAGMA will attempt to factor n using F, $m1$ and $m2$, automatically selecting the other parameters (see below) for the algorithm.

The automatically chosen parameters are NOT optimal in general, and therefore no conclusions should be drawn about the speed of the implementation or the algorithm itself based on the use of this function.

For example, note that the default algebraic factor base size of NFS is chosen to be rather large to decrease the likelihood of running out of useful relations. This slows the algorithm considerably, since it increases the size of the matrix to be reduced – but it also means that the algorithm should succeed in finding a factor unless one chooses a really bad polynomial.

```
NumberFieldSieve(n, F, m1, m2)
```

```
NFS(n, F, m1, m2)
```

Performs the factorization of an integer n using the Number Field Sieve with algebraic polynomial F, where the integers $m1$ and $m2$ satisfy $F(m1, m2) \equiv 0 \bmod n$. Returns a nontrivial factor of n if one is found, or 0 otherwise.

18.14.3 Factoring with NFS Processes

An NFS Process (an object of category NFSProc) encapsulates the data of a MAGMA NFS factorization. It contains the number n to be factored, the algebraic polynomial F and the integers m_1 and m_2. It also provides access to a number of NFS algorithm parameters (such as approximate factor base sizes). These parameters are attributes of the NFS process. If any of the parameters are not set, sensible (but not necessarily optimal) defaults will be provided by MAGMA.

The NFS algorithm is divided into four stages:

1. Sieving
2. Auxiliary data gathering
3. Linear algebra
4. Factorization

The stages are described in detail below.

After creating an NFS process for the factorization attempt, the user should proceed through each of the four stages in the above order.

```
NFSProcess(n, F, m1, m2)
```

Given a (composite) integer n, a bivariate homogeneous integer polynomial F, and nonzero integers $m1$ and $m2$ such that $F(m1, m2) \equiv 0 \bmod n$, this function creates an NFS process object for an NFS factorization of n.

Example H18E10

The attributes associated with an NFS process are:

```
> ListAttributes(NFSProc);
AlgebraicError              OutputFilename
AlgebraicFBBound            RationalError
AlgebraicLargePrimeBound    RationalFBBound
CacheSize                   RationalLargePrimeBound
F                           m1
Firstb                      m2
Lastb                       n
Maximuma
```

`OutputFilename` is the base name for NFS-generated data files. These files (and their actual names) are discussed below.
`AlgebraicFBBound` is the upper bound for smooth primes in the algebraic-side factor base, and `RationalFBBound`, the upper bound for smooth primes in rational-side factor base.
`Maximuma` bounds the sieve interval for a: NFS will sieve for relations with $|a| \leq$ `Maximuma`.
`Firstb` is the first value of b to sieve on, and `Lastb` is the last.
`AlgebraicLargePrimeBound` gives the upper bound for "large" (non-smooth) primes in the algebraic-side factor base. Similarly, `RationalLargePrimeBound` is the upper bound for the rational side.
`AlgebraicError` defines an "error" tolerance for logarithm arithmetic on algebraic side. Similarly, `RationalError` defines an "error" tolerance for the rational side.
`CacheSize` is a flag reflecting the computer cache memory size, for optimisation.

18.14.3.1 Attribute Selection

As a guideline for the selection of attributes, we include here a few examples of attributes that we have determined to be good for the MAGMA NFS implementation.

Example H18E11

Sample attributes for a 70-digit number:

```
> n := 52358696802333662953669045107254580530431112410356788979338022350609 27;
> R<X,Y> := PolynomialRing(Integers( ), 2);
> F := 2379600*X^4 - 12052850016*X^3*Y - 13804671642407*X^2*Y^2 +
>      11449640164912254*X*Y^3 + 7965530070546332840*Y^4 ;
> m1 := 6848906180202117;
> m2 := 1;
> P := NFSProcess(n,F,m1,m2);
> P`AlgebraicFBBound := 8*10^5;
> P`RationalFBBound := 6*10^5;
> P`OutputFilename := "/tmp/nfs_70_digit";
> P`Maximuma := 4194280;
> P`AlgebraicError := 16;
> P`RationalError := 14;
```

Example H18E12

Sample attributes for an 80-digit number:

```
> n := 187183186635768649345112272295104022206327935038373865025390693348907\
> 2483083589;
> P<X,Y> := PolynomialRing(Integers(),2);
> F := 15901200*X^4 + 9933631795*X^3*Y - 112425819157429*X^2*Y^2 -
>     231659214929438137*X*Y^3 - 73799500175565303965*Y^4;
> m1 := 1041619817688573426;
> m2 := 1;
> P := NFSProcess(n, F, m1, m2);
> P`AlgebraicFBBound := 8*10^5;
> P`RationalFBBound := 6*10^5;
> P`OutputFilename := "/tmp/nfs_80_dgit";
> P`Maximuma := 10485760;
> P`AlgebraicError := 16;
> P`RationalError := 14;
```

Example H18E13

Sample attributes for an 87-digit number:

```
> n := 12118618732463427472219179104631767765107839384219612469780841876821498\
> 2402918637227743;
> P<X,Y> := PolynomialRing(Integers(),2);
> F := 190512000*X^4 - 450872401242*X^3*Y +
>     1869594915648551*X^2*Y^2 + 2568544235742498*X*Y^3 -
>     93229655834198010101044*Y^4;
> m1 := 282411707411952732l1;
> m2 := 1;
> P := NFSProcess(n, F, m1, m2);
> P`AlgebraicFBBound := 16*10^5;
> P`RationalFBBound := 10^6;
> P`OutputFilename := "/tmp/nfs_87_digit";
> P`Maximuma := 2^24;
> P`AlgebraicError := 24;
> P`RationalError := 18;
```

The best choice for the factor base size depends on many variables, including the average log size and the Murphy α parameter (defined in [Mur99]) for the polynomial F. Our polynomials above are quite good: if the user does not know much about determining the quality of polynomials, then he or she should use much larger factor bases.

18.14.3.2 The Sieving stage

MAGMA's NFS uses a "line-by-line" (or "classical") sieving algorithm. Future versions may include lattice sieving.

The line-by-line siever sieves values of $F(a, b)$ on the algebraic side and corresponding values $a \cdot m2 - b \cdot m1$ on the rational side. This is done by fixing a value of b (beginning with the parameter `Firstb`, if supplied), then sieving all values of a between $-a_0$ and a_0, where a_0 is approximately equal to the parameter `Maximuma` (some rounding off is done to make sure that the sieve interval length is divisible by a high power of 2). When this is completed b is incremented, and the next value of b is processed.

The sieving continues until either the maximum value of b (specified by the parameter `Lastb`) has been reached, or until enough relations are obtained to complete the factorization. If `Lastb` is not defined, the sieve simply continues until enough relations are found. The number of relations required may be determined by the function `NumberOfRelationsRequired`.

"Cycles" among partial relations are counted after every 256 iterations.

The sieve implementation uses (rounded natural) logarithms of primes to mark the sieve interval. Moreover, the implementation does not sieve with prime powers. Therefore, we must allow for some error in scanning the sieve arrays for useful relations; the acceptable sieve threshold errors for each side are defined by the `AlgebraicError` and `RationalError` parameters. If, in addition, the user wants to take advantage of large prime relations (recommended), then larger error terms should be used. The implementation will keep relations having up to 2 large primes on each side, but will only find such relations if the user selects large enough sieve threshold error bounds. The user should be cautious when sieving for (and subsequently using) relations with large primes, as they greatly increase overall disk space requirements. Some experimentation may be required in order to determine the best error bounds for speed or disk space optimization purposes.

The `CacheSize` parameter may be used to take advantage of the cache memory size of the computer: a value of 1 indicates a small cache size, 2 a medium cache size, and 3 a for large cache size.

`NumberOfRelationsRequired(P)`

The minimum number of relations required for an NFS factor attempt with NFS process P.

`FindRelations(P)`

Given an NFS process P for factoring an integer n, generates relations to factor n with the Number Field Sieve algorithm. Returns the number of full relations plus the number of cycles found.

18.14.3.3 The Auxiliary data stage

In this stage of the algorithm, "cycles"[LD95] are detected in the partial relations from the sieving stage, and quadratic characters are calculated for the relations. This greatly improves the efficiency of the NFS.

In a typical factorization, the user should call the procedures `CreateCycleFile` and `CreateCharacterFile` in succession.

`CreateCycleFile(P)`

Creates a file with all the cycle information that the NFS algorithm requires to complete the matrix reduction and final factorization stages for the NFS process P.

`CycleCount(P)`

Returns the number of cycles in the partial relations of the NFS process P. This function is mainly intended for factoring with multiple processors.

`CycleCount(fn)`

Returns the number of cycles in the partial data file corresponding to the base file name fn. This function is mainly intended for factoring with multiple processors.

`CreateCharacterFile(P)`

Creates a file with the quadratic character data for the full relations and cycles in the NFS process P.

`CreateCharacterFile(P, cc)`

Creates a file with the quadratic character data for the full relations and cycles in the NFS process P. There are cc sets of 32 quadratic character columns created.

18.14.3.4 Finding dependencies: the Linear algebra stage

In this stage, the relations are collected together to form a matrix, and then block Lanczos reduction is applied to find linear dependencies among the relations. These dependencies become candidates for factorization.

`FindDependencies(P)`

Finds dependencies between relations in the NFS process P.

18.14.3.5 The Factorization stage

In this stage, number field square roots are extracted and we attempt to factor the dependencies found in the linear algebra stage.

`Factor(P)`

Try to factor with each dependency in the NFS process P until a proper factor is found. Returns the factor, or 0 if no factor is found.

`Factor(P,k)`

Attempt to factor with the k-th dependency in the NFS process P. Returns a proper factor if found, 0 otherwise.

18.14.4 Data files

Many data files are ued for an NFS factorization. The user can control the names and location of the files by specifying the `OutputFilename` parameter; then all output files will have names beginning with the `OutputFilename` string, with a range of suffixes depending on their purpose.

In general, all files are appended to rather than overwritten; so to avoid inconsistencies (and to save disk space) the user should call `RemoveFiles` after a successful factorization.

When distributing factorizations, or collecting results from sieving stages that have been broken up into several runs for some reason (for example, if a process has been interrupted), MAGMA provides the function `MergeFiles`. This takes a sequence of base filenames (which are treated as if they were the value for `OutputFilename`), and reads in the corresponding relation and partial relation files; it then combines the contents of these files, removing duplicates and corrupted lines of data, and places the results into new relation and partial relation files.

`RemoveFiles(P)`

Deletes any data files created by the NFS process P.

`MergeFiles(S, fn)`

Merges the NFS relation files named in the sequence S (and their associated partial relation files) into a pair of new relation and partial relation files, while removing duplicate and corrupted lines of data; returns the number of relations and the number of partial relations in the new output files. The combined full relations are stored in a file named fn, and the partial relations in a file named fn_partials.

18.14.4.1 MAGMA native NFS data files

Here we describe the files used in a typical MAGMA NFS factorization. These files all use formats peculiar to `Magma`'s NFS.

The first kind of file created by NFS stores the relations generated in the sieving stage by the `FindRelations` procedure. The name of the file is precisely the `OutputFilename` string.

NFS also stores partial relations generated in the sieving stage; these are stored in a file named `OutputFilename`_partials.

Whenever cycles [LD95] are counted (for example, in `CycleCount`, a file named `OutputFilename`_cycles is created to store them in. Some other files are also created and then deleted during the cycle counting process.

The quadratic characters calculated in `CreateCharacterFile` are stored in a file named `OutputFilename`_cc.

The linear algebra stage creates a file named `OutputFilename`_null_space, which lists relations making up null space vectors for the NFS matrix.

18.14.5 Distributing NFS Factorizations

MAGMA provides a number of tools for distributing the sieving and final factoring stages over a number of computers.

To distribute the sieving stage, each processor should get a unique range of b-values to sieve and unique data file names. During the sieving, the user must manually check when the combined data has enough relations to factor the number. To do this, the data files must first be merged using `MergeFiles`, and then the cycles can be counted with `CycleCount`. If the combined number of full relations plus the number of cycles exceeds the size of both factor bases combined, then the user can proceed to the other stages of the factorization attempt using the merged data file name.

To distribute the factorization stage, the user may choose a dependency for each process to factor, then call `Factor(P,k)` where P is the NFS process and k the number specifying the dependency to factor, with a different value of k for each process.

Example H18E14

Here we demonstrate a distributed NFS factorization (of a very small n) over two processes, A and B – which may be on different machines, or different magma processes on the same machine, or even in the same magma process.
We begin with process A:

```
> R<X,Y> := PolynomialRing(Integers( ),2);
> n := 704787824974797479087234958341;
> F := 814*X^4 + 3172*X^3*Y - 49218*X^2*Y^2 - 142775*X*Y^3
>      - 65862*Y^4;
> m1 := 3050411;
> m2 := 1;
> A := NFSProcess(n,F,m1,m2);
> A`Firstb := 0;
> A`Lastb := 99;
> A`OutputFilename := "/tmp/nfs-distrib-A";
> FindRelations(A);
3852
```

Now, process B, with n, F, $m1$ and $m2$ as above:

```
> B := NFSProcess(n,F,m1,m2);
> B`Firstb := 99;
> B`Lastb := 199;
> B`OutputFilename := "/tmp/nfs-distrib-B";
> FindRelations(B);
2455
```

Then later, on a single machine,

```
> input_files := ["/tmp/nfs-distrib-A","/tmp/nfs-distrib-B"];
> P := NFSProcess(n,F,m1,m2);
> P`OutputFilename := "/tmp/nfs-distrib-all";
> MergeFiles(input_files, P`OutputFilename);
```

```
4162 25925
> CycleCount(P);
4368
> CreateCycleFile(P);
> CreateCharacterFile(P);
> FindDependencies(P);
```

Now, the final factorization stage may be distributed over more than one processor also. We attempt to factor a relation on A:

```
> A‘OutputFilename := "/tmp/nfs-distrib-all";
> Factor(A,9);  // factor dependency 9
0
```

No factor was found on machine A, but meantime on B:

```
> B‘OutputFilename := "/tmp/nfs-distrib-all";
> Factor(P,1);  // factor dependency 1
94899629
> n mod $1, n div $1;
0 74266575624650207929
```

We have a successful factorisation.

18.14.6 MAGMA and CWI NFS Interoperability

At the time of writing, the record NFS factorizations were lead by the CWI group and by people using CWI's or Arjen Lenstra's code. The CWI tools use a different data file format to MAGMA's native format, but MAGMA supplies some tools to allow users to assist in CWI factorization attempts.

The user may generate relations in CWI relation format, rather than MAGMA native format, by using `FindRelationsInCWIFormat`. The user should note that relations in CWI format cannot at present be used in the Auxiliary data, Linear algebra or Factorization stages of the MAGMA NFS.

Alternatively, assuming some MAGMA NFS relations have already been computed for a process, then the user may use the procedure `ConvertToCWIFormat` to convert the relation data files from MAGMA native format to CWI format. The resulting data file is named `OutputFilename`_CWI_format, and will contain both the full and partial relations of the process.

`FindRelationsInCWIFormat(P)`

Given an NFS process P for factoring an integer n, generates relations to factor n with the Number Field Sieve algorithm, in a file format suitable for use with CWI's NFS tools. Returns the number of relations found.

`ConvertToCWIFormat(P, pb)`

Converts the relation files of the NFS process P to CWI format, storing primes only greater than or equal to the prime printing bound pb. The resulting data file name will be named `P‘OutputFilename`_CWI_format.

18.14.7 Tools for Finding a Suitable Polynomial

MAGMA does not provide a function to select an optimal polynomial for the factorization of a given number. However, MAGMA does provide some functions that are useful for the implementation of the polynomial selection algorithms developed by Peter Montgomery and Brian Murphy in [Mur99].

The functions `BaseMPolynomial`, `MurphyAlphaApproximation`, `OptimalSkewness`, `BestTranslation`, `PolynomialSieve`, and `DickmanRho`, will be useful for those wanting to implement polynomial selection routines within the MAGMA interpreter language.

`BaseMPolynomial(n, m, d)`

Given integers n, m and d, returns a homogeneous bivariate polynomial $F = \sum_{i=0}^{d} c_i X^i Y^{d-i}$ such that the coefficients c_i give a base m representation of n: that is, $\sum_{i=0}^{d} c_i m^i = n$. The coefficients also satisfy $|c_i| \leq m/2$.

This polynomial F may be used to factorize n using the number field sieve (with $m_1 := m$ and $m_2 := 1$).

This function requires that $d \geq 2$ and $n \geq m^d$.

`MurphyAlphaApproximation(F, b)`

Given a univariate or homogeneous bivariate polynomial F, return an approximation of the α value of F, using primes less than the positive integer bound b.

The α value of a polynomial is defined in [Mur99].

Since random sampling is used for primes dividing the discriminant, successive calls to this function will give slightly different results.

`OptimalSkewness(F)`

Given a univariate or homogeneous bivariate polynomial F, return its optimal skewness and corresponding average log size.

The optimal skewness and average log size values are defined in [Mur99].

Example H18E15

This example illustrates an effective (though not optimal) method for finding a "good" polynomial for use in NFS factorizations.
Here we search for a degree $d = 4$ polynomial to use in factoring a 52-digit integer n.
We define the rating of a polynomial to be the sum of the α value and corresponding "average log size" (see [Mur99]).
We then proceed by iterating over base m polynomials with successive leading coefficients (with the values of m near $(m^{1/d} m^{1/d+1})^{1/2}$, and chosen to minimize the second-to-leading coefficient), and choosing as a result the polynomial with the smallest rating.

```
> n := RandomPrime(90)*RandomPrime(90);
> n;
359635470725625320407673937416777014871521894980388 9
> d := 4;
> approx_m := Iroot( Iroot( n, d+1 ) * Iroot( n, d ) , 2 );
> leading_coeff := n div approx_m^d;
```

```
> leading_coeff;
143082
> m := Iroot( n div leading_coeff, d );
> P<X,Y> := PolynomialRing( Integers(), 2 );
> F<X,Y> := BaseMPolynomial(n,m,d);
> F;
143082*X^4 + 463535*X^3*Y - 173869838910*X^2*Y^2 + 167201617413*X*Y^3 +
    159859288415*Y^4
> skew, als := OptimalSkewness( F );
> alpha := MurphyAlphaApproximation( F, 2000 );
> rating := als + alpha;
> rating;
23.14371454891457519331 4917
>
> best_rating := rating;
> best_m := m;
> for i in [1..100] do
>   leading_coeff := leading_coeff + 1;
>   m := Iroot( n div leading_coeff, d );
>   F<X,Y> := BaseMPolynomial(n,m,d);
>   skew, als := OptimalSkewness( F );
>   alpha := MurphyAlphaApproximation( F, 2000 );
>   rating := als + alpha;
>   if rating lt best_rating then
>     best_rating := rating;
>     best_m := m;
>   end if;
> end for;
> best_rating;
20.899568473033257031950385
> best_m;
398116527578
> F<X,Y> := BaseMPolynomial(n,best_m,d);
> F;
143160*X^4 + 199085*X^3*Y - 9094377652*X^2*Y^2 - 93898749030*X*Y^3 -
    169859083883*Y^4
> OptimalSkewness( F );
165.514255523681640625 20.969934467920612180646408
> MurphyAlphaApproximation( F, 2000 );
-0.054271615763014144950015084 2
> time NFS( n, F, best_m, 1 );
...
```

`BestTranslation(F, m, a)`

Given a univariate or homogeneous bivariate polynomial F, an integer m, and real value a (which should be the average log size of F for some optimal skewness), returns a polynomial G and an integer m' such that $G(m') = F(m)$, together with the average log size and optimal skewness of G. The translation G is selected such that the average log size is a local minimum.

`PolynomialSieve(F, m, J0, J1,MaxAlpha)`

`PrimeBound` RngIntElt *Default :* 1000

Given a homogeneous bivariate integer polynomial F of degree d, together with integers m, J_0 and J_1 and a real value `MaxAlpha`, returns a list of tuples, each of which contains a polynomial $G = F + j_1 x^2 y^{d-2} - (|j_0| + j_1 m)xy^{d-1} + (j_0 m)y^d$, where $|j_0| \leq J_0$ and $|j_1| \leq J_1$ such that the α value (see [Mur99]) of G is "better" (that is, lower) than `MaxAlpha`.

Each tuple contains the data <average log size + α, skewness, α, G, m, j_0, j_1 >.

If the optional parameter `PrimeBound` is set, it is used as an upper bound for primes used to calculate α.

18.15 Bibliography

[**AHU74**] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[**AM93**] A. O. L. Atkin and F. Morain. Elliptic curves and primality proving. *Math. Comp.*, 61:29 – 68, 1993.

[**Bre80**] R. P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20:176–184, 1980.

[**BtR92**] R. P. Brent and H. J. J te Riele. Factorizations of $a^n \pm 1$, $13 \leq a < 100$. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, 1992. URL: ftp://nimbus.anu.edu.au/pub/Brent.

[**Jae93**] G. Jaeschke. On strong pseudoprimes to several bases. *Math. Comp.*, 61:915 – 926, 1993.

[**Knu97**] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, Massachusetts, 3rd edition, 1997.

[**LD95**] Arjen K. Lenstra and Bruce Dodson. NFS with four large primes: An explosive experiment. In Don Coppersmith, editor, *Advances in cryptology—CRYPTO 1995*, volume 963 of *LNCS*, pages 372–385, Berlin, 1995. Springer.

[**Mar95**] G. Marsaglia. DIEHARD: a battery of tests of randomness. URL:http://stat.fsu.edu/pub/diehard/, 1995.

[**Mar00**] G. Marsaglia. The Monster, a random number generator with period 10^{2857} times as long as the previously touted longest-period one. Preprint, 2000.

[**Mon92**] Peter Lawrence Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization*. PhD thesis, University of California, Los Angeles, 1992.

[**Mur99**] Brian Murphy. *Polynomial selection for the number field sieve integer factorisation algorithm*. PhD thesis, Oxford University, 1999. URL:http://web.comlab.ox.ac.uk/oucl/work/richard.brent/ftp/Murphy-thesis.ps.gz.

[**Sch71**] Arnold Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

[**vzGG99**] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, 1999.

[**Web95**] Kenneth Weber. The Accelerated Integer GCD Algorithm. *ACM Transactions on Mathematical Software*, 21(1):111–122, 1995.

19 INTEGER RESIDUE CLASS RINGS

Chapter 19
INTEGER RESIDUE CLASS RINGS

19.1 Introduction

This chapter presents the machinery provided in MAGMA for computing in quotient rings of the ring of integers **Z**, that is, integer residue class rings. The first half of the chapter describes operations with ideals of **Z** and their quotient rings while the second half provides an introduction to computing with Dirichlet characters.

19.2 Ideals of Z

The theory of ideals of **Z** is very elementary but for completeness the general machinery for ring ideals applies. Such ideals will have type `RngInt`, that is, the same type as the ring of integers itself (`ideal<Integers() | 1>`).

In the case of **Z** any subring is an ideal so that the `sub`-constructor creates the same object as does the `ideal`-constructor.

`ideal< R | a >`

Given the ring of integers **Z** and an integer a, return the ideal of **Z** generated by a.

Example H19E1

We construct some ideals of **Z**.

```
> Z := IntegerRing();
> I13 := ideal< Z | 13 >;
> I13;
Ideal of Integer Ring generated by 13
> 1 in I13;
false
> 0 in I13;
true
> -13 in I13;
true
> I0 := ideal< Z | 0 >;
> 0 in I0;
true
> 1 in I0;
false
```

We check that that **Z** is regarded as an ideal.

```
> I1 := ideal< Z | 1 >;
```

```
> I1 eq Z;
true
```

19.3 Z as a Number Field Order

A collection of functions are provided that make **Z** behave like an order of a number field. Note however, that **Z** is not of type `RngOrd`. If complete compatibility is necessary, the user should create the maximal order of a degree 1 extension of **Q**.

`Decomposition(R, p)`

Returns the ideal decomposition of the prime p, i.e. a list `[ < ideal<Z|p>, 1> ]` as in the number field case.

`Generator(I)`

A generator for the given ideal.

`RamificationIndex(I, p)`

`RamificationIndex(I)`

The ramification index of I over **Z** which is always 1.

`Degree(I)`

The inertia degree of the ideal I, which is always 1.

`TwoElementNormal(I)`

Two integers that generate the ideal I. In this case the generator is returned twice.

`ChineseRemainderTheorem(I, J, a, b)`

The Chinese remainder theorem for ideals. Given ideals I and J of **Z** together with integers a and b, an integer x such that $x - a \in I$ and $x - b \in J$ is returned.

`Valuation(x, I)`

The valuation of the integer x at the prime ideal I.

`ClassRepresentative(I)`

The representative of the ideal I of **Z** in the basis of the class group.

19.4 Residue Class Rings

The ring $\mathbf{Z}/m\mathbf{Z}$ consists of representatives for the residue classes of integers modulo $m > 1$. This Section describes the operations in MAGMA for such rings and their elements.

At any stage during a session, MAGMA will have at most one copy of $\mathbf{Z}/m\mathbf{Z}$ present, for any $m > 1$. In other words, different names for the same residue class ring will in fact be different references to the same structure. This saves memory and avoids confusion about different but isomorphic structures.

If m is a prime number, the ring $\mathbf{Z}/m\mathbf{Z}$ forms a field; however, MAGMA has special functions for dealing with finite fields. The operations described here should *not* be used for finite field calculations: the implementation of finite field arithmetic in MAGMA as described in Chapter 21 takes full advantage of the special structure of finite fields and leads to superior performance.

19.4.1 Creation

In addition to the general quotient constructor, a number of abbreviations are provided for computing residue class rings.

```
quo< Z | I >
```

Given the ring of integers Z, and an ideal I, create the residue class ring modulo the ideal.

```
quo< Z | m >
```

Given the ring of integers Z, and an integer $m \neq 0$, create the residue class ring $\mathbf{Z}/m\mathbf{Z}$.

```
ResidueClassRing(m)
IntegerRing(m)
Integers(m)
RingOfIntegers(m)
```

Given an integer greater than zero, create the residue class ring $\mathbf{Z}/m\mathbf{Z}$.

```
ResidueClassRing(Q)
IntegerRing(Q)
Integers(Q)
```

Create the residue class ring $\mathbf{Z}/m\mathbf{Z}$, where m is the integer corresponding to the factorization sequence Q. This is more efficient than creating the ring by m alone, since the factorization Q will be stored so it can be reused later.

Example H19E2

We construct a residue ring having modulus the largest prime not exceeding 2^{16}.

```
> p := PreviousPrime(2^16);
> p;
65521
> R := ResidueClassRing(p);
Residue class ring of integers modulo 65521
```

Now we try to find an element x in R such that $x^3 = 23$.

```
> exists(t){x : x in R | x^3 eq 23};
true
> t;
12697
```

19.4.2 Coercion

As can be seen from the tables in Chapter 17, automatic coercion takes place between $\mathbf{Z}/m\mathbf{Z}$ and $\mathbf{Z}$ so that a binary operation like + applied to an element of $\mathbf{Z}/m\mathbf{Z}$ and an integer will result in a residue class from $\mathbf{Z}/m\mathbf{Z}$.

Using `!`, elements from a prime field $\mathbf{F}_p$ can be coerced into $\mathbf{Z}/p\mathbf{Z}$, and elements from $\mathbf{Z}/p\mathbf{Z}$ can be coerced into $\mathbf{F}_{p^r}$. Also, transitions between $\mathbf{Z}/m\mathbf{Z}$ and $\mathbf{Z}/n\mathbf{Z}$ can be made using `!` provided that m divides n or n divides m. In cases where there is a choice – such as when an element r from $\mathbf{Z}/m\mathbf{Z}$ is coerced into $\mathbf{Z}/n\mathbf{Z}$ with m dividing n – the result will be the residue class containing the representative for r.

Example H19E3

```
> r := ResidueClassRing(3) ! 5;
> r;
2
> ResidueClassRing(6) ! r;
2
```

So the representative 2 of 5 mod 3 is mapped to the residue class 2 mod 6, and not to 5 mod 6.

19.4.3 Elementary Invariants

`Characteristic(R)` `# R`

`Modulus(R)`

Given a residue class ring $R = \mathbf{Z}/m\mathbf{Z}$, this function returns the common modulus m for the elements of R.

`FactoredModulus(R)`

Given a residue class ring $R = \mathbf{Z}/m\mathbf{Z}$, this function returns the factorization of the common modulus m for the elements of R.

19.4.4 Structure Operations

`AdditiveGroup(R)`

Given $R = \mathbf{Z}/m\mathbf{Z}$, create the abelian group of integers modulo m under addition. This returns the finite additive abelian group A (of order m) together with a map from A to the ring $\mathbf{Z}/m\mathbf{Z}$, sending $A.1$ to 1.

`MultiplicativeGroup(R)`
`UnitGroup(R)`

Given $R = \mathbf{Z}/m\mathbf{Z}$, create the multiplicative group of R as an abelian group. This returns an (additive) abelian group A of order $\phi(m)$, together with a map from A to R.

`sub< R | n >`

Given R, the ring of integers modulo m or an ideal of it, and an element n of R create the ideal of R generated by n.

`Set(R)`

Create the enumerated set consisting of the elements of the residue class ring R.

`Category(R)` `Parent(R)` `PrimeRing(R)`
`Center(R)`

19.4.5 Ring Predicates and Booleans

IsCommutative(R) IsUnitary(R)
IsFinite(R) IsOrdered(R)
IsField(R) IsEuclideanDomain(R)
IsPID(R) IsUFD(R)
IsDivisionRing(R) IsEuclideanRing(R)
IsPrincipalIdealRing(R) IsDomain(R)
R eq R R ne R

19.4.6 Homomorphisms

Ring homomorphisms with domain $\mathbf{Z}/m\mathbf{Z}$ are completely determined by the image of 1. As usual (see Chapter 18), we require our homomorphisms to map 1 to 1. Therefore, the general homomorphism constructor with domain $\mathbf{Z}/m\mathbf{Z}$ needs no arguments.

hom< R -> S | >

Given a residue class ring R, and a ring S, create a homomorphism from R to S, determined by $f(1_R) = 1_S$. Note that it is the responsibility of the user that the map defines a homomorphism!

19.5 Elements of Residue Class Rings

19.5.1 Creation

elt< R | k >

Create the residue class containing the integer k in residue class ring R.

R ! k

Create the residue class containing k in the residue class ring R. Here k is allowed to be either an integer, or an element of the finite field $\mathbf{F}_p$ in the case $R = \mathbf{Z}/p\mathbf{Z}$, or an element of $S = \mathbf{Z}/n\mathbf{Z}$ for a multiple or divisor n of m (with $R = \mathbf{Z}/m\mathbf{Z}$).

One(R) Identity(R)
Zero(R) Representative(R)

These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively, in any $\mathbf{Z}/m\mathbf{Z}$.

Random(R)

Create a "random" residue class in R.

19.5.2 Arithmetic Operators

`+ n` `- n`

`m + n` `m - n` `m * n` `n ^ k` `m / n` `m div n`

`m +:= n` `m -:= n` `m *:= n` `m /:= n` `m ^:= k`

19.5.3 Equality and Membership

`m eq n` `m ne n`

`n in R` `n notin R`

19.5.4 Parent and Category

`Parent(n)` `Category(n)`

19.5.5 Predicates on Ring Elements

`IsZero(n)` `IsOne(n)` `IsMinusOne(n)`

`IsNilpotent(n)` `IsIdempotent(n)`

`IsUnit(n)` `IsZeroDivisor(n)` `IsRegular(n)`

`IsIrreducible(n)` `IsPrime(n)`

19.5.6 Solving Equations over $\mathbf{Z}/m\mathbf{Z}$

`Solution(a, b)`

Given elements a and b of $\mathbf{Z}/m\mathbf{Z}$, return a solution x to the linear congruence $a \cdot x = b \in \mathbf{Z}/m\mathbf{Z}$. An error is signalled if no solution exists.

`IsSquare(n)`

`Factorization` [<RngIntElt, RngIntElt>]

Given an element $n \in \mathbf{Z}/m\mathbf{Z}$ this function returns `true` if there exists $a \in \mathbf{Z}/m\mathbf{Z}$ such that $a^2 = n \in \mathbf{Z}/m\mathbf{Z}$, `false` otherwise. If n is a square, a square root a is also returned. If m is large and its prime factorization is known, the computation may be speeded up by assigning the factorization sequence for m to the optional argument `Factorization`.

`Sqrt(a)`

`SquareRoot(a)`

`Factorization` [<RngIntElt, RngIntElt>]

Given an element a of the ring $\mathbf{Z}/m\mathbf{Z}$, this function returns an element b of $\mathbf{Z}/m\mathbf{Z}$ such that $b^2 = a \in \mathbf{Z}/m\mathbf{Z}$, if such an element exists, and an error otherwise. If m is large and its prime factorization is known, the computation may be speeded up by assigning the factorization sequence for m to the optional argument `Factorization`.

`AllSquareRoots(a)`

`AllSqrts(a)`

`Factorization` [<RngIntElt, RngIntElt>]

Return a sequence containing all square roots of the element a in a residue class ring $\mathbf{Z}/m\mathbf{Z}$. If the modulus m is large and its prime factorization is known, the computation may be speeded up by assigning the factorization sequence for m to the optional argument `Factorization`.

Example H19E4

We construct the residue class ring having modulus 2340 and find all the square roots of 1404.

```
> R := ResidueClassRing(2340);
Residue class ring of integers modulo 2340
> x := R!1404;
> sqrts := AllSquareRoots(x);
> sqrts;
[ 78, 312, 468, 702, 858, 1092, 1248, 1482, 1638,
  1872, 2028, 2262 ]
> [ y^2 : y in sqrts ];
[ 1404, 1404, 1404, 1404, 1404, 1404, 1404, 1404,
  1404, 1404, 1404, 1404 ]
```

So 1404 has 12 square roots!

19.6 Ideal Operations

`ideal< R | a1, ..., ar >`

The ideal of the residue ring R generated by the greatest common divisor of the elements a_i and the modulus of R.

`GreatestCommonDivisor(a, b)`
`Gcd(a, b)`
`GCD(a, b)`

Greatest common divisor of the elements a and b of R, that is, a generator for the R-ideal $(a) + (b)$.

`GreatestCommonDivisor(Q)`
`Gcd(Q)`
`GCD(Q)`

Greatest common divisor of the sequence of elements Q, that is, a generator for the R-ideal generated by the elements in Q.

`LeastCommonMultiple(a, b)`
`Lcm(a, b)`
`LCM(a, b)`

Least common multiple of the elements a and b of R, that is, a generator for the R-ideal $(a) \cap (b)$.

`LeastCommonMultiple(Q)`
`Lcm(Q)`
`LCM(Q)`

Least common multiple of the sequence of elements Q, that is, a generator for the R-ideal formed by the intersection of the principal ideals generated by elements of Q.

`I + J` `I * J` `I meet J`
`a in I` `a notin I`
`I eq J` `I ne J`
`I subset J` `I notsubset J`

19.7 The Unit Group

`UnitGroup(R)`

Given $R = \mathbf{Z}/m\mathbf{Z}$, construct the unit group of R as an abelian group. This returns an (additive) abelian group A of order $\phi(m)$, together with a map from A to R.

`IsPrimitive(n)`

Returns `true` if the element $n \in \mathbf{Z}/m\mathbf{Z}$ is primitive, that is, if it generates the multiplicative group of $\mathbf{Z}/m\mathbf{Z}$, `false` otherwise.

`PrimitiveElement(R)`

`PrimitiveRoot(R)`

Given $R = \mathbf{Z}/m\mathbf{Z}$, this function returns a generator for the group of units of R if this group is cyclic, and returns 0 otherwise. Thus a valid generator is only returned if $m = 2, 4, p^t$ or $2p^t$, with p an odd prime and $t \geq 1$.

`Order(a)`

Given an element a belonging to $\mathbf{Z}/m\mathbf{Z}$, return the multiplicative order $k \geq 1$ of a if a is in the unit group $(\mathbf{Z}/m\mathbf{Z})^*$, and zero if a is not a unit.

`Normalize(x)`

`Normalise(x)`

Given an element $x \in R = \mathbf{Z}/m\mathbf{Z}$, this function returns the unique canonical associate $y \in R$ of x and a unit $u \in R$ such that $u \cdot x = y$. The canonical associate of x is the GCD of x and m, considered as natural integers (unless x is 0, in which case it is 0).

Example H19E5

We determine the unit group of the ring with modulus 735 and then verify its order by comparing it with $\phi(m)$.

```
> m := 735;
> R := ResidueClassRing(m);
Residue class ring of integers modulo 735
> U, psi := UnitGroup(R);
> U;
Abelian Group isomorphic to Z/2 + Z/2 + Z/84
Defined on 3 generators
Relations:
    2*U.1 = 0
    4*U.2 = 0
    42*U.3 = 0
> #U;
336
> EulerPhi(735);
```

```
336
```

So the order of U is equal to $\phi(m)$ as it should be. Finally, we look for three elements of R that generate the unit group.

```
> gens := [ psi(U.i) : i in [1..3] ]; gens;
> [ Order(x) : x in gens ];
[ 2, 4, 42 ]
```

Example H19E6

We construct a residue class ring $R = \mathbf{Z}/m\mathbf{Z}$ having cyclic unit group. By a theorem of Gauss, the ring R has cyclic unit group precisely when $n = 4, n = p^e$, or $n = 2p^e$, and p is an odd prime.

```
> R := IntegerRing(50);
> U, psi := UnitGroup(R);
Abelian Group isomorphic to Z/20
Defined on 1 generator
Relations:
    20*U.1 = 0
> w := PrimitiveElement(R);
> w;
3
> Order(w);
20
```

We verify that the powers of w are precisely the elements of the unit group U.

```
> powers := { w^i : i in [0..19] };
> powers;
{ 29, 1, 31, 3, 33, 7, 37, 9, 39, 11, 41, 13, 43, 17, 47, 19, 49, 21, 23, 27 }
> powers eq { psi(u) : u in U };
true
```

19.8 Dirichlet Characters

Let R be a ring. Then a *Dirichlet character over R of modulus N* is a homomorphism

$$\varepsilon : (\mathbf{Z}/N\mathbf{Z})^* \to R^*,$$

where R^* is the group of invertible elements of R. We extend ε to a set theoretic map on the whole of $\mathbf{Z}$ by defining $\varepsilon(x) = 0$ if $\gcd(x, N) \neq 1$. The *conductor* of ε is the smallest positive integer M such that the homomorphism $(\mathbf{Z}/N\mathbf{Z})^* \to R^*$ factors through $(\mathbf{Z}/M\mathbf{Z})^*$ via the natural map $(\mathbf{Z}/N\mathbf{Z})^* \to (\mathbf{Z}/M\mathbf{Z})^*$.

19.8.1 Creation

`DirichletGroup(N)`

The group of Dirichlet characters modulo N with image in `RationalField()`. Note that this is a group of exponent at most 2.

`DirichletGroup(N,R)`

The group of Dirichlet characters modulo N with image in the ring R. Here R can be the integers, rationals, a number field or a finite field.

`DirichletGroup(N,R,z,r)`

The group of Dirichlet characters mod N with image in the order-r cyclic subgroup of the ring R generated by the root of unity z. Here z must be an element of R of exact order r.

`FullDirichletGroup(N)`

The group of Dirichlet characters modulo N taking values in the mth cyclotomic field, where m is the exponent of the unit group modulo N. (This is a shortcut for the previous command.)

`BaseExtend(G, R)`
`BaseExtend(G, R, z)`

The group of Dirichlet characters corresponding to G with values in the ring R. In the second form, the distinguished root of unity of the base ring of G is identified with the given element z.

`AssignNames(~G, S)`

Assign names to the generators of the Dirichlet group G.

19.8.2 Element Creation

`Elements(G)`

A sequence containing all Dirichlet characters in the Dirichlet group G.

`Random(G)`

A random element of the Dirichlet group G.

`G . i`

The ith generator of the group G.

`G ! x`

This coerces the given element x into the Dirichlet group G. Here x may be a Dirichlet character belonging to a different group, or a sequence of integers specifying an element of the `AbelianGroup` of G.

`KroneckerCharacter(D)`

`KroneckerCharacter(D, R)`

The Kronecker character $n \mapsto (d/n)$, where d is the fundamental discriminant associated to the integer D.

When a ring R is given, this is returned as a character with values in R.

19.8.3 Properties of Dirichlet Groups

`BaseRing(G)`

The ring in which characters in G take values.

`Modulus(G)`

The integer N such that G is a group of Dirichlet characters on $\mathbf{Z}/N$.

`Order(G)`

The order of the Dirichlet group G.

`Exponent(G)`

The exponent of the Dirichlet group G.

`AbelianGroup(G)`

This returns a finite abelian group isomorphic to the given group G of Dirichlet characters (as an abstract group), and secondly returns a map from the abstract group to G.

It is necessary to use this function in order to make group theoretic constructions involving G.

`NumberOfGenerators(G)`

The number of generators of the Dirichlet group G.

`Generators(G)`

A sequence containing generators for the Dirichlet group G.

`G . i`

The ith generator of the group G.

`UnitGenerators(G)`

This returns an ordered sequence of integers that reduce to "canonical" generators of the unit group of Z/N, where N is the modulus of G.

19.8.4 Properties of Elements

`BaseRing(chi)`

The ring in which the Dirichlet character χ takes values.

`Modulus(chi)`

The modulus of the group of Dirichlet characters that contains χ.

`Conductor(chi)`

The minimal conductor of the Dirichlet character χ. (That is, the smallest integer M such that `chi` is well-defined on the unit group of Z/M.)

`ElementToSequence(chi)`

A sequence of integers specifying the Dirichlet character χ (in terms of generators of the group containing χ).

`x eq y`

Return `true` iff the given characters have the same modulus and values.

`Order(chi)`

The order of the given element χ in a group of Dirichlet characters.

`IsTrivial(chi)`

Returns `true` if and only if the Dirichlet character χ has order 1.

`IsPrimitive(chi)`

Returns `true` iff the Dirichlet character χ is primitive (equivalently, if its conductor equals its modulus).

`AssociatedPrimitiveCharacter(chi)`

The primitive character modulo the conductor of χ which takes the same values (on units) as χ.

`IsEven(chi)`

Returns `true` if and only if `Evaluate(chi,-1)` is equal to 1. Note that in characteristic 0, the space of modular forms of weight k and character χ is zero if χ is even and k is odd.

`IsOdd(chi)`

Returns `true` if and only if `Evaluate(chi,-1)` is equal to -1. Note that in characteristic 0, the space of modular forms of weight k and character χ is zero if χ is odd and k is even.

`IsTotallyEven(chi)`

For a Dirichlet character χ, this is `true` if and only if every character in the `Decomposition` of χ (into prime power components) is even.

`Decomposition(chi)`

This decomposes the Dirichlet character χ as a product of characters with prime power moduli. The function returns a list (not a sequence) containing these characters (which do not belong to the same group).

`GaloisConjugacyRepresentatives(G)`

`GaloisConjugacyRepresentatives(seq)`

This returns a sequence containing one representative from each Galois conjugacy class (over $\mathbf{Q}$) of characters corresponding to a character in the given group or the given sequence.

`MinimalBaseRingCharacter(chi)`

The returns a character which is the same as χ, except which takes values in the smallest possible subring of the base ring of χ.

19.8.5 Evaluation

`Evaluate(chi,n)`

`chi(n)`

The value of the Dirichlet character χ at the integer n.

`ValueList(chi)`

A sequence containing the values $[\chi(1), .., \chi(N)]$ of the given character χ, where N is the modulus of χ.

The list of values is stored; then in later calls to `Evaluate`, the stored value is returned.

`ValuesOnUnitGenerators(chi)`

A sequence containing the values of χ on the ordered sequence of elements of $\mathbf{Z}/m$ given by `UnitGenerators(Parent(chi))`, where m is the modulus of χ.

`OrderOfRootOfUnity(r, n)`

Given an element r of some ring which is *assumed* to satisfy $r^n = 1$, this returns the smallest integer m such that $r^m = 1$.

(This provides a convenient way to calculate the order of values of non-real characters.)

19.8.6 Arithmetic

`x * y`

`x / y`

The product or quotient (respectively) of the Dirichlet characters x and y. This is a Dirichlet character of modulus equal to the least common multiple of the moduli of x and y. The base rings and chosen roots of unity of the parents of x and y are equal.

`x ^ n`

The Dirichlet character x raised to the power of n, where n is any integer.

`x ^ phi`

The image of the Dirichlet character x under the automorphism ϕ.

`Sqrt(x)`

Given a Dirichlet character x of odd order, this returns a square root of x (in the same group).

19.8.7 Example

Example H19E7

We begin by constructing the group of characters $(\mathbf{Z}/5\mathbf{Z})^* \to \mathbf{Q}^*$.

```
> G<a> := DirichletGroup(5);  G;  // The default base field is Q.
Group of Dirichlet characters of modulus 5 over Rational Field
> #G;
2
> [Evaluate(a, n) : n in [1..5]];
[ 1, -1, -1, 1, 0 ]
> Eltseq(a);
[ 2 ]
> a eq G![2];
true
> IsEven(a);
true
> IsOdd(a);
false
> IsTrivial(a);
false
```

Next we create a character by building it up "locally".

```
> G1<a4> := DirichletGroup(4);
> Conductor(a4);
4
> G2<a5> := DirichletGroup(25);
```

```
> Conductor(a5);
5
> eps := a4*a5;
> Modulus(eps);
100
> Conductor(eps);
20
> Evaluate(eps,7) eq Evaluate(a4,7)*Evaluate(a5,7);
true
```

Characters can be constructed over various fields.

```
> G<a> := DirichletGroup(7,GF(7));
> #G;
6
> Evaluate(a,2);
2
>
> G<a3,a5> := DirichletGroup(15,CyclotomicField(EulerPhi(15)));
> G;
Group of Dirichlet characters of modulus 15 over Cyclotomic Field of
order 8 and degree 4
> #G;
8
> Conductor(a3);
3
> Conductor(a5);
5
> Order(a5);
4
> Evaluate(a5,2);
zeta_8^2
```

If D is a fundamental discriminant, then `KroneckerCharacter(D)` is the quadratic Dirichlet character corresponding to the quadratic field $\mathbf{Q}(\sqrt{D})$. The following code verifies that `KroneckerCharacter` and `KroneckerSymbol` agree in the case $D = 209$.

```
> chi := KroneckerCharacter(209);
> for n in [1..209] do
>    assert Evaluate(chi,n) eq KroneckerSymbol(209,n);
> end for;
```

If E is an elliptic curve with newform f_E, then the twist E_D corresponds to f_E twisted by this character, as illustrated below.

```
> E := EllipticCurve(CremonaDatabase(),"11A");
> f := qEigenform(E,8); f;
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
> chi := KroneckerCharacter(-7);
> qEigenform(QuadraticTwist(E,-7),8);
q - 2*q^2 + q^3 + 2*q^4 - q^5 - 2*q^6 + O(q^8)
```

```
> R<q> := Parent(f);
> &+[Evaluate(chi,n)*Coefficient(f,n)*q^n : n in [1..7]] + O(q^8);
q - 2*q^2 + q^3 + 2*q^4 - q^5 - 2*q^6 + O(q^8)
```

---

20 RATIONAL FIELD

Chapter 20
RATIONAL FIELD

20.1 Introduction

This Chapter describes functions relating to the field of rational numbers **Q**. Note that most functions for rational integers can be found in Chapter 18.

The rational field **Q** is automatically created when Magma is started up. That means that in **Q**, unlike most other structures, arithmetic can be done without the need to create the structure explicitly first. The same is true for the ring of integers.

In order to be compatible with the other rings and fields, `Q.1` will return 1.

20.1.1 Representation

Rational numbers are stored as pairs of numerator and denominator. Whenever a rational number is created, it will be put in reduced form (coprime numerator and denominator, positive denominator). It is well possible that a rational number has denominator 1, and thus represents a rational integer; in such cases it will however never automatically be converted into an integer (that is, its type will not be changed).

20.1.2 Coercion

The tables in Chapter 17 describe which coercions of rational numbers are allowed, and which will take place automatically when necessary. As a general rule, automatic coercion occurs between elements of **Q** and elements of any ring R of characteristic 0. That means, for example, that addition of any rational number and an element r of such ring can be performed without the need to coerce the elements first; the result will be in the larger of **Q** and R (usually R, unless R is a subring of **Q** such as **Z**). The most important exceptions to the above rule are those cases where the result would lie in a structure strictly larger than both **Q** and R. Examples of this are $R = \mathbf{Z}[x]$, and the result would generally be in $\mathbf{Q}(x)$, and $R = O_K$, an order in a number field (and the result could be in K).

Example H20E1

We give three examples of successful automatic coercion, and one where it does not work. Note that in the third case the result, although being integral, is still in the rational field.

```
> 1/2 + elt< CyclotomicField(3) | 1,2>;
1/2*(4*zeta_3 + 3)
> 1/2 - 0.12345;
0.37655
> 1/2 * 2;
1
> Parent(1/2 * 2);
```

```
Rational Field
>  R<x> := PolynomialRing(Integers());
>  // The following produces an error:
>  1/2 + x;
>> 1/2 + x;
       ^
Runtime error in '+': Bad argument types
```

20.1.3 Homomorphisms

Since homomorphisms are generally only allowed to be unitary, the specification of ring homomorphisms from $Q = \mathbf{Q}$ to a ring R is particularly simple: the image is completely determined by the image of 1, which we require to be 1 in R, so

```
hom< Q -> R | >
```

suffices.

Note that MAGMA allows the user to define maps with `hom` that are not proper homomorphisms; this is sometimes useful, as the example below shows.

Example H20E2

Suppose we wish to coerce rational numbers with denominator not divisible by 11 into the ring $\mathbf{Z}/11\mathbf{Z}$ in the obvious way by sending r/s to rs^{-1} mod 11. The coercion rules do not allow you to do so using `!`, but a simple 'homomorphism' will work.

```
> Z11 := Integers(11);
> Q := RationalField();
> h := hom< Q -> Z11 | >;
> h(1/2);
6
```

20.2 Creation Functions

20.2.1 Creation of Structures

The rational field **Q** is automatically created when MAGMA is started up. Nevertheless, it may be necessary to formally create the rational field, for instance if it is to be used as the coefficient ring for a polynomial ring. There is a unique rational field structure in MAGMA, that is, multiple calls to the creation function `RationalField()` will return the same object (and not an isomorphic copy), so no memory will be wasted.

`Rationals()`

`RationalField()`

Create the field **Q** of rational numbers.

`MaximalOrder(Q)`

`IntegerRing(Q)`

`IntegerRing()`

`Integers()`

`RingOfIntegers(Q)`

Create the field **Z** of rational integers.

`FieldOfFractions(Z)`

The function `FieldOfFractions` returns the field **Q** when R is either the ring **Z** of rational integers, or the field **Q** itself.

`Completion(Q, P)`

`Precision` RNGINTELT *Default :* ∞

Computes the completion of `Q` at the integral prime ideal `P` together with the injection into the completion.

The parameter `Precision` may be used to specify a particular precision.

20.2.2 Creation of Elements

Unlike elements of other structures, rational numbers and integers can be created as literals without the need to define the parent field **Q** or the parent ring **Z** first, since these structures are loaded whenever MAGMA is started up.

`a / b`

Given integers a and $b \neq 0$, form the rational number a/b (in reduced form). Of course a and be are allowed to be given as expressions defining integers.

`Q ! [a]`

The inverse function to `Eltseq`, returns `Q!a`.

```
Q ! [a, b]
```
```
elt< Q | a, b >
```

Given the rational field **Q**, and integers a, b (with $b \neq 0$), construct the rational number a/b, in reduced form.

```
Q ! a
```

Given the rational field **Q**, and an integer a, create the rational number $a = a/1$ in **Q**. Also, any element from a quadratic, cyclotomic or number field (or an order of such) that is rational can be coerced into the rational field this way.

```
One(Q)
```
```
Identity(Q)
```
```
Zero(Q)
```
```
Representative(Q)
```

These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively, in the rational field **Q**.

```
RootOfUnity(n, Q)
```

This function returns, in general, for a positive integer n and a cyclotomic field Q a primitive n-th root of unity in Q; if Q is the rational field, n must be 1 or 2, and the result will be 1 or -1 in Q accordingly.

```
Random(Q, m)
```

This function returns a random rational number with random numerator in $[-u..u]$ and random denominator in $[1..u]$, where u is the absolute value of m.

20.3 Structure Operations

20.3.1 Related Structures

```
Category(Q)
```
```
Parent(Q)
```
```
PrimeField(Q)
```
```
IntegralBasis(Q)
```

An integral basis for Q as a number field as a sequence of elements of Q (giving the sequence containing 1 for the rational field).

```
MinimalField(q)
```

Return the least cyclotomic field containing the cyclotomic field element q; if q is rational this returns the rational field.

`MinimalField(S)`

Returns the minimal cyclotomic field containing the cyclotomic field elements in the enumerated set S; this will return the rational field if all elements of S are rational numbers.

`BaseField(Q)`

In analogy to the number fields, returns the coefficient field of Q which will be $\mathbf{Q}$.

`Basis(Q)`

`AbsoluteBasis(Q)`

A basis for Q as a $\mathbf{Q}$-vector space, i.e. [1].

`UnitGroup(Q)`

The unit group of the maximal order of $\mathbf{Q}$ (i.e. of $\mathbf{Z}$).

`ClassGroup(Q)`

The class group of the ring of integers $\mathbf{Z}$ of $\mathbf{Q}$ (which is trivial).

`AutomorphismGroup(Q)`

`AutomorphismGroup(Q, Q)`

The group of $\mathbf{Q}$ automorphisms of $\mathbf{Q}$, ie. a trivial finitely presented group, the parent structure for $\mathbf{Q}$-automorphisms and a map from the group to actual field automorphisms. In this case, of course the only $\mathbf{Q}$-automorphism will be the identity.

`Algebra(Q, Q)`

The field of the rational number form canonically an algebra. This function returns an associative $\mathbf{Q}$-algebra isomorphic to $\mathbf{Q}$ and the map from the algebra to $\mathbf{Q}$.

`VectorSpace(Q, Q)`

The field of the rational number form canonically a vector space. This function returns a $\mathbf{Q}$-vector space isomorphic to $\mathbf{Q}$ and the map from the vector space to $\mathbf{Q}$.

`Decomposition(Q, p)`

For a prime p or for the "infinite prime" `Infinity()` compute the decomposition in $\mathbf{Q}$ as a number field. This returns a list of length one containing a 2-tuple describing the splitting behaviour: the first component contains p and the second it's ramification degree, ie. 1.

20.3.2 Numerical Invariants

The functions below are defined for the rational field **Q** mainly because it often arises as a degenerate case of quadratic or cyclotomic field constructions. See the corresponding Chapters 35 and 36 for more.

`Characteristic(Q)`

`Conductor(Q)`

The smallest positive integer n such that Q is contained in the cyclotomic field $\mathbf{Q}(\zeta_n)$. For the rational field this is 1.

`Degree(Q)`
`AbsoluteDegree(Q)`

The degree of Q as a number field (which is 1 for the rational field).

`Discriminant(Q)`
`AbsoluteDiscriminant(Q)`

The field discriminant of Q (which is 1 for the rational field).

`DefiningPolynomial(Q)`

An irreducible polynomial over **Q** a root of which generates Q as a number field (for the rational field this returns the linear polynomial $x - 1$).

`Signature(Q)`

The signature (number of real embeddings and pairs of complex embeddings) of **Q**.

20.3.3 Ring Predicates and Booleans

`IsCommutative(Q)` `IsUnitary(Q)`
`IsFinite(Q)` `IsOrdered(Q)`
`IsField(Q)` `IsEuclideanDomain(Q)`
`IsPID(Q)` `IsUFD(Q)`
`IsDivisionRing(Q)` `IsEuclideanRing(Q)`
`IsPrincipalIdealRing(Q)` `IsDomain(Q)`
`Q eq R` `Q ne R`

20.4 Element Operations

A variety of different types of operations are provided for rational elements including arithmetic operations, comparison and predicates and converting to a sequence.

20.4.1 Parent and Category

`Parent(r)` `Category(r)`

20.4.2 Arithmetic Operators

`+ a` `- a`

`a + b` `a - b` `a * b` `a ^ k` `a / b`

`a +:= b` `a -:= b` `a *:= b` `a /:= b` `a ^:= k`

20.4.3 Numerator and Denominator

`Numerator(q)`

The (integer) numerator of the rational number q in reduced form.

`Denominator(q)`

The (integer) denominator of the rational number q in reduced form. This will always be a positive integer.

Example H20E3

Rational numbers are always immediately put in reduced form, that is, the greatest common divisor of numerator and denominator is taken out, and the denominator will be positive.

```
> Numerator(10/-4);
-5
> Denominator(10/-4);
2
```

20.4.4 Equality and Membership

`a eq b` `a ne b`

`a in R` `a notin R`

20.4.5 Predicates on Ring Elements

`IsIntegral(q)`

Returns `true` if the rational number q is an element of the ring of integers, `false` otherwise.

`IsZero(a)` `IsOne(a)` `IsMinusOne(a)`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsUnit(a)` `IsZeroDivisor(a)` `IsRegular(a)`

`IsIrreducible(a)` `IsPrime(a)`

20.4.6 Comparison

`a gt b` `a ge b` `a lt b` `a le b`

`Maximum(a, b)` `Maximum(Q)`

`Minimum(a, b)` `Minimum(Q)`

20.4.7 Conjugates, Norm and Trace

`ComplexConjugate(q)`

The complex conjugate of q, which will be the rational number q itself.

`Conjugate(q)`

The conjugate of q, which will be the rational number q itself.

`Norm(q)`

`Norm(q)`

The norm (in $\mathbf{Q}$) of q, which will be the rational number q itself.

`Trace(q)`

The trace (in $\mathbf{Q}$) of q, which will be the rational number q itself.

`MinimalPolynomial(q)`

Returns the minimal polynomial of the rational number q, which is the monic linear polynomial with constant coefficient q in a univariate polynomial ring R over the rational field. (If R has not been created before with a name for its indeterminate, `$.1-q` will be returned.)

20.4.8 Absolute Value and Sign

`AbsoluteValue(q)`

`Abs(q)`

The absolute value $|q|$ of a rational number q.

`Sign(q)`

Returns the sign of the rational number q, which is one of the integers -1, 0, 1, corresponding to the cases $q < 0$, $q = 0$, and $q > 0$.

`Height(q)`

The height of $q = r/s$. For r and s coprime, the height is defined as the maximum of the absolute value of r and s.

20.4.9 Rounding and Truncating

`Ceiling(q)`

The ceiling of the rational number q, that is, the least integer greater than or equal to q.

`Floor(q)`

The floor of the rational number q, that is, the largest integer less than or equal to q.

`Round(q)`

This function returns the integer value of the rational number q rounded to the nearest integer. In the case of a tie, rounding is done away from zero (that is, $i + \frac{1}{2}$ is rounded to $i + 1$, for non-negative integers i and $i - \frac{1}{2}$ is rounded to $i - 1$, for non-positive integers i).

`Truncate(q)`

This function returns the integer truncation of the rational number q, that is the integral part of q. Thus the effect is that of rounding towards 0.

`Qround(q, M)`

`ContFrac` BoolElt *Default :* `false`

Finds an rational approximation d of q such that the denominator of d is bounded by M. If `ContFrac` is given then an optimal approximation is computed using the continued fraction process. By default d is obtained by some rounding procedure which is faster but gives worse results.

20.4.10 Rational Reconstruction

Under certain circumstances it is useful to have a partial inverse of the function $\psi_m : \mathbf{Q} \to \mathbf{Z}/m\mathbf{Z}$ of taking residues modulo m (where the obvious value of ψ_m is only defined for rational numbers with denominator in smallest terms coprime to m); the partial inverse of the function is sometimes referred to as 'rational reconstruction'. For $s \in \mathbf{Z}/m\mathbf{Z}$ the value of $\psi^{-1}(s)$ is the rational number r for which $\psi_m(r) = s$ and, in addition, the absolute values of both the numerator and denominator of r are at most $\sqrt{m/2}$; such r does not always exist, but if r exists it is unique.

`RationalReconstruction(s)`

Given an element s of a ring S of m elements, return a Boolean flag indicating whether or not a rational number r exists such that for the representation $r = n/d$ in minimal terms it holds that $n \cdot d^{-1} \equiv s \bmod m$, $|n| \leq \sqrt{m/2}$ and $0 < d \leq \sqrt{m/2}$. If the flag is true, the element r is also returned. The ring S is allowed to be a residue class ring `Integers(m)` or a finite field of prime cardinality $p = m$: `FiniteField(p)`.

In addition, s is allowed to be a matrix over a prime finite field, in which case the existence (and, if possible, value) of a rational reconstruction of the matrix is determined.

20.4.11 Valuation

`Valuation(x, p)`

`Valuation(x, I)`

The valuation v of the rational number x at the prime p (the prime ideal I). This is the difference of the valuations of the numerator and denominator of x. The optional second return value is the rational u such that $x = p^v u$.

20.4.12 Sequence Conversions

`ElementToSequence(a)`

`Eltseq(a)`

The sequence $[a]$ for compatibility with the other field types.

21 FINITE FIELDS

Chapter 21
FINITE FIELDS

21.1 Introduction

MAGMA provides a powerful environment for computing with lattices of finite fields. Complete freedom in the manner in which fields are constructed is allowed, while assuring compatibility. Finite fields of various kinds are supported, with optimized representations for each kind. For a detailed description of how finite fields are presented in MAGMA, see [BCS97].

21.1.1 Representation of Finite Fields

In MAGMA, arithmetic in small non-prime finite fields is carried out using tables of Zech logarithms. While this ensures that finite field arithmetic is fast, its use is limited to finite fields of small cardinality.

Larger finite fields are internally represented as polynomial rings over a small finite field. It is possible for the user to specify his own irreducible polynomial (although internally an alternative representation may well be used).

Although two finite fields of the same cardinality are isomorphic, in practical applications it is often important to be guaranteed to work in a field defined by a specific polynomial. Moreover, in passing between fields and subfields, choices regarding the embeddings have to be made, so that these embeddings are *compatible* (so that 'diagrams commute'). The scheme implemented in MAGMA and described in [BCS97] ensures that this is so.

21.1.2 Conway Polynomials

To avoid ambiguities when talking about (small) finite fields, *Conway polynomials* have been defined and calculated by R. Parker. The Conway polynomial $C_{p,n}$ is the lexicographically first monic irreducible, primitive polynomial of degree n over $\mathbf{F}_p$ with the property that it is consistent with all $C_{p,m}$ for m dividing n. Consistency of $C_{p,n}$ and $C_{p,m}$ for m dividing n means that for a root α of $C_{p,n}$ it holds that $\beta = \alpha^{\frac{p^n-1}{p^m-1}}$ is a root of $C_{p,m}$. Lexicographically first is with respect to the system of representatives $-\frac{p-1}{2}, \ldots, -1, 0, 1, \ldots, \frac{p-1}{2}$ for the residue classes modulo p, ordered via $0 < -1 < 1 < -2 < \cdots \frac{p-1}{2}$ (and we only need to compare polynomials of the same degree).

To compute the Conway polynomial $C_{p,n}$ one needs to know all Conway polynomials $C_{p,m}$ for m dividing n, and as far as we know, no essentially better method is known than enumerating and testing the primitive polynomials of degree n in lexicographical order.

Conway polynomials are used in MAGMA by default for the construction of $\mathbf{F}_{p^n}$ using `FiniteField(p, n)` or its synonyms, whenever the Conway polynomial is available. However, it must be stressed that Conway polynomials are **only** used in MAGMA to provide

standard defining polynomials for $\mathbf{F}_{p^n}$ – the special properties of Conway polynomials are never used because they are totally irrelevant in the scheme implemented in MAGMA!

21.1.3 Ground Field and Relationships

Throughout this Chapter we will use the notions of *ground field* and *prime field* in the following way. The prime field of a finite field F is the unique field of cardinality p, the characteristic of F. Here we mean unique not just in the mathematical sense, but in the sense that all prime fields of the same cardinality are identical in MAGMA, and their elements are denoted $0, 1, \ldots, p-1$. The ground field of F is the field over which F is created as an extension. If F was not explicitly created as an extension of a finite field E by using `ext`, its ground field will be the prime field. Printing in MAGMA always takes place with respect to the ground field, that is, elements of F are expressed as polynomials in the generator `F.1` of the field with coefficients in the ground field; there is one exception to this rule: if the field F is small enough to be represented by means of Zech logarithms, the printing of elements is in the form of powers of the primitive element (see the option on `AssertAttribute` below for ways of changing that).

It should be kept in mind that finite fields may be related mathematically without MAGMA being aware of the relation between them. This happens for example when two fields of dividing degrees are created as *extensions* of one field; although an isomorphic image of the smaller field will be contained in the larger, MAGMA will not establish this relation (unless the user explicitly asks for it, using the `Embed` function). However, all `subfields` of one common overfield in MAGMA will have their inclusion relations set up automatically.

21.2 Creation Functions

Since V2.13, a database of low-term irreducible polynomials over $\mathbf{F}_2$ is available for all degrees up to 90000 (see the function `IrreducibleLowTermGF2Polynomial` below). Thus one can create the finite field $\mathbf{F}_{2^k}$ for k within this range and compute within the field without any delay in the creation. Advantage is also taken of the special form of the defining polynomial.

Previous to V2.11, sparse trinomial/pentanomial irreducible polynomials (see the function `IrreducibleSparseGF2Polynomial`) were used by default for constructing $GF(2^k)$ when k is beyond the Conway range. To enable compatibility with older versions, one may select these sparse polynomials with the parameter `Sparse` in the creation functions.

21.2.1 Creation of Structures

`FiniteField(q)`

`GaloisField(q)`

`GF(q)`

`Optimize` BOOLELT *Default :* `true`

Sparse BoolElt *Default :* `false`

Given $q = p^n$, where p is a prime, create the finite field $\mathbf{F}_q$. If p is very big, it is advised to use the form `FiniteField(p, n)` described below instead, because Magma will first attempt to factor q completely.

The primitive polynomial used to construct $\mathbf{F}_q$ when $n > 1$ will be a Conway polynomial, if it is available. If the parameter `Optimize` is `false`, then no optimized representation (i.e., by using Zech logarithm tables or internal multi-step extensions) will be constructed for the new field which means that the time to create the field will be trivial but arithmetic operations in the field may be slower – this is useful if say one wishes to just compute a few trivial operations on a few elements of the field alone.

If $q = 2^k$ and k is beyond the Conway range, then a low-term irreducible is used (see `IrreducibleLowTermGF2Polynomial` below). Setting the parameter `Sparse` to `true` will cause a sparse polynomial to be used instead if possible (see `IrreducibleSparseGF2Polynomial` below).

`FiniteField(p, n)`

`GaloisField(p, n)`

`GF(p, n)`

Check BoolElt *Default :* `true`

Optimize BoolElt *Default :* `true`

Sparse BoolElt *Default :* `false`

Given a prime p and an exponent $n \geq 1$, create the finite field $\mathbf{F}_{p^n}$. The primitive polynomial used to construct $\mathbf{F}_q$ when $n > 1$ will be a Conway polynomial, if it is available.

By default p is checked to be a strong pseudoprime for 20 random bases b with $1 < b < p$; if the parameter `Check` is `false`, then no check is done on p at all (this is useful when p is very large and one does not wish to perform an expensive primality test on p).

The parameters are as above.

`ext< F | n >`

Optimize BoolElt *Default :* `true`

Sparse BoolElt *Default :* `false`

Given a finite field F and a positive integer n, create an extension G of degree n of F, as well as the embedding map $\phi : F \to G$. The parameter `Optimize` has the same behaviour as that for the `FiniteField` function. If F is a default field, then G will also be a default field (so its ground field will be the prime field). Otherwise, the ground field of G will be F.

The parameters are as above.

`ext< F | P >`

`Optimize` BoolElt *Default :* `true`

Given a finite field F and a polynomial P of degree n over F, create an extension $G = F[\alpha]$ of degree n of F, as well as the natural embedding map $\phi : F \to G$; the polynomial P must be irreducible over F, and α is one of its roots. Thus the defining polynomial of G over F will be P. The parameter `Optimize` has the same behaviour as that for the `FiniteField` function. The ground field of G will be F.

The parameter is as above.

`ExtensionField< F, x | P >`

Given a finite field F, a literal identifier x, and a polynomial P of degree n over F presented as a (polynomial) expression in x, create an extension $G = F[x]$ of degree n of F, as well as the natural embedding map $\phi : F \to G$; the polynomial P must be irreducible over F, and x is one of its roots. Thus the defining polynomial of G over F will be P. The parameter `Optimize` has the same behaviour as in the `FiniteField` function.

`RandomExtension(F, n)`

Given a finite field F and a degree n, return the extension of F by a random degree-n irreducible polynomial over F.

`SplittingField(P)`

Given a univariate polynomial P over a finite field F, create the minimal splitting field of P, that is, the smallest-degree extension field G of F such that P factors completely into linear factors over G.

`SplittingField(S)`

Given a set S of univariate polynomials each over a finite field F, create the minimal splitting field of S, that is, the smallest-degree extension field G of F such that for every polynomial P of S, P factors completely into linear factors over G.

`sub< F | d >`

`Optimize` BoolElt *Default :* `true`

`Sparse` BoolElt *Default :* `false`

Given a finite field F of cardinality p^n and a positive divisor d of n, create a subfield E of F of degree d, as well as the embedding map $\phi : E \to F$.

The parameters are as above.

`sub< F | f >`

`Optimize`	BoolElt	*Default :* `true`
`Sparse`	BoolElt	*Default :* `false`

Given a finite field F and an element f of F, create the subfield E of F generated by f, together with the embedding map $\phi : E \to F$. The map and field are constructed so that $\phi(w) = f$, where w is the generator of E (that is, `E.1`).

The parameters are as above.

`GroundField(F)`

`BaseField(F)`

Given a finite field F, return its ground field. If F was constructed as an extension of the field E, this function returns E; if F was not explicitly constructed as an extension then the prime field is returned.

`PrimeField(F)`

The subfield of F of prime cardinality.

`IsPrimeField(F)`

Returns whether field F is a prime field.

`F meet G`

Given finite fields F and G of the same characteristic p, return the finite field that forms the intersection $F \cap G$.

`CommonOverfield(K, L)`

Given finite fields K and L, both of characteristic p, return the smallest field which contains both of them.

Example H21E1

To define the field of 7 elements, use

```
> F7 := FiniteField(7);
```

We can define the field of 7^4 elements in several different ways. We can use the Conway polynomial:

```
> F<z> := FiniteField(7^4);
> F;
Finite field of size 7^4
```

We can define it as an extension of the field of 7 elements, using the internal polynomial:

```
> F<z> := ext< F7 | 4 >;
> F;
Finite field of size 7^4
```

We can supply our own polynomial, say $x^4 + 4x^3 + 2x + 3$:

```
> P<x> := PolynomialRing(F7);
```

```
> p := x^4+4*x^3+2*x+3;
> F<z> := ext< F7 | p >;
> F;
Finite field of size 7^4
```

We can define it as an extension of the field of 7^2 elements:

```
> F49<w> := ext< F7 | 2 >;
> F<z> := ext< F49 | 2 >;
> F;
Finite field of size 7^4
```

21.2.2 Creating Relations

`Embed(E, F)`

Given finite fields E and F of cardinality p^d and p^n, such that d divides n, assert the embedding relation between E and F. That is, an isomorphism between E and the subfield of F of cardinality p^d is chosen and set up, and can be used from then on to move between the fields E and F. See [BCS97] for details as to how this is done. If both E and F have been defined with Conway polynomials then the isomorphism will be such that the generator β of F is mapped to $\alpha^{\frac{p^n-1}{p^d-1}}$, where α is the generator of F.

`Embed(E, F, x)`

Given finite fields E and F of cardinality p^d and p^n such that d divides n, as well as an element $x \in F$, assert the embedding relation between E and F mapping the generator of E to x. The element x must be a root of the polynomial defining E over the prime field. Thus an isomorphism between E and the subfield of F of cardinality p^d is set up, and can be used from then on to move between the fields E and F.

21.2.3 Special Options

For finite fields for which the complete table of Zech logarithms is stored (and which must therefore be small), printing of elements can be done in two ways: either as powers of the primitive element or as polynomials in the generating element.

Note that power printing is not available in all cases where the logarithm table is stored however (the defining polynomial may not be primitive); for convenience element of a prime field are always printed as integers, and therefore power printing on prime fields will not work. Also, if a field is created with a generator that is not primitive, then power printing will be impossible.

`AssertAttribute(FldFin, "PowerPrinting", l)`

This attribute is used to change the *default* printing for all (small) finite fields created after the `AssertAttribute` command is executed. If l is `true` all elements of finite fields small enough for the Zech logarithms to be stored will be printed by default as a power of the primitive element – see `PrimitiveElement`). If l is `false` every finite field element is printed by default as a polynomial in the generator `F.1` of degree less than n over the ground field. The default can be overruled for a particular finite field by use of the `AssertAttribute` option listed below. The value of this attribute is obtained by use of `HasAttribute(FldFin, "PowerPrinting")`.

`SetPowerPrinting(F, l)`

`AssertAttribute(F, "PowerPrinting", l)`

Given a finite field F, the Boolean value l can be used to control the printing of elements of F, provided that F is small enough for the table of Zech logarithms to be stored. If l is `true` all elements will be printed as a power of the primitive element – see `PrimitiveElement`). If l is `false` (which is the only possibility for big fields), every element of F is printed as a polynomial in the generator `F.1` of degree less than n over the ground field of F, where n is the degree of F over its ground field. The function `HasAttribute(F, "PowerPrinting")` may be used to obtain the current value of this flag.

`HasAttribute(FldFin, "PowerPrinting", l)`

This function is used to find the current default printing style for all (small) finite fields. It returns `true` (since this attribute is always defined for FldFin), and also returns the current value of the attribute. The procedure `AssertAttribute(FldFin, "PowerPrinting", l)` may be used to control the value of this flag.

`HasAttribute(F, "PowerPrinting")`

Given a finite field F that is small enough for the table of Zech logarithms to be stored, returns `true` if the attribute `"PowerPrinting"` is defined, else returns `false`. If the attribute is defined, the function also returns the value of the attribute. The procedure `AssertAttribute(F, "PowerPrinting", l)` may be used to control the value of this flag.

`AssignNames(~F, [f])`

Procedure to change the name of the generating element in the finite field F to the contents of the string f. When F is created, the name will be `F.1`.

This procedure only changes the name used in printing the elements of F. It does *not* assign to an identifier called f the value of the generator in F; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies F, it is necessary to have a reference `~F` to F in the call to this function.

```
Name(F, 1)
```

Given a finite field F, return the element which has the name attached to it, that is, return the element `F.1` of F.

21.2.4 Homomorphisms

```
hom< F -> G | x >
```

Given a finite field F, create a homomorphism with F as its domain and G as its codomain. If F is a prime field, then the right hand side in the constructor must be empty; in this case the ring homomorphism is completely determined by the rule that the map must be unitary, that is, 1 of F is mapped to 1 of G. If F is not of prime cardinality, then the homomorphism must be specified by supplying one element x in the codomain, which serves as the image of the generator of the field F over its prime field. Note that it is the responsibility of the user that the map defines a homomorphism.

21.2.5 Creation of Elements

```
F . 1
```

The generator for F as an algebra over its ground field. Thus, if F was defined by the polynomial $P = P(X)$ over E, so $F \cong E[X]/P(X)$, then `F.1` is the image of X in F.

If F is a prime field, then $1 = 1_F$ will be returned.

```
elt< F | a >
F ! a
```

Given a finite field F create the element specified by a; here a is allowed to be an element coercible into F, which means that a may be

(i) an element of F;

(ii) an element of a subfield of F;

(iii) an element of an overfield of F that lies in F;

(iv) an integer, to be identified with a modulo the characteristic of F;

(v) a sequence of elements of the ground field E of F, of length equal to the degree of F over E. In this case the element $a_0 + a_1 w + \cdots + a_{n-1} w^{n-1}$ is created, where $a = [a_0, \ldots a_{n-1}]$ and w is the generator `F.1` of F over E.

`elt< F | a0, ..., an−1 >`

Given a finite field F with generator w of degree n over the ground field E, create the element $a_0 + a_1 w + \cdots + a_{n-1} w^{n-1} \in F$, where $a_i \in E$ $(0 \leq i \leq n-1)$. If the a_i are in some subfield of E or the a_i are integers, they will be coerced into the ground field.

`One(F)` `Identity(F)`

`Zero(F)` `Representative(F)`

These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively, in any finite field.

`Random(F)`

Create a 'random' element of finite field F.

21.2.6 Special Elements

`F . 1`

`Generator(F)`

Given a finite field F, this function returns the element f of F that generates F over its ground field E, so $F = E[f]$. This is the same as the element `F.1`.

`Generator(F, E)`

Given a finite field F and a subfield E of F, this function returns an element f of F that generates F over E, so $F = E[f]$. Note that this element may be different from the element `F.1`, but if `F.1` works it will be returned.

`PrimitiveElement(F)`

Given a finite field F, this function returns a primitive element for F, that is, a generator for the multiplicative group F^* of F. Note that this may be an element different from the generator `F.1` for the field as an algebra. This function will return the same element upon different calls with the same field; the primitive element that is returned is the one that is used as basis for the `Log` function.

`SetPrimitiveElement(F, x)`

(Procedure.) Given a finite field F and a *primitive* element x of F, set the internal primitive element p of F to be x. If the internal primitive element p of F has already been computed or set, x must equal it. The function `Log` (given one argument) returns the logarithm of its argument with respect to the base p; this function thus allows one to specify which base should be used. (One can also use `Log(x, b)` for a given base but setting the primitive element and using `Log(x)` will be faster if many logarithms are to be computed.)

`NormalElement(F)`

Given a finite field $F = \mathbf{F}_{p^n}$, this function returns a normal element for F over the ground field G, that is, an element $\alpha \in F$ such that $\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}$ forms a basis for F over G, where q is the cardinality of G, and n the degree for F over G. Two calls to this function with the same field may result in different normal elements.

`NormalElement(F, E)`

Given a finite field $F = \mathbf{F}_{q^n}$ and a subfield $E = \mathbf{F}_q$, this function returns a normal element for F over E, that is, an element $\alpha \in F$ such that $\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}$ forms a basis for F over E.

21.2.7 Sequence Conversions

`SequenceToElement(s, F)`

`Seqelt(s, F)`

Given a sequence $s = [s_0, \ldots, s_{n-1}]$ of elements of a finite field E, of length equal to the degree of the field F over its subfield E, construct the element $s = s_0 + s_1 w + \cdots + s_{n-1} w^{n-1}$ of F, where w is the generator `F.1` of F over E.

`ElementToSequence(a)`

`Eltseq(a)`

Given an element a of the finite field F, return the sequence of coefficients $[a_0, \ldots, a_{n-1}]$ in the ground field E of F such that $a = a_0 + a_1 w + \cdots + a_{n-1} w^{n-1}$, with w the generator of F over E, and n the degree of F over E.

`ElementToSequence(a, E)`

`Eltseq(a, E)`

Given an element a of the finite field F, return the sequence of coefficients $[a_0, \ldots, a_{n-1}]$ in the subfield E of F such that $a = a_0 + a_1 w + \cdots + a_{n-1} w^{n-1}$, with w the generator of F over E, and n the degree of F over E.

21.3 Structure Operations

21.3.1 Related Structures

`Category(F)` `Parent(F)` `Centre(F)`
`PrimeRing(F)` `PrimeField(F)`
`FieldOfFractions(F)`

`AdditiveGroup(F)`

Given $F = \mathbf{F}_q$, create the finite additive abelian group A of order $q = p^r$ that is the direct sum of r copies of the cyclic group of order p, together with the corresponding isomorphism from the group A to the field F.

`MultiplicativeGroup(F)`
`UnitGroup(F)`

Given $F = \mathbf{F}_q$, create the multiplicative group of R as an abelian group. This returns the (additive) cyclic group A of order $q-1$, together with a map from A to $F \setminus 0$, sending 1 to a primitive element of F.

`Set(F)`

Create the enumerated set consisting of the elements of finite field F.

`VectorSpace(F, E)`

Given a finite field F that is an extension of degree n of E, define the natural isomorphism between F and the n-dimensional vector space E^n. The function returns two values:

(a) A vector space $V \cong E^n$;

(b) The isomorphism $\phi : F \to V$.

The basis of V is chosen to correspond with the power basis α^0, α^1, ..., α^{n-1} of F, where α is the generator returned by `Generator(F, E)`, so that $V = E \cdot 1 \times E \cdot \alpha \times \cdots \times E \cdot \alpha^{n-1}$ and $\phi : \alpha^i \to e_{i+1}$, (for $i = 0, \ldots, n-1$), where e_i is the basis vector of V having all components zero, except the i-th, which is one.

`VectorSpace(F, E, B)`

Given a finite field F that is an extension of degree n of E, define the isomorphism between F and the n-dimensional vector space E^n defined by the basis B for F over E. The function returns two values:

(a) A vector space $V \cong E^n$;

(b) The isomorphism $\phi : F \to V$.

The basis of V is chosen to correspond with the basis $B = \beta_1, \beta_2, \ldots, \beta_n$ of F over E, as specified by the user, so that $V = E \cdot \beta_1 \times E \cdot \beta_2 \times \cdots \times E \cdot \beta_n$. $\phi : \beta_i \to e_i$, (for $i = 1, \ldots, n$), where e_i is the basis vector of V having all components zero, except the i-th, which is one.

`MatrixAlgebra(F, E)`

Let F be a finite field that is an extension of degree n of E. The function returns two values:

(a) A matrix algebra A of degree n, such that A is isomorphic to F;

(b) An isomorphism $\phi : F \to A$.

The matrix algebra A will be the subalgebra of the full algebra of $n \times n$ matrices over E generated by the companion matrix C of the defining polynomial of F over E. The generator `Generator(F, E)` of F over E is thus mapped to C.

`MatrixAlgebra(A, E)`

Let F be a finite field. Let A be a matrix algebra over F, and E be a subfield of F. The function returns two values:

(a) A matrix algebra N over E isomorphic to A, obtained from A by expanding each component of an element of A into the block matrix associated with it;

(b) An E-isomorphism $\phi : A \to N$.

N is A considered as an E-matrix algebra.

Example H21E2

Given the field F of 7^4 elements defined as an extension of the field $F49$ of 7^2 elements as above, we can construct two vector spaces, one of dimension 2, and the other of dimension 4:

```
> F7 := FiniteField(7);
> F49<w> := ext< F7 | 2 >;
> F<z> := ext< F49 | 2 >;
> v2, i2 := VectorSpace(F, F49);
> v2;
Full Vector space of degree 2 over GF(7^2)
> i2(z^12);
(   w w^28)
> v4, i4 := VectorSpace(F, PrimeField(F));
> v4;
Full Vector space of degree 4 over GF(7)
> i4(z^12);
(5 3 6 4)
```

`GaloisGroup(K, k)`

Compute the Galois group (which is of course cyclic) of K/k as a permutation group. The group is returned as well as the roots of the defining polynomial of K/k in a compatible ordering.

`AutomorphismGroup(K, k)`

Computes the (cyclic) group of k-automorphisms of K. The group is returned as well as a sequence of all automorphisms and a map sending an element of the abstract automorphism group to an explicit automorphism.

21.3.2 Numerical Invariants

`Characteristic(F)` `# F`

`Degree(F)`

The absolute degree of F, that is, the degree over its prime subfield.

`Degree(F, E)`

Given a finite field F that has been constructed as an extension of a field E, return the degree of F over E.

21.3.3 Defining Polynomial

`DefiningPolynomial(F)`

Given a finite field F that has been constructed as an extension of a field E, return the polynomial with coefficients in E that was used to define F as an extension of E. This is the minimum polynomial of `F.1`.

`DefiningPolynomial(F, E)`

Given a finite field F and a subfield E, return the polynomial with coefficients in E used to define F as an extension of E. This is the same as the minimum polynomial of the generator `Generator(F, E)` over E.

21.3.4 Ring Predicates and Booleans

`IsConway(F)`

Given a finite field F, this function returns `true` iff F is defined over its prime field using a Conway polynomial.

`IsDefault(F)`

Given a finite field F, this function returns `true` iff F is a default field.

`IsCommutative(F)` `IsUnitary(F)`

`IsFinite(F)` `IsOrdered(F)`

`IsField(F)` `IsEuclideanDomain(F)`

`IsPID(F)` `IsUFD(F)`

`IsDivisionRing(F)` `IsEuclideanRing(F)`
`IsPrincipalIdealRing(F)` `IsDomain(F)`
`F eq G` `F ne G`

21.3.5 Roots

`Roots(f)`

Given a polynomial f over a finite field F, this function finds all roots of f in F, and returns a sorted sequence of tuples (pairs), each consisting of a root of f in F and its multiplicity.

`RootsInSplittingField(f)`

Given a univariate polynomial f over a finite field K, compute the minimal splitting field S of f as an extension field of K, and return the roots of f in S, together with S. Using this function will be faster than computing the roots of f anew over the splitting field.

`FactorizationOverSplittingField(f)`

Given a univariate polynomial f over a finite field K, compute the minimal splitting field S of f as an extension field of K, and return the factorization (into linears) of f over S, together with S. Using this function will be faster than factorizing f anew over the splitting field.

`RootOfUnity(n, K)`

Return a primitive n-th root of unity in the smallest possible extension field of K.

Example H21E3

We compute the roots of a certain degree-20 polynomial f in its minimal splitting field.

```
> K := GF(2);
> P<x> := PolynomialRing(GF(2));
> f := x^20 + x^11 + 1;
> Factorization(f);
[
    <x^3 + x^2 + 1, 1>,
    <x^8 + x^7 + x^3 + x^2 + 1, 1>,
    <x^9 + x^7 + x^6 + x^4 + 1, 1>
]
> time r, S<w> := RootsInSplittingField(f);
Time: 0.040
```

We note that the splitting field S has degree 72 and there are 20 roots of f in S of course. We check that the evaluation of f at each root is zero.

```
> S;
```

```
Finite field of size 2^72
> DefiningPolynomial(S);
x^72 + x^48 + x^47 + x^44 + x^38 + x^35 + x^32 + x^31 + x^30 +
    x^29 + x^27 + x^25 + x^23 + x^22 + x^21 + x^18 + x^15 +
    x^12 + x^8 + x^4 + 1
> #r;
20
> r[1];
<w^68 + w^67 + w^64 + w^62 + w^60 + w^59 + w^56 + w^50 + w^49 +
    w^48 + w^47 + w^44 + w^43 + w^39 + w^37 + w^35 + w^33 + w^32
    + w^30 + w^29 + w^28 + w^25 + w^21 + w^19 + w^18 + w^16 +
    w^15 + w^14 + w^12 + w^10 + w^6 + w, 1>
> [IsZero(Evaluate(f, t[1])): t in r];
[ true, true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true ]
```

21.4 Element Operations

See also Section 17.5.

21.4.1 Arithmetic Operators

`+ a` `- a`

`a + b` `a - b` `a * b` `a / b`

`a ^ k`

`a +:= b` `a -:= b` `a *:= b`

21.4.2 Equality and Membership

`a eq b` `a ne b`

`a in F` `a notin F`

21.4.3 Parent and Category

`Parent(a)` `Category(a)`

21.4.4 Predicates on Ring Elements

`IsZero(a)` `IsOne(a)` `IsMinusOne(a)`
`IsNilpotent(a)` `IsIdempotent(a)`
`IsUnit(a)` `IsZeroDivisor(a)` `IsRegular(a)`
`IsIrreducible(a)` `IsPrime(a)`

`IsPrimitive(a)`

Returns `true` if and only if the element a of F is a primitive element for F (i.e., if and only if the multiplicative order of a is $\#F - 1$).

`IsPrimitive(f)`

Given a univariate polynomial $f \in F[x]$, over a finite field F, such that the degree of f is greater than or equal to 1, this function returns `true` if and only if f defines a primitive extension $G = F[x]/f$ of F (that is, x is primitive in G).

`IsNormal(a)`

Returns `true` if and only if the element a of F generates a normal basis for the field over the ground field, that is, if and only if $a, a^q, \ldots, a^{q^{n-1}}$ form a basis for F over the ground field $G = \mathbf{F}_q$.

`IsNormal(a, E)`

Returns `true` if and only if the element a of the finite field F with q^n elements generates a normal basis for F over its subfield E, that is, if and only if $a, a^q, \ldots, a^{q^{n-1}}$ form a basis for F over E for $q = \#E$.

`IsSquare(a)`

Given a finite field element $a \in F$, this function returns either `true` and an element $b \in F$ such that $b^2 = a$, or it returns `false` in the case that such an element does not exist.

21.4.5 Minimal and Characteristic Polynomial

`MinimalPolynomial(a)`

The minimal polynomial of the element a of the field F, relative to the ground field of F. This is the unique minimal-degree monic polynomial with coefficients in the ground field, having a as a root.

`MinimalPolynomial(a, E)`

The minimal polynomial of the element a of the field F, relative to the subfield E of F. This is the unique minimal-degree monic polynomial with coefficients in E, having a as a root.

`CharacteristicPolynomial(a)`

Given an element a of a finite field F, return the characteristic polynomial of a with respect to the ground field of F. (This polynomial is the characteristic polynomial of the companion matrix of a written as a polynomial over the ground field, and is a power of the minimal polynomial.)

`CharacteristicPolynomial(a, E)`

Given an element a of a finite field F, return the characteristic polynomial of a with respect to the subfield E of F. (This polynomial is the characteristic polynomial of the companion matrix of a written as a polynomial over E, and is a power of the minimal polynomial over E.)

21.4.6 Norm, Trace and Frobenius

`Norm(a)`

The norm of the element a from the field F to the ground field of F.

`Norm(a, E)`

The relative norm of the element a from the field F, with respect to the subfield E of F. The result is an element of E.

`AbsoluteNorm(a)`
`NormAbs(a)`

The absolute norm of the element a, that is, the norm to the prime subfield of the parent field F of a.

`Trace(a)`

The trace of the element a from the field F to the ground field of F.

`Trace(a, E)`

The relative trace of the element a from field F, with respect to the subfield E of F. The result is an element of E.

`AbsoluteTrace(a)`
`TraceAbs(a)`

The trace of the element a, that is, the trace to the prime subfield of the parent field F of a.

`Frobenius(a)`

The Frobenius image of a w.r.t. the ground field of K; i.e., $a^{\#G}$, where G is the ground field of the parent of a.

`Frobenius(a, r)`

The r-th Frobenius image of a w.r.t. the ground field of K; i.e., $a^{(\#G)^r}$, where G is the ground field of the parent of a.

`Frobenius(a, E)`

The Frobenius image of x w.r.t. E; i.e., $x^{\#E}$.

`Frobenius(a, E, r)`

The Frobenius image of x w.r.t. E; i.e., $x^{(\#E)^r}$.

`NormEquation(K, y)`

Given a finite field K and an element y of a subfield S of K, return whether an element $x \in K$ exists such that $\mathrm{Norm}(x, S) = y$, and, if so, such an element x (in K).

`Hilbert90(a, q)`

Given an element a of some finite field k and a power q of the characteristic of k, return a solution of the Hilbert 90 equation $x^q x^{-1} = a$. Note that the solution may be in an finite-degree extension of k.

`AdditiveHilbert90(a, q)`

Given an element a of some finite field k and a power q of the characteristic of k, return a solution of the additive Hilbert 90 equation $x^q - x = a$. Note that the solution may be in an finite-degree extension of k.

21.4.7 Order and Roots

`Order(a)`

The multiplicative order of the non-zero element a of the field F.

`FactoredOrder(a)`

The multiplicative order of the non-zero element a of the field F as a factorization sequence.

`SquareRoot(a)`

`Sqrt(a)`

The square root of the non-zero element a from the field F, i.e., an element y of F such that $y^2 = a$. An error results if a is not a square.

`Root(a, n)`

The n-th root of the non-zero element a from the field F, i.e., an element y of F such that $y^n = a$. An error results if no such root exists.

`IsPower(a, n)`

Given a finite field element $a \in F$, and an integer $n > 0$, this function returns either `true` and an element $b \in F$ such that $b^n = a$, or it returns `false` in the case that such an element does not exist.

`AllRoots(a, n)`

Given a finite field element $a \in F$, and an integer $n > 0$, return a sequence containing all of the n-th roots of a which lie in the same field F.

Example H21E4

Given the fields F and $F49$ defined above, we can use the following functions:

```
> F7 := FiniteField(7);
> F49<w> := ext< F7 | 2 >;
> F<z> := ext< F49 | 2 >;
> Root(z^73, 7);
z^1039
> Trace(z^73);
0
> Trace(z^73, F49);
w^44
> Norm(z^73);
3
> Norm(z^73, F49);
w^37
> Norm(w^37);
3
> MinimalPolynomial(z^73);
x^2 + w^20*x + w^43
> MinimalPolynomial(z^73, F7);
x^4 + 4*x^2 + 4*x + 3
```

We now demonstrate the `NormEquation` function.

```
> Norm(z);
3
> NormEquation(F, F7!3);
true z
> Norm(z^30, F49);
w^30
> Parent(z) eq F;
true
> NormEquation(F, w^30);
true z^30
```

21.5 Polynomials for Finite Fields

`IrreduciblePolynomial(F, n)`

Given a finite field F and a positive integer $n > 1$, return a polynomial of degree n that is irreducible over F. If a Conway polynomial or a sparse polynomial is available, then it is returned.

`RandomIrreduciblePolynomial(F, n)`

Given a finite field F and a positive integer $n > 1$, return a random irreducible polynomial of degree n that is irreducible over F. The polynomial will be dense in general (that is, a Conway or stored sparse polynomial is not used).

`IrreducibleLowTermGF2Polynomial(n)`

Given an integer n in the range $1 \leq n \leq 100000$, return the irreducible polynomial f of the form $x^n + g$ where the degree of g is minimal and g is the first such polynomial in lexicographical order.

This uses a database of low-term irreducible polynomials over $\mathbf{F}_2$, constructed by Allan Steel in 2004 (thanks are expressed to William Stein for providing machines for some of the computations).

`IrreducibleSparseGF2Polynomial(n)`

Given an integer n in the range $4 \leq n \leq 12800$, return the irreducible polynomial f of the form $x^n + g$ where g has 2 non-zero terms if possible and 4 non-zero terms if not; g is the first such polynomial in lexicographical order in either case.

This uses a database of sparse irreducible polynomials over $\mathbf{F}_2$ constructed by Allan Steel in 1998.

`PrimitivePolynomial(F, m)`

Given a finite field F and a positive integer $m > 1$, construct a polynomial f of degree m that is primitive over F. Thus, f is irreducible over F, and it has a primitive root of the degree m extension field of F as a root.

`AllIrreduciblePolynomials(F, m)`

Given a finite field F and a positive integer $m > 1$, construct the set of all monic polynomials of degree m that are irreducible over F.

`ConwayPolynomial(p, n)`

Given a prime p and an exponent $n \geq 1$, return the Conway polynomial of degree n over $\mathbf{F}_p$. The Conway polynomial is defined in the introduction. Note that this polynomial is read in from a table containing Conway polynomials for a limited range of p, n only.

`ExistsConwayPolynomial(p, n)`

Given a prime p and an exponent $n > 1$, return `true` and the Conway polynomial if it is known for the field $\mathbf{F}_p$, `false` otherwise.

21.6 Discrete Logarithms

Let K be a field of cardinality $q = p^k$, with p prime. MAGMA contains several advanced algorithms for computing discrete logarithms of elements of K. The two main kinds of algorithms used are as follows: (1) *Pohlig-Hellman* [PH78]: The running time is usually proportional to the square root of the largest prime l dividing $q - 1$; this is combined with the Shanks baby-step/giant-step algorithm (when l is very small) or the Pollard-ρ algorithm. (2) *Index-Calculus*: There is first a precomputation stage which computes and stores all the logarithms of a *factor base* (all the elements of the field corresponding to irreducible polynomials up to some bound) and then each subsequent individual logarithm is computed by expressing the given element in terms of the factor base.

The different kinds of finite fields in MAGMA are handled as follows (in this order):

(a) *Small Fields (any characteristic)*:

If the largest prime l dividing $q - 1$ is reasonably small (typically, less than 2^{36}), the Pohlig-Hellman algorithm is used (the characteristic p is irrelevant).

(b) *Large Prime* :

Suppose K is a prime field (so $q = p$). Then the *Gaussian integer sieve* [COS86, LO91a] is used if p has at least 4 bits but no more than 400 bits, $p - 1$ is not a square, and one of the following is a quadratic residue modulo p: -1, -2, -3, -7, or -11. If the Gaussian integer sieve cannot be used and if p is no more than 300-bits, then the *linear sieve* [COS86, LO91a] is used. The precomputation stage always takes place and typically requires a lot more time than for computing individual logarithms (and may also require a lot of memory for large fields). Thus, the first call to the function `Log` below may take much more time than for subsequent calls. Also, for large prime fields, in comparison to the Gaussian method the linear sieve requires much more time and memory than the Gaussian method for the precomputation stage, and therefore it is only used when the Gaussian integer algorithm cannot be used. See the example H27E3 in the chapter on sparse matrices for an explanation of the basic linear sieve algorithm and for more information on the sparse linear algebra techniques employed.

(c) *Small Characteristic, Non-prime* :

Since V2.19, if K is a finite field of characteristic p, where p is less than 2^{30}, then an implementation by Allan Steel of Coppersmith's index-calculus algorithm [Cop84, GM93, Tho01] is used. (Strictly speaking, Coppersmith's algorithm is for the case $p = 2$ only, but a straightforward generalization is used when $p > 2$.) A suite of external auxiliary tables boost the algorithm so that the precomputation stage computation to determine the logarithms of a factor base can be avoided for a large number of fields of very small characteristic. This means that logarithms of individual elements can be computed immediately if a relevant table is present for the specific field. By default, tables are included in the standard Magma distribution at least for all fields of characteristic 2, 3, 5 or 7 with cardinality up to 2^{200}. The user can optionally download a much larger suite of tables from the Magma optional downloads page `http://magma.maths.usyd.edu.au/magma/download/db/` (files `FldFinLog_2.tar.gz`, etc.; about 5GB total).

(d) *Large Characteristic, Non-prime* :

In all other cases, the Pohlig-Hellman algorithm is used.

`Log(x)`

The discrete logarithm of the non-zero element x from the field F, i.e., the unique integer k such that $x = w^k$ and $0 \leq k < (\#F - 1)$, where w is the primitive element of F (as returned by `PrimitiveElement`). Default parameters are automatically chosen if an index-calculus method is used (use `Sieve` below to set parameters). See also the procedure `SetPrimitiveElement`.

`Log(b, x)`

The discrete logarithm to the base b of the non-zero element x from the field F, i.e., the unique integer k such that $x = b^k$ and $0 \leq k < (\#F - 1)$. If b is not a primitive element, then in some unusual cases the algorithm may take much longer than normal.

`ZechLog(K, n)`

The Zech logarithm $Z(n)$ of the integer n for the field field F, which equals the logarithm to base w of $w^n + 1$, where w is the primitive element of F. If w^n is the minus one element of K, then -1 is returned.

`Sieve(K)`

`Lanczos` BoolElt *Default :* `false`

(Procedure.) Call the Gaussian integer sieve on the prime finite field K if possible; otherwise call the linear sieve on K (assuming K is not too small).

If the parameter `Lanczos` is set to `true`, then the *Lanczos* algorithm [LO91b, Sec. 3] will be used for the linear algebra phase. This is generally *very much slower* than the default method (often 10 to 50 times slower), but it will take considerably less memory, so may be preferable for extremely large fields. See also the function `ModularSolution` in the chapter on sparse matrices for more information.

`SetVerbose("FFLog", v)`

(Procedure.) Set the verbose printing level for the finite field logarithm algorithm to be v. Currently the legal values for v are 0, 1, and 2. If the level is 1, information is printed whenever the logarithm of an element is computed (unless the field is very small, in which case a lookup table is used). The value of 2 will print a very large amount of information.

Example H21E5

We demonstrate the Log function.

```
> F<z> := FiniteField(7^4);
> PrimitiveElement(F);
z;
> Log(z);
1
> Log(z^2);
2
> Log(z + 1);
419
> z^419 eq z + 1;
true
> b := z + 1;
> b;
z^419
> Log(b, b);
1
> Log(b, z);
779
> b^779 eq z;
true
```

We now do similar things for a larger field of characteristic 2, which will use Coppersmith's algorithm to compute the logarithms.

```
> F<z> := GF(2, 73);
> Factorization(#F-1);
[ <439, 1>, <2298041, 1>, <9361973132609, 1> ]
> PrimitiveElement(F);
z
> time Log(z + 1);
429570031703221890 8392
Time: 5.400
> z^429570031703221890 8392;
z + 1
> time Log(z + 1);
429570031703221890 8392
Time: 0.000
> time Log(z^2);
2
Time: 0.000
> time Log(z^2134914112412412);
2134914112412412
Time: 0.000
> b := z + 1;
> b;
z + 1
```

```
> time Log(b, b);
1
Time: 0.010
> time Log(b, z);
2260630912967574270198
Time: 0.000
> b^2260630912967574270198;
z
```

21.7 Permutation Polynomials

Let K be a finite field. A polynomial representing (by the evaluation map) a bijection of K into itself is known as a *permutation polynomial*. The Dickson polynomials of the first and second kind are permutation polynomials when certain conditions are satisfied.

`DicksonFirst(n, a)`

Given a positive integer n, this function constructs the Dickson polynomial of the first kind $D_n(x, a)$ of degree n, where $D_n(x, a)$ is defined by

$$D_n(x,a) = \sum_{i=0}^{\lfloor n/2 \rfloor} \frac{n}{n-i} \binom{n-i}{i} (-a)^i x^{n-2i}.$$

`DicksonSecond(n, a)`

Given a positive integer n, this function constructs the Dickson polynomial of the second kind $E_n(x, a)$ of degree n, where $E_n(x, a)$ is defined by

$$E_n(x,a) = \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n-i}{i} (-a)^i x^{n-2i}.$$

`IsProbablyPermutationPolynomial(p)`

`NumAttempts` RngIntElt *Default :* 100

Let p denote a polynomial defined over a finite field K. A probabilistic test is applied to determine whether the mapping on K defined by p is a bijection. The function returns `true` if the test succeeds for each of n attempts, otherwise `false`. By default, n is taken to be 100; a different value for n can be specified by use of the parameter `NumAttempts`.

Example H21E6

Let K be a finite field of cardinality q. By a theorem of Nöbauer, the Dickson polynomial of the first kind of degree n is a permutation polynomial for K if and only if $(n, q^2 - 1) = 1$. Consider $K = \mathbf{F}_{16}$.

```
> Factorization(16^2 - 1);
[ <3, 1>, <5, 1>, <17, 1> ]
```

Thus, $D_n(x, a)$ will be a permutation polynomial for K providing that n is coprime to 3, 5 and 17.

```
> K<w> := GF(16);
> R<x> := PolynomialRing(K);
> a := w^5;
> p1 := DicksonFirst(3, a);
> p1;
x^3 + w^5*x
> #{ Evaluate(p1, x) : x in K };
11
> IsProbablyPermutationPolynomial(p1);
false
```

So $D_3(x, a)$ is not a permutation polynomial. However, $D_4(x, a)$ is a permutation polynomial:

```
> p1 := DicksonFirst(4, a);
> p1;
x^7 + w^5*x^5 + x
> #{ Evaluate(p1, x) : x in K };
16
> IsProbablyPermutationPolynomial(p1);
true
```

21.8 Bibliography

[**BCS97**] Wieb Bosma, John Cannon, and Allan Steel. Lattices of Compatibly Embedded Finite Fields. *J. Symbolic Comp.*, 24(3):351–369, 1997.

[**Cop84**] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Trans. Inform. Theory*, IT–30(4):587–594, July 1984.

[**COS86**] D. Coppersmith, A. M. Odlyzko, and R. Schroeppel. Discrete logarithms in GF(p). *Algorithmica*, 1:1–15, 1986.

[**GM93**] D. M. Gordon and K. S. McCurley. Massively parallel computation of discrete logarithms. In Ernest F. Brickell, editor, *Advances in Cryptology—CRYPTO 1992*, volume 740 of *LNCS*, pages 312–323. Springer-Verlag, 1993. Proc. 12th Annual International Cryptology Conference, Santa Barbara, Ca, USA, August 16–20, 1992.

[**LO91a**] B. A. LaMacchia and A. M. Odlyzko. Computation of Discrete Logarithms in Prime Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of *INCS*, pages 616–618. Springer-Verlag, 1991.

[**LO91b**] B. A. LaMacchia and A. M. Odlyzko. Solving Large Sparse Linear Systems over Finite Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of *LNCS*, pages 109–133. Springer-Verlag, 1991.

[**PH78**] S. C. Pohlig and M. E. Hellman. An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance. *IEEE Trans. Inform. Theory*, 24:106–110, 1978.

[**Tho01**] Emmanuel Thomé. Computation of discrete logarithms in $\mathbf{F}_{2^{607}}$. In Colin Boyd and Ed Dawson, editors, *Advances in Cryptology—AsiaCrypt 2001*, volume 2248 of *LNCS*, pages 107–124. Springer-Verlag, 2001. Proc. 7th International Conference on the Theory and Applications of Cryptology and Information Security, Dec. 9–13, 2001, Gold Coast, Queensland, Australia.

22 NEARFIELDS

Chapter 22
NEARFIELDS

22.1 Introduction

In 1905, in the course of proving the independence of the field postulates, L. E. Dickson [Dic05a] (p. 203) introduced the first example of a nearfield. His example is a set of 9 elements with operations of addition and multiplication which satisfy all the axioms of a field except for the commutative law of multiplication and the right distributive law. Later that year Dickson [Dic05b] published a more extensive collection of examples: an infinite series obtained by twisting the multiplication of a Galois field and seven "irregular" examples.

The terminology 'nearfield' seems to have introduced by Zassenhaus in his 1935 paper [Zas35] where he showed that the only finite nearfields (endliche Fastkörper) are those due to Dickson.

The irregular nearfields are often referred to as Zassenhaus nearfields and the nearfields in the infinite series are called Dickson nearfields.

In the papers of Dickson and Zassenhaus the nearfields are left-distributive but for the purposes of the MAGMA implementation we consider only right-distributive nearfields.

Nearfields are important in group theory, geometry and a combination of these two fields. On the one hand, the finite sharply doubly transitive permutation groups are in one-to-one correspondence with the finite nearfields and on the other hand, nearfields coordinatise a class of translation planes [Hal59, L69] and they are the starting point for the construction of the Hughes planes [Dem71, Hug57]. Furthermore, every sharply transitive collineation group of projective space over a finite field is a quotient of the group of units of a nearfield [EK63] (see also, [Dem68, §1.4, n° 17]).

22.2 Nearfield Properties

A (right-distributive) *nearfield* is a set N containing elements 0 and 1 and with binary operations $+$ and $\circ$ such that

NF1: $(N, +)$ is an abelian group and 0 is its identity element. Let $N^\times$ denote the set of non-zero elements of N.

NF2: $(N^\times, \circ)$ is a group and 1 is its identity element.

NF3: $a \circ 0 = 0 \circ a = 0$ for all $a \in N$.

NF4: $(a + b) \circ c = a \circ c + b \circ c$ for all $a, b, c \in N$.

A subset S of a nearfield N is a *sub-nearfield* if $(S, +)$ and $(S \setminus \{0\}, \circ)$ are groups. The sub-nearfield *generated* by a subset X is the intersection of all sub-nearfields containing X. The *prime field* $\mathcal{P}(N)$ of N is the sub-nearfield generated by 1.

The inverse of $x \in N^\times$ is written $x^{[-1]}$. But where no confusion is possible we write multiplication of nearfield elements x and y as xy rather than $x \circ y$ and we write the inverse of x as x^{-1}. (In the MAGMA code we use "*" as the symbol for multiplication.)

If N is a finite nearfield, the prime field of N is a Galois field $\mathbf{F}_p$ for some prime p and p is the *characteristic* of N.

A nearfield of characteristic p is a vector space over its prime field and therefore its cardinality is p^n for some n. Every field is a nearfield.

If N is a nearfield, the *centre* of N is the set

$$\mathcal{Z}(N) = \{\, x \in N \mid xy = yx \text{ for all } y \in N \,\}$$

and the *kernel* of N is the subfield

$$\mathcal{K}(N) = \{\, x \in N \mid x(y+z) = xy + xz \text{ for all } y, z \in N \,\}.$$

It is clear that $\mathcal{Z}(N) \subseteq \mathcal{K}(N)$ but equality need not hold because, in general, $\mathcal{Z}(N)$ need not be closed under addition. Furthermore, the prime field $\mathcal{P}(N)$ need not be contained in $\mathcal{Z}(N)$. However, for the Dickson nearfields $\mathcal{Z}(N) = \mathcal{K}(N)$.

If N is a nearfield, then $\mathcal{Z}(N) = \bigcap\{\mathcal{K}(N)^x \mid x \in N,\ x \neq 0\}$.

22.2.1 Sharply Doubly Transitive Groups

A group G acting on a set Ω is *sharply doubly transitive* if G is doubly transitive on Ω and only the identity element fixes two points.

If G is a finite sharply doubly transitive group on Ω then

1. The set M consisting of the identity element and the elements of G without fixed points is an elementary abelian normal subgroup of G of order p^n for some n and some prime p.
2. Addition and multiplication between elements of Ω can be defined so that Ω becomes a nearfield and so that the group G is isomorphic to the group of all affine transformations $v \mapsto va + b$ of Ω, where $a \in \Omega^\times$ and $b \in \Omega$.

There is a converse to this theorem, namely if N is a nearfield, the group of all transformations $v \mapsto va + b$ acts sharply doubly transitively on N.

Let F be the prime field of N, regard N as a vector space over F and define $\mu : N^\times \to \mathrm{GL}(N)$ by $v^{\mu(a)} = va$. Then for all $a \in N^\times$, $a \neq 1$, the linear transformation $\mu(a)$ is fixed-point-free. Furthermore, μ defines an isomorphism between the multiplicative group $N^\times$ and its image in $\mathrm{GL}(N)$.

Suppose that $G = H \ltimes M$ is a sharply doubly transitive group of degree p^n, as above. The centre of G is trivial and M is a minimal normal subgroup. Thus if Ω' is a minimal permutation representation we may suppose that it is primitive. Then M is transitive on Ω' and since M is abelian, it acts regularly on Ω'. Thus p^n is the minimal degree of a faithful permutation representation of G.

22.3 Constructing Nearfields

There are two types of finite nearfield: the *regular* nearfields of Dickson and the *irregular* nearfields of Zassenhaus. In order to accommodate both types MAGMA has a 'virtual type' `Nfd` and types `NfdDck` and `NfdZss` which inherit from `Nfd`.

22.3.1 Dickson Nearfields

In order to begin exploring `Nfd` types in MAGMA we need a way to create instances of nearfields and their elements. As already mentioned there is a large class of nearfields first described by L. E. Dickson [Dic05a, Dic05b] in 1905 and in this section we describe how to construct them in MAGMA.

The nearfields resulting from this construction will be called *Dickson* (or *regular*) nearfields.

If p is a prime and if the positive integers h and v satisfy

- *if r is a prime or 4 and if r divides v, then r divides $p^h - 1$*
  then (p, h, v) is a *Dickson triple*.

If we write $q = p^h$, the condition above is equivalent to

- *All prime factors of v divide $q - 1$ and $q \equiv 3 \bmod 4$ implies $v \not\equiv 0 \bmod 4$.*
  We call (q, v) a *Dickson pair*.

`DicksonPairs(p, hlo, hhi, vlo, vhi )`

The list of Dickson pairs (q, v) for prime p, where `hlo` and `hhi` are the lower and upper bounds on h and where `vlo` and `vhi` are the lower and upper bounds on v.

`DicksonPairs(p, h1, v1)`

The list of Dickson pairs (p^h, v) for the prime p, where `h1` and `v1` are upper bounds on h and v.

`DicksonTriples(p, hb, vb)`

The list of Dickson triples (p, h, v) for the prime p, where `hb` and `vb` are bounds on h and v.

Example H22E1

For each Dickson pair (equivalently Dickson triple), there is at least one Dickson nearfield.

```
> DicksonPairs(5,3,4,4,5);
[
    [ 125, 4 ],
    [ 625, 4 ]
]
> DicksonPairs(5,4,5);
[
    [ 5, 1 ],
    [ 5, 2 ],
```

```
    [ 5, 4 ],
    [ 25, 1 ],
    [ 25, 2 ],
    [ 25, 3 ],
    [ 25, 4 ],
    [ 125, 1 ],
    [ 125, 2 ],
    [ 125, 4 ],
    [ 625, 1 ],
    [ 625, 2 ],
    [ 625, 3 ],
    [ 625, 4 ]
]
> DicksonTriples(5,4,5);
[
    [ 5, 1, 1 ],
    [ 5, 1, 2 ],
    [ 5, 1, 4 ],
    [ 5, 2, 1 ],
    [ 5, 2, 2 ],
    [ 5, 2, 3 ],
    [ 5, 2, 4 ],
    [ 5, 3, 1 ],
    [ 5, 3, 2 ],
    [ 5, 3, 4 ],
    [ 5, 4, 1 ],
    [ 5, 4, 2 ],
    [ 5, 4, 3 ],
    [ 5, 4, 4 ]
]
```

The isomorphism type of a Dickson nearfield depends on the choice of primitive element of the underlying Galois field. It has been shown by Lüneburg [L$\ddot{7}$1] that if ϕ is the Euler phi-function and g is the order of p modulo v, there are $\phi(v)/g$ isomorphism classes of Dickson nearfields with the same Dickson triple (p, h, v).

The default nearfield will use the 'standard' primitive element of the field. The other variants with the same Dickson pair can be obtained by providing an integer s coprime to v. Internally this is converted to a suitable integer e coprime to $q^v - 1$ such that $s \equiv e \pmod{v}$.

`NumberOfVariants(q, v)`

The number of non-isomorphic nearfields with Dickson pair (q, v).

`NumberOfVariants(N)`

The number of variants of the Dickson nearfield N.

`VariantRepresentatives(q, v)`

Representatives for the variant parameter of nearfields with Dickson pair (q, v).

Example H22E2

For each Dickson pair there can be several variants. The variant representative can be used when constructing the corresponding Dickson nearfield.

```
> NumberOfVariants(625,4);
2
> VariantRepresentatives(625,4);
[ 1, 3 ]
```

`DicksonNearfield(q, v : parameters)`

`Variant`	RngIntElt	*Default :* 1
`LargeMatrices`	BoolElt	*Default :* `false`

Create a Dickson nearfield from the Dickson pair (q, v). The `Variant` parameter is an integer s which can be used to specify the choice of primitive element (see the discussion following the intrinsic `DicksonTriples`). The parameter `LargeMatrices` is used only when the group of units of the nearfield is requested. The default is to represent the group of units as a matrix group defined over the kernel of the nearfield. But if `LargeMatrices` is `true`, the matrices are defined over the prime field.

Example H22E3

As indicated in the previous example, up to isomorphism, there are two Dickson nearfields with Dickson pair $(625, 4)$.

```
> D := DicksonNearfield(625,4);
> D3 := DicksonNearfield(625,4 : Variant := 3);
> D5 := DicksonNearfield(625,4 : Variant := 5);
> D eq D3;
false
> D3 eq D5;
false
> D eq D5;
true
> D;
Nearfield D of Dickson type defined by the pair (625, 4)
Order = 152587890625
```

22.3.2 Zassenhaus Nearfields

It was shown by Zassenhaus [Zas35] that in addition to the regular nearfields there are seven *irregular* nearfields. Zassenhaus gave constructions but did not prove their uniqueness. The proofs in [Zas35] are known to contain gaps. Perhaps the most reliable account of the existence and uniqueness of the irregular nearfields is the PhD thesis of Dancs-Groves [Gro74].

The seven finite nearfields which are not Dickson nearfields are the *Zassenhaus* nearfields.

Zassenhaus nearfields can be distinguished from regular nearfields by the fact that the multiplicative group of a finite nearfield N is metacyclic if and only if N is regular.

As a consequence, a Zassenhaus nearfield cannot occur as a subfield of a Dickson nearfield.

`ZassenhausNearfield(n)`

Creates the nth Zassenhaus nearfield.

Example H22E4

The orders of the Zassenhaus nearfields are 5^2, 11^2, 7^2, 23^2, 11^2, 29^2 and 59^2.

```
> for n := 1 to 7 do ZassenhausNearfield(n); end for;
Irregular nearfield Z with Zassenhaus number 1
Order = 25
Irregular nearfield Z with Zassenhaus number 2
Order = 121
Irregular nearfield Z with Zassenhaus number 3
Order = 49
Irregular nearfield Z with Zassenhaus number 4
Order = 529
Irregular nearfield Z with Zassenhaus number 5
Order = 121
Irregular nearfield Z with Zassenhaus number 6
Order = 841
Irregular nearfield Z with Zassenhaus number 7
Order = 3481
```

22.4 Operations on Elements

22.4.1 Nearfield Arithmetic

The operations of addition, subtraction and negation are inherited from the underlying Galois field.

The operation of multiplication distinguishes a nearfield from a field. In a nearfield, multiplication is not commutative and the left distributive law fails.

```
+ a
- a
a + b
a - b
a * b
a / b
a ^ k
a +:= b
a -:= b
a *:= b
```

```
Inverse(a)
```

The inverse of a.

22.4.2 Equality and Membership

```
a eq b
a ne b
a in N
a notin N
```

22.4.3 Parent and Category

```
Parent(a)
Category(a)
```

```
N ! x
```

```
Element(N, x)
```

Create a nearfield element from a finite field element.

```
ElementToSequence(x)
```

Create a sequence from an element x of a nearfield.

22.4.4 Predicates on Nearfield Elements

```
IsZero(a)
IsUnit(a)
IsIdentity(a)
```

Example H22E5

This example illustrates some of the basic operations available on nearfields and their elements. There is a strong connection with the arithmetic of the underlying Galois field of a nearfield D, which is available as the attribute D`gf.

```
> D := DicksonNearfield(3^2,2);
> K := D`gf;
> x := Element(D,K.1);
> x;
$.1
> Parent(x);
Nearfield D of Dickson type defined by the pair (9, 2)
Order = 81
> x^2;
$.1^10
> Identity(D);
1
> assert x ne Identity(D);
> assert x eq x;
> Zero(D);
0
> Parent(Zero(D));
Nearfield D of Dickson type defined by the pair (9, 2)
Order = 81
> assert not IsZero(D!1);
> assert not IsZero(x);
> assert IsZero(Zero(D));
> K<z> := GF(3,4);
> x := Element(D,z^61);
> y := Element(D,z^54);
> assert x + y eq Element(D,z^61+z^54);
> assert x - y eq Element(D,z^61-z^54);
> x*y;
z^35
> x/y;
z^7
> x^y;
z^29
```

Example H22E6

A nearfield is right-distributive, but unlike a Galois field, multiplication is not commutative and the left-distributive law may fail.

```
> N := DicksonNearfield(3^2,4);
> F<a> := N`gf;
> x := Element(N,a^5215);
> y := Element(N,a^5140);
```

```
> z := Element(N,a^5819);
> x*y eq y*x;
false
> x*(y+z) eq x*y+x*z;
false
> (y+z)*x eq y*x+z*x;
true
```

22.5 Operations on Nearfields

`N eq M` `N ne M`

`#N`

`Cardinality(N)`

The cardinality of the nearfield N.

`Random(N)`

A random element of the nearfield N.

`Identity(N)`

The multiplicative identity of the nearfield N.

`Zero(N)`

The additive identity of the nearfield N.

`PrimeField(N)`

The prime field of the nearfield N.

`Kernel(N)`

Return the kernel of the nearfield N as a finite field.

22.6 The Group of Units

If N is a nearfield and $F = \mathcal{K}(N)$ is its kernel, N is a vector space over F and for all $u \in N^\times$, the map $x \mapsto x \circ u$ is an F-linear transformation. This action of $N^\times$ on the non-zero elements of the vector space is transitive and fixed-point-free.

Similarly, we may regard N as a vector space over its prime field and again the elements of $N^\times$ act as linear transformations. In the following code the vector space E could be either a vector space over the kernel or a vector space of the prime field. The default setting is to use the kernel. But if the parameter `LargeMatrices` is set to `true` when a regular nearfield is first defined, the prime field will be used. For irregular nearfields the kernel coincides with the prime field.

Let (p, h, v) be the Dickson triple for N, let ζ be a primitive element of $K = \mathbf{F}_{q^v}$ and put $A = \langle \zeta^v \rangle$. Then A is a group of order $m = (q^v - 1)/v$ and the elements $s_i = \zeta^{(q^i-1)/(q-1)}$ $(1 \le i \le v)$ are coset representatives for A in $K^\times$. Let Φ denote the Frobenius automorphism $x \mapsto x^q$ of K and define $\rho : K^\times \to \mathrm{Gal}(K/\mathbf{F}_p)$ by $\rho(u) = \Phi^i$ if $u \in s_i A$; that is, letting automorphisms of K act on the right, we have $x^{\rho(u)} = x^{q^i}$. The map ρ is not a homomorphism. However, its image is the cyclic group of order v generated by $\Phi = \rho(\zeta)$ and the fixed field of $\mathrm{im}\rho$ is $\mathbf{F}_q$; thus $\mathrm{im}\rho$ may be identified with $\mathrm{Gal}(K/\mathbf{F}_q)$.

The underlying set of N is identified with K and multiplication in N is defined to be $w \circ u = w^{\rho(u)}u$.

The group U of units of the Dickson nearfield $D = D(p, h, v, \zeta)$ has generators a and b and relations $a^m = 1$, $b^v = a^t$ and $b^{-1}ab = a^q$, where $q = p^h$, $m = (q^v - 1)/v$ and $t = m/(q-1)$. Furthermore, Ellers and Karzel [EK64] show that $\gcd(v, t) = \gcd(q - 1, t) \le 2$. Equality holds if and only if $v \equiv 2 \pmod 4$ and $q \equiv 3 \pmod 4$ and this in turn is equivalent to the Sylow 2-subgroup of U being a generalised quaternion group.

The centre of D is $\mathbf{F}_q$ and its group of units is generated by ζ^{vt}.

`UnitGroup(N)`

`UnitGroup(GrpPerm, N)`

`UnitGroup(GrpPC, N)`

The unit group of the nearfield N.

Example H22E7

In this example we construct the group of units of a subnearfield.

```
> N := DicksonNearfield(3^3,13);
> zeta := N`prim;
> x := N!(zeta^((3^39-1) div (3^13-1)));
> S := sub< N | x >;
> U := UnitGroup(S);
> IsAbelian(U);
true
> Factorisation(#N);
[ <3, 39> ]
```

```
> Factorisation(#S);
[ <3, 13> ]
> Factorisation(#Kernel(N));
[ <3, 3> ]
> S;
Nearfield S of Dickson type defined by the pair (1594323, 1)
Order = 1594323
```

`Order(x)`

The order of the unit x of a nearfield.

As a matrix group, the unit group U of a nearfield acts regularly on the non-zero vectors of the underlying vector space E and consequently the affine group $E \cdot U$ is sharply two-transitive. All sharply two-transitive groups occur in this way.

`AffineGroup(N)`

`AffineGroup(GrpPerm, N)`

`AffineGroup(GrpPC, N)`

The sharply two-transitive affine group associated with a nearfield, returned as a matrix group.

If $\Gamma = \mathrm{Gal}(K/\mathbf{F}_p)$ and $S = \Gamma \ltimes K^\times$ is the semidirect product of Γ and $K^\times$, then $D^\times \to S : w \mapsto \rho(w)w$ is an embedding of the multiplicative group $D^\times$ of $D = D(p, h, v, \zeta)$ in S, where multiplication in S is defined by

$$(\gamma_1 a_1)(\gamma_2 a_2) = \gamma_1\gamma_2 a_1^{\gamma_2} a_2.$$

If U is the image of $D^\times$ in S, then $\Gamma \cap U = 1$, $\Gamma U = S$ and $K^\times \cap U = A = \langle \zeta^v \rangle$. In fact, from the definition of ρ, we have $UK^\times = \Gamma_0 \ltimes K^\times$, where $\Gamma_0 = \mathrm{Gal}(K/\mathbf{F}_q)$. This is the *extended unit group* of the Dickson nearfield D.

`ExtendedUnitGroup(D)`

The extended unit group of a Dickson nearfield.

22.7 Automorphisms

`IsIsomorphic(N1, N2)`

Test whether the regular nearfields N_1 and N_2 are isomorphic. If they are, return an isomorphism.

`AutomorphismGroup(N)`

The automorphism group A of the regular nearfield N and a map giving the action of A on N.

22.8 Nearfield Planes

A nearfield N is said to be *planar* if the mapping $x \mapsto -xa + xb$ is a permutation of N whenever $a \neq b$. Every finite nearfield is planar.

Given a finite nearfield N, there is an *affine* plane $\mathcal{A}$ with point set $N \times N$ and lines given by the equations

$$y = xm + b$$
$$x = c$$

Let $\mathcal{P}$ be the corresponding projective plane, obtained from $\mathcal{A}$ by adjoining a line L_∞ called the *line at infinity*. We label the points of $\mathcal{P}$ with triples of elements of N as follows.

(1) For every point (x, y) of $\mathcal{A}$ there is a point $[1, x, y]$ of $\mathcal{P}$.

(2) For every m there is an "ideal" point $[0, 1, m]$ of $\mathcal{P}$ which lies on every line $y = xm + b$ $(b \in N)$ and on L_∞.

(3) There is a point $[0, 0, 1]$ of $\mathcal{P}$ which lies on every line $x = c$ and on L_∞.

The lines of $\mathcal{P}$ may also be labelled by triples of elements of N: the line $y = xm + b$ corresponds to the triple $[-b, -m, 1]$ and the line $x = c$ corresponds to $[-c, 1, 0]$. The line L_∞ is labelled $[1, 0, 0]$. A point $\pi = [w, x, y]$ is incident with a line $L = [a, b, c]$ if and only if $wa + xb + yc = 0$.

Every collineation of $\mathcal{A}$ extends to a collineation of $\mathcal{P}$.

`ProjectivePlane( N : parameters)`

`Check` BoolElt *Default :* `false`

The finite projective plane coordinatised by the nearfield N. The points of the nearfield plane are represented as triples of Galois field elements.

Example H22E8

```
> N := DicksonNearfield(3,2);
> pl := ProjectivePlane(N);
> A := AutomorphismGroup(pl);
> #A;
311040
> CompositionFactors(A);
    G
    |  Cyclic(2)
    *
    |  Alternating(5)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
```

```
|  Cyclic(2)
*
|  Cyclic(2)
*
|  Cyclic(3)
*
|  Cyclic(3)
*
|  Cyclic(3)
*
|  Cyclic(3)
1
```

22.8.1 Hughes Planes

In 1957 Hughes [Hug57] discovered a class of finite projective planes constructed from the Dickson nearfields which have rank 2 over their kernel. Neither these planes nor their duals are translation planes and therefore they cannot be obtained by the coordinatisation method of the previous section. Hughes' methods required the kernel to be central but in 1960 the construction was generalised by Rosati [Ros60] to include the Zassenhaus nearfields (see also Dembowski [Dem68, §5.4] and [Dem71]). For simplicity of notation we shall use the term 'Hughes plane' to include both Hughes planes and generalised Hughes planes.

`HughesPlane(N : parameters)`

`Check` BoolElt *Default :* `false`

The Hughes plane based on the nearfield N.

Example H22E9

We construct the Desarguesian projective plane `PG(2,49)` and then, using nearfields of order 49, we construct three non-Desarguesian projective planes. These four planes can be distinguished by the orders of their collineation groups.

```
> DP := FiniteProjectivePlane(49); // Desarguesian plane
> DP;
Projective Plane PG(2, 49)
> CD  := CollineationGroup(DP);
> FactoredOrder(CD);
[ <2, 10>, <3, 3>, <5, 2>, <7, 6>, <19, 1>, <43, 1> ]
> N := DicksonNearfield(7,2);
> NP := ProjectivePlane(N);
> NP;
Projective Plane of order 49
> CN := CollineationGroup(NP);
> FactoredOrder(CN);
```

```
[ <2, 10>, <3, 2>, <7, 4> ]
> Z := ZassenhausNearfield(3);
> #Z;
49
> ZP := ProjectivePlane(Z);
> CZ := CollineationGroup(ZP);
> FactoredOrder(CZ);
[ <2, 9>, <3, 3>, <7, 4> ]
> HP := HughesPlane(N);
> HP;
Projective Plane of order 49
> CH := CollineationGroup(HP);
> FactoredOrder(CH);
[ <2, 6>, <3, 3>, <7, 3>, <19, 1> ]
> CompositionFactors(CH);
    G
    |  Cyclic(3)
    *
    |  A(2, 7)               = L(3, 7)
    *
    |  Cyclic(2)
    1
```

22.9 Bibliography

[**Dem68**] Peter Dembowski. *Finite geometries.* Ergebnisse der Mathematik und ihrer Grenzgebiete, Band 44. Springer-Verlag, Berlin, 1968.

[**Dem71**] Peter Dembowski. Generalized Hughes planes. *Canad. J. Math.*, 23:481–494, 1971.

[**Dic05a**] Leonard Eugene Dickson. Definitions of a group and a field by independent postulates. *Trans. Amer. Math. Soc.*, 6(2):198–204, 1905.

[**Dic05b**] Leonard Eugene Dickson. On finite algebras. *Nachr. Kgl. Ges. Wiss. Göttingen, Math.-phy. Klasse*, pages 358–393, 1905.

[**EK63**] Erich Ellers and Helmut Karzel. Kennzeichnung elliptischer Gruppenräume. *Abh. Math. Sem. Univ. Hamburg*, 26:55–77, 1963.

[**EK64**] Erich Ellers and Helmut Karzel. Endliche Inzidenzgruppen. *Abh. Math. Sem. Univ. Hamburg*, 27:250–264, 1964.

[**Gro74**] Susan Dancs Groves. *Locally finite near-fields.* PhD thesis, Australian National University, 1974.

[**Hal59**] Marshall Hall, Jr. *The theory of groups.* The Macmillan Co., New York, N.Y., 1959.

[**Hug57**] D. R. Hughes. A class of non-Desarguesian projective planes. *Canad. J. Math.*, 9:378–388, 1957.

[**L$\ddot{\text{o}}$9**] Heinz Lüneburg. *Lectures on projective planes.* University of Illinois, Chicago, 1969.

[**L$\ddot{\text{u}}$1**] Heinz Lüneburg. Über die Anzahl der Dickson'schen Fastkörper gegebener Ordnung. In *Atti del Convegno di Geometria Combinatoria e sue Applicazioni (Univ. Perugia, Perugia, 1970)*, pages 319–322. Ist. Mat., Univ. Perugia, Perugia, 1971.

[**Ros60**] Luigi Antonio Rosati. Su una generalizzazione dei piani di Hughes. *Atti Accad. Naz. Lincei Rend. Cl. Sci. Fis. Mat. Nat. (8)*, 29:303–308 (1961), 1960.

[**Zas35**] Hans Zassenhaus. Über endliche Fastkörper. *Abh. Math. Sem. Univ. Hamburg*, 11:187–220, 1935.

23 UNIVARIATE POLYNOMIAL RINGS

Chapter 23
UNIVARIATE POLYNOMIAL RINGS

23.1 Introduction

Univariate polynomial rings may be defined over any ring R. Let us denote the univariate polynomial ring in indeterminate x over the coefficient ring R by $P = R[x]$.

There are two kinds of polynomials in MAGMA: *univariate* polynomials, represented as vectors of coefficients; and *multivariate polynomials* represented in distributive form (linear sums of coefficient-monomial pairs). In this chapter we discuss univariate polynomials.

23.1.1 Representation

The vector representation enables fast arithmetic on univariate polynomials, but it requires considerable amounts of memory for multivariate polynomials; therefore, only univariate polynomial rings using the vector representation can be created directly (but, if one insists, it is possible to create univariate polynomial rings over univariate polynomial rings, etc.). Multivariate polynomials can be stored efficiently in distributive form, but the arithmetic operations on polynomials of one variable stored in this way may be considerably slower.

23.2 Creation Functions

23.2.1 Creation of Structures

There are two different ways to create polynomial rings, corresponding to the different internal representations (vector versus distributive — see the introductory section): `PolynomialRing(R)` and `PolynomialRing(R, n)`. The latter should be used to create multivariate polynomials; the former should be used for univariate polynomials.

`PolynomialAlgebra(R)`

`PolynomialRing(R)`

`Global` BOOLELT *Default :* `true`

Create a univariate polynomial ring over the ring R. The ring is regarded as an R-algebra via the usual identification of elements of R and the constant polynomials. The polynomials are stored in vector form, which allows fast arithmetic. It is not recommended to use this function recursively to build multivariate polynomial rings. The angle bracket notation can be used to assign names to the indeterminate, e.g.: `P<x> := PolynomialRing(R)`.

By default, the unique *global* univariate polynomial ring over R will be returned; if the parameter `Global` is set to `false`, then a non-global univariate polynomial ring over R will be returned (to which a separate name for the indeterminate can be assigned).

Example H23E1

We demonstrate the difference between global and non-global rings. We first create the global univariate polynomial ring over **Q** twice.

```
> Q := RationalField();
> P<x> := PolynomialRing(Q);
> PP := PolynomialRing(Q);
> P;
Univariate Polynomial Ring in x over Rational Field
> PP;
Univariate Polynomial Ring in x over Rational Field
> PP.1;
x
```

PP is identical to P. We now create non-global univariate polynomial rings (which are also different to the global polynomial ring P). Note that elements of all the rings are mathematically equal by automatic coercion.

```
> Pa<a> := PolynomialRing(Q: Global := false);
> Pb<b> := PolynomialRing(Q: Global := false);
> Pa;
Univariate Polynomial Ring in a over Rational Field
> Pb;
Univariate Polynomial Ring in b over Rational Field
> a;
a
> b;
b
> P;
Univariate Polynomial Ring in x over Rational Field
> x;
x
> x eq a; // Automatic coercion
true
> x + a;
2*x
```

23.2.2 Print Options

The `AssignNames` and `Name` functions can be used to associate a name with the indeterminate of a polynomial ring after creation.

```
AssignNames(~P, s)
```

Procedure to change the name of the indeterminate of a polynomial ring P. The indeterminate will be given the name of the string in the sequence s.

This procedure only changes the name used in printing the elements of P. It does *not* assign to identifiers corresponding to the strings the indeterminates in P; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies P, it is necessary to have a reference ~P to P in the call to this function.

```
Name(P, i)
```

Given a polynomial ring P, return the i-th indeterminate of P (as an element of P).

23.2.3 Creation of Elements

The easiest way to create polynomials in a given ring is to use the angle bracket construction to attach names to the indeterminates, and to use these names to express polynomials (see the examples). Below we list other options.

```
P . 1
```

Return the indeterminate for the polynomial ring P, as an element of P.

```
elt< P | a0, ..., ad >
```

Given a polynomial ring $P = R[x]$ and elements $a_0, \ldots, a_d$ coercible into the coefficient ring R, return the polynomial $a_0 + a_1 x_n + \cdots + a_d x_n^d$ as an element of P.

```
P ! s
```

```
elt< P | s >
```

Coerce the element s into the polynomial ring $P = R[x]$. The following possibilities for s exist.

(a) s is an element of P: it is returned unchanged;

(b) s is an element of a ring that can be coerced into the coefficient ring R of P: the constant polynomial s is returned;

(c) $s = \sum_j s_j y^j$ is an element of a univariate polynomial ring whose coefficient ring elements s_j can be coerced into R: the polynomial $\sum_j r_j x^j$ is returned, where r_j is the result of coercing s_j into R;

(c) s is a sequence: if s is empty then the zero element of P is returned, and if it is non-empty but the elements of the sequence can be coerced into R then the polynomial $\sum_j s[j] x_n^{j-1}$ is returned.

Note that constant polynomials may be coerced into their coefficient rings.

`Polynomial(Q)`

Given a sequence Q of elements from a ring R, create the polynomial over R whose coefficients are given by Q. This is equivalent to `PolynomialRing(Universe(Q))!Q`.

`Polynomial(R, Q)`

Given a ring R and sequence Q of elements from a ring S, create the polynomial over R whose coefficients are given by the elements of Q, coerced into S. This is equivalent to `PolynomialRing(R)!ChangeUniverse(Q, R)`.

`Polynomial(R, f)`

Given a ring R and a polynomial f over a ring S, create the polynomial over R obtained from f by coercing its coefficients into S. This is equivalent to `PolynomialRing(R)!f`.

`One(P)` `Identity(P)` `Zero(P)` `Representative(P)`

Example H23E2

The easiest way to create the polynomial $x^3 + 3x + 1$ (over the integers) is as follows.

```
> P<x> := PolynomialRing(Integers());
> f := x^3+3*x+1;
> f;
x^3 + 3*x + 1
```

Alternative ways to create polynomials are given by the element constructor (rarely used) and the `!` operator:

```
> P<x> := PolynomialAlgebra(Integers());
> f := elt< P | 2, 3, 0, 1 >;
> f;
x^3 + 3*x + 2
> P ! [ 2, 3, 0, 1 ];
x^3 + 3*x + 2
```

Note that it is important to realize that a sequence is coerced into a polynomial ring by coercing its entries into the coefficient ring, and it is not attempted first to coerce the sequence as a whole into the coefficient ring:

```
> Q := RationalField();
> Q ! [1, 2];
1/2
> P<x> := PolynomialRing(Q);
> P ! [1,2];
2*x + 1
> P ! Q ! [1,2];
1/2
> P ! [ [1,2], [2,3] ];
2/3*x + 1/2
```

23.3 Structure Operations

23.3.1 Related Structures

The main structure related to a polynomial ring is its coefficient ring. Univariate polynomial rings belong to the MAGMA category `RngUPol`.

`BaseRing(P)`

`CoefficientRing(P)`

`CoefficientRing(f)`

Return the coefficient ring of polynomial ring P (the parent of f).

`Category(P)` `Parent(P)` `PrimeRing(P)`

23.3.2 Changing Rings

The `ChangeRing` function enables changing coefficient rings on a polynomial ring.

`ChangeRing(P, S)`

Given a polynomial ring $P = R[x]$, together with a ring S, construct the polynomial ring $Q = S[y]$, together with the homomorphism h from P to Q. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. The homomorphism h will apply this coercion to the coefficients of elements in P to return elements of Q. The usual angle bracket notation can be used for indeterminate names on the result.

`ChangeRing(P, S, f)`

Given a polynomial ring $P = R[x]$, together with a ring S and a map $f : R \to S$, construct the polynomial ring $Q = S[y]$ together with the homomorphism h from P to Q obtained by applying h to the coefficients of elements of P. The usual angle bracket notation can be used for indeterminate names on the result.

Example H23E3

In the first example of `ChangeRing` below we use automatic coercion of integers to rationals to go from $\mathbf{Z}[x]$ to $\mathbf{Q}[y]$. In fact `!` can be used for this as well. In the second example we use a map to obtain a non-standard embedding (not mapping 1 to 1) of $\mathbf{Z}$ in $\mathbf{Q}$.

```
> Z := Integers();
> Q := RationalField();
> P<x> := PolynomialRing(Z);
> S<y>, h := ChangeRing(P, Q);
> h(x^3-2*x+5);
y^3 - 2*y + 5
> S ! (x^3-2*x+5);
y^3 - 2*y + 5
> m := hom< Z -> Q | x :-> 3*x >;
```

```
> S<y>, h := ChangeRing(P, Q, m);
> h(x^3-2*x+5);
3*y^3 - 6*y + 15
```

23.3.3 Numerical Invariants

The characteristic can be obtained for any polynomial ring, the rank for free polynomial rings and the cardinality only for finite quotients.

`Rank(P)`

Return the rank of the polynomial ring P, defined as the maximal number of independent indeterminates in P over its coefficient ring; for univariate polynomial rings this will therefore always return 1.

`#P`

Return the number of elements of P; this will only return an integer value if P is finite, which for polynomial rings can only happen for quotients of polynomial rings over finite coefficient rings.

`Characteristic(P)`

23.3.4 Ring Predicates and Booleans

The usual ring functions returning Boolean values are available on polynomial rings.

`IsCommutative(P)` `IsUnitary(P)` `IsFinite(P)` `IsOrdered(P)`
`IsField(P)` `IsEuclideanDomain(P)` `IsPID(P)` `IsUFD(P)`
`IsDivisionRing(P)` `IsEuclideanRing(P)` `IsDomain(P)`
`IsPrincipalIdealRing(P)` `P eq Q` `P ne Q` `P lt Q`
`P gt Q` `P le Q` `P ge Q`

23.3.5 Homomorphisms

A ring homomorphism taking a polynomial ring $R[x]$ as its domain requires 2 pieces of information, namely, a map (homomorphism) telling how to map the coefficient ring R, together with the image of the indeterminate x. The map may be omitted.

`hom< P -> S | f, y >`
`hom< P -> S | y >`

Given a polynomial ring $P = R[x]$, a ring S, a map $f : R \to S$ and an element $y \in S$, create the homomorphism $g : P \to S$ given by that $g(\sum s_i x^i) = \sum f(s_i) y^i$.

The coefficient ring map may be omitted, in which case the coefficients are mapped into S by the unitary homomorphism sending 1_R to 1_S. Also, the image y is allowed to be from a structure that allows automatic coercion into S.

Example H23E4

In this example we map $\mathbf{Z}[x]$ into the reals by sending x to $1/2$. Note that we do not have a choice for the coefficient map (since we require it to be unitary), and also that we give the image of x as a rational number that is automatically coerced into the reals.

```
> Z := Integers();
> P<x> := PolynomialRing(Z);
> Re := RealField(20);
> half := hom< P -> Re | 1/2 >;
> half(x^3-3*x+5);
3.625
```

23.4 Element Operations

The categories for elements in univariate polynomial rings and their quotients are are `RngUPolElt` and `RngUPolResElt`

23.4.1 Parent and Category

`Parent(p)` `Category(p)`

23.4.2 Arithmetic Operators

The usual unary and binary ring operations are available for univariate polynomials, with the following notable restrictions.

Since inverses cannot generally be obtained in polynomial rings, division (using `/`) of polynomials is not allowed, and neither are negative powers. For polynomial rings over fields division by elements of the coefficient field are allowed.

The operators `div` and `mod` give results corresponding to the quotient and the remainder of division of the arguments. See the section on quotient and remainder for details.

`+ a` `- a`

`a + b` `a - b` `a * b` `a ^ k` `a / b`

`a div b` `a mod b`

`a +:= b` `a -:= b` `a *:= b`

23.4.3 Equality and Membership

`a eq b` `a ne b`

`a in R` `a notin R`

23.4.4 Predicates on Ring Elements

The list belows contains the general ring element predicates. Note that not all functions are available for every coefficient ring.

`IsZero(a)` `IsOne(a)` `IsMinusOne(a)`
`IsNilpotent(a)` `IsIdempotent(a)`
`IsUnit(a)` `IsZeroDivisor(a)` `IsRegular(a)`
`IsIrreducible(a)` `IsPrime(a)` `IsMonic(a)`

23.4.5 Coefficients and Terms

`Coefficients(p)`
`ElementToSequence(p)`
`Eltseq(p)`

The coefficients of the polynomial $p \in R[x]$ in ascending order, as a sequence of elements of R.

`Coefficient(p, i)`

Given a polynomial $p \in R[x]$ and an integer $i \geq 0$, return the coefficient of the i-th power of x in f. (If i exceeds the degree of f then zero is returned.) The return value is an element of R.

`MonomialCoefficient(p, m)`

Given elements p and m of a polynomial ring $P = R[x]$, where m is a monomial (that is, has exactly one non-zero base coefficient, which must be 1), return the coefficient of m in p, as an element of the coefficient ring R.

`LeadingCoefficient(p)`

Return the coefficient of the highest occurring power of x in $p \in R[x]$, as an element of the coefficient ring R.

`TrailingCoefficient(p)`

Return the coefficient of the lowest occurring power of x in $p \in R[x]$, as an element of the coefficient ring R.

`ConstantCoefficient(p)`

Return the constant term, ie. the coefficient of x^0 as an element of the coefficient ring R.

`Terms(p)`

Return the non-zero terms of the polynomial $p \in P = R[x]$ in ascending order with respect to the degree, as a sequence of elements of P with ascending degrees.

`LeadingTerm(p)`

Return the term of $p \in P = R[x]$ with the highest occurring power of x, as an element of P. The coefficient of the result will be the leading coefficient of p.

`TrailingTerm(p)`

Return the term of $p \in P = R[x]$ with the lowest occurring power of x, as an element of P. The coefficient of the result will be the trailing coefficient of p.

`Monomials(p)`

The monomials of the univariate p, matching up with `Coefficients(p)`, that is a sequence of powers of the indeterminate up to the degree of p.

`Support(p)`

Given a polynomial $p \in R[x]$, return the positions in p for which there are non-zero coefficients, and the corresponding coefficients.

`Round(p)`

Given a polynomial $p \in P = R[x]$ where R is a subring of the real field (the ring of integers $\mathbf{Z}$, the rational field $\mathbf{Q}$, or a real field), return the polynomial in $Z[x]$ obtained from p by rounding all the coefficients of p.

`Valuation(p)`

The valuation of a polynomial $p \in R[x]$, that is, the exponent of the largest power of x which divides p. Note that the zero polynomial has valuation ∞.

23.4.6 Degree

`Degree(p)`

The degree of a polynomial $p \in R[x]$, that is, the exponent of the largest power of x that occurs with non-zero coefficient. Note that the zero polynomial has degree -1.

23.4.7 Roots

`Roots(p)`

`Max` RngIntElt *Default :*

Given a polynomial p over one of a certain collection of coefficient rings, this function returns a sorted sequence of pairs of coefficient ring element and integer, where the ring element is a root of p in the coefficient ring, and the integer its multiplicity. Currently the coefficient rings that are allowed comprise complex and real fields, integers and rationals, finite fields and residue class rings with prime modulus. If the parameter `Max` is set to a non-negative number m, at most m roots are returned.

`Roots(p, S)`

Given a polynomial p over one of a certain collection of coefficient rings as well as a ring S into which the coefficients of p can be coerced automatically, this function returns a sorted sequence of pairs of ring element and integer, where the ring element is a root of p in the ring S, and the integer its multiplicity. Currently the coefficient rings that are allowed comprise complex and real fields, integers and rationals, finite fields and residue class rings with prime modulus.

`HasRoot(p)`

Given a polynomial p over the coefficient ring R this function returns `true` iff p has a root in R. If the result is `true`, the function also returns a root of p as a second return value. Currently the coefficient rings that are allowed comprise complex and real fields, integers and rationals, finite fields and residue class rings with prime modulus. Note that particularly for finite fields, this method may be much faster than the computation of all roots of the polynomial.

`HasRoot(p, S)`

Given a polynomial p over the coefficient ring R and a ring S which contains R, this function returns `true` iff p has a root in S. If the result is `true`, the function also returns a root of p in S as a second return value. Currently the coefficient rings that are allowed comprise complex and real fields, integers and rationals, finite fields and residue class rings with prime modulus. Note that particularly for finite fields, this method may be much faster than the computation of all roots of the polynomial.

`SmallRoots(p, N, X)`

`Bits`	BoolElt	*Default :* `false`
`Beta`	RngElt	*Default :* 1.0
`Exponent`	RngElt	*Default :*
`Finalshifts`	RngElt	*Default :*
`Direct`	BoolElt	*Default :* `false`

Given a monic non-zero univariate integer polynomial p and two positive integers N and X, this function returns all x_0's such that $|x_0| \leq X$ and $P(x_0) = 0$ $[N]$, as long as $X \leq 0.5 \cdot N^{1/d}$, where d is the degree of p.

This function implements Coppersmith's algorithm to compute the small roots of a univariate polynomial modulo an integer [Cop96], as described in Alexander May's PhD thesis [May03]. It relies upon the LLL algorithm for reducing euclidean lattices [LLL82]. It is frequently used for cryptanalysing public-key cryptosystems (see the example below).

When `Bits` is set to `true`, the input X is read as 2^X.

The parameter `Beta` can be set to any value in $(0.0, 1.0]$. The routine will then find all x_0's such that $|x_0| \leq X$ and $P(x_0) = 0 \cong N'$, as long as $X \leq 0.5 \cdot N^{\beta^2/d}$, where d is the degree of p and $N' \geq N^\beta$ is any divisor of N.

The `Exponent` and `Finalshifts` specify the shape of the lattice basis to be reduced. If `Exponent` is m, then p^m will be the highest power of p used to build the lattice basis, and if `Finalshifts` is t, t shifts of p^m will be used. Unless requested by the user, these parameters are chosen automatically.

Finally, the `Direct` option allows the user to require the lattice basis to be reduced at once, and not progressively while constructed. This is can be slower.

Example H23E5

We show how to use the `SmallRoots` routine to factor an RSA modulus when some most significant bits of one of the factors is known. We first generate an RSA modulus.

```
> SetSeed(1);
> F<x> := PolynomialRing (Integers());
> length := 1024;
> p:=NextPrime (2^(Round(length/2)): Proof:=false);
> pi:=Pi(RealField());
> q:=NextPrime (Round (pi*p): Proof:=false);
> N := p*q;
```

Suppose that N is known, as well as an approximation of the factor q:

```
> hidden:=220;
> approxq := q+Random(2^hidden-1);
```

Our goal is to recover `q` from our knowledge of `approxq`. We are therefore interested in the small roots of the polynomial $x - approxq$ modulo q, whose multiple N is known.

```
> A:=x-approxq;
> time perturb:=SmallRoots (A, N, hidden : Bits, Beta:=0.5)[1];
Time 0.050
> q eq approxq-perturb;
true
```

`SetVerbose("SmallRoots", v)`

(Procedure.) Set the verbose printing level for the `SmallRoots` routine to be v. Currently the legal values for v are `true`, `false`, 0, 1 or 2 (`false` is the same as 0, and `true` is the same as 1).

23.4.8 Derivative, Integral

`Derivative(p)`

Given a polynomial $p \in P$, return the derivative of p as an element of P.

`Derivative(p, n)`

Given a polynomial $p \in P$ and an integer $n \geq 0$, return the n-th derivative of p as an element of P.

`Integral(p)`

Given a polynomial $p \in P$ over a field of characteristic zero, return the formal integral of p as an element of P.

23.4.9 Evaluation, Interpolation

`Evaluate(p, r)`

Given an element p of a polynomial ring P and an element r of a ring S, return the value of p evaluated at r. If r can be coerced into the coefficient ring R of P, the result will be an element of R. If r cannot be coerced to the coefficient ring, then an attempt is made to do a generic evaluation of p at r. In this case, the result will be an element of S.

`Interpolation(I, V)`

This function finds a univariate polynomial that evaluates to the values V in the interpolation points I. Let K be a field and $n > 0$ an integer; given sequences I and V, both consisting of n elements of K, return the unique univariate polynomial p over K of degree less than n such that $p(I[i]) = V[i]$ for each $1 \leq i \leq n$.

23.4.10 Quotient and Remainder

`Quotrem(f, g)`

Given elements f and g of the polynomial ring $P = R[x]$, this function returns polynomials q (quotient) and r (remainder) in P such that $f = q \cdot g + r$, and the degree of r is minimal. The leading coefficient of g has to be a non-zero divisor in R. If the leading coefficient of g is a unit, then the degree of r will be strictly less than that of g (taking the degree of 0 to be -1). Over the integers ($R = \mathbf{Z}$) this will be true in general when the leading coefficient of g divides that of f.

`f div g`

The quotient of the polynomial f by g, which is the first return value of `Quotrem` described above.

`IsDivisibleBy(a, b)`

Return whether the polynomial f is exactly divisible by the polynomial g; that is, whether there exists q with $f = qg$. If so, return also the exact divisor q.

`ExactQuotient(f, g)`

Assuming that the polynomial f is exactly divisible by the polynomial g, return the exact quotient of f by g (as a polynomial in the same polynomial ring). An error results if g does not divide f exactly.

`f mod g`

The remainder of division of the polynomial f by g, which is the second return value of `Quotrem` described above.

`Valuation(f, g)`

The exponent of the highest power of the polynomial g which divides the polynomial f.

`Reductum(f)`

The reductum of a polynomial f, which is the polynomial obtained by removing the leading term of f.

`PseudoRemainder(f, g)`

Given polynomials f, g in $P = R[x]$, where R is an integral domain, this function returns the pseudo-remainder r of f and g defined as follows. Let d be the maximum of 0 and $\deg(f) - \deg(g) + 1$, and let c be the leading coefficient of g; then r will be the unique polynomial in P such that $c^d \cdot f = q \cdot g + r$ and the degree of r is less than that of g (possibly -1 for $r = 0$).

`EuclideanNorm(p)`

Return the Euclidean norm of the univariate polynomial $p \in P$, where the Euclidean norm is the function that makes P into a Euclidean ring, which is the degree function.

23.4.11 Modular Arithmetic

The following functions allow modular arithmetic for univariate polynomials over a field without the need to move into the quotient ring. See also the description of `mod` in the section on quotient and remainder.

`Modexp(f, n, g)`

Given univariate polynomials f and g in $K[x]$ over a field K, return $f^n \bmod g$ as an element of $K[x]$. Here n must be a non-negative integer, and g is allowed to be a constant polynomial.

`ChineseRemainderTheorem(X, M)`

`CRT(X, M)`

Given two sequences X and M of polynomials where the elements in M are assumed to be pairwise coprime, find a single polynomial t that solve the modular equation $X_i = t$ modulo M_i.

23.4.12 Other Operations

`ReciprocalPolynomial(f)`

The reciprocal of the given univariate polynomial.

`PowerPolynomial(f,n)`

The polynomial whose roots are the nth powers of the roots of the given polynomial (which should have coefficients in some field).

`f ^ M`

The transformation of the univariate polynomial f under the linear fractional transformation given by the 2 by 2 matrix M (obtained by homogenizing f and making a linear substitution).

23.5 Common Divisors and Common Multiples

The functions in this section are restricted to univariate polynomials over a field, over the integers, or over a residue class ring of integers with prime modulus, or any polynomial ring over these.

23.5.1 Common Divisors and Common Multiples

```
GreatestCommonDivisor(f, g)
```
```
Gcd(f, g)
```
```
GCD(f, g)
```

Given univariate polynomials f and g over the ring R, this function returns the greatest common divisor (GCD) of f and g. The valid coefficient rings are those which themselves have a GCD algorithm for their elements (which includes most commutative rings in MAGMA).

If either of the inputs is zero, then the result is the other input (and if the inputs are both zero then the result is zero). The result is normalized (see the function Normalize), so the result is always unique.

For polynomials over finite fields, the simple Euclidean algorithm is used, since this is efficient (there is no intermediate coefficient blowup).

For polynomials over the integers or rationals, a combination of two algorithms is used: (1) the heuristic evaluation 'GCDHEU' algorithm of Char et al. ([CGG89] and [GCL92, section 7.7]), suitable for small to moderate-degree dense polynomials; (2) a modular algorithm similar to that presented in [vzGG99, Algorithm 6.38] or [GCL92, section 7.4] (although lifting all the way up to a bound is not used since it is completely unnecessary for correctness in this algorithm!).

For polynomials over an algebraic number field, quadratic field, or cyclotomic field, a fast modular algorithm is used, which maps the field to a residue class polynomial ring modulo a small prime.

Since V2.10, for polynomials over an algebraic function field or polynomial quotient ring over a function field, a new fast modular algorithm of Allan Steel (to be published) is used, which evaluates and interpolates for each base transcendental variable.

For polynomials over another polynomial ring or function field, the polynomials are first "flattened" to be inside a multivariate polynomial ring over the base coefficient ring, then the appropriate (multivariate) algorithm is used for that base coefficient ring.

For polynomials over any other ring, the generic subresultant algorithm [Coh93, section 3.3] is used.

```
ExtendedGreatestCommonDivisor(f, g)
```
```
Xgcd(f, g)
```
```
XGCD(f, g)
```

The extended greatest common divisor of polynomials f and g in a univariate polynomial ring P: the function returns polynomials c, a and b in P with $\deg(a) < \deg(g)$ and $\deg(b) < \deg(f)$ such that c is the monic GCD of f and g, and $c = a \cdot f + b \cdot g$. The multipliers a and b are unique if f and g are both non-zero. The coefficient ring must be a field.

For polynomials over the rational field, a modular algorithm due to Allan Steel (unpublished) is used; over other fields the basic Euclidean algorithm is used.

`LeastCommonMultiple(f, g)`

`Lcm(f, g)`

`LCM(f, g)`

The least common multiple of polynomials f and g in a univariate polynomial ring P. The LCM of zero and anything else is zero. The result is normalized (see the function `Normalize`), so the result is always unique. The valid coefficient rings are as for the function `GCD`, above.

The LCM is effectively computed as `Normalize((F div GCD(F, G)) * G)`, for non-zero inputs.

`Normalize(f)`

Given a univariate polynomial f over the ring R, this function returns the unique normalized polynomial g which is associate to f (so $g = uf$ for some unit in R). This is chosen so that if R is a field then g is monic, if R is $\mathbf{Z}$ then the leading coefficient of g is positive, if R is a polynomial ring itself, then the leading coefficient of g is recursively normalized, and so on for other rings.

23.5.2 Content and Primitive Part

`Content(p)`

The content of p, that is, the greatest common divisor of the coefficients of p as an element of the coefficient ring.

`PrimitivePart(p)`

The primitive part of p, being p divided by the content of p.

`ContentAndPrimitivePart(p)`

`Contpp(p)`

The content (the greatest common divisor of the coefficients) of p, as an element of the coefficient ring, as well as the primitive part (p divided by the content) of p.

23.6 Polynomials over the Integers

The functions in this section are available for univariate polynomials over the integers only.

`Sign(p)`

The sign of the leading coefficient of p.

`AbsoluteValue(p)`

`Abs(p)`

Returns either p or $-p$ according to which one has non-negative leading coefficient.

`MaxNorm(p)`

The maximum of the absolute values of the coefficients of p.

`SumNorm(p)`

The sum of the coefficients of p.

`DedekindTest(p, m)`

Given a monic polynomial p (univariate or multivariate in one variable) and a prime number m, this returns true if p satisfies the Dedekind criterion at m, and false otherwise. The Dedekind criterion is satisfied at m if and only if the equation order corresponding to p is locally maximal at m [PZ89, p. 295].

23.7 Polynomials over Finite Fields

The functions in this section are available for univariate polynomials over finite fields only.

`PrimePolynomials(R, d)`

`PrimePolynomials(R, d, n)`

A sequence of all monic prime polynomials of R of degree d, resp. a sequence of n monic prime polynomials of R of degree d.

`RandomPrimePolynomial(R, d)`

A random monic prime polynomial of R of degree d.

`NumberOfPrimePolynomials(q, d)`

`NumberOfPrimePolynomials(K, d)`

`NumberOfPrimePolynomials(R, d)`

The number of monic prime polynomials of degree d over the respective finite field.

`JacobiSymbol(a,b)`

The Jacobi symbol (a/b) of the two polynomials $a, b \in \mathbf{F}_q[x]$ where q must be odd. If b is irreducible, the symbol equals 0 if b divides a. It equals 1 if a is a square mod b and -1 otherwise. The symbol then extends multiplicatively to all non-constant polynomials b.

23.8 Factorization

This section describes the functions for polynomial factorization and associated computations. These are available for several kinds of coefficient rings.

23.8.1 Factorization and Irreducibility

`Factorization(f)`
`Factorisation(f)`

`Al` MonStgElt *Default : "Default"*

Given a univariate polynomial f over the ring R, this function returns the factorization of f as a factorization sequence Q, that is, a sequence of pairs, each consisting of an irreducible factor q_i a positive integer k_i (its multiplicity). Each irreducible factor is normalized (see the function `Normalize`), so the expansion of the factorization sequence is the unique canonical associate of f. The function also returns the unit u of R giving the normalization, so $f = u \cdot \prod_i q_i^{k_i}$.

The coefficient ring R must be one of the following: a finite field $\mathbf{F}_q$, the ring of integers $\mathbf{Z}$, the field of rationals $\mathbf{Q}$, an algebraic number field $\mathbf{Q}(\alpha)$, a local ring, or a polynomial ring, function field (rational or algebraic) or finite-dimensional affine algebra (which is a field) over any of the above.

For factorization over very small finite fields, the Berlekamp algorithm is used by default, which depends on fast linear algebra (see, for example, [Knu97, section 4.6.2] or [vzGG99, section 14.8]). For medium to large finite fields, the von zur Gathen/Kaltofen/Shoup algorithm ([vzGS92, KS95, Sho95]) is used by default. The parameter `Al` may be used to specify the factorization algorithm over finite fields. The possible values are:

(1) `"Default"`: The default strategy, whereby an appropriate choice will be made.

(2) `"BerlekampSmall"` or `"BerlekampLarge"` for the Berlekamp algorithm (see [Knu97, pp. 446–447] for the difference between these two variants).

(3) `"GKS"` for the von zur Gathen/Kaltofen/Shoup algorithm.

Since V2.8 (July 2001), Magma uses the algorithm of Mark van Hoeij [vH02, vH01] to factor polynomials over the integers or rationals. First a factorization of f is found modulo a suitable small prime, then Hensel lifting is applied, as in the standard Berlekamp-Zassenhaus (BZ) algorithm [Knu97, p. 452]. The Hensel lifting is performed using Victor Shoup's 'tree lifting' algorithm, as described in [vzGG99, Sec. 15.5]. Easy factors are also detected at various stages, if possible, using heuristics developed by Allan Steel. But the final search for the correct combination of modular factors (which has exponential worst-case complexity in the standard BZ algorithm) is now performed by van Hoeij's algorithm, which efficiently finds the correct combinations by solving a Knapsack problem via the LLL lattice-basis reduction algorithm [LLL82].

van Hoeij's new algorithm is *much* more efficient in practice than the original lattice-based factoring algorithm proposed in [LLL82]: the lattice constructed in

van Hoeij's algorithm has dimension equal to the number of modular factors (not the degree of the input polynomial), and the entries of the lattice are very much smaller. Many polynomials can now be easily factored which were out of reach for any previous algorithm (see the examples below).

For polynomials over algebraic number fields, algebraic function fields and affine algebras, the norm-based algorithm of Trager [Tra76] is used, which performs a suitable substitution and resultant computation, and then factors the resulting polynomial with one less variable. In characteristic zero, the difficult case (where there are very many factors of this integral polynomial modulo any prime) is now easily handled by van Hoeij's combination algorithm above. In small characteristic, where inseparable field extensions may occur, an algorithm of Allan Steel ([Ste05]) is used.

`HasPolynomialFactorization(R)`

Given a ring R, return whether factorization of polynomials over R is allowed in MAGMA.

`SetVerbose("PolyFact", v)`

(Procedure.) Change the verbose printing level for all polynomial factorization algorithms to be v. Currently the legal levels are 0, 1, 2 or 3.

`FactorisationToPolynomial(f)`

`Facpol(f)`

Given a sequence of tuples, each consisting of pairs of irreducible polynomials and positive integer exponents, return the product polynomial.

Example H23E6

To demonstrate the power of the van Hoeij combination algorithm, in this example we factor Swinnerton-Dyer polynomials, which are worse-case inputs for the Berlekamp-Zassenhaus factorization algorithm for polynomials over $\mathbf{Z}$.

The n-th Swinnerton-Dyer polynomial is defined to be

$$\prod(x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \cdots \pm \sqrt{p_n}),$$

where p_i is the i-th prime and the product runs over all 2^n possible combinations of $+$ and $-$ signs. This polynomial lies in $\mathbf{Z}[x]$, has degree 2^n, is irreducible over $\mathbf{Z}$, and has at least 2^{n-1} factors modulo any prime. This last fact is easy to see, since, given any finite field K, the polynomial must split into linear factors over a quadratic extension of K, so it will have only linear or quadratic factors over K. See also [vzGG99, section 15.3] for further discussion.

In this example, we use the function `SwinnertonDyerPolynomial` to construct the polynomials (see Example H40E2 in the chapter on algebraically closed fields for an explanation of how this function works).

First we display the first 4 polynomials.

```
> P<x> := PolynomialRing(IntegerRing());
```

```
> SwinnertonDyerPolynomial(1);
x^2 - 2
> SwinnertonDyerPolynomial(2);
x^4 - 10*x^2 + 1
> SwinnertonDyerPolynomial(3);
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
> SwinnertonDyerPolynomial(4);
x^16 - 136*x^14 + 6476*x^12 - 141912*x^10 + 1513334*x^8 - 7453176*x^6 +
    13950764*x^4 - 5596840*x^2 + 46225
> IsIrreducible($1);
true
```

We note the degree patterns of the factorizations of the first eight Swinnerton-Dyer polynomials over the three finite fields $\mathbf{F}_3$, $\mathbf{F}_{23}$ and $\mathbf{F}_{503}$. There are only linear or quadratic factors, as expected.

```
> for i := 1 to 8 do
>    f := SwinnertonDyerPolynomial(i);
>    printf "%o:", i;
>    for p in [3, 23, 503] do
>         L := Factorization(PolynomialRing(GF(p)) ! f);
>         printf " %o", {* Degree(t[1])^^t[2]: t in L *};
>     end for;
>     "";
> end for;
1: {* 2 *} {* 1^^2 *} {* 1^^2 *}
2: {* 2^^2 *} {* 1^^4 *} {* 1^^4 *}
3: {* 1^^4, 2^^2 *} {* 2^^4 *} {* 2^^4 *}
4: {* 1^^8, 2^^4 *} {* 2^^8 *} {* 2^^8 *}
5: {* 1^^8, 2^^12 *} {* 2^^16 *} {* 2^^16 *}
6: {* 1^^16, 2^^24 *} {* 2^^32 *} {* 2^^32 *}
7: {* 1^^48, 2^^40 *} {* 2^^64 *} {* 2^^64 *}
8: {* 1^^96, 2^^80 *} {* 1^^16, 2^^120 *} {* 2^^128 *}
```

We now construct the 6-th polynomial, note its largest coefficient, and then factor it; it takes only a second to prove that it is irreducible, even though there are 32 modular factors.

```
> sd6 := SwinnertonDyerPolynomial(6);
> Degree(sd6);
64
> Max([Abs(x): x in Coefficients(sd6)]);
177108072043062916168515897889215259945 6
> time L := Factorization(sd6);
Time: 1.009
> #L;
1
```

Now we factor the 7-th polynomial!

```
> sd7 := SwinnertonDyerPolynomial(7);
> Degree(sd7);
```

```
128
> Max([Abs(x): x in Coefficients(sd7)]);
85783447140360187781662744163364252674665633803596496806969245 87\
4401145842570683324825619
> time L := Factorization(sd7);
Time: 11.670
> #L;
1
```

We now factor the product of the 6-th and 7-th polynomials. This has degree 192 and has at least 96 factors modulo any prime! But the van Hoeij algorithms easily finds the correct factors over the integers.

```
> p := sd6*sd7;
> Degree(p);
192
> Max([Abs(x): x in Coefficients(p)]);
46178075233031441597519883536198372339486796800578859978206259 79\
48178917111255021010981707011266628489195528524859249200516300 8
31
> time L := Factorization(p);
Time: 16.840
> #L;
2
> L[1,1] eq sd6;
true
> L[2,1] eq sd7;
true
```

See also Example H40E2 in the chapter on algebraically closed fields for a generalization of the Swinnerton-Dyer polynomials.

`SquarefreeFactorization(f)`

Given a univariate polynomial f over the ring R, this function returns the square-free factorization of f as a sequence of pairs, each consisting of a (not necessarily irreducible) factor and an integer indicating the multiplicity. The factors do not contain the square of any non-constant polynomial.

The coefficient ring R must be the integer ring or any field. The algorithm works by computing the GCD of f with its derivative and repeating as necessary (special considerations are also necessary for characteristic p).

`DistinctDegreeFactorization(f)`

`Degree` RngIntElt *Default :* 0

Given a squarefree univariate polynomial $f \in F[x]$ with F a finite field, this function returns the distinct-degree factorization of f as a sequence of pairs, each consisting of a degree d, together with the product of the degree-d irreducible factors of f.

If the optional parameter `Degree` is given a value $L > 0$, then only (products of) factors up to degree L are returned.

`EqualDegreeFactorization(f, d, g)`

Given a squarefree univariate polynomial $f \in F[x]$ with F a finite field, and integer d and another polynomial $g \in F[x]$ such that F is known to be the product of distinct degree-d irreducible polynomials alone, and g is x^q mod f, where q is the cardinality of F, this function returns the irreducible factors of f as a sequence of polynomials (no multiplicities are needed).

If the conditions are not satisfied, the result is unpredictable. This function allows one to split f, assuming that one has computed f in some special way.

`IsIrreducible(f)`

Given a univariate polynomial f over the ring R, this function returns returns `true` if and only f is irreducible over R. The conditions on R are the same as for the function `Factorization` above.

`IsSeparable(f)`

Given a polynomial $f \in K[x]$ such that f is a polynomial of degree ≥ 1 and K is a field allowing polynomial factorization, this function returns `true` iff f is separable.

`QMatrix(f)`

Given a univariate polynomial f of degree d over a finite field F this function returns the Berlekamp Q-matrix associated with f, which is an element of the degree $d-1$ matrix algebra over F.

23.8.2 Resultant and Discriminant

`Discriminant(f)`

The discriminant D of $f \in R[x]$ is returned. The discriminant is an element of R that can be defined by $D = c_n^{2n-2} \prod_{i \neq j}(\alpha_i - \alpha_j)$, where c_n is the leading coefficient of f and the α_i are the zeros of f (in some algebraic closure of R). The coefficient ring R must be a domain.

`Resultant(f, g)`

The resultant of univariate polynomials f and g (of degree m and n) in $R[x]$, which is by definition the determinant of the Sylvester matrix for f and g (a matrix of rank $m+n$ containing coefficients of f and g as entries). The resultant is an element of R. The coefficient ring R must be a domain.

`CompanionMatrix(f)`

Given a monic univariate polynomial f of degree d over some ring R, return the companion matrix of f as an element of the full matrix algebra of degree $d-1$ over R. The companion matrix for $f = a_0 + a_1 x + \cdots + a_{d-1}x^{d-1} + x^d$ is given by

$$\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ -a_0 & -a_1 & -a_2 & \cdots - a_{d-2} & -a_{d-1} \end{pmatrix}.$$

23.8.3 Hensel Lifting

`HenselLift(f, s, P)`

Given the sequence of irreducible factors s modulo some prime p of the univariate integer polynomial f, return the Hensel lifting into the polynomial ring P, which must be the univariate polynomial ring over a residue class ring modulo some power of p. Thus given $f \equiv \prod_i s_i \bmod p$, this returns $f \equiv \prod_i t_i \bmod p^k$ for some $k \geq 1$, as a sequence of polynomials in $\mathbf{Z}/p^k\mathbf{Z}$. The factorization of f modulo p must be squarefree, that is, s should not contain repeated factors.

Example H23E7

```
> R<x> := PolynomialRing(Integers());
> b := x^5 - x^3 + 2*x^2 - 2;
> F<f> := PolynomialRing(GF(5));
> s := [ w[1] : w in Factorization( F ! b ) ];
> s;
[
    f + 1,
    f + 3,
    f + 4,
    f^2 + 2*f + 4
]
> T<t> := PolynomialRing(Integers(5^3));
> h := HenselLift(b, s, T);
> h;
[
    t + 1,
    t + 53,
    t + 124,
    t^2 + 72*t + 59
]
> &*h;
t^5 + 124*t^3 + 2*t^2 + 123
```

23.9 Ideals and Quotient Rings

Currently it is only possible to create ideals and quotient rings in univariate polynomial rings over fields. Note that these are principal ideal domains: all ideals can be generated by a single element.

23.9.1 Creation of Ideals and Quotients

`ideal< R | a1, ..., ar >`

Given a univariate polynomial ring R over a field K, this function returns the ideal of R generated by the elements $a_1, \ldots, a_r \in R$. This is the same as the ideal generated by the greatest common divisor of the elements a_i in R. The function returns the ideal as a subring of R, generated by a single element.

`quo< R | I >`

`quo< R | a1, ..., ar >`

Given an ideal I in the univariate polynomial ring R (over a field), return the quotient R/I, as well as the projection map $h : R \to R/I$. The ideal I may either be specified as an ideal or by a list a_1, a_2, ..., a_r, of generators. The angle bracket notation can be used to assign names to the indeterminates: `Q<q> := quo< I | I >;`.

23.9.2 Ideal Arithmetic

Since ideals of R are regarded as subrings of R, the ring R itself is a valid ideal as well.

`I + J`

Given ideals I and J in the same polynomial ring R, this function returns the sum of the ideals I and J, which is the ideal generated by the generators of I and those of J. Since we require R to be a principal ideal domain, the resulting ideal will be simply generated by the greatest common divisor of `I.1` and `J.1`.

`I * J`

Given ideals I and J in the same polynomial ring R, this function returns the product of the ideals I and J, which is the ideal generated by the products of the generators of I and those of J. Since we require R to be a principal ideal domain, the resulting ideal will be simply generated by `I.1 * J.1`.

`I meet J`

Given ideals I and J in the same polynomial ring R, this function returns the intersection of the ideals I and J. Since we require R to be a principal ideal domain, the resulting ideal will equal the product of I and J and be simply generated by `I.1 * J.1`.

`a in I`

Given an element a of a polynomial ring P as well as an ideal I of P, this function returns `true` if and only if a is contained in I, and `false` otherwise.

`a notin I`

Given an element a of a polynomial ring P as well as an ideal I of P, this function returns `false` if and only if a is contained in I, and `true` otherwise.

`I eq J`

Given two ideals I and J in the same polynomial ring R this returns `true` if and only if I and J are the same, and `false` otherwise.

`I ne J`

Given two ideals I and J in the same polynomial ring R this returns `false` if and only if I and J are the same, and `true` otherwise.

`I subset J`

Given two ideals I and J in the same polynomial ring R this returns `true` if and only if I is contained in J, and `false` otherwise.

`I notsubset J`

Given two ideals I and J in the same polynomial ring R this returns `false` if and only if I is contained in J, and `true` otherwise.

23.9.3 Other Functions on Ideals

Since ideals are considered as subrings of polynomial rings, and in particular are in the same MAGMA category as polynomial rings, most of the function listed in this chapter for polynomial rings do also apply to ideals, but some restrictions apply. Thus it will be possible to get the coefficient ring but it will not be possible to use `ChangeRing` to change it. We list some functions here that additional comments.

`I . 1`

Given an ideal I in a univariate polynomial ring R, return the generator of I in R as an element of I.

23.9.4 Other Functions on Quotients

Contrary to ideals, quotient rings form a separate MAGMA category. Only very few functions are available on these rings; however most element functions for polynomial rings apply to elements of quotients as well, in particular the coefficient, term and degree functions.

`Modulus(Q)`

Given a quotient ring $Q = R[x]/I$ of the univariate polynomial ring $R[x]$ obtained by factoring out by the ideal I, return the generator for I as an element of R.

`PreimageRing(Q)`

If Q is the quotient $Q = R/I$ for some univariate polynomial ring R, this function returns R.

23.10 Special Families of Polynomials

23.10.1 Orthogonal Polynomials

`ChebyshevFirst(n)`

`ChebyshevT(n)`

Given a positive integer n, this function constructs the Chebyshev polynomial of the first kind $T_n(x)$, where $T_n(x)$ is defined by $T_n(x) = \cos n\theta$ with $x = \cos\theta$.

`ChebyshevSecond(n)`

`ChebyshevU(n)`

Given a positive integer n, this function constructs the Chebyshev polynomial of the second kind, $U_n(x)$, of degree $n-1$. The polynomial is defined by

$$U_n(x) = \frac{1}{n}T_n'(x) = \frac{\sin n\theta}{\sin\theta}$$

where $x = \cos\theta$.

`LegendrePolynomial(n)`

Given a positive integer n, this function constructs the Legendre polynomial $P_n(x)$ of degree n, where $P_n(x)$ is defined by

$$P_0(x) = 1,\ P_1(x) = x,$$
$$P_n(x) = \frac{1}{n}((2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)).$$

`LaguerrePolynomial(n)`

`LaguerrePolynomial(n, m)`

Given a positive integer n, this function constructs the Laguerre polynomial $L_n^m(x)$ of degree n with parameter m. If m is omitted, it is assumed to be zero if it is not specified. The polynomial satisfies the recurrence relation

$$L_0(x) = 1,\ L_1(x) = 1 + m - x,$$
$$L_n(x) = \frac{1}{n}(((2n + m - 1) - x)L_{n-1}^m(x) - (n - 1 + m)L_{n-2}^m(x)).$$

`HermitePolynomial(n)`

Given a positive integer n, this function constructs the Hermite polynomial $H_n(x)$ of degree n, where $H_n(x)$ is defined by

$$H_0(x) = 1,\ H_1(x) = 2x,$$
$$H_n(x) = 2xH_{n-1}(x) - 2nH_{n-2}(x).$$

`GegenbauerPolynomial(n, m)`

Given a positive integer n and an integer m, this function constructs the Gegenbauer polynomial $C_n^m(x)$ of degree n with parameter m, where $C_n^m(x)$ is defined by

$$C_0^m(x) = 1,\ C_1^m(x) = 2mx,$$
$$C_n^m(x) = \frac{1}{n}(2(n - 1 + m)xC_{n-1}^m(x) - (n + 2m - 2)C_{n-2}^m(x)).$$

23.10.2 Permutation Polynomials

`DicksonFirst(n, a)`

Given a positive integer n, this function constructs the Dickson polynomial of the first kind $D_n(x, a)$ of degree n, where $D_n(x, a)$ is defined by

$$D_n(x, a) = \sum_{i=0}^{\lfloor n/2 \rfloor} \frac{n}{n - i}\binom{n - i}{i}(-a)^i x^{n-2i}.$$

`DicksonSecond(n, a)`

Given a positive integer n, this function constructs the Dickson polynomial of the second kind $E_n(x, a)$ of degree n, where $E_n(x, a)$ is defined by

$$E_n(x, a) = \sum_{i=0}^{\lfloor n/2 \rfloor} \binom{n - i}{i}(-a)^i x^{n-2i}.$$

23.10.3 The Bernoulli Polynomial

`BernoulliPolynomial(n)`

Given a positive integer n, this function constructs the n-th Bernoulli polynomial.

23.10.4 Swinnerton-Dyer Polynomials

`SwinnertonDyerPolynomial(n)`

Given a positive integer n, this function constructs the n-th Swinnerton-Dyer polynomial, which is defined to be

$$\prod(x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \cdots \pm \sqrt{p_n}),$$

where p_i is the i-th prime and the product runs over all 2^n possible combinations of $+$ and $-$ signs. This polynomial lies in $\mathbf{Z}[x]$, has degree 2^n, and is irreducible over $\mathbf{Z}$.

See Example H23E6 above which explains more about this class of polynomials, and see also Example H40E2 in the chapter on algebraically closed fields to see how these polynomials are constructed and also for a generalization.

23.11 Bibliography

[**CGG89**] Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation. *J. Symbolic Comp.*, 7(1):31–48, 1989.

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**Cop96**] Don Coppersmith. Finding a small root of a univariate modular equation. In *Advances in Cryptology—EuroCrypt 1996*, volume 1070 of *LNCS*, pages 155–165. Springer, 1996.

[**GCL92**] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer, Boston/Dordrecht/London, 1992.

[**Knu97**] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, Massachusetts, 3rd edition, 1997.

[**KS95**] Erich Kaltofen and Victor Shoup. Subquadratic-time factoring of polynomials over finite fields. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, pages 398–406. ACM, 1995.

[**LLL82**] Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[**May03**] Alexander May. *New RSA Vulnerabilities Using Lattice Reduction Methods*. Dissertation, University of Paderborn, 2003.

[**PZ89**] Michael E. Pohst and Hans Zassenhaus. *Algorithmic Algebraic Number Theory*. Encyclopaedia of mathematics and its applications. Cambridge University Press, Cambridge, 1989.

[**Sho95**] Victor Shoup. A New Polynomial Factorization Algorithm and its Implementation. *J. Symbolic Comp.*, 20(4):363–397, 1995.

[**Ste05**] Allan Steel. Conquering Inseparability: Primary Decomposition and Multivariate Factorization over Algebraic Function Fields of Positive Characteristic. *J. Symbolic Comp.*, 40(3):1053–1075, 2005.

[**Tra76**] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, pages 196–208. ACM press, 1976.

[**vH01**] Mark van Hoeij. Factoring Polynomials and 0-1 vectors. In *Proceedings of the Cryptography and Lattices Conference (CaLC 2001), Brown University, Providence, RI, USA, March 29-30, 2001*, pages 142–146. Springer, 2001.

[**vH02**] Mark van Hoeij. Factoring Polynomials and the knapsack problem. *J. Number Th.*, 95(2):167–189, 2002.
URL:http://www.math.fsu.edu/∼hoeij/paper/knapsack.ps.

[**vzGG99**] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, 1999.

[**vzGS92**] Joachim von zur Gathen and Victor Shoup. Computing Frobenius Maps And Factoring Polynomials. *Computational Complexity*, 2:187–224, 1992.

24 MULTIVARIATE POLYNOMIAL RINGS

Chapter 24
MULTIVARIATE POLYNOMIAL RINGS

24.1 Introduction

This chapter describes multivariate polynomial rings in MAGMA. A multivariate polynomial ring in any number of variables $n \geq 1$ can be created over an arbitrary coefficient ring R, and we will denote it by $P = R[x_1, \ldots, x_n]$. Certain functions, however, will only apply for coefficient rings satisfying certain conditions.

MAGMA contains a powerful system for computing with ideals of multivariate polynomial rings. This is based on the construction of Gröbner bases of such ideals. This chapter only deals with polynomial rings and operations on their elements; see Chapter 105 for the details concerning ideals and Gröbner bases.

Permutation and matrix groups have a natural action on multivariate polynomial rings. This leads to the subject of invariant rings of finite groups, which is covered in Chapter 110. See also the chapters on affine algebras (Chapter 108) and on modules over affine algebras (Chapter 109).

24.1.1 Representation

Let P be the polynomial ring $R[x_1, \ldots, x_n]$ of rank n over a ring R. A *monomial* (or *power product*) of P is a product of powers of the variables of P, that is, an expression of the form $x_1^{e_1} \cdots x_n^{e_n}$ with $e_i \geq 0$ for $1 \leq i \leq n$. Multivariate polynomials in MAGMA are stored efficiently in distributive form, using arrays of coefficient-monomial pairs, where the coefficient is in the base ring R. The word 'term' will always refer to a coefficient multiplied by a monomial.

Various orders can be applied to the monomials, and these are of great importance when dealing with Gröbner bases. A polynomial ring in MAGMA may defined with a certain monomial order, but as this does not affect the basic arithmetic operations in the polynomial ring, these orders are not described here but in the chapter dealing with Gröbner bases (see Section 105.2).

Since V2.7 (June 2000), a new generalized monomial representation has been developed, which uses differing byte sizes for monomials depending on the size of the monomials encountered. Monomial overflow is rigorously detected, and the system automatically extends the byte size of the monomials in the background if possible. Thus there is no need for the user to know beforehand the maximum degree which may occur, and much memory is also saved for low- to medium-degree computations. The total degree of any monomial may now be anything up to $2^{30} - 1 = 1073741823$.

It is possible but not advised to use distributive 'multivariate' polynomials in one single variable. (See Chapter 23 which devoted to univariate polynomial rings.)

24.2 Polynomial Rings and Polynomials

24.2.1 Creation of Polynomial Rings

Multivariate polynomial rings are created from a coefficient ring, the number of indeterminates, and a monomial order. If no order is specified, the monomial order is taken to be the lexicographical order (see Section 105.2 for details).

`PolynomialRing(R, n)`

`PolynomialAlgebra(R, n)`

`Global` BoolElt *Default :* `false`

Create a multivariate polynomial ring in $n > 0$ indeterminates over the ring R. The ring is regarded as an R-algebra via the usual identification of elements of R and the constant polynomials. The lexicographical ordering on the monomials is used for this default construction (see Section 105.2). The angle bracket notation can be used to assign names to the indeterminates; e.g.: `P<x, y> := PolynomialRing(R, 2);` etc.

By default, a *non-global* polynomial ring will be returned; if the parameter `Global` is set to `true`, then the unique global polynomial ring over R with n variables will be returned. This may be useful in some contexts, but a non-global result is returned by default since one often wishes to have several rings with the same numbers of variables but with different variable names (and create mappings between them, for example). Explicit coercion is always allowed between polynomial rings having the same number of variables (and suitable base rings), whether they are global or not, and the coercion maps the i-variable of one ring to the i-th variable of the other ring.

`PolynomialRing(R, n, order)`

`PolynomialAlgebra(R, n, order)`

Create a multivariate polynomial ring in $n > 0$ indeterminates over the ring R with the given order *order* on the monomials. See Section 105.2 for details.

Example H24E1

We show the use of angle brackets for generator names.

```
> Z := IntegerRing();
> S := PolynomialRing(Z, 2);
```

If we define S this way, we can only refer to the indeterminates by $S.1$ and $S.2$ (see below). So we could assign these generators to variables, say x and y, as follows:

```
> x := S.1;
> y := S.2;
```

In this case it is easy to construct polynomials, but printing is slightly awkward:

```
> f := x^3*y +3*y^2;
```

```
> f;
$.1^3*$.2 + 3*$.2^2
```

To overcome that, it is possible to assign names to the indeterminates that are used in the printing routines, using the `AssignNames` function, before assigning to x and y.

```
> AssignNames(~S, ["x", "y"]);
> x := S.1; y := S.2;
> f := x^3*y +3*y^2;
> f;
x^3*y + 3*y^2
```

Alternatively, we use the angle brackets to assign generator names that will be used in printing as well:

```
> S<x, y> := PolynomialRing(Z, 2);
> f := x^3*y +3*y^2;
> f;
x^3*y + 3*y^2
```

Example H24E2

We demonstrate the difference between global and non-global rings. We first create the global multivariate polynomial ring over $\mathbf{Q}$ with 3 variables twice.

```
> Q := RationalField();
> P<x,y,z> := PolynomialRing(Q, 3: Global);
> PP := PolynomialRing(Q, 3: Global);
> P;
Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
> PP;
Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
> PP.1;
x
```

PP is identical to P. We now create default (non-global) multivariate polynomial rings (which are also different to the global polynomial ring P). Explicit coercion is allowed, and maps the i-variable of one ring to the i-th variable of the other ring.

```
> P1<a,b,c> := PolynomialRing(Q, 3);
> P2<d,e,f> := PolynomialRing(Q, 3);
> P1;
Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: a, b, c
> P2;
Polynomial ring of rank 3 over Rational Field
```

```
Lexicographical Order
Variables: d, e, f
> a;
a
> d;
d
> P1 ! d;
a
> P ! e;
y
```

24.2.2 Print Names

The `AssignNames` and `Name` functions can be used to associate names with the indeterminates of polynomial rings after creation.

`AssignNames(~P, s)`

Procedure to change the name of the indeterminates of a polynomial ring P. The i-th indeterminate will be given the name of the i-th element of the sequence of strings s (for $1 \leq i \leq \#s$); the sequence may have length less than the number of indeterminates of P, in which case the remaining indeterminate names remain unchanged.

This procedure only changes the name used in printing the elements of P. It does *not* assign to identifiers corresponding to the strings the indeterminates in P; to do this, use an assignment statement, or use angle brackets when creating the polynomial ring.

Note that since this is a procedure that modifies P, it is necessary to have a reference `~P` to P in the call to this function.

`Name(P, i)`

Given a polynomial ring P, return the i-th indeterminate of P (as an element of P).

24.2.3 Graded Polynomial Rings

It is possible within MAGMA to assign weights to the variables of a multivariate polynomial ring. This means that monomials of the ring then have a *weighted degree* with respect to the weights of the variables. Such a multivariate polynomial ring is called *graded* or *weighted*. Since this subject is intimately related to ideals, it is covered in the chapter on ideals and Gröbner bases (see Subsection 105.3.2).

24.2.4 Creation of Polynomials

The easiest way to create polynomials in a given ring is to use the angle bracket construction to attach variables to the indeterminates, and then to use these variables to create polynomials (see the examples). Below we list other options.

```
P . i
```

Return the i-th indeterminate for the polynomial ring P in n variables ($1 \leq i \leq n$) as an element of P.

```
elt< R | a >
```

```
R ! s
```

```
elt< R | s >
```

This element constructor can only be used for trivial purposes in multivariate polynomial rings: given a polynomial ring $P = R[x_1, \ldots, x_n]$ and an element a that can be coerced into the coefficient ring R, the constant polynomial a is returned; if a is in P already it will be returned unchanged.

```
MultivariatePolynomial(P, f, i)
```

```
MultivariatePolynomial(P, f, v)
```

Given a multivariate polynomial ring $P = R[x_1, \ldots, x_n]$, as well as a polynomial f in a univariate polynomial ring $R[x]$ over the same coefficient ring R, return an element q of P corresponding to f in the indeterminate $v = x_i$; that is, $q \in P$ is defined by $q = \sum_j f_j x_i^j$ where $f = \sum_j f_j x^j$. The indeterminate x_i can either be specified as a polynomial $v = x_i$ in P, or by simply providing the integer i with $1 \leq i \leq n$.

The inverse operation is performed by the `UnivariatePolynomial` function.

```
One(P)
```

```
Identity(P)
```

```
Zero(P)
```

```
Representative(P)
```

24.3 Structure Operations

24.3.1 Related Structures

The main structure related to a polynomial ring is its coefficient ring. Multivariate polynomial rings belong to the MAGMA category `RngMPol`.

```
BaseRing(P)
```

```
CoefficientRing(P)
```

Return the coefficient ring of polynomial ring P.

```
Category(P)
```

```
Parent(P)
```

```
PrimeRing(P)
```

24.3.2 Numerical Invariants

Note that the # operator only returns a value for finite (quotients of) polynomial rings.

```
Rank(P)
```

Return the number of indeterminates of polynomial ring P over its coefficient ring.

```
Characteristic(P)
```
```
# P
```

24.3.3 Ring Predicates and Booleans

The usual ring functions returning Boolean values are available on polynomial rings.

```
IsCommutative(P)
```
```
IsUnitary(P)
```
```
IsFinite(P)
```
```
IsOrdered(P)
```
```
IsField(P)
```
```
IsEuclideanDomain(P)
```
```
IsPID(P)
```
```
IsUFD(P)
```
```
IsDivisionRing(P)
```
```
IsEuclideanRing(P)
```
```
IsDomain(P)
```
```
IsPrincipalIdealRing(P)
```
```
P eq Q
```
```
P ne Q
```

24.3.4 Changing Coefficient Ring

The ChangeRing function enables the changing of the coefficient ring of a polynomial ring.

```
ChangeRing(P, S)
```

Given a polynomial ring $P = R[x_1, \ldots, x_n]$ of rank n with coefficient ring R, together with a ring S, construct the polynomial ring $Q = S[x_1, \ldots, x_n]$. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S.

24.3.5 Homomorphisms

In its general form, a ring homomorphism taking a polynomial ring $R[x_1, \ldots, x_n]$ as domain requires $n + 1$ pieces of information, namely, a map (homomorphism) telling how to map the coefficient ring R together with the images of the n indeterminates.

```
hom< P -> S | f, y1, ..., yn >
```
```
hom< P -> S | y1, ..., yn >
```

Given a polynomial ring $P = R[x_1, \ldots, x_n]$, a ring S, a map $f : R \to S$ and n elements $y_1, \ldots, y_n \in S$, create the homomorphism $g : P \to S$ by applying the rules that $g(rx_1^{a_1} \cdots x_n^{a_n}) = f(r)y_1^{a_1} \cdots y_n^{a_n}$ for monomials and linearity, that is, $g(M + N) = g(M) + g(N)$.

The coefficient ring map may be omitted, in which case the coefficients are mapped into S by the unitary homomorphism sending 1_R to 1_S. Also, the images y_i are allowed to be from a structure that allows automatic coercion into S.

Example H24E3

In this example we map $\mathbf{Q}[x, y]$ into the number field $\mathbf{Q}(\sqrt[3]{2}, \sqrt{5})$ by sending x to $\sqrt[3]{2}$ and y to $\sqrt{5}$ and the identity map on the coefficients (which we omit).

```
> Q := RationalField();
> R<x, y> := PolynomialRing(Q, 2);
> A<a> := PolynomialRing(IntegerRing());
> N<z, w> := NumberField([a^3-2, a^2+5]);
> h := hom< R -> N | z, w >;
> h(x^11*y^3-x+4/5*y-13/4);
-40*w*z^2 - z + 4/5*w - 13/4
```

24.4 Element Operations

24.4.1 Arithmetic Operators

The usual unary and binary ring operations are available for multivariate polynomials.

For polynomial rings over fields division by elements of the coefficient field are allowed (with the result in the original polynomial ring). The operator `div` has slightly different semantics from the univariate case: if b divides a, that is, if there exists a polynomial $q \in P$ such that $a = b \cdot q \in P$ then q will be the result of `a div b`, but if such polynomial does not exist an error results.

`+ a` `- a`

`a + b` `a - b` `a * b` `a ^ k` `a / b` `a div b`

`a +:= b` `a -:= b` `a *:= b` `a div:= b`

24.4.2 Equality and Membership

`a eq b` `a ne b`

`a in R` `a notin R`

24.4.3 Predicates on Ring Elements

The list belows contains the general ring element predicates. Also, the `IsDivisibleBy` function allows a divisibility test, and the `IsAlgebraicallyDependent` function determines if a set of ring elements is algebraically dependent. Note that not all functions are available for every coefficient ring.

`IsDivisibleBy(a, b)`

Given elements a, b in a multivariate polynomial ring P, this function returns whether the polynomial a is divisible by b in P, that is, if and only if there exists $q \in P$ such that $a = q \cdot b$. If `true` is returned, the quotient polynomial q is also returned.

`IsAlgebraicallyDependent(S)`

Returns `true` iff the set S of multivariate polynomials is algebraically dependent.

`IsZero(f)` `IsOne(f)` `IsMinusOne(f)`

`IsNilpotent(f)` `IsIdempotent(f)`

`IsUnit(f)` `IsZeroDivisor(f)` `IsRegular(f)`

`IsIrreducible(f)` `IsPrime(f)`

24.4.4 Coefficients, Monomials and Terms

Many of the functions in this subsection come in three different forms: one in which no variable is specified, which usually returns values in the coefficient ring, and two in which a particular variable is referred, either by name or by number, and these usually return values in the polynomial ring itself.

`Coefficients(f)`

Given a multivariate polynomial f with coefficients in R, this function returns a sequence of 'base' coefficients, that is, a sequence of elements of R occurring as coefficients of the monomials in f. Note that the monomials are ordered, and that the sequence of coefficients corresponds exactly to the sequence of monomials returned by `Monomials(f)`.

`Coefficients(f, i)`

`Coefficients(f, v)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns a sequence of coefficients with respect to a given variable $v = x_i$, that is, the function returns a sequence of elements of P that form the coefficients of the powers of v (in ascending order) when f is regarded as a polynomial $\sum_j c_j x_i^j$; note that the variable x_i itself will not occur in the coefficients. There are two ways to indicate with respect to which variable the coefficients are to be taken: either one specifies

i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Coefficient(f, i, k)`

`Coefficient(f, v, k)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the coefficient of $v^k = x_i^k$, that is, the function returns the element of P that forms the coefficient of the k-th power of x_i, when f is regarded as a polynomial $\sum_j c_j x_i^j$; note that the variable x_i itself will not occur in the coefficient. There are two ways to indicate with respect to which variable the coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`LeadingCoefficient(f)`

Given a multivariate polynomial f with coefficients in R, this function returns the leading coefficient of f as an element of R; this is the coefficient of the leading monomial of f, that is, the first among the monomials occurring in f with respect to the ordering of monomials used in P.

`LeadingCoefficient(f, i)`

`LeadingCoefficient(f, v)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the element of P that forms the coefficient of the largest power of $v = x_i$ that occurs with non-zero coefficient in f, when f is regarded as a polynomial $\sum_j c_j x_i^j$; note that the variable x_i itself will not occur in the coefficient. There are two ways to indicate with respect to which variable the leading coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Length(f)`

Given a multivariate polynomial f, return the length of f, i.e., the number of terms of f.

`TrailingCoefficient(f)`

Given a multivariate polynomial f with coefficients in R, this function returns the trailing coefficient of f as an element of R; this is the coefficient of the trailing monomial of f, that is, the last among the monomials occurring in f with respect to the ordering of monomials used in P.

`TrailingCoefficient(f, i)`

`TrailingCoefficient(f, v)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the element of P that forms the coefficient of the least power of $v = x_i$ that occurs with non-zero coefficient in f, when f is regarded as a polynomial $\sum_j c_j x_i^j$; note that the variable x_i itself will not occur in the coefficient. There are two ways to indicate with respect to which variable the leading coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`MonomialCoefficient(f, m)`

Given a multivariate polynomial f and a monomial m, both in $P \in R[x_1, \ldots, x_n]$, this function returns the coefficient with which m occurs in f as an element of R.

`Monomials(f)`

Given a multivariate polynomial $f \in P$, this function returns a sequence of monomials, that is, a sequence of monomial elements of P occurring in f. Note that the monomials in P are ordered, and that the sequence of monomials corresponds exactly to the sequence of coefficients returned by `Coefficients(f)`.

`CoefficientsAndMonomials(f)`

Given a multivariate polynomial $f \in P$, this function returns parallel sequences C and M of the coefficients and monomials, respectively, of f. Thus this function is equivalent to calling `Coefficients` and `Monomials` separately, but is more efficient (particularly for large polynomials) since only one scan of the polynomial needs to be done.

`LeadingMonomial(f)`

Given a multivariate polynomial $f \in P$ this function returns the leading monomial of f, that is, the first monomial element of P that occurs in f, with respect to the ordering of monomials used in P.

`Terms(f)`

Given a multivariate polynomial $f \in P$, this function returns the sequence of (non-zero) terms of f as elements of P. The terms are ordered according to the ordering on the monomials in P. Consequently the i-th element of this sequence of terms will be equal to the product of the i-th element of the sequence of coefficients and the i-th element of the sequence of monomials.

`Terms(f, i)`

`Terms(f, v)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns a sequence of terms with respect to a given variable $v = x_i$, that is, the function returns a sequence of elements of P that form the terms (ascending order) of f regarded as a polynomial $\sum_j c_j x_i^j$. There are two ways to indicate with respect to which variable the terms are to be ordered: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Term(f, i, k)`

`Term(f, v, k)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the k-th term of f (with $k \geq 0$), that is, the function returns the term of f involving the k-th power of x_i, when f is regarded as a polynomial $\sum_j c_j x_i^j$. There are two ways to indicate with respect to which variable the term is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`LeadingTerm(f)`

Given a multivariate polynomial $f \in P$, this function returns the leading term of f as an element of P; this is the product of the leading monomial and the leading coefficient that is, the first among the monomial terms occurring in f with respect to the ordering of monomials used in P.

`LeadingTerm(f, i)`

`LeadingTerm(f, v)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the element of P that forms the leading term of f when f is regarded as a polynomial $\sum_j c_j x_i^j$. Thus it is the term involving the largest power of x_i that occurs with non-zero coefficient. There are two ways to indicate with respect to which variable the leading coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`TrailingTerm(f)`

Given a multivariate polynomial $f \in P$, this function returns the trailing term of f as an element of P; this is the last among the monomial terms occurring in f with respect to the ordering of monomials used in P.

`TrailingTerm(f, i)`

`TrailingTerm(f, v)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the element of P that forms the trailing term of f when f is regarded as a polynomial $\sum_j c_j x_i^j$. Thus it is the term involving the least power of x_i that occurs with non-zero coefficient. There are two ways to indicate with respect to which variable the leading coefficient is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Exponents(f)`

Given a single term f (a polynomial having exactly one term) in a polynomial ring of rank n, return the exponents of the monomial of f, as a sequence of length n of integers. (The coefficient of f is ignored; it need not be 1.)

`Monomial(P, E)`

Given a multivariate polynomial ring $P = R[x_1, \ldots, x_n]$, and a sequence E of non-negative integers, return the monomial ${x_1}^{E[1]} \ldots {x_n}^{E[n]}$ in P. This function is a semi-inverse of `Exponents`.

`Polynomial(C, M)`

Given a length-k sequence C of coefficients in a ring R and a length-k sequence M of monomials of a polynomial ring R, return the multivariate polynomial $f \in R$ whose coefficients are C and monomials are M. (Thus for any $f \in R$, `Polynomial(Coefficients(f), Monomials))` equals f.)

Example H24E4

In this and the next example we illustrate the coefficient and term functions, using the polynomial in three variables x, y, z over the rational field that is given by $f = (2x + y)z^3 + 11xyz + x^2y^2$.

```
> R<x, y, z> := PolynomialAlgebra(RationalField(), 3);
> f := (2*x+y)*z^3+11*x*y*z+x^2*y^2;
> f;
x^2*y^2 + 11*x*y*z + 2*x*z^3 + y*z^3
> Coefficients(f);
[ 1, 11, 2, 1 ]
> Monomials(f);
[
   x^2*y^2,
   x*y*z,
   x*z^3,
   y*z^3
]
> CoefficientsAndMonomials(f);
[ 1, 11, 2, 1 ]
```

```
[
    x^2*y^2,
    x*y*z,
    x*z^3,
    y*z^3
]
> Terms(f);
[
   x^2*y^2,
   11*x*y*z,
   2*x*z^3,
   y*z^3
]
> Coefficients(f, y);
[
   2*x*z^3,
   11*x*z + z^3,
   x^2
]
> Terms(f, 2);
[
   2*x*z^3,
   11*x*y*z + y*z^3,
   x^2*y^2
]
> MonomialCoefficient(f, x*y*z);
11
> LeadingTerm(f);
x^2*y^2
> LeadingTerm(f, z);
2*x*z^3 + y*z^3
> LeadingCoefficient(f, z);
2*x + y
> Polynomial([1, 2, 3], [x*y, y, z^2]);
x*y + 2*y + 3*z^2
```

24.4.5 Degrees

`Degree(f, i)`

`Degree(f, v)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the degree of f in $v^k = x_i^k$, that is, the function returns the degree of f when it is regarded as a polynomial $\sum_j c_j x_i^j$. The resulting integer is thus the largest power of x_i occurring in any monomial of f. There are two ways to indicate with respect to

which variable the degree is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P). If f is the zero polynomial, the return value is always -1.

`TotalDegree(f)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the total degree of f, which is the maximum of the total degrees of all monomials that occur in f. The total degree of a monomial m is the sum of the exponents of the indeterminates that make up m. Note that this ignores the weights on the variables if there are any (see the section on graded polynomial rings below). If f is the zero polynomial, the return value is -1.

`LeadingTotalDegree(f)`

Given a multivariate polynomial $f \in P = R[x_1, \ldots, x_n]$, this function returns the leading total degree of f, which is the total degree of the leading monomial of f. If f is the zero polynomial, the return value is -1.

24.4.6 Univariate Polynomials

`IsUnivariate(f)`

Given a multivariate polynomial $f \in R[x_1, \ldots, x_n]$, this function returns whether f is in fact a univariate polynomial in one of its indeterminates $x_1, \ldots, x_n$. If true is returned, then the function also returns a univariate version u of f and (the first) i such that f is univariate in x_i. Note that there will only be ambiguity about i if f is a constant polynomial. The univariate polynomial u will be an element of $R[x]$ with the same coefficients as f.

`IsUnivariate(f, i)`
`IsUnivariate(f, v)`

Given a multivariate polynomial $f \in R[x_1, \ldots, x_n]$, this function returns whether f is in fact a univariate polynomial in x_i. If true is returned, then the function also returns a univariate version u of f, which will be an element of the univariate polynomial ring $R[x]$ with the same coefficients as f. The indeterminate x_i should either be specified as a (polynomial) argument v or as an integer i.

`UnivariatePolynomial(f)`

Given a multivariate polynomial $f \in R[x_1, \ldots, x_n]$, which is known to be a univariate polynomial in x_i for some i with $1 \leq i \leq n$, return a univariate version u of f, which will be an element of the univariate polynomial ring $R[x]$ with the same coefficients as f.

Example H24E5

Suppose we have two bivariate polynomials f and g over some ring.

```
> P<x,y> := PolynomialRing(GF(5), 2);
> f := x^2 - y + 3;
> g := y^3 - x*y + x;
```

If we compute the resultant in either variable of the two polynomials, then we can apply `UnivariatePolynomial` to this to obtain a univariate version of it, from which we can compute the roots.

```
> ry := Resultant(f, g, y);
> ry;
4*x^6 + x^4 + x^3 + 3*x^2 + 2*x + 3
> Roots(UnivariatePolynomial(ry));
[ <3, 1> ]
> Evaluate(f, x, 3);
4*y + 2
> Evaluate(g, x, 3);
y^3 + 2*y + 3
> GCD($1, $2);
y + 3
> rx := Resultant(f, g, x);
> rx;
y^6 + 4*y^3 + 3*y + 3
> Roots(UnivariatePolynomial(rx));
[ <2, 1> ]
> Evaluate(f, y, 2);
x^2 + 1
> Evaluate(g, y, 2);
4*x + 3
> GCD($1, $2);
x + 2
```

24.4.7 Derivative, Integral

`Derivative(f, i)`

`Derivative(f, v)`

Given a multivariate polynomial $f \in P$, return the derivative of f with respect to the variable $v = x_i$, as an element of P. There are two ways to indicate with respect to which variable the derivative is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Derivative(f, k, i)`

`Derivative(f, k, v)`

Given a multivariate polynomial $f \in P$ and an integer $k > 0$, return the k-th derivative of f with respect to the variable $v = x_i$, as an element of P. There are two ways to indicate with respect to which variable the derivative is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`Integral(f, i)`

`Integral(f, v)`

Given a multivariate polynomial $f \in P$ over a field of characteristic zero, return the formal integral of f with respect to $v = x_i$ as an element of P. There are two ways to indicate with respect to which variable the integral is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

`JacobianMatrix( [ f ] )`

Creates the matrix with (i, j)'th entry the partial derivative of the i'th polynomial in the list with the j'th indeterminate of its parent ring.

24.4.8 Evaluation, Interpolation

`Evaluate(f, s)`

Given an element f of a polynomial ring $P = R[x_1, \ldots, x_n]$ and a sequence or tuple s of ring elements of length n, return the value of f at s, that is, obtained by substituting $x_i = s[i]$. If the elements of s can be lifted into the coefficient ring R, then the result will be an element of R. If the elements of s cannot be lifted to the coefficient ring, then an attempt is made to do a generic evaluation of f at s. In this case, the result will be of the same type as the elements of s.

`Evaluate(f, i, r)`

`Evaluate(f, v, r)`

Given an element f of a multivariate polynomial ring P and a ring element r return the value of f when the variable $v = x_i$ is evaluated at r. If r can be coerced into the coefficient ring of P, the result will be an element in P again. Otherwise the other variables of P must be coercible into the parent of r, and the result will have the same parent as r.

`Interpolation(I, V, i)`

`Interpolation(I, V, v)`

Let K be a field, and $P = K[x_1, \ldots, x_n]$ a multivariate polynomial ring over K; let $v = x_i$ be the i-th indeterminate of P. Given a sequence I of elements of K (the interpolation points) and a sequence V of elements of P (the interpolation values), both sequences of length $k > 0$, return the unique polynomial $f \in P$ of degree less than k in the variable x_i such that $f(I[j]) = V[j]$, for $j = 1, \ldots, k$. The variable x_i may not occur anywhere in the values V. There are two ways to indicate with respect to which variable to interpolate: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable or the variable v itself (as a polynomial).

Example H24E6

We define $P = \mathbf{Q}[x, y, z]$, and give an example of interpolation. We find a polynomial which, when evaluated in the first variable x in the rational points $1, 2, 3$, yields $y, z, y + z$ respectively. We check the result by evaluating.

```
> Q := RationalField();
> P<x, y, z> := PolynomialRing(Q, 3);
> f := Interpolation([Q | 1, 2, 3], [y, z, y + z], 1);
> f;
x^2*y - 1/2*x^2*z - 4*x*y + 5/2*x*z + 4*y - 2*z
> [ Evaluate(f, 1, v) : v in [1, 2, 3] ];
[
   y,
   z,
   y + z
]
```

24.4.9 Quotient and Reductum

`f div g`

`ExactQuotient(f, g)`

The quotient of the multivariate polynomial f by g in $R[x_1, \ldots, x_n]$, provided the result lies in P again. Here R must be a domain. If a polynomial q in P exists such that $f = q \cdot g$ then it will be returned, but if does not exist an error results.

`Reductum(f)`

The reductum of a polynomial f, which is the polynomial obtained by removing the leading term of f.

`Reductum(f, i)`

`Reductum(f, v)`

The reductum of a multivariate polynomial $f \in R[x_1, \ldots, x_n]$ obtained by removing the leading term with respect to the variable $v = x_i$. Here either v must be specified as a polynomial, or x_i must be specified by providing the integer i, with $1 \leq i \leq n$.

24.4.10 Diagonalizing a Polynomial of Degree 2

We provide two basic tools that deal with polynomial diagonalization.

`SymmetricBilinearForm(f)`

The symmetric bilinear form (as a matrix) of a multivariate polynomial of degree 2.

`DiagonalForm(f)`

The diagonal form of the multivariate polynomial of degree 2. Also returns the transformation matrix.

Example H24E7

```
> Q := RationalField();
> PR<x, y, z> := PolynomialRing(Q, 3);
> g := 119/44*x^2 - 93759/41440*x*y + 390935/91427*x*z
>      + 212/243*x - 3/17*y^2 + 52808/172227*y*z
>      - 287/227*y + 537/934*z^2 - 127/422*z;
> SymmetricBilinearForm(g);
[       119/44  -93759/82880 390935/182854       106/243]
[ -93759/82880         -3/17  26404/172227      -287/454]
[390935/182854  26404/172227       537/934      -127/844]
[      106/243      -287/454      -127/844             0]
> DiagonalForm(g);
119/44*x^2 - 15798558582429/4*y^2 +
34932799628335074761085292707227419544217/934*z^2 -
17658873286101893452437121055688364561927521739811614723483771045740414637 1/2
>
> bl := SymmetricBilinearForm(g);
> NBL := Matrix(PR, bl);
> D, T := OrthogonalizeGram(bl);
> NT := Matrix(PR, T);
> C := Matrix(PR, [[x,y,z,1]]);
> NC := C * NT;
> NCT := Transpose(NC);
> (NC * NBL * NCT)[1][1] eq DiagonalForm(g);
true
```

The last few statements demonstrate how the polynomial's diagonal form is obtained from its symmetric bilinear form. Note also that since the polynomial g is not homogeneous its symmetric bilinear form is given on four variables, the fourth variable being a homogenizing variable.

24.5 Greatest Common Divisors

The functions in this section can be applied to multivariate polynomials over any ring which has a GCD algorithm.

24.5.1 Common Divisors and Common Multiples

`GreatestCommonDivisor(f, g)`

`Gcd(f, g)`

`GCD(f, g)`

The greatest common divisor of f and g in a multivariate polynomial ring P. If either of the inputs is zero, then the result is the other input (and if the inputs are both zero then the result is zero). The result is normalized (see the function `Normalize`), so the result is always unique.

The valid coefficient rings are those which themselves have a GCD algorithm for their elements (which includes most commutative rings in MAGMA).

For polynomials over the integers or rationals, a combination of three algorithms is used: (1) the heuristic evaluation 'GCDHEU' algorithm of Char et al. ([CGG89] and [GCL92, section 7.7]), suitable for moderate-degree dense polynomials with several variables; (2) the EEZ-GCD algorithm of Wang ([Wan80, MY73] and [GCL92, section 7.6]), based on evaluation and sparse ideal-adic multivariate Hensel lifting ([Wan78] and [GCL92, section 6.8]), suitable for sparse polynomials; (3) a recursive multivariate evaluation-interpolation algorithm (similar to that in [GCL92, section 7.4]), which in fact works generically over $\mathbf{Z}$ or most fields.

For polynomials over any finite field or any field of characteristic zero besides $\mathbf{Q}$, the generic recursive multivariate evaluation-interpolation algorithm (3) above is used, which effectively takes advantage of any fast modular algorithm for the base univariate polynomials (e.g., for number fields). See the function `GreatestCommonDivisor` in the univariate polynomials chapter for details of univariate GCD algorithms.

For polynomials over another polynomial ring or rational function field, the polynomials are first "flattened" to be inside a multivariate polynomial ring over the base coefficient ring, then the appropriate algorithm is used for that base coefficient ring.

For polynomials over any other ring, the generic subresultant algorithm [Coh93, section 3.3] is called recursively on a subring with one less variable.

`GCD(Q)`

Given a sequence Q of polynomials, return the GCD of the elements of Q. If Q has length 0 and universe P, then the zero element of P is returned.

`LeastCommonMultiple(f, g)`

`Lcm(f, g)`

`LCM(f, g)`

The least common multiple of f and g in a multivariate polynomial ring P. The LCM of zero and anything else is zero. The result is normalized (see the function `Normalize`), so the result is always unique. The valid coefficient rings are as for the function `GCD`, above.

The LCM is effectively computed as `Normalize((F div GCD(F, G)) * G)`, for non-zero inputs.

`LCM(Q)`

Given a sequence Q of polynomials, return the LCM of the elements of Q. If Q has length 0 and universe P, then the one element of P is returned.

`Normalize(f)`

Given a polynomial f over the base ring R, this function returns the unique normalized polynomial g which is associate to f (so $g = uf$ for some unit in R). This is chosen so that if R is a field then g is monic, if R is $\mathbf{Z}$ then the leading coefficient of g is positive, if R is a polynomial ring itself, then the leading coefficient of g is recursively normalized, and so on for other rings.

`ClearDenominators(f)`

`ClearDenominators(Q)`

Given a polynomial f over a field K such that K is the field of fractions of a domain D, the first function computes the lowest common denominator L of the coefficients of f and returns the polynomial $g = L \cdot f$ over D with cleared denominators, and L. The second function returns the sequence of polynomials derived from independently clearing the denominators in each polynomial in the given sequence Q.

24.5.2 Content and Primitive Part

`Content(f)`

The content of f, that is, the greatest common divisor of the coefficients of f as an element of the coefficient ring.

`PrimitivePart(f)`

The primitive part of f, being f divided by the content of f.

`ContentAndPrimitivePart(f)`

`Contpp(f)`

The content (the greatest common divisor of the coefficients) of f, as an element of the coefficient ring, as well as the primitive part (f divided by the content) of f.

24.6 Factorization and Irreducibility

We describe the functions for multivariate polynomial factorization and associated computations.

`Factorization(f)`

Given a multivariate polynomial f over the ring R, this function returns the factorization of f as a factorization sequence Q, that is, a sequence of pairs, each consisting of an irreducible factor q_i a positive integer k_i (its multiplicity). Each irreducible factor is normalized (see the function `Normalize`), so the expansion of the factorization sequence is the unique canonical associate of f. The function also returns the unit u of R giving the normalization, so $f = u \cdot \prod_i q_i^{k_i}$.

The coefficient ring R must be one of the following: a finite field $\mathbf{F}_q$, the ring of integers $\mathbf{Z}$, the field of rationals $\mathbf{Q}$, an algebraic number field $\mathbf{Q}(\alpha)$, or a polynomial ring, function field (rational or algebraic) or finite-dimensional affine algebra (which is a field) over any of the above.

For bivariate polynomials, a polynomial-time algorithm in the same spirit as van Hoeij's Knapsack factoring algorithm [vH02] is used.

For polynomials over the integers or rationals, an algorithm similar to that presented in [Wan78] and [GCL92, section 6.8], based on evaluation and sparse ideal-adic multivariate Hensel lifting, is used.

For polynomials over any finite field, a similar algorithm is used, with a few special modifications for non-zero characteristic (see, for example, [BM97]).

For polynomials over algebraic number fields and affine algebras, a multivariate version of the norm-based algorithm of Trager [Tra76] is used, which performs a suitable substitution and multivariate resultant computation, and then factors the resulting integral multivariate polynomial.

Each of these algorithms reduces to univariate factorization over the base ring; for details of how this factorization is done in each case, see the function `Factorization` in the univariate polynomial rings chapter.

For polynomials over another polynomial ring or function field, the polynomials are first "flattened" to be inside a multivariate polynomial ring over the base coefficient ring, then the appropriate algorithm is used for that base coefficient ring.

`SquarefreeFactorization(f)`

Return the squarefree factorization of the multivariate polynomial f as a sequence of tuples of length 2, each consisting of a (not necessarily irreducible) factor and an integer indicating the multiplicity. The factors do not contain the square of any polynomial of degree greater than 0. The allowable coefficient rings are the same as those allowable for the function `Factorization`.

`SquarefreePart(f)`

Return the squarefree part of the multivariate polynomial f, which is the largest (normalized) divisor g of f which is squarefree.

`IsIrreducible(f)`

Given a multivariate polynomial f over a ring R, this function returns whether f is irreducible over R. The allowable coefficient rings are the same as those allowable for the function `Factorization`.

`SetVerbose("PolyFact", v)`

(Procedure.) Change the verbose printing level for all polynomial factorization algorithms to be v. Currently the legal levels are 0, 1, 2 or 3.

Example H24E8

We create a polynomial f in the polynomial ring in three indeterminates over the ring of integers by multiplying together various trinomials. The resulting product f has 461 terms and total degree 15. We then factorize f to recover the trinomials.

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> f := &*[x^i+y^j+z^k: i,j,k in [1..2]];
> #Terms(f);
461
> TotalDegree(f);
15
> time Factorization(f);
[
    <x + y + z, 1>,
    <x + y + z^2, 1>,
    <x + y^2 + z, 1>,
    <x + y^2 + z^2, 1>,
    <x^2 + y + z, 1>,
    <x^2 + y + z^2, 1>,
    <x^2 + y^2 + z, 1>,
    <x^2 + y^2 + z^2, 1>
]
Time: 0.290
```

Example H24E9

We construct a Vandermonde matrix of rank 6, find its determinant, and factorize that determinant.

```
> // Create polynomial ring over R of rank n
> PRing := function(R, n)
>     P := PolynomialRing(R, n);
>     AssignNames(~P, ["x" cat IntegerToString(i): i in [1..n]]);
>     return P;
> end function;
>
> // Create Vandermonde matrix of rank n
> Vandermonde := function(n)
```

```
>     P := PRing(IntegerRing(), n);
>     return MatrixRing(P, n) ! [P.i^(j - 1): i, j in [1 .. n]];
> end function;
>
> V := Vandermonde(6);
> V;
[   1   x1 x1^2 x1^3 x1^4 x1^5]
[   1   x2 x2^2 x2^3 x2^4 x2^5]
[   1   x3 x3^2 x3^3 x3^4 x3^5]
[   1   x4 x4^2 x4^3 x4^4 x4^5]
[   1   x5 x5^2 x5^3 x5^4 x5^5]
[   1   x6 x6^2 x6^3 x6^4 x6^5]
> D := Determinant(V);
> #Terms(D);
720
> TotalDegree(D);
15
> time Factorization(D);
[
    <x5 - x6, 1>,
    <x4 - x6, 1>,
    <x4 - x5, 1>,
    <x3 - x6, 1>,
    <x3 - x5, 1>,
    <x3 - x4, 1>,
    <x2 - x6, 1>,
    <x2 - x5, 1>,
    <x2 - x4, 1>,
    <x2 - x3, 1>,
    <x1 - x6, 1>,
    <x1 - x5, 1>,
    <x1 - x4, 1>,
    <x1 - x3, 1>,
    <x1 - x2, 1>
]
Time: 0.030
```

Example H24E10

We construct a polynomial $A2$ in three indeterminates a, b, and c over the rational field such that $A2$ is the square of the area of the triangle with side lengths a, b, c. Using elementary trigonometry one can derive the expression $(4 * a^2 * b^2 - (a^2 + b^2 - c^2)^2)/16$ for $A2$. Factorizing $A2$ gives a nice formulation of the square of the area which is similar to that given by *Heron's formula*.

```
> P<a, b, c> := PolynomialRing(RationalField(), 3);
> A2 := 1/16 * (4*a^2*b^2 - (a^2 + b^2 - c^2)^2);
> A2;
-1/16*a^4 + 1/8*a^2*b^2 + 1/8*a^2*c^2 - 1/16*b^4 + 1/8*b^2*c^2 - 1/16*c^4
```

```
> F, u := Factorization(A2);
> F;
[
    <a - b - c, 1>,
    <a - b + c, 1>,
    <a + b - c, 1>,
    <a + b + c, 1>
]
> u;
-1/16
```

Example H24E11

We factorize a multivariate polynomial over a finite field.

```
> Frob := function(G)
>     n := #G;
>     I := {@ g: g in G @};
>     P := PolynomialRing(GF(2), n);
>     AssignNames(~P, [CodeToString(96 + i): i in [1 .. n]]);
>     M := MatrixRing(P, n);
>     return M ! &cat[
>         [P.Index(I, I[i] * I[j]): j in [1 .. n]]: i in [1 .. n]
>     ];
> end function;
> A := Frob(Sym(3));
> A;
[a b c d e f]
[b c a f d e]
[c a b e f d]
[d e f a b c]
[e f d c a b]
[f d e b c a]
> Determinant(A);
a^6 + a^4*d^2 + a^4*e^2 + a^4*f^2 + a^2*b^2*c^2 +
    a^2*b^2*d^2 + a^2*b^2*e^2 + a^2*b^2*f^2 + a^2*c^2*d^2 +
    a^2*c^2*e^2 + a^2*c^2*f^2 + a^2*d^4 + a^2*d^2*e^2 +
    a^2*d^2*f^2 + a^2*e^4 + a^2*e^2*f^2 + a^2*f^4 + b^6 +
    b^4*d^2 + b^4*e^2 + b^4*f^2 + b^2*c^2*d^2 + b^2*c^2*e^2
    + b^2*c^2*f^2 + b^2*d^4 + b^2*d^2*e^2 + b^2*d^2*f^2 +
    b^2*e^4 + b^2*e^2*f^2 + b^2*f^4 + c^6 + c^4*d^2 +
    c^4*e^2 + c^4*f^2 + c^2*d^4 + c^2*d^2*e^2 + c^2*d^2*f^2
    + c^2*e^4 + c^2*e^2*f^2 + c^2*f^4 + d^6 + d^2*e^2*f^2 +
    e^6 + f^6
> time Factorization(Determinant(A));
[
    <a + b + c + d + e + f, 2>,
    <a^2 + a*b + a*c + b^2 + b*c + c^2 + d^2 + d*e + d*f +
```

```
        e^2 + e*f + f^2, 2>
]
Time: 0.049
```

24.7 Resultants and Discriminants

`Resultant(f, g, i)`

`Resultant(f, g, v)`

The resultant of multivariate polynomials f and g in $P = R[x_1, \ldots, x_n]$ with respect to the variable $v = x_i$, which is by definition the determinant of the Sylvester matrix for f and g when considered as polynomials in the single variable x_i. The result will be an element of P again. The coefficient ring R must be a domain. There are two ways to indicate with respect to which variable the integral is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

The algorithm used is the modular interpolation method, as given in [GCL92, pp. 412–413].

`Discriminant(f, i)`

`Discriminant(f, v)`

The discriminant D of $f \in R[x_1, \ldots, x_n]$ is returned, where f is considered as a polynomial in $v = x_i$. The result will be an element of P again. The coefficient ring R must be a domain. There are two ways to indicate with respect to which variable the integral is to be taken: either one specifies i, the integer $1 \leq i \leq n$ that is the number of the variable (upon creation of P, corresponding to `P.i`) or the variable v itself (as an element of P).

24.8 Polynomials over the Integers

The functions in this section are available for multivariate polynomials over the integers only.

`Sign(f)`

The sign of the leading coefficient of f.

`AbsoluteValue(f)`

`Abs(f)`

Return either f or $-f$ according to which one has non-negative leading coefficient.

`MaxNorm(f)`

The maximum of the absolute values of the coefficients of f.

`SumNorm(f)`

The sum of the base coefficients of f.

24.9 Bibliography

[**BM97**] Laurent Bernardin and Michael B. Monagan. Efficient Multivariate Factorization Over Finite Fields. In *Proceedings of AAECC*, volume 1255 of *LNCS*, pages 15–28. Springer-Verlag, 1997.

[**CGG89**] Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation. *J. Symbolic Comp.*, 7(1):31–48, 1989.

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**GCL92**] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Kluwer, Boston/Dordrecht/London, 1992.

[**MY73**] J. Moses and D.Y.Y. Yun. The EZ GCD algorithm. *Proc. ACM Annual Conference*, 73(2):159–166, 1973.

[**Tra76**] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, pages 196–208. ACM press, 1976.

[**vH02**] Mark van Hoeij. Factoring Polynomials and the knapsack problem. *J. Number Th.*, 95(2):167–189, 2002.
URL:http://www.math.fsu.edu/∼hoeij/paper/knapsack.ps.

[**Wan78**] Paul S. Wang. An improved multivariate polynomial factoring algorithm. *Math. Comp.*, 32(144):1215–1231, 1978.

[**Wan80**] Paul S. Wang. The EEZ-GCD algorithm. *SIGSAM Bulletin*, 14(2):50–60, 1980.

25 REAL AND COMPLEX FIELDS

Chapter 25
REAL AND COMPLEX FIELDS

25.1 Introduction

Real and complex numbers can only be stored in the computer effectively in approximations. MAGMA provides a number of facilities for calculating with such approximations. Most of these facilities are based upon the C libraries MPFR, which provides algorithms for manipulating real numbers with *exact rounding*, and MPC, an extension of MPFR to handle complex numbers. More specifically, the MFPR library extends the semantics of the ANSI/IEEE-754 standard for double-precision numbers — which are used in virtually all major programming languages — to handle real numbers of arbitrary precision. The precise semantics of MPFR give the user fine control over precision loss, which is a tremendous advantage when working with reals and complexes. MAGMA currently uses MPFR 2.4.1 and MPC 0.8. Documentation for algorithms used in MPFR can be found at `mpfr.org`.

As MPFR and MPC are works in progress, they do not yet provide a complete framework for working with the reals and complexes. For those functions that these libraries are missing, Magma falls back to algorithms taken from PARI. The documentation of MFPR and MPC provide a list of the functions that they provide. Assume that each intrinsic uses MPFR unless otherwise stated.

Although we use the terms *real field* and *complex field* for MAGMA structures containing real or complex approximations, it should be noted that such a subset of the real or complex field may not even form a commutative ring. Never the less, the real and complex fields are considered to be fields by MAGMA, they comprise objects of type `FldRe` and `FldCom` with elements of type `FldReElt` and `FldComElt` respectively.

25.1.1 Overview of Real Numbers in MAGMA

Real numbers are stored internally as expansions $\sum b_i 2^i$. Complex numbers consist of a pair of real numbers of identical precision. Each real or complex number is associated with a corresponding field structure, which has the same precision as all of its elements. MAGMA stores a list of real and complex fields that have been created during a session, and it is guaranteed that any two fields of the same fixed precision are the same. This means in particular that changing the name of $\sqrt{-1}$ (see `AssignNames` below) on one of the complex fields of precision r will change the name on every complex field of that same precision. As a convenience, MAGMA allows real and complex numbers of differing precisions to be used in the same expression; internally, MAGMA implicitly reduces the precision of the higher precision element to the precision of the lower element.

While internally we store real numbers in base two, when creating real or complex fields the precision is by default specified in the number of *decimal* digits, not binary digits, required. It is possible to specify the precision in binary digits if needed (see the documentation for `RealField` for details).

Example H25E1

We show how to create and manipulate real numbers. In particular, note that there is an inherent loss of precision in the conversion between base 10 and base 2 representations of some real numbers.

```
> S1 := RealField(20);
> S2 := RealField(10);
> a := S1 ! 0.5;
> a;
0.50000000000000000000
> b := S2 ! 0.05;
> b;
0.05000000000
> a + b;
0.5500000000
> Precision(a + b);
10
```

A warning is in place here; in the examples above, the real number on the right hand side had to be constructed in some real field *before* it could be coerced into S_1 and S_2. That real field is the so-called *default real field*. In these examples it is assumed that the default field has sufficiently large precision to store the real numbers on the right accurately to the last digit.

25.1.2 Coercion

Automatic coercion ensures that all functions listed below that take an element of some real field as an argument, will also accept an integer or a rational number as an argument; in this case the integer or rational number will be coerced automatically into the default real field. For the binary operations (such as `+, *`) coercion also takes place: if one argument is real and the other is integral or rational, automatic coercion will put them both in the parent field of the real argument. If the arguments are real numbers of different fixed precision, the result will have the smaller precision of the two.

The same coercion rules apply for functions taking a complex number as an argument; in that case real numbers will be valid input as well: if necessary reals, rationals or integers will be coerced into the appropriate complex field.

Elements of quadratic and cyclotomic fields that are real can be coerced into any real field using `!`; any quadratic or cyclotomic field element can be coerced by `!` into any complex field. Functions taking real or complex arguments will not *automatically* coerce such arguments though.

25.1.3 Homomorphisms

The only homomorphisms that have a real field or a complex field as domain are the coercion functions. Therefore, homomorphisms from the reals or complexes may be specified as follows.

```
hom< R -> S | >
```

Here S must be a structure into which all elements of the real or complex field R are coercible, such as another real or complex field, or a polynomial ring over one of these. These homomorphisms can also be obtained as map by using the function `Coercion`, also called `Bang`.

Example H25E2

Here are two equivalent ways of creating the embedding function from a real field into a polynomial ring over some complex field.

```
> Re := RealField(20);
> PC<x, y> := PolynomialRing(ComplexField(8), 2);
> f := hom< Re -> PC | >;
> bangf := Bang(Re, PC);
> f(Pi(Re));
3.1415927
> f(Pi(Re)) eq bangf(Pi(Re));
true
```

25.1.4 Special Options

When MAGMA is started up, real and complex fields of precision 30 are created by default. They serve (among other things) as a parent for reals that are created as literals, such as 1.2345, in the same way as the default ring of integers is the parent for literal integers. It is possible to change this default real field with `SetDefaultRealField`.

Finally, `AssignNames` can be used to change the name for $\sqrt{-1}$ in a complex field.

```
SetDefaultRealField(R)
```

Procedure to change the default parent for literal real numbers to the real field R. This parent is the real field of precision 30 by default.

```
GetDefaultRealField()
```

Return the current parent for literal real numbers.

`AssignNames(~C, [s])`

Procedure to change the name of the purely imaginary element $\sqrt{-1}$ in the complex field C to the contents of the string s. When C is created, the name is "C.1"; suitable choices of s might be `"i"`, `"I"` or `"j"`.

This procedure only changes the name used in printing the elements of C. It does *not* assign to an identifier called s the value of $\sqrt{-1}$ in C; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies C, it is necessary to have a reference `~C` to C in the call to this function.

`Name(C, 1)`

Given a complex field C, return the element which has the name attached to it, that is, return the purely imaginary element $\sqrt{-1}$ of C.

25.1.5 Version Functions

The following intrinsics retrieve the versions of MPFR, MPC and GMP which the current MAGMA is using.

`GetGMPVersion()`
`GetMPFRVersion()`
`GetMPCVersion()`

The version of GMP, MPFR or MPC being used.

25.2 Creation Functions

We describe the creation of real and complex fields and their elements.

25.2.1 Creation of Structures

At the time MAGMA is loaded, a real field is automatically created. This is used as the default parent for literal reals and real values returned by MAGMA.

`RealField(p)`

`Bits` BOOLELT *Default :* `false`

Given a positive integer p, create and return a version R of the real field $\mathbf{R}$ in which all calculations are correct to precision p. If the parameter `Bits` is `true`, then the precision p is specified as the number of binary digits. If `Bits` is `false`, then the precision is given as the number of decimal digits — this is translated into a binary precision of $\lceil \log_2 10^p \rceil$.

`RealField()`

Return the default real field.

`ComplexField(p)`

`Bits` BoolElt *Default :* `false`

Given a positive integer p, create and return a version C of the complex field $\mathbf{C}$ in which all calculations are correct to precision p. If the parameter `Bits` is `true`, then the precision p is specified as the number of binary digits. If `Bits` is `false`, then the precision is given as the number of decimal digits — this is translated into a binary precision of $\lceil \log_2 10^p \rceil$.

By default no name is given to $\sqrt{-1}$; this may be changed with `AssignNames`. Angle brackets, e.g. `C<i> := ComplexField(20)`, may be used to assign $\sqrt{-1}$ to an identifier.

`ComplexField()`

Return the default complex field.

By default no name is given to $\sqrt{-1}$; this may be changed with `AssignNames`. Angle brackets, e.g. `C<i> := ComplexField()`, may be used to assign $\sqrt{-1}$ to an identifier.

`ComplexField(R)`

Return the complex field which has real subfield R; in other words, return the complex field with the same precision as the real field R.

Example H25E3

It is convenient to use i to define elements of a complex field. It is also possible to change the default printing of i, using `AssignNames`, as follows. Note that the latter procedure does not assign to an identifier, it only changes the printing.

```
> C<i> := ComplexField(20);
> Pi(C)+ 1/4*i;
3.14159265358979323850 + 0.25000000000000000000*i
> AssignNames(~C, ["k"]);
> Pi(C)+ 1/4*i;
3.14159265358979323850 + 0.25000000000000000000*k
> k := Name(C, 1);
> Pi(C)+ 1/4*k;
3.14159265358979323850 + 0.25000000000000000000*k
```

25.2.2 Creation of Elements

`d . eefpg`

`d . eEfPg`

Given a succession of literal decimal digits d, a succession of literal decimal digits e, a succession of literal decimal digits f, and an integer g, construct the real number $r = d.e \times 10^f$. If specified, the effect of g is to create r as an element of the real field of precision g.

If g is omitted (together with `p` or `P`), the real number will be created as an element of the default real field.

Both d and f may include a leading sign `+` or `-`; leading zeroes in d and f are ignored. If e consists entirely of zeroes it may be omitted together with the `.` and if f is zero it may be omitted together with `E` (or `e`). But note that if all of e, f and g are omitted the result will be an integer.

```
elt< R | m, n >
```

Given the real field R, an element m coercible into R and an integer n, construct the real number $m \times 2^n$ in R.

```
elt< C | x, y >
```

```
C ! [x, y]
```

Given the complex field C and elements x and y coercible into the real field underlying C, construct the complex number $x + y$i.

```
R ! a
```

Given an integer, a rational number, a quadratic or cyclotomic number field element a, this returns an element from the real field R that best approximates a. An error results if a is a non-real quadratic or cyclotomic field element.

If R is a field of precision r and a is an element of a real field S of precision s then:

if a is an element of a real field S of precision $s \geq r$, then an element of R approximating a to r digits is returned;

if a is an element of a real field S of precision $s < r$, then an element of R is returned approximating a, obtained by padding with zeroes until the required precision r is reached;

```
C ! a
```

Given an integer, a rational number, a quadratic or cyclotomic number field element a, this returns an element from the complex field C that best approximates a. The rules of coercion for the real and imaginary parts are the same as those for coercion into a real field.

Example H25E4

We create the real number 1.2345 in many ways. We assume that the default real field has not been changed.

```
> x := 1.2345;
> x, Parent(x);
1.234500000000000000000000000000 Real field of precision 30
> SetDefaultRealField(RealField(20));
> x1 := 1.2345;
> x1, Parent(x1);
1.2345000000000000000 Real field of precision 20
> x2 := 12345e-4;
> x2, Parent(x2);
1.2345000000000000000 Real field of precision 20
> x3 := 1.2345p10;
> x3, Parent(x3);
1.234500000 Real field of precision 10
> x4 := 12345e-4p8;
> x4, Parent(x4);
1.2345000 Real field of precision 8
> x5 := RealField(12) ! 1.2345;
> x5, Parent(x5);
1.23450000000 Real field of precision 12
```

The following generic element constructions are available; they return the 1 and 0 element of a real or complex field.

`One(R)` `Identity(R)`
`Zero(R)` `Representative(R)`

25.3 Structure Operations

25.3.1 Related Structures

`Category(R)` `Parent(R)`
`PrimeField(R)`

25.3.2 Numerical Invariants

`Characteristic(R)`

25.3.3 Ring Predicates and Booleans

IsCommutative(R) IsUnitary(R)
IsFinite(R) IsOrdered(R)
IsField(R) IsEuclideanDomain(R)
IsPID(R) IsUFD(R)
IsDivisionRing(R) IsEuclideanRing(R)
IsPrincipalIdealRing(R) IsDomain(R)
R eq S R ne S

25.3.4 Other Structure Functions

Precision(R)

Return the decimal precision p to which calculations are performed in the real or complex field R.

BitPrecision(R)

Return the (internally used) bit precision p to which calculations are performed in the real or complex field R.

25.4 Element Operations

25.4.1 Generic Element Functions and Predicates

All predicates on real or complex numbers that check whether these numbers are equal to an integer do so within the given precision of the parent field. Thus `IsOne(c)` for an element of a complex domain of precision 20 returns true if and only if the real part equals one and the imaginary part equals 0 up to 20 decimals.

Parent(r) Category(r)
IsZero(r) IsOne(r) IsMinusOne(r)
IsUnit(r) IsZeroDivisor(r)
IsIdempotent(r) IsNilpotent(r)
IsIrreducible(r) IsPrime(r)

25.4.2 Comparison of and Membership

The (in)equality test on real numbers of only test for equality up to the given precision. Equality testing on complex numbers is done by testing the real and imaginary parts.

The comparison functions `gt, ge, lt, le` are not defined for complex numbers.

```
a eq b    a ne b
a in R    a notin R
a gt b    a ge b    a lt b    a le b
Maximum(a, b)    Minimum(a, b)
Maximum(Q)    Minimum(Q)
```

25.4.3 Other Predicates

`IsIntegral(c)`

Returns `true` if and only if the real or complex number c is a rational integer.

`IsReal(c)`

Returns `true` if the complex number c is real, `false` otherwise. This checks whether the digits of the imaginary part of c are 0 up to the precision of the parent complex field.

25.4.4 Arithmetic

The binary operations `+, -, *, /` allow combinations of arguments from the integers, the rationals, and real and complex fields; automatic coercion is applied where necessary (see the Introduction).

```
+ r    - r
r + s    r - s    r * s    r / s    r ^ k
r +:= s    r -:= s    r *:= s    r /:= s    r ^:= s
```

25.4.5 Conversions

Here we list various ways to convert between integers, reals of fixed precision, complexes and their various representations, other than by the creation functions and `!`. See also the rounding functions in a later section.

`MantissaExponent(r)`

Given a real number r, this function returns a real number m (the mantissa of r) and an integer e (the exponent of r) such that $1 \leq m < 10$ and $r = m \times 2^e$.

`ComplexToPolar(c)`

Given a complex number c, return the modulus $m \geq 0$ and the argument a (with $-\pi \leq a \leq \pi$) of c as real numbers to the same precision as c.

`PolarToComplex(m, a)`

Given real numbers m and a, construct the complex number me^{ia}. The result will have the smaller of the precisions of m and a; each of m and a is allowed to be an integer or rational number; if both are integral or rational then the result will have the default precision, otherwise the result will be of the same precision as the real argument.

`Argument(c)`

`Arg(c)`

Given a complex number c, return the real number (to the same precision) that is the argument (in radians between $-\pi$ and π) of c.

`Modulus(c)`

Given a complex number c, return the real number (to the same precision as c) that is the modulus of c.

`Real(c)`

`Re(c)`

Given a complex number $c = x + y\mathrm{i}$, return the real part x of c (as a real number to the same precision as c).

`Imaginary(c)`

`Im(c)`

Given a complex number $c = x + y\mathrm{i}$, return the imaginary part y of c (as a real number to the same precision as c).

25.4.6 Rounding

`Round(r)`

Given a real number r, return the integer i for which $|r - i|$ is a minimum. i.e., the integer closest to r. If there are two such integers, the one of larger magnitude is chosen (rounding away from zero). Given a (non-real) complex number r, return the Gaussian integer i for which $|r - i|$ is a minimum, i.e. the Gaussian integer closest to r.

`Truncate(r)`

Given a real number r, return $\lfloor r \rfloor$ if r is positive, and return $-\lfloor -r \rfloor + 1$ if r is negative. Thus, the effect of this function is to round towards zero.

`Ceiling(r)`

`Ceiling(r)`

The ceiling of the real number r, i.e. the smallest integer greater than or equal to r.

`Floor(r)`

`Floor(r)`

The floor of the real number r, i.e. the greatest integer less than or equal to r.

25.4.7 Precision

`Precision(c)`

Given a real or complex number c belonging to the real or complex field C, return the decimal precision p to which calculations are performed in C.

`BitPrecision(c)`

Given a real or complex number c belonging to the real or complex field C, return the (internally used) bit precision p to which calculations are performed in C.

`Precision(L)`

`Precision(L)`

Gives a sequence of real or complex numbers, return the precision p of their parent field.

`ChangePrecision(r, n)`

`ChangePrecision(c, n)`

Coerces the real (r) or complex (c) number into a field of precision n.

25.4.8 Constants

Let R denote a real or complex field. The functions described below will return an approximation of certain constants to the precision associated with a given real or complex field R. If R is real, a real number is returned; if R is complex, a complex number with imaginary part zero is returned.

`Catalan(R)`

The value of Catalan's constant computed to the accuracy associated with the real or complex field R. Catalan's constant is the sum

$$\sum_{k=0}^{\infty}(-1)^k(2k+1)^{-2}.$$

MPFR calculates this constant using formula (31) of Victor Adamchik's document "33 representations for Catalan's constant"*, for more information see `mpfr.org`.

---

* `http://www-2.cs.cmu.edu/~adamchik/articles/catalan/catalan.htm`

`EulerGamma(R)`

The value of Euler's constant

$$\gamma = \lim_{n\to\infty}(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} - \log n) \approx 0.57721566$$

computed to the precision of R.

`Pi(R)`

The value of π computed to the precision of R.

25.4.9 Simple Element Functions

`AbsoluteValue(r)`

`Abs(r)`

The absolute value of the real or complex number r.

`Sign(r)`

Return one of the integer values $+1$, 0, -1 depending upon whether the real number r is positive, zero or negative, respectively.

`ComplexConjugate(c)`

`Conjugate(c)`

The complex conjugate $x - y\mathrm{i}$ of a complex number $x + y\mathrm{i}$.

`Norm(c)`

The real norm of a real or complex number c; note that for complex $c = x + y\mathrm{i}$ this returns $x^2 + y^2$, while for elements of real domains it just returns the absolute value. The result lies in the same field as the argument.

`Root(r, n)`

Given a real number R and a positive integer n, calculate $\sqrt[n]{r}$ (using Newton's method without divisions) with the same precision. If n is even then r must be non-negative.

`SquareRoot(c)`

`Sqrt(c)`

Given a real or complex number c, return the square root of r as an element of the same field to which r belongs.

```
Distance(x, L)
Distance(x, L)
Distance(x, L)
Distance(x, L)
```

Given a sequence L of real or complex numbers and an additional number x compute the distance between x and L, ie. $\min_{y \in L} |x-y|$, that is the shortest distance between x and any element of L. Furthermore, the index in L of an element realising the distance is returned as a second argument.

```
Diameter(L)
Diameter(L)
```

Given a sequence L of real of complex numbers, compute the diameter of the set defined by L, ie. the smallest distance between distinct elements of L.

25.4.10 Roots

MAGMA contains a very powerful algorithm for finding highly accurate approximations to the complex roots of a polynomial; it is based on Xavier Gourdon's implementation of Schönhage's algorithm, which we will summarize below.

Given a polynomial $p = a_0 + a_1 z + \cdots + a_n z^n \in \mathbf{C}[z]$, define the norm of p, $|p|$, by

$$|p| = |a_0| + |a_1| + \cdots + |a_n|.$$

Schönhage's algorithm (given in his technical report of 1982 [Sch82]) takes as input a univariate polynomial p in $\mathbf{C}[z]$ and a positive real number ε, and finds linear factors $L_j = u_j z - v_j$ $(j = 1, \ldots, n = \deg(p))$ such that

$$|p - L_1 \cdots L_n| < \varepsilon |p|.$$

The parameter ε may be chosen so as to find the roots of p to within a certain ε', and this is how the function `Roots` described below works (when run with Schönhage's algorithm).

The algorithm uses the concept of a 'splitting circle' to find polynomials F and G such that $|p - FG| < \varepsilon_1 |p|$ for some ε_1 depending on ε.

This splitting circle method can then be applied recursively to F and G until we have only linear factors, as required.

The splitting circle method works as follows. For the purposes of this discussion assume that p is monic. Suppose we know a circle Γ such that, for some integer k with $0 < k < n$, there are k roots of p (say $u_1, \ldots, u_k$) which lie inside Γ, and the other $n - k$ roots $(u_{k+1}, \ldots, u_n)$ lie outside Γ. Note that the circle Γ is chosen so that the roots of p are not too close to it. Then we can write $p = FG$, where $F = (z - u_1) \ldots (z - u_k)$ and $G = (z - u_{k+1}) \ldots (z - u_n)$. Through shifts and scalings, we may assume that $\Gamma = \{c \in \mathbf{C} : \|z\| = 1\}$.

For m in $\{1, \ldots, k\}$, let s_m denote the m-th power sum of the roots of p which lie inside the splitting circle. That is,

$$s_m = u_1^m + \cdots + u_k^m.$$

The residue theorem can then be used to calculate s_m $(1 \leq m \leq k)$:

$$s_m = \frac{1}{2\pi i} \int_\Gamma z^m \frac{p'(z)}{p(z)}\, dz.$$

where the integration can be computed to the required precision by the discrete sum

$$s_m \approx \frac{1}{N} \sum_{j=0}^{N-1} \frac{p'(\omega^j)}{p(\omega^j)} \omega^{(m+1)j}.$$

for a large enough integer N, where $\omega = \exp(2\pi i/N)$.

The coefficients of the polynomial F can then be computed from the Newton sums s_m $(1 \leq m \leq k)$ using the classical Newton formulae. Then set $G = p/F$.

The integer N above needed to get F and G to the required precision can be quite large. It is more efficient to use a smaller value of N to give an approximation F_0 of F, and then use the following refining technique.

Define G_0 (an approximation of G) by $p = F_0 G_0 + r$, where $\deg(r) < \deg(F_0)$. We want polynomials f and g such that $F_1 = F_0 + f$ and $G_1 = G_0 + g$ are better approximations of F and G. Now

$$p - F_1 G_1 = p - F_0 G_0 - f G_0 - g F_0 - fg.$$

Hence choosing f and g such that

$$p - F_0 G_0 = f G_0 + g F_0$$

will lead to a second order error.

The Euclidean algorithm could be used to find f and g, but this is numerically unstable. It suffices to find polynomials H (called the auxiliary polynomial) and L such that

$$1 = H G_0 + L F_0,$$

where $\deg(H) < \deg(F_0)$ and $\deg(L) < \deg(G_0)$.

The polynomial H can be calculated using the formula

$$H(z) = \frac{1}{2\pi i} \int_\Gamma \frac{1}{(F_0 G_0)(t)} \frac{F_0(z) - F_0(t)}{z - t}\, dt.$$

Again, rather than computing the integral to the required precision directly, we find only an approximation H_0 and then refine it using Newton iteration:

$$H_{m+1} \equiv H_m(2 - H_m G_0) \pmod{F_0}.$$

Assuming that $|H - H_0|$ is small, the sequence (H_m) converges quadratically to H.

Once H is known to a large enough precision, f can be computed by

$$f \equiv H(p - F_0 G_0) \pmod{F_0}.$$

This gives us the new approximation F_1 of F, and G_1 is computed by division of p by F_1. We repeat this process until

$$|p - FG| < \varepsilon_1 |p|$$

and we are done.

The problem remains to find the splitting circle.

This relies mainly on the computation of the moduli of the roots of p. Let

$$r_1(p) \leq r_2(p) \leq \cdots \leq r_n(p)$$

denote the moduli of the roots of p in ascending order. For each k, the computation of $r_k(p)$ with a small number of digits can be achieved in a reliable way using the Graeffe process. The Graeffe process is a root squaring step transforming any given polynomial p into a polynomial q of the same degree whose roots are the square of the roots of p.

By the use of a suitable shift, we may assume that the sum of the roots of p is zero. If $p(0) = 0$, then we have found a factorization $p \approx FG$ with $F = z$ and $G = p/z$. If not, then the computation of the maximum root modulus $r_n(p)$ allows us to scale p so that its maximum root modulus is now close to 1. For $j = 0, 1, 2, 3$, set

$$q_j(z) = p(z + 2i^j).$$

Then amongst these four polynomials there exists q such that

$$\frac{r_n(q)}{r_1(q)} = \exp(\Delta),$$

with $\Delta > 0.3$. A dichotomic process from the computation of some $r_j(q)$ can then be applied to find k $(1 \leq k \leq n-1)$ such that

$$\frac{r_{k+1}(q)}{r_k(q)} > \exp(\frac{\Delta}{n-1}).$$

Then the circle $\{c : \|c\| = \sqrt{r_k(q) r_{k+1}(q)}\}$ is a suitable splitting circle, with the roots not too close to it.

`Roots(p)`

`Al`	MonStgElt	*Default :* "*Schonhage*"
`Digits`	RngIntElt	*Default :*

Given a univariate polynomial p over a real or complex field, this returns a sequence of complex approximations to the roots of p. The elements of this sequence are of the form $< r, m >$, where r is a root and m its multiplicity.

The algorithm used to find the roots of p may be specified by using the optional argument `Al`. This must be one of `"Schonhage"` (which is the default), `"Laguerre"`, `"NewtonRaphson"` or `"Combination"` (a combination of Laguerre and Newton-Raphson). When using the (default) Schönhage algorithm, the roots given are correct to within an absolute error of 10^{-d}, where d is the value of `Digits`. This algorithm gives correct results in all cases. When using the other algorithms (for which correct answers are not guaranteed in all cases), the results are found with `Digits` significant figures. The default value for `Digits` is the current precision of the free real field.

PARI is used here for complex polynomials.

Warning: Beware of the problems of floating point numbers. Because real numbers are stored in the computer with finite precision, you may not be finding the roots of the polynomial you want. If you know the polynomial exactly, you should enter it with exact (that is, integer or rational) coefficients. This is illustrated in the following example.

Example H25E5

```
> P<z> := PolynomialRing(ComplexField());
> p := (z-1.1)^6;
> p;
z^6 - 6.60000000000000000000000000001*z^5 +
    18.15000000000000000000000000000*z^4 -
    26.62000000000000000000000000000*z^3 +
    21.96150000000000000000000000000*z^2 -
    9.66306000000000000000000000003*z +
    1.77156100000000000000000000001
> R := Roots(p);
> R;
[ <1.100013305965906054216519998 57, 1>,
<1.100006652894308609692986689 17 +
1.152330449581798256514866512 57E-5*i, 1>,
<1.100006652894308609692986689 17 -
1.152330449581798256514866512 57E-5*i, 1>,
<1.099986694211380305218347312 34, 1>,
<1.099993347017048210589579655 37 +
1.152315096132698580046691677 11E-5*i, 1>,
<1.099993347017048210589579655 37 -
1.152315096132698580046691677 11E-5*i, 1> ]
> P<x> := PolynomialRing(Rationals());
> q := (x-11/10)^6;
> Roots(q);
[ <11/10, 6> ]
```

The function `RootsNonExact` (below) is more suitable for non-exact polynomials.

`RootsNonExact(p)`

Given a polynomial p of degree n defined over a real or complex field, returns a sequence $[v_1, \ldots, v_n]$ of complex numbers such that

$$|p - a(z - v_1) \ldots (z - v_n)| < 10^{-d}|p|,$$

where a is the leading coefficient of p, and d is the precision of the field.

A second sequence $[e_1, \ldots, e_n]$ of (free) real numbers may also be returned. Given any polynomial $\hat{p}$ such that $|p - \hat{p}| < 10^{-d}|p|$, we can write $\hat{p} = a(z - u_1) \ldots (z - u_n)$ with $|v_i - u_i| < e_i$. In some cases, such error bounds cannot be derived, because the value d of `Digits` is too small for the given polynomial. In these cases, this second sequence is not returned.

This function acknowledges the fact that the polynomial p may not be the exact polynomial wanted, but only an approximation (to a certain number of decimal places), and so the roots of the true polynomial can only be found to a limited number of decimal places. Increasing the precision will decrease the errors on the 'roots'.

Example H25E6

```
> P<z> := PolynomialRing(ComplexField());
> p := (z-1.1)^6;
> R, E := RootsNonExact(p);
> R;
[ 1.10001483296913451410370191006 -
8.56404454142796527111307103383E-6*i,
1.10001483296913451410370191006 +
8.56404454142796527111304767692E-6*i,
1.09998516742199321904393975959 +
8.56336708832137724325720239457E-6*i,
1.09999999960887226685235833026 +
1.71274116266516824685125851069E-5*i,
1.09998516742199321904393771623 -
8.56336708832137723945115457519E-6*i,
1.09999999960887226685236037365 -
1.71274116266516824723187272572E-5*i ]
> E;
[ 0.0048231441542156971991062164306 6,
0.0048231441542156971991062164306 6,
0.0048230140891973860561847686767 6,
0.0048230791126115946099162101745 6,
0.0048230140891973860561847686767 6,
0.0048230791126115946099162101745 6 ]
```

`HenselLift(f, R, k)`

`HenselLift(f, R, k)`

Let f be a real or complex polynomial and x an approximation to a single zero of f. This function will apply the Newton-iteration to improve the accuracy of the root to the precision indicated by k.

25.4.11 Continued Fractions

The following functions use the continued fraction expansion of real numbers to get Diophantine approximations. They were obtained from corresponding PARI implementations.

`ContinuedFraction(r)`

`ContinuedFraction(r)`

`Bound` RNGINTELT *Default :* -1

Given an element r from a real field, return a sequence of integers s that form the partial quotient for the (regular) continued fraction expansion for r, so r is approximately equal to

$$s_1 + \cfrac{1}{s_2 + \cfrac{1}{s_3 + \cdots + \cfrac{1}{s_n}}}.$$

The length n of the sequence is determined in such a way that the last significant partial quotient is obtained (determined by the precision with which r is known), unless the optional integer argument `Bound` is used to limit the length.

`BestApproximation(r, n)`

Given an element r from a real field and a positive integer n, this function determines a rational approximation to r with denominator not exceeding n. The approximation is at least as close as the best continued fraction convergent with denominator not exceeding n. PARI is used here.

`Convergents(s)`

Given a sequence s of n non-negative integers (forming the partial fractions of a real number r, say), this function returns a 2×2 matrix with integer coefficients

$$\begin{pmatrix} p_n & p_{n-1} \\ q_n & q_{n-1} \end{pmatrix};$$

the quotients p_{n-1}/q_{n-1} and p_n/q_n form the last two convergents for r as provided by s.

25.4.12 Algebraic Dependencies

`LinearRelation(q: parameters)`

`LinearRelation(v: parameters)`

`Al`	MonStgElt	*Default :* "*Hastad*"

Given a sequence q or a vector v with entries from a complex field, return an integer sequence or vector forming the coefficients for a (small) linear dependency among the entries. The algorithm used may be specified by the optional parameter `Al`. The default is `"Hastad"`, which uses a variation of the LLL algorithm due to Hastad, Lagarias and Schnorr; the alternative is `"LLL"`, which uses a straight LLL algorithm. Pari is used here. The new version of this function is `IntegerRelation`.

`AllLinearRelations(q,p)`

Given a sequence q with entries from a real or complex field, return the lattice of all (small) integer linear dependencies among the entries. The precision, p, given as second argument is used for two purposes. First "small" is defined to be any relation such that the sum of the digits of the coefficients is less than p. Second, a linear relation must be zero to within 10^{-p}.

`PowerRelation(r, k: parameters)`

`Al`	MonStgElt	*Default :* "*Hastad*"
`Precision`	RngIntElt	*Default :*

Given an element r from a real or complex field, and an integer $k > 0$, return a univariate integer polynomial of degree at most k having r as an approximate root. The parameters here have the same usage and meaning as for `LinearRelation`. Pari is used here. The new version of this function is `MinimalPolynomial`.

25.5 Transcendental Functions

25.5.1 Exponential, Logarithmic and Polylogarithmic Functions

In this section the exponential and logarithmic functions to the natural base e are described, as well as the conversion to the logarithm with respect to any base.

The power series expansions are

$$e^z = \sum_{n=0}^{\infty} \frac{z^n}{n!}, \quad \ln(1+z) = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{z^n}{n}.$$

Further information on the Dilog and Polylog functions can be found in Lewin [Lew81].

`Exp(f)`

Given a power series f defined over a real or complex field, return the exponential power series of f.

`Exp(c)`

Given an arbitrary real or complex number c, return the exponential e^c of c. Here c is allowed to be free or of fixed precision, and the result is in the same field as c.

`Log(f)`

Given a power series f defined over a real or complex field, return the logarithm of f. The valuation of f must be zero.

`Log(c)`

Given a non-zero real or complex number c, return the logarithm of c (to the natural base e). The principal value with imaginary part in $(-\pi, \pi]$ is chosen. The result will be a complex number, unless the argument is real and positive, in which case a real number is returned.

`Log(b, r)`

Given non-negative real numbers b and r, return the logarithm $\log_b(r)$ of a to the base b. Automatic coercion is applied if necessary.

`Dilog(s)`

For a given complex s, this returns the value of the principal branch of the *dilogarithm* $\mathrm{Li}_2(s)$, which can be defined by

$$\mathrm{Li}_2(s) = -\int_0^s \frac{\log(1-s)}{s} ds,$$

and forms the analytic continuation of the power series

$$\sum_{n=1}^{\infty} \frac{s^n}{n^2},$$

(which is convergent for $|s| \leq 1$). For large values of the argument a functional equation like

$$\mathrm{Li}_2(\frac{-1}{s}) + \mathrm{Li}_2(-s) = 2\mathrm{Li}_2(-1) - \frac{1}{2}\log^2(s)$$

should be used.

`Polylog(m, f)`

For an integer $m \geq 2$ and power series f defined over a real or complex field, return the m-th polylogarithm of the series f. The valuation of f must be positive for $m > 1$.

`Polylog(m, s)`

For given integer $m \geq 2$ and complex s this returns the value of the principal branch of the *polylogarithm* $\mathrm{Li}_m(s)$, defined for $m \geq 3$ by

$$\mathrm{Li}_m(s) = \int_0^s \frac{\mathrm{Li}_{m-1}(s)}{s} ds$$

(and for $m = 2$ as the dilogarithm Li_2). Then Li_m is the analytic continuation of

$$\sum_{n=1}^{\infty} \frac{s^n}{n^m},$$

(which is convergent for $|s| \leq 1$). For large values of the argument a functional equation like

$$(-1)^m \mathrm{Li}_m(\frac{-1}{s}) + \mathrm{Li}_m(-s) = -\frac{1}{m!}\log^m(s) + 2\sum_{r=1}^{\lfloor m/2 \rfloor} \frac{\log^{m-2r}(s)}{(m-2r)!}\mathrm{Li}_{2r}(-1)$$

should be used. PARI is used here.

`PolylogD(m, s)`

`PolylogDold(m, s)`

`PolylogP(m, s)`

Given integer $m \geq 2$ and complex s, this returns the value of the principal branch of the modified versions $\tilde{D}_m, D_m$ and P_m of the polylogarithm $\mathrm{Li}_m(s)$; all of these satisfy functional equations of the form $f_m(1/s) = (-1)^m f_m(s)$. For their definition and main properties, see Zagier [Zag91]. PARI is used here.

25.5.2 Trigonometric Functions

The trigonometric functions may be computed for real and complex arguments or for power series defined over a real or complex field. The basic power series expansions are

$$\sin(z) = \sum_{n=0}^{\infty} \frac{(-1)^{n+1} z^{2n+1}}{(2n+1)!}, \quad \cos(z) = \sum_{n=0}^{\infty} \frac{(-1)^n z^{2n}}{(2n)!}.$$

Euler's formulas relate these with the exponential functions via

$$\sin(z) = \frac{e^{\mathrm{i}z} - e^{-\mathrm{i}z}}{2\mathrm{i}}, \quad \cos(z) = \frac{e^{\mathrm{i}z} + e^{-\mathrm{i}z}}{2}.$$

`Sin(f)`

Given a power series f defined over a real or complex field, return the power series $\sin(f)$.

`Sin(c)`

Given a real or complex number c, return the value $\sin(c)$.

`Cos(f)`

Given a power series f defined over a real or complex field, return the power series $\cos(f)$.

`Cos(c)`

Given a real or complex number c, return the value $\cos(c)$.

`Sincos(f)`

Given a power series f defined over a real or complex field, return the two power series $\sin(f)$ and $\cos(f)$.

`Sincos(s)`

Given a real or complex number s, return the two values $\sin(s)$ and $\cos(s)$.

`Tan(f)`

Given a power series f defined over the real or complex field, return the power series $\tan(f)$.

`Tan(c)`

Given a real or complex number c, return the value $\tan(c) = \frac{\sin(c)}{\cos(c)}$. Note that c should not be too close to one of the zeroes $(\pi/2 + n \cdot \pi)$ of $\cos(z)$.

`Cot(f)`

Given a power series f defined over a real or complex field having valuation zero, return the power series $\cot(f)$.

`Cot(c)`

Given a real or complex number c, return the value $\cot(c) = \cos(c)/\sin(c)$. Note that c should not be too close to one of the zeroes $n \cdot \pi$ of $\sin(z)$.

`Sec(f)`

Given a power series f defined over a real or complex field, return the power series $\sec(f)$.

`Sec(c)`

Given a real or complex number c, return the value $\sec(c) = 1/\cos(c)$. Note that c should not be too close to one of the zeroes $(\pi/2 + n \cdot \pi)$ of $\cos(z)$.

`Cosec(f)`

Given a power series f defined over a real or complex field having valuation zero, return the power series $\mathrm{cosec}(f)$.

`Cosec(c)`

Given a real or complex number c, return the value $\mathrm{cosec}(c) = 1/\sin(c)$. Note that c should not be too close to one of the zeroes $n \cdot \pi$ of $\sin(z)$.

25.5.3 Inverse Trigonometric Functions

The inverse trigonometric functions are all available for arbitrary real or complex arguments. The principal values are chosen as indicated.

We mention the power series expansions for the inverse of the sine and tangent functions (for $|z| \leq 1$):

$$\arcsin(z) = \sum_{n=0}^{\infty} \left(\prod_{k=1}^{2n} k^{(-1)^{k-1}}\right) \frac{z^{2n+1}}{2n+1},$$

$$\arctan(z) = \sum_{n=1}^{\infty} (-1)^n \frac{z^{2n+1}}{2n+1}.$$

The important relations with the logarithmic function include

$$\arcsin(z) = \frac{1}{\mathrm{i}} \log(\mathrm{i}z + \sqrt{1-z^2}),$$

$$\arccos(z) = \frac{1}{\mathrm{i}} \log(z + \sqrt{z^2-1}),$$

$$\arctan(z) = \frac{1}{2\mathrm{i}} \log\left(\frac{1+\mathrm{i}z}{1-\mathrm{i}z}\right).$$

`Arcsin(f)`

Given a power series f defined over a real or complex field. return the inverse sine of the power series f.

`Arcsin(r)`

Given a real or complex number s, return a value t such that $\sin(t) = s$. The principal value with real part in $[-\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless s is real and $-1 \leq s \leq 1$, in which case a free real number is returned.

`Arccos(f)`

Given a power series f defined over a real or complex field. return the inverse cosine of the power series f.

`Arccos(r)`

Given a real or complex number s, return a value t such that $\cos(t) = s$. The principal value with real part in $[0, \pi]$ is chosen. The return value is a complex number, unless s is real and $-1 \leq s \leq 1$, in which case a free real number is returned.

`Arctan(f)`

Given a power series f defined over the real or complex field, return the inverse tangent of the power series f.

`Arctan(r)`

Given a real or complex number s, return a value t such that $\tan(t) = s$. The principal value with real part in $(-\pi/2, \pi/2)$ is chosen. The return value is a complex number, unless s is real, in which case a free real number is returned.

`Arctan(x, y)`

`Arctan2(x, y)`

Given the real numbers x and y, return the value v of $\arctan(y/x)$ determined by the choice of signs for x and y. That is, the value v is chosen in $(-\pi, \pi)$ in such a way that the signs of x and $\sin(v)$ coincide, as well as the signs of y and $\cos(v)$. An error occurs if x and y are both zero; if y is zero and x non-zero, the value returned is $\text{sign}(x) \cdot \pi/2$.

The arguments are allowed to be in any real field (automatic coercion is used whenever necessary).

`Arccot(r)`

Given a real or complex number s, return a value t such that $\cot(t) = s$. The principal value with real part in $(-\pi/2, \pi/2)$ is chosen. The return value is a complex number, unless s is real, in which case a real number is returned.

`Arcsec(r)`

Given a real or complex number s, return a value t such that $\sec(t) = s$. The principal value with real part in $[0, \pi/2) \cup (\pi/2, \pi]$ is chosen. The return value is a complex number, unless s is real, in which case a real number is returned.

`Arccosec(r)`

Given a real or complex number s, return a value t such that $\text{cosec}(t) = s$. The principal value with real part in $[-\pi/2, 0) \cup (0, \pi/2]$ is chosen. The return value is a complex number, unless s is real, in which case a real number is returned.

25.5.4 Hyperbolic Functions

The hyperbolic functions are available for real and complex arguments, as specified below.

The hyperbolic functions are defined using

$$\sinh(z) = \frac{e^z - e^{-z}}{2},$$
$$\cosh(z) = \frac{e^z + e^{-z}}{2}.$$

`Sinh(f)`

Given a power series f defined over a real or complex field, return the hyperbolic sine of the power series f.

`Sinh(s)`

Given a real or complex number s, return $\sinh(s)$. The result will be a real or complex value, in accordance with the argument.

`Cosh(f)`

Given a power series f defined over a real or complex field, return the hyperbolic cosine of the power series f.

`Cosh(r)`

Given a real or complex number s, return $\cosh(s)$. The result will be a real or complex value, in accordance with the argument.

`Tanh(f)`

Given a power series f defined over a real or complex field, return the hyperbolic tangent of the power series f.

`Tanh(r)`

Given a real or complex number s, return $\tanh(s) = \frac{\sinh(s)}{\cosh(s)}$. The result will be a real or complex value, in accordance with the argument.

`Coth(r)`

Given a real or complex number s, return $\coth(s) = \frac{\cosh(s)}{\sinh(s)}$. The result will be a real or complex value, in accordance with the argument.

`Sech(r)`

Given a real or complex number s, return $\mathrm{sech}(s) = 1/\cosh(s)$. The result will be a real or complex value, in accordance with the argument.

`Cosech(r)`

Given a real or complex number s, return $\mathrm{cosech}(s) = 1/\sinh(s)$. The result will be a real or complex value, in accordance with the argument.

25.5.5 Inverse Hyperbolic Functions

The inverse hyperbolic functions are available for real or complex arguments. The principal values are chosen as indicated.

`Argsinh(f)`

Given a power series f defined over a real or complex field, return the inverse hyperbolic sine of the power series f.

`Argsinh(r)`

Given a real or complex number s, return t such that $\sinh(t) = s$; the principal value with imaginary part in $[\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless the argument is real, in which case a real number is returned.

`Argcosh(f)`

Given a power series f defined over a real or complex field, return the inverse hyperbolic cosine of the power series f.

`Argcosh(r)`

Given a real or complex number s, return t such that $\cosh(t)) = s$; the principal value with imaginary part in $[0, \pi]$ is chosen. The return value is a complex number, unless the argument is real and $s \geq 1$, in which case a real number is returned.

`Argtanh(f)`

Given a power series f defined over a real or complex field, return the inverse hyperbolic tangent of the power series f.

`Argtanh(s)`

Given a real or complex number s, return t such that $\tanh(t) = s$; the principal value with imaginary part in $[\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless the argument is real and $-1 < s < 1$, in which case a real number is returned.

`Argsech(s)`

Given a real or complex number s, return t such that $\mathrm{sech}(t)) = s$; the principal value with imaginary part in $[0, \pi]$ is chosen. The return value is a complex number, unless the argument is real and $|s| \geq 1$, in which case a real number is returned.

`Argcosech(s)`

Given a real or complex number s, return t such that $\mathrm{cosech}(t)) = s$; the principal value with imaginary part in $[-\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless the argument is real, in which case a real number is returned.

`Argcoth(s)`

Given a real or complex number s, return t such that $\coth(t)) = s$; the principal value with imaginary part in $[-\pi/2, \pi/2]$ is chosen. The return value is a complex number, unless the argument is real and $0 < s \leq 1$, in which case free real number is returned.

25.6 Elliptic and Modular Functions

More information on elliptic functions can be found for example in Chandrasekharan [Cha85], and for modular functions and their use see Koblitz [Kob84].

25.6.1 Eisenstein Series

Let $f(z)$ be a modular function. Then $f(z)$ may be written as a Fourier series

$$f(z) = \sum_{n \in \mathbf{Z}} a_n q^n,$$

where $q = e^{2\pi i z}$, which has at most finitely many nonzero coefficients a_n with $n < 0$. Such a Fourier expansion of a modular function is called its *q-expansion*. In this and the next section we present intrinsics for q-expansions of the Eisenstein series and the Weierstrass $\wp$-function.

Let z be a point in the upper half-plane and let L be a lattice in $\mathbf{C}$. The Eisenstein series are defined as the coefficients of the Laurent Series expansion of the Weierstrass $\wp$-function:

$$\wp(z, L) = \frac{1}{z^2} + \sum_{2 \leq k} G_k(L)(2k-1)z^{2k-2}$$

where $G_k(L)$ are the Eisenstein series. The normalization $E_{2n}(z) = \frac{1}{2\zeta(2n)} G_{2n}(z)$ ensures that $E_{2n}(z)$ has a *rational* q-expansion.

`Eisenstein(k, z)`

`Precision` RngIntElt *Default :*

Given a positive even integer $k = 2n$ and a complex power series z with positive valuation, return the q-expansion of the normalized Eisenstein series $E_{2n}(z)$. If z has finite precision this is the default for `Precision` otherwise the default precision of the parent of z is used.

`Eisenstein(k, t)`

Given a positive even integer $k = 2n$ and a point t in the upper half plane, return the value of $E_{2n}(z)$ at t.

`Eisenstein(k, L)`

Given a positive even integer $k = 2n$ and a lattice $L = [a, b]$ in the complex plane, return the value of the Eisenstein series $E_{2n}(z)$ relative to the lattice L.

`Eisenstein(k, F)`

Given a positive even integer $k = 2n$ and a binary quadratic form $F = ax^2 + bxy + cy^2$, return the value of the Eisenstein series $E_{2n}(z)$ at the point $\tau = \left(-b + \sqrt{b^2 - 4ac}\right)/(2a)$ where z is $e^{2\pi * \mathrm{i} * \tau}$.

Example H25E7

We compute the q-expansion for the normalized Eisenstein series $E_4(z)$.

```
> C<i> := ComplexField();
> R<z> := PowerSeriesRing(C);
> E4<q> := Eisenstein(4, z);
> E4;
1.0000000000000000000000000000 +
    240.0000000000000000000000000000*q +
    2160.0000000000000000000000000000*q^2 +
    6720.0000000000000000000000000000*q^3 +
    17520.0000000000000000000000000000*q^4 +
    30240.0000000000000000000000000000*q^5 +
    60480.0000000000000000000000000000*q^6 +
    82560.0000000000000000000000000000*q^7 +
    140400.0000000000000000000000000000*q^8 +
    181680.0000000000000000000000000000*q^9 +
    272160.0000000000000000000000000000*q^10 +
    319680.0000000000000000000000000000*q^11 +
    490560.0000000000000000000000000000*q^12 +
    527520.0000000000000000000000000000*q^13 +
    743040.0000000000000000000000000000*q^14 +
    846720.0000000000000000000000000000*q^15 +
    1123440.0000000000000000000000000000*q^16 +
    1179360.0000000000000000000000000000*q^17 +
    1635120.0000000000000000000000000000*q^18 +
    1646400.0000000000000000000000000000*q^19 + O(q^20)
```

We now evaluate this series at the point $z1 = 2.5 + i$. Since the expansion is in terms of q rather than z we first must calculate the point $q1$ that corresponds to $z1$.

```
> q1 := Exp(2*Pi(RealField())*i*(2.5 +i));
> Evaluate(E4, q1);
0.55930285285619077376676241 1942 +
3.67329046709782088758389413820E-31*i
```

If we are interested only in the value of E_4 at a single point, then we can compute it directly:

```
> Eisenstein(4, 2.5 + i);
```

```
0.55930285285619077376676241 1942 +
3.67329046709782088758389413820E-31*i
```

25.6.2 Weierstrass Series

`WeierstrassSeries(z, q)`

`Precision` RngIntElt *Default :*

Return a normalized q-expansion of the Weierstrass $\wp$-function:

$$\wp(z, L) = \frac{1}{z^2} + \sum_{2 \leq k} G_k(L)(2k-1)z^{2k-2}$$

where $G_k(L)$ are the Eisenstein series and

$$\texttt{WeierstrassSeries}(z, q) = (2\pi i)^{-2}\wp(q, z/(2\pi i))$$

Each term is an Eisenstein series, calculated to precision `Precision`, which is by default the precision of q.

`WeierstrassSeries(z, t)`

Given a complex power series z with positive valuation and a point $t = \tau$ in the upper-half complex plane, return the normalized q-expansion of the Weierstrass $\wp$-function. This is equivalent to evaluating the q-series expansion at $q = e^{2\pi i\tau}$.

`WeierstrassSeries(z, L)`

Given a complex power series z with positive valuation and a lattice $L = [a, b]$ in the complex plane, returns the normalized q-expansion of the Weierstrass $\wp$-function relative to the lattice L.

`WeierstrassSeries(z, F)`

Given a complex power series z with positive valuation and a binary quadratic form $F = ax^2 + bxy + cy^2$, this function returns the q-expansion of the Weierstrass $\wp$-function at $\tau = \left(-b + \sqrt{b^2 - 4ac}\right)/(2a)$

25.6.3 The Jacobi θ and Dedekind η-functions

The first Jacobi θ-function, $\theta(q,z)$, is defined by

$$\theta(q,z) = \frac{1}{\mathrm{i}} \sum_{n=-\infty}^{\infty} (-1)^n q^{(n+\frac{1}{2})^2} e^{(2n+1)\mathrm{i}z} = 2\sum_{n=0}^{\infty} (-1)^n q^{(n+\frac{1}{2})^2} \sin(2n+1)z.$$

Defined this way, θ satisfies $\theta(q,-z) = -\theta(q,z)$, it is periodic with period 2π in the second variable: $\theta(q,z+2\pi) = \theta(q,z)$, and its zeroes are of the form $m_1\pi + m_2\frac{\log x}{\mathrm{i}}$ for any integers m_1, m_2.

`JacobiTheta(q, z)`

For a real or complex number q satisfying $|q| < 1$, return the first of Jacobi's theta functions $\theta(q,z)$ as a power series expansion in z, a series over the complex numbers. PARI is used here.

`JacobiTheta(q, z)`

For real or complex numbers q, z satisfying $|q| < 1$, return the value of $\theta(q,z)$, the first of Jacobi's theta functions. PARI is used here.

`JacobiThetaNullK(q, k)`

For integer $k \geq 0$, return the k-th derivative $\theta^{(k)}(q,0)$ of $\theta(q,z)$ at $z = 0$. PARI is used here.

`DedekindEta(z)`

Given a complex power series z with positive valuation, return the q-expansion of Dedekind's η-function. Note that the unnormalized series is returned, that is, the factor $q^{1/24}$ is *not* removed. See [Lan87].

`DedekindEta(s)`

For complex argument s with positive imaginary part, this returns the actual value of Dedekind's η-function which is defined by

$$\eta(s) = e^{\frac{2\pi\mathrm{i}s}{24}} \left(1 + \sum_{n=1}^{\infty} (-1)^n (q^{n(3n-1)/2} + q^{n(3n+1)/2})\right)$$

where $q = e^{2\pi\mathrm{i}s}$.

25.6.4 The j-invariant and the Discriminant

The discriminant of the elliptic curve corresponding to the complex lattice L_τ, spanned by 1 and τ is given by

$$\Delta(\tau) = q\left(1 + \sum_{n=1}^{\infty}(-1)^n(q^{n(3n-1)/2} + q^{n(3n+1)/2})\right)$$

where $q = e^{2\pi i\tau}$.

`jInvariant(q)`

Given a power series q over a real or complex field with positive valuation, return the q-expansion of the elliptic j-invariant. The expansion begins with

$$j(q) = q^{-1} + 744 + 196884q + \cdots.$$

Note that:

$$j(q) = \frac{E_4(q)^3}{\Delta(q)}$$

where $E_4(q) =$ `Eisenstein(4, q)` and $\Delta(q) =$ `Delta(q)`.

`jInvariant(s)`

For complex argument s with positive imaginary part, this returns the value of the elliptic j-invariant at s. This is a modular function of weight 0 whose Fourier expansion starts with

$$j(s) = e^{-2\pi is} + 744 + 196884e^{2\pi is} + \cdots.$$

`jInvariant(L)`

Given a lattice $L = [a, b]$ in the complex plane, this function returns the value of the elliptic j-invariant of L. This is the j-invariant of τ where $\tau = a/b$ or $\tau = b/a$, whichever is in the upper half complex plane.

`jInvariant(F)`

For a binary quadratic form $F = ax^2 + bxy + cy^2$ with negative discriminant, this returns the elliptic j-invariant of F. This is the j-invariant of τ where $\tau = \left(-b + \sqrt{b^2 - 4ac}\right)/(2a)$.

`Delta(z)`

Given a complex power series z, this function returns a q-series expansion of the discriminant $\Delta(z)$.

`Delta(t)`

Given a point t in the upper half plane, return the q-series expansion of the discriminant $\Delta(q)$ evaluated at $q = e^{2\pi i t}$.

`Delta(L)`

Given a pair $L = [a,b]$ of complex numbers generating a lattice in $\mathbf{C}$, return the q-series expansion of the discriminant $\Delta(q)$ evaluated at $q = e^{2\pi i \tau}$ where $\tau = a/b$ or $\tau = b/a$, whichever is in the upper half complex plane.

25.6.5 Weber's Functions

`WeberF(s)`

For complex argument s in the upper half-plane, this returns the value of Weber's function f, defined in such a way that

$$j(s) = \frac{(f(s)^{24} - 16)^3}{f(s)^{24}}.$$

`WeberF2(g)`

For a complex power series g having positive valuation, this function returns the q-expansion of Weber's f_2 function

$$f_2(x) = \frac{\eta(2x)\sqrt{2}}{\eta(x)}$$

defined in such a way that

$$j(s) = \frac{(f_2(s)^{24} + 16)^3}{f_2(s)^{24}}.$$

`WeberF1(s)`
`WeberF2(s)`

For complex number s lying in the upper half-plane, these return the value of Weber's functions f_1 and f_2, defined in such a way that

$$j(s) = \frac{(f_{1/2}(s)^{24} + 16)^3}{f_{1/2}(s)^{24}}.$$

In fact, f_2 is as defined above and

$$f_1(x) = f_2(-1/x) = \eta(x/2)/\eta(x)$$

Example H25E8

We compute the q-expansion for the Weber function $f_2(z)$.

```
> C<i> := ComplexField();
> R<x> := PowerSeriesRing(C);
> f2<q> := WeberF2(x);
> f2;
1.41421356237309504880168872421 +
    (1.41421356237309504880168872421 +
    0.37024024484653052058465674917 2*i)*q +
    (1.36574922765338060759226121771 +
    0.37024024484653052058465674917 2*i)*q^2 +
    (2.77996279002647565639394994192 +
    0.36601093379329241948227297708 1*i)*q^3 +
    (2.78023959778761313408864734217 +
    0.73625117863982294006692972625 3*i)*q^4 +
    (4.14598882544099374168090855987 +
    0.73626567226030370903683781952 8*i)*q^5 +
    (5.56020175541059398072755542234 +
    1.10227660605359612851911079661*i)*q^6 +
    ...
```

25.7 Theta Functions

One of the main tools for working with analytic Jacobians is the theta function. For instance it is used by FromAnalyticJacobian on page 4209 and RosenhainInvariants on page 4216. For $c \in \mathbf{R}^{2g}$ let c' be the first g entries and c'' the second g entries of c. For such a c, $z \in \mathbf{C}^g$ and τ an element of Siegel upper half-space the classical multi-variable theta function is defined by

$$\theta[c](z,\tau) = \sum_{m \in \mathbf{Z}^g} \exp(\pi i^t (m+c')\tau(m+c') + 2\pi i^t (m+c')(z+c'')).$$

The vector c is called the characteristic of the theta function.

`Theta(char, z, tau)`

This computes the multidimensional theta function with characteristic *char* (a $2g \times 1$ matrix) at z (a $g \times 1$ matrix) and τ (a symmetric $g \times g$ matrix with positive definite imaginary part).

`Theta(char, z, A)`

This computes the multidimensional theta function with characteristic *char* (a $2g \times 1$ matrix) at z (a $g \times 1$ matrix) and τ, the small period matrix of the analytic Jacobian A. This function caches the values of theta null values ($z = 0$) at half-integer characteristics.

25.8 Gamma, Bessel and Associated Functions

As a general reference to the functions described in this section (and much more), we refer the reader to Whittaker and Watson [WW15].

`Gamma(f)`

Return the Gamma function $\Gamma(f)$ of the series f. f must be defined over the free real or complex field, the valuation of f must be 0 and the constant term of f must be 1.

`Gamma(r)`

`Gamma(r)`

Given a real or complex number s (not equal to $0, -1, -2, \ldots$), calculate the value $\Gamma(s)$ of the *gamma function* at s. For s with positive real part this is the value of

$$\Gamma(s) = \int_0^\infty u^{s-1} e^{-u} du.$$

For other s (not a non-positive integer) the function is defined by analytic continuation, and it satisfies the product formula

$$\frac{1}{s\Gamma(s)} = e^{\gamma s} \prod_{n=1}^{\infty} (1 + \frac{s}{n}) e^{-s/n}.$$

The function Γ also satisfies

$$\Gamma(s)\Gamma(1-s) = \frac{\pi}{\sin(\pi s)},$$

and

$$\Gamma(s+1) = s\Gamma(s).$$

`Gamma(r, s)`

`Complementary`	BoolElt	*Default :* `false`
`Gamma`	FldReElt	*Default :*

For real numbers s, t this returns the value of the incomplete gamma function

$$\gamma(s,t) = \int_0^t u^{s-1} e^{-u} du.$$

The optional argument `Complementary` can be used to find the complement

$$\int_t^\infty u^{s-1} e^{-u} du$$

instead. There is a second optional argument that may be used in the computation of the incomplete gamma value; the free real value of `Gamma` should be the value of $\Gamma(s)$, in which case $\gamma(s,t)$ may be computed as the difference between the given value for $\Gamma(s)$ and that of the complementary γ at s, t. Pari is used here.

`GammaD(s)`

For free real s (such that $s + \frac{1}{2}$ is not a non-positive integer) this returns the value of $\Gamma(s + \frac{1}{2})$. For integer values of s this is faster than `Gamma(s+(1/2))`, because Legendre's doubling formula

$$\Gamma(s + \frac{1}{2}) = 2^{1-2s}\sqrt{\pi}\frac{\Gamma(2s)}{\Gamma(s)}$$

is used. PARI is used here.

`LogGamma(f)`

Return the Log-Gamma function $\mathrm{Log}(\Gamma(f))$ of the series f. f must be defined over a real or complex field, the valuation of f must be 0 and the constant term of f must be 1.

`LogGamma(r)`

For real or complex s (not a non-positive integer) return the value of the principal branch of the logarithm of the gamma function of s.

`LogDerivative(s)`

`Psi(s)`

For real or complex s (not a non-positive integer) return the principal value of the logarithmic derivative

$$\Psi(s) = \frac{d\log\Gamma(s)}{ds} = \frac{\Gamma'(s)}{\Gamma(s)},$$

of the gamma function, which allows the expansion

$$\Psi(s) = -\gamma - \frac{1}{s} + s\sum_{n=1}^{\infty}\frac{1}{n(s+n)};$$

here γ is Euler's gamma. PARI is used here.

`BesselFunction(n, r)`

Given a small integer n and a real number r, calculate the value of the *Bessel function* $y = J_n(r)$, of the first kind of order n. Results for negative arguments are defined by: $J_{-n}(r) = J_n(-r) = (-1)^n J_n(r)$. The Bessel function of the first kind of order n is defined by

$$J_n(x) = \frac{1}{2\pi\mathrm{i}}\left(\frac{z}{2}\right)^n \int_{-\infty}^{0^+} u^{-n-1}e^{u-\frac{z^2}{4t}}\,du,$$

and satisfies

$$J_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k z^{n+2k}}{2^{n+2k} k! \Gamma(n+k+1)}.$$

`BesselFunctionSecondKind(n, r)`

Given a small integer n and a real number r, calculate the value of the *Bessel function* $y = Y_n(r)$, of the second kind of order n. Results for negative arguments are defined by: $Y_{-n}(r) = -(-1)^n Y_n(r)$, $Y_n(-r)$ is not a real number. The Bessel function of the second kind of order n satisfies the Bessel differential equation.

`JBessel(n, s)`

Given a small integer n and a real number s, calculate the value of the *Bessel function* of the first kind of half integral index $n + \frac{1}{2}$, $J_{n+\frac{1}{2}}$, defined as above. PARI is used here.

`KBessel(n, s)`

`KBessel2(n, s)`

Given a complex n and a positive real s, compute the value of the *modified Bessel function of the second kind* $K_n(s)$, which may be defined by

$$K_n(s) = \frac{\pi}{2} \left(\mathrm{i}^n J_{-n}(\mathrm{i}s) - \mathrm{i}^{-n} J_n(s) \right) \cot(n\pi).$$

The function `KBessel2` is an alternative (often faster) implementation of this function. PARI is used here.

25.9 The Hypergeometric Function

For more information on the Hypergeometric Series, see Husemöller [Hus87], page 176.

`HypergeometricSeries(a,b,c, z)`

Return the hypergeometric series $F(a, b, c; z)$ defined by

$$F(a, b, c; z) = \sum_{0 \le n} \frac{(a)_n (b)_n}{n! (c)_n z^n}$$

where $(a)_n = a(a+1)\cdots(a+n-1)$.

`HypergeometricU(a, b, s)`

For positive real s and complex arguments a and b this function returns the value of the confluent hypergeometric function $U(a, b, s)$. This can be defined by

$$U(a, b, s) = \frac{1}{\Gamma(a)} \int_{u=0}^{\infty} e^{-su} u^{a-1} (1+u)^{b-a-1)} du.$$

PARI is used here.

25.10 Other Special Functions

`ArithmeticGeometricMean(x, y)`

`AGM(f, g)`

Return the hyperbolic arithmetic-geometric mean of the series f and g defined over a field. The valuations of f and g must be equal.

`ArithmeticGeometricMean(x, y)`

`AGM(x, y)`

Returns the arithmetic-geometric mean of the real or complex numbers x and y, defined as the limit of either of the sequences x_i, y_i where $x_0 = x$, $y_0 = y$ and $x_{i+1} = (x_i + y_i)/2, y_{i+1} = \sqrt{x_i y_i}$. The function calculates both sequences, and when the numbers are within the desired precision of each other, it returns one of them.

`BernoulliNumber(n)`

For a non-negative integer n, return the value of the n-th Bernoulli number B_n, defined by

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!}.$$

`BernoulliApproximation(n)`

For a non-negative integer n, return an approximation in the field of real numbers to the value of the n-th Bernoulli number B_n, defined by

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!}.$$

`DawsonIntegral(r)`

Given a real number r, compute the value of *Dawson's integral*,

$$e^{-x^2} \cdot \int_0^x e^{u^2} du,$$

at $x = r$. The mp real package is used here.

`ErrorFunction(r)`

`Erf(r)`

Given a real number r, calculate the value of the *error function* erf. This is the value of

$$\sqrt{\frac{4}{\pi}} \cdot \int_0^x e^{-u^2} du,$$

at $x = r$ for $r > 0$, and for $r < 0$ it is defined by $\mathrm{erf}(x) = -\mathrm{erf}(-x)$, while $\mathrm{erf}(0) = 0$.

`ComplementaryErrorFunction(r)`

`Erfc(r)`

Given a real number r, calculate the value of the *complementary error function*. This is the value of $y = \operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$. for the error function erf as defined above.

`ExponentialIntegral(r)`

Given a real number r, calculate the value of the *exponential integral*, that is, the principal value of

$$\int_{-\infty}^{x} \frac{e^u}{u} du$$

at $x = r$.

`ExponentialIntegralE1(r)`

Given a real number r, calculate the value of the *exponential integral E1*, that is, the principal value of

$$\int_{x}^{\infty} \frac{e^u}{u} du$$

at $x = r$.

`LogIntegral(r)`

Given a non-negative real number r that is not equal to 1, evaluate the *logarithmic integral* $y = \operatorname{li}(x)$ at $x = r$. This integral is defined to be the principal value of

$$\int_{0}^{x} \frac{1}{\log(u)} du.$$

The mp real package is used here.

`ZetaFunction(s)`

`ZetaFunction(R, n)`

These functions calculate values of the Riemann ζ-function, which is the analytic continuation of

$$\zeta(z) = \sum_{i=1}^{\infty} \frac{1}{i^z}$$

(convergent for $\operatorname{Re}(z) > 1$). The version with one argument takes a real or complex number $r \neq 1$ and returns a real or complex number.

The version with two arguments is much more restricted; it takes a real field R and an integer $n \neq 1$, and returns $\zeta(n)$ in R.

MPFR uses the algorithm of Jean-Luc Rémy and Sapphorain Pétermann [PR06].

25.11 Numerical Functions

This section contains some functions for numerical analysis, taken from Pari.

25.11.1 Summation of Infinite Series

There are three functions for evaluating infinite sums of real numbers. The sum should be specified as a map m from the integers to the real field, such that $m(n)$ is the n^{th} term of the sum. The summation begins at term i. The precision of the result will be the default precision of the real field.

`InfiniteSum(m, i)`

An approximation to the infinite sum $m(i)+m(i+1)+m(i+2)+\cdots$. This function also works for maps to the complex field.

`PositiveSum(m, i)`

An approximation to the infinite sum $m(i)+m(i+1)+m(i+2)+\cdots$. Designed for series in which every term is positive, it uses van Wijngaarden's trick for converting the series into an alternating one. Due to the stopping criterion, terms equal to 0 will create problems and should be removed.

`AlternatingSum(m, i)`

`Al`	MonStgElt	*Default :* "*Villegas*"

An approximation to the infinite sum $m(i)+m(i+1)+m(i+2)+\cdots$. Designed for series in which the terms alternate in sign. The optional argument `Al` can be used to specify the algorithm used. The possible values are `"Villegas"` (the default), and `"EulerVanWijngaarden"`. Due to the stopping criterion, terms equal to 0 will create problems and should be removed.

25.11.2 Integration

A number of 'Romberg-like' integration methods have been taken from Pari. The precision should not be made too large for this, and singularities are not allowed in the interval of integration (including its boundaries).

`Interpolation(P, V, x)`

Using Neville's algorithm, interpolate the value of x under a polynomial p such that $p(P[i]) = V[i]$. An estimate of the error is also returned.

`RombergQuadrature(f, a, b: parameters)`

`Precision`	FldReElt	*Default :* $1.0e-6$
`MaxSteps`	RngIntElt	*Default :* 20
`K`	RngIntElt	*Default :* 5

Using Romberg's method of order $2K$, approximate the integral of f from a to b. The desired accuracy may be specified by setting the `Precision` parameter, and the order of the algorithm by changing `K`. The algorithm ceases after `MaxSteps` iterations if the desired accuracy has not been achieved.

`SimpsonQuadrature(f, a, b, n)`

Using Simpson's rule on n sub-intervals, approximate the integral of f from a to b.

`TrapezoidalQuadrature(f, a, b, n)`

Using the trapezoidal rule on n sub-intervals, approximate the integral of f from a to b.

25.11.3 Numerical Derivatives

There is also a function to compute the NumericalDerivative of a function. This works via computing enough interpolation points and using a Taylor expansion.

`NumericalDerivative(f, n, z)`

Given a suitably nice function f, compute a numerical approximation to the nth derivative at the point z.

Example H25E9

```
> f := func<x|Exp(2*x)>;
> NumericalDerivative(f, 10, ComplexField(30)! 1.0) / f (1.0);
1024.000000000000000000000000000
> NumericalDerivative(func<x|LogGamma(x)>,1,ComplexField()!3.0);
0.92278433509846713939348790991 8
> Psi(3.0); // Psi is Gamma'/Gamma
0.92278433509846713939348790991 8
```

25.12 Bibliography

[**Cha85**] K. Chandrasekharan. *Elliptic Functions*, volume 281 of *Grundlehren der mathematischen Wissenschaften*. Springer, Berlin, 1985.

[**Hus87**] Dale Husemöller. *Elliptic Curves*, volume 111 of *Graduate Texts in Mathematics*. Springer, New York, 1987.

[**Kob84**] Neal Koblitz. *Introduction to Elliptic Curves and Modular Forms*, volume 97 of *Graduate Texts in Mathematics*. Springer, New York, 1984.

[**Lan87**] Serge Lang. *Elliptic Functions*, volume 112 of *Graduate Texts in Mathematics*. Springer, New York, 1987.

[**Lew81**] Leonard Lewin. *Polylogarithms and associated functions*. North Holland, New York, 1981.

[**PR06**] Y.-F. S. Pétermann and Jean-Luc Rémy. Arbitrary Precision Error Analysis for computing $\zeta(s)$ with the Cohen-Olivier algorithm: Complete description of the real case and preliminary report on the general case. Research Report 5852, INRIA, 2006. URL:http://www.inria.fr/rrrt/rr-5852.html.

[**Sch82**] A. Schönhage. The fundamental theorem of algebra in terms of computational complexity. Technical report, Univ. Tübingen, 1982.

[**vdGOS91**] G. van der Geer, F. Oort, and J. Steenbrink, editors. *Arithmetic Algebraic Geometry*, volume 89 of *Progress in Mathematics*, Basel, 1991. Birkhäuser Verlag.

[**WW15**] E. T. Whittaker and G. N. Watson. *A course of modern analysis.* Cambridge University Press, Cambridge, 2nd edition, 1915.

[**Zag91**] Don Zagier. Polylogarithms, Dedekind Zeta Functions, and the Algebraic K-Theory of Fields. In van der Geer et al. [vdGOS91], pages 377–390.

PART IV
MATRICES AND LINEAR ALGEBRA

26 MATRICES

Chapter 26
MATRICES

26.1 Introduction

This chapter describes all the basic operations available for creating and working with matrices. Matrices arise in many different contexts and there are several types of matrix within MAGMA, but most of the operations listed here apply to all types of matrix.

The parent of any matrix will be one of several types of matrix-structure (module, matrix algebra, matrix group, etc.), and each of these matrix-structure types are described in other chapters, together with operations peculiar to their elements.

26.2 Creation of Matrices

This section describes the elementary constructs provided for creating a matrix or vector. For each of the following functions, the parent of the result will be as follows:

(a) If the result is a vector then its parent will be the appropriate R-space (of type `ModTupRng` or `ModTupFld`).

(b) If the result is a square matrix then its parent will be the appropriate matrix algebra (of type `AlgMatElt`).

(c) If the result is a non-square matrix then its parent will be the appropriate R-matrix space (of type `ModMatRng` or `ModMatFld`).

A matrix or a vector may also be created by coercing a sequence of ring elements into the appropriate parent matrix structure. There is also a virtual type `Mtrx` and all matrix types inherit from `Mtrx`. While writing package intrinsics, an argument should be declared to be of type `Mtrx` if it is a general matrix.

26.2.1 General Matrix Construction

`Matrix(R, m, n, Q)`

Given a ring R, integers $m, n \geq 0$ and a sequence Q, return the $m \times n$ matrix over R whose entries are those specified by Q, coerced into R. Either of m and n may be 0, in which case Q must have length 0 (and may even be null), and the $m \times n$ zero matrix over R is returned. There are several possibilities for Q:

(a) The sequence Q may be a sequence of length mn containing elements of a ring S, in which case the entries are given in row-major order. In this case, the function is equivalent to `MatrixRing(R, n)!Q` if $m = n$ and `RMatrixSpace(R, m, n)!Q` otherwise.

(b) The sequence Q may be a sequence of tuples, each of the form `<i, j, x>`, where $1 \leq i \leq m$, $1 \leq j \leq n$, and $x \in S$ for some ring S. Such a tuple specifies that

the (i, j)-th entry of the matrix is x. If an entry position is not given then its value is zero, while if an entry position is repeated then the last value overrides any previous value(s). This case is useful for creating sparse matrices.

(c) The sequence Q may be a sequence of m sequences, each of length n and having entries in a ring S, in which case the rows of the matrix are specified by the inner sequences.

(d) The sequence Q may be a sequence of m vectors, each of length n and having entries in a ring S, in which case the rows of the matrix are specified by the vectors.

Example H26E1

This example demonstrates simple ways of creating matrices using the general `Matrix(R, m, n, Q)` function.

(a) Defining a 2×2 matrix over $\mathbf{Z}$:

```
> X := Matrix(IntegerRing(), 2, 2, [1,2, 3,4]);
> X;
[1 2]
[3 4]
> Parent(X);
Full Matrix Algebra of degree 2 over Integer Ring
```

(b) Defining a 2×3 matrix over $\mathbf{F}_{23}$:

```
> X := Matrix(GF(23), 2, 3, [1,-2,3, 4,100,-6]);
> X;
[ 1 21  3]
[ 4  8 17]
> Parent(X);
Full KMatrixSpace of 2 by 3 matrices over GF(23)
```

(c) Defining a sparse 5×10 matrix over $\mathbf{Q}$:

```
> X := Matrix(RationalField(), 5, 10, [<1,2,23>, <3,7,11>, <5,10,-1>]);
> X;
[ 0 23  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0 11  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0 -1]
> Parent(X);
Full KMatrixSpace of 5 by 10 matrices over Rational Field
```

(c) Defining a sparse 10×10 matrix over $\mathbf{F}_{101}$:

```
> X := Matrix(GF(101), 10, 10, [<2*i-1, 2*j-1, i*j>: i, j in [1..5]]);
> X;
[  1   0   2   0   3   0   4   0   5   0]
[  0   0   0   0   0   0   0   0   0   0]
```

```
[  2   0   4   0   6   0   8   0  10   0]
[  0   0   0   0   0   0   0   0   0   0]
[  3   0   6   0   9   0  12   0  15   0]
[  0   0   0   0   0   0   0   0   0   0]
[  4   0   8   0  12   0  16   0  20   0]
[  0   0   0   0   0   0   0   0   0   0]
[  5   0  10   0  15   0  20   0  25   0]
[  0   0   0   0   0   0   0   0   0   0]
> Parent(X);
Full Matrix Algebra of degree 10 over GF(101)
```

26.2.2 Shortcuts

The following functions are "shortcut" versions of the previous general creation function, where some of the arguments are omitted since they can be inferred by MAGMA.

`Matrix(m, n, Q)`

Given integers $m, n \geq 0$ and a sequence Q of length mn containing elements of a ring R, return the $m \times n$ matrix over R whose entries are the entries of Q, in row-major order. Either of m and n may be 0, in which case Q must have length 0 and some universe R. This function is equivalent to `MatrixRing(Universe(Q), n)!Q` if $m = n$ and `RMatrixSpace(Universe(Q), m, n)!Q` otherwise.

`Matrix(m, n, Q)`

Given integers m and n, and a sequence Q consisting of m sequences, each of length n and having entries in a ring R, return the $m \times n$ matrix over R whose rows are given by the inner sequences of Q.

`Matrix(Q)`

Given a sequence Q of m vectors, each of length n over a ring R, return the $m \times n$ matrix over R whose rows are the entries of Q.

`Matrix(R, n, Q)`

Given a ring R, an integer $n \geq 0$ and a sequence Q of length l containing elements of a ring S, such that n divides l, return the $(l/n) \times n$ matrix over R whose entries are the entries of Q, coerced into R, in row-major order. The argument n may be 0, in which case Q must have length 0 (and may even be null), in which case the 0×0 matrix over R is returned. This function is equivalent to `MatrixRing(R, n)!Q` if $l = n^2$ and `RMatrixSpace(R, #Q div n, n)!Q` otherwise.

`Matrix(n, Q)`

Given an integer $n \geq 0$ and a sequence Q of length l containing elements of a ring R, such that n divides l, return the $(l/n) \times n$ matrix over R whose entries are the entries of Q, in row-major order. The argument n may be 0, in which case Q must have length 0 and some universe R, in which case the 0×0 matrix over R is returned. This function is equivalent to `MatrixRing(Universe(Q), n)!Q` if $l = n^2$ and `RMatrixSpace(Universe(Q), #Q div n, n)!Q` otherwise.

`Matrix(Q)`

Given a sequence Q consisting of m sequences, each of length n and having entries in a ring R, return the $m \times n$ matrix over R whose rows are given by the inner sequences of Q.

`Matrix(R, Q)`

Given a sequence Q consisting of m sequences, each of length n and having entries in a ring S, return the $m \times n$ matrix over R whose rows are given by the inner sequences of Q, with the entries coerced into R.

Example H26E2

The first matrix in the previous example may be created thus:

```
> X := Matrix(2, [1,2, 3,4]);
> X;
[1 2]
[3 4]
> X := Matrix([[1,2], [3,4]]);
> X;
[1 2]
[3 4]
```

The second matrix in the previous example may be created thus:

```
> X := Matrix(GF(23), 3, [1,-2,3, 4,100,-6]);
> X;
[ 1 21  3]
[ 4  8 17]
> Parent(X);
Full KMatrixSpace of 2 by 3 matrices over GF(23)
> X := Matrix(GF(23), [[1,-2,3], [4,100,-6]]);
> X;
[ 1 21  3]
[ 4  8 17]
> X := Matrix([[GF(23)|1,-2,3], [4,100,-6]]);
> X;
[ 1 21  3]
[ 4  8 17]
```

26.2.3 Construction of Structured Matrices

`ZeroMatrix(R, m, n)`

Given a ring R and integers $m, n \geq 0$, return the $m \times n$ zero matrix over R.

`ScalarMatrix(n, s)`

Given an integer $n \geq 0$ and an element s of a ring R, return the $n \times n$ scalar matrix over R which has s on the diagonal and zeros elsewhere. The argument n may be 0, in which case the 0×0 matrix over R is returned. This function is equivalent to `MatrixRing(Parent(s), n)!s`.

`ScalarMatrix(R, n, s)`

Given a ring R, an integer $n \geq 0$ and an element s of a ring S, return the $n \times n$ scalar matrix over R which has s, coerced into R, on the diagonal and zeros elsewhere. n may be 0, in which case in which case the 0×0 matrix over R is returned. This function is equivalent to `MatrixRing(R, n)!s`.

`DiagonalMatrix(R, n, Q)`

Given a ring R, an integer $n \geq 0$ and a sequence Q of n ring elements, return the $n \times n$ diagonal matrix over R whose diagonal entries correspond to the entries of Q, coerced into R.

`DiagonalMatrix(R, Q)`

Given a ring R and a sequence Q of n ring elements, return the $n \times n$ diagonal matrix over R whose diagonal entries correspond to the entries of Q, coerced into R.

`DiagonalMatrix(Q)`

Given a sequence Q of n elements from a ring R, return the $n \times n$ diagonal matrix over R whose diagonal entries correspond to the entries of Q.

`Matrix(A)`

Given a matrix A of any type, return the same matrix but having as parent the appropriate matrix algebra if A is square, or the appropriate R-matrix space otherwise. This is useful, for example, if it is desired to convert a matrix group element or a square R-matrix space element to be an element of a general matrix algebra.

`LowerTriangularMatrix(Q)`

Given a sequence Q of length l containing elements of a ring R, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ lower-triangular matrix F over R such that the entries of Q describe the lower triangular part of F, in row major order.

`LowerTriangularMatrix(R, Q)`

Given a ring R and a sequence Q of length l containing elements of a ring S, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ lower-triangular matrix F over R such that the entries of Q, coerced into R, describe the lower triangular part of F, in row major order.

`UpperTriangularMatrix(Q)`

Given a sequence Q of length l containing elements of a ring R, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ upper-triangular matrix F over R such that the entries of Q describe the upper triangular part of F, in *row major* order.

`UpperTriangularMatrix(R, Q)`

Given a ring R and a sequence Q of length l containing elements of a ring S, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ upper-triangular matrix F over R such that the entries of Q, coerced into R, describe the upper triangular part of F, in *row major* order.

`SymmetricMatrix(Q)`

Given a sequence Q of length l containing elements of a ring R, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ symmetric matrix F over R such that the entries of Q describe the lower triangular part of F, in row major order. This function allows the creation of symmetric matrices without the need to specify the redundant upper triangular part.

`SymmetricMatrix(R, Q)`

Given a ring R and a sequence Q of length l containing elements of a ring S, such that $l = \binom{n+1}{2} = n(n+1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ symmetric matrix F over R such that the entries of Q, coerced into R, describe the lower triangular part of F, in row major order. This function allows the creation of symmetric matrices without the need to specify the redundant upper triangular part.

`AntisymmetricMatrix(Q)`

Given a sequence Q of length l containing elements of a ring R, such that $l = \binom{n}{2} = n(n-1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ antisymmetric matrix F over R such that the entries of Q describe the proper lower triangular part of F, in row major order. The diagonal of F is zero and the proper upper triangular part of F is the negation of the proper lower triangular part of F.

`AntisymmetricMatrix(R, Q)`

Given a ring R and a sequence Q of length l containing elements of a ring S, such that $l = \binom{n}{2} = n(n-1)/2$ for some integer n (so l is a triangular number), return the $n \times n$ antisymmetric matrix F over R such that the entries of Q, coerced into R, describe the proper lower triangular part of F, in row major order.

`PermutationMatrix(R, Q)`

Given a ring R and a sequence Q of length n, such that Q is a permutation of $[1, 2, \ldots, n]$, return the n by n permutation matrix over R corresponding Q.

`PermutationMatrix(R, x)`

Given a ring R and a permutation x of degree n, return the n by n permutation matrix over R corresponding x.

Example H26E3

This example demonstrates ways of creating special matrices.
(a) Defining a 3×3 scalar matrix over $\mathbf{Z}$:

```
> S := ScalarMatrix(3, -4);
> S;
[-4  0  0]
[ 0 -4  0]
[ 0  0 -4]
> Parent(S);
Full Matrix Algebra of degree 3 over Integer Ring
```

(b) Defining a 3×3 diagonal matrix over $\mathbf{F}_{23}$:

```
> D := DiagonalMatrix(GF(23), [1, 2, -3]);
> D;
[ 1  0  0]
[ 0  2  0]
[ 0  0 20]
> Parent(D);
Full Matrix Algebra of degree 3 over GF(23)
```

(c) Defining a 3×3 symmetric matrix over $\mathbf{Q}$:

```
> S := SymmetricMatrix([1, 1/2,3, 1,3,4]);
> S;
[  1 1/2   1]
[1/2   3   3]
[  1   3   4]
> Parent(S);
Full Matrix Algebra of degree 3 over Rational Field
```

(d) Defining $n \times n$ lower- and upper-triangular matrices for various n:

```
> low := func<n | LowerTriangularMatrix([i: i in [1 .. Binomial(n + 1, 2)]])>;
```

```
> up := func<n | UpperTriangularMatrix([i: i in [1 .. Binomial(n + 1, 2)]])>;
> sym := func<n | SymmetricMatrix([i: i in [1 .. Binomial(n + 1, 2)]])>;
> low(3);
[1 0 0]
[2 3 0]
[4 5 6]
> up(3);
[1 2 3]
[0 4 5]
[0 0 6]
> sym(3);
[1 2 4]
[2 3 5]
[4 5 6]
> up(6);
[ 1  2  3  4  5  6]
[ 0  7  8  9 10 11]
[ 0  0 12 13 14 15]
[ 0  0  0 16 17 18]
[ 0  0  0  0 19 20]
[ 0  0  0  0  0 21]
```

26.2.4 Construction of Random Matrices

`RandomMatrix(R, m, n)`

Given a *finite* ring R and positive integers m and n, construct a random $m \times n$ matrix over R.

`RandomUnimodularMatrix(M, n)`

Given positive integers M and n, construct a random integral $n \times n$ matrix having determinant 1 or -1. Most entries will lie in the range $[-M, M]$.

`RandomSLnZ(n, k, l)`

A random element of $SL_n(\mathbf{Z})$, obtained by multiplying l random matrices of the form $I + E$, where E has exactly one nonzero entry, which is off the diagonal and has absolute value at most k.

`RandomGLnZ(n, k, l)`

A random element of $GL_n(\mathbf{Z})$, obtained in a similar way to `RandomSLnZ`.

`RandomSymplecticMatrix(g, m)`

Given positive integers n and m, construct a (somewhat) random $2n \times 2n$ symplectic matrix over the integers. The entries will have the same order of magnitude as `m`.

26.2.5 Creating Vectors

`Vector(n, Q)`

Given an integer n and a sequence Q of length n containing elements of a ring R, return the vector of length n whose entries are the entries of Q. The integer n may be 0, in which case Q must have length 0 and some universe R. This function is equivalent to `RSpace(Universe(Q), n)!Q`.

`Vector(Q)`

Given a sequence Q of length l containing elements of a ring R, return the vector of length l whose entries are the entries of Q. The argument Q may have length 0 if it has a universe R (i.e., it may not be null). This function is equivalent to `RSpace(Universe(Q), #Q)!Q`.

`Vector(R, n, Q)`

Given a ring R, an integer n, and a sequence Q of length n containing elements of a ring S, return the vector of length n whose entries are the entries of Q, coerced into R. The integer n may be 0, in which case Q must have length 0 (and may even be null). This function is equivalent to `RSpace(R, n)!Q`.

`Vector(R, Q)`

Given a ring R and a sequence Q of length l containing elements of a ring S, return the vector of length l whose entries are the entries of Q, coerced into R. The argument Q may have length 0 and may be null. This function is equivalent to `RSpace(R, #Q)!Q`.

26.3 Elementary Properties

The following functions yield elementary properties of matrices and may be applied to matrices of any type, including vectors.

`NumberOfRows(A)`
`Nrows(A)`

Given an $m \times n$ matrix A, return m, the number of rows of A.

`NumberOfColumns(A)`
`Ncols(A)`

Given an $m \times n$ matrix A, return n, the number of columns of A.

`NumberOfNonZeroEntries(A)`
`NNZEntries(A)`

Given a matrix A, return the number of non-zero entries in A.

`Density(A)`

Given a matrix A, return the density of A as a real number, which is the number of non-zero entries in A divided by the product of the number of rows of A and the number of columns of A (or zero if A has zero rows or columns).

`BaseRing(A)`

`CoefficientRing(A)`

Given a matrix A with entries lying in a ring R, return R.

`ElementToSequence(A)`

`Eltseq(A)`

Given a matrix A over the ring R having m rows and n columns, return the entries of A, in row-major order, as a sequence of mn elements of R.

`RowSequence(A)`

Returns the entries of A as a sequence of rows where a row is represented as a sequence of entries of A.

26.4 Accessing or Modifying Entries

26.4.1 Indexing

The following functions and operators enable one to access individual entries or rows of matrices or vectors.

`A[i]`

Given a matrix A over the ring R having m rows and n columns, and an integer i such that $1 \leq i \leq m$, return the i-th row of A, as a vector of length n.

`A[i, j]`

Given a matrix A over the ring R having m rows and n columns, integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, return the (i, j)-th entry of A, as an element of the ring R.

`A[Q]`

`A[i .. j]`

Given a matrix A over the ring R having m rows and n columns, and a sequence Q of integers in the range $[1..m]$, return the sequence consisting of the rows of A specified by Q. This is equivalent to `[A[i]: i in Q]]`. If Q is a range, then the second form `A[i .. j]` may be used to specify the range directly.

`A[i] := v`

Given a matrix A over the ring R having m rows and n columns, an integer i such that $1 \leq i \leq m$, and a vector v over R of length n, modify the i-th row of A to be v. The integer 0 may also be given for v, indicating the zero vector.

`A[i, j] := x`

Given a matrix A over the ring R having m rows and n columns, integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, and a ring element x coercible into R, modify the (i, j)-th entry of A to be x.

Example H26E4

This example demonstrates simple ways of accessing the entries of matrices.

```
> X := Matrix(4, [1,2,3,4, 5,4,3,2, 1,2,3,4]);
> X;
[1 2 3 4]
[5 4 3 2]
[1 2 3 4]
> X[1];
(1 2 3 4)
> X[1, 2];
2
> X[1, 2] := 23;
> X;
[ 1 23  3  4]
[ 5  4  3  2]
[ 1  2  3  4]
> X[3] := Vector([9,8,7,6]);
> X[2] := 0;
> X;
[ 1 23  3  4]
[ 0  0  0  0]
[ 9  8  7  6]
```

26.4.2 Extracting and Inserting Blocks

The following functions enable the extraction of certain rows, columns or general submatrices, or the replacement of a block by another matrix.

`Submatrix(A, i, j, p, q)`

`ExtractBlock(A, i, j, p, q)`

Given an $m \times n$ matrix A and integers i, j, p and q such that $1 \leq i \leq i + p \leq m + 1$ and $1 \leq j \leq j + q \leq n + 1$, return the $p \times q$ submatrix of A rooted at (i, j). Either or both of p and q may be zero, while i may be $m + 1$ if p is zero and j may be $n + 1$ if q is zero.

`SubmatrixRange(A, i, j, r, s)`

`ExtractBlockRange(A, i, j, r, s)`

Given an $m \times n$ matrix A and integers i, j, r and s such that $1 \leq i$, $i - 1 \leq r \leq m$, $1 \leq j$, and $j - 1 \leq s \leq n$, return the $r - i + 1 \times s - j + 1$ submatrix of A rooted at the (i, j)-th entry and extending to the (r, s)-th entry, inclusive. r may equal $i - 1$ or s may equal $j - 1$, in which case a matrix with zero rows or zero columns, respectively, will be returned.

`Submatrix(A, I, J)`

Given an $m \times n$ matrix A and integer sequences I and J, return the submatrix of A given by the row indices in I and the column indices in J.

`InsertBlock(A, B, i, j)`

`InsertBlock(~A, B, i, j)`

Given an $m \times n$ matrix A over a ring R, a $p \times q$ matrix B over R, and integers i and j such that $1 \leq i \leq i + p \leq m + 1$ and $1 \leq j \leq j + q \leq n + 1$, insert B at position (i, j) in A. In the functional version (A is a value argument), this function returns the new matrix and leaves A untouched, while in the procedural version ($\sim A$ is a reference argument), A is modified in place so that the $p \times q$ submatrix of A rooted at (i, j) is now equal to B.

`RowSubmatrix(A, i, k)`

Given an $m \times n$ matrix A and integers i and k such that $1 \leq i \leq i + k \leq m + 1$, return the $k \times n$ submatrix of X consisting of rows $[i \ldots i + k - 1]$ inclusive. The integer k may be zero and i may also be $m + 1$ if k is zero, but the result will always have n columns.

`RowSubmatrix(A, i)`

Given an $m \times n$ matrix A and an integer i such that $0 \leq i \leq m$, return the $i \times n$ submatrix of X consisting of the first i rows. The integer i may be 0, but the result will always have n columns.

`RowSubmatrixRange(A, i, j)`

Given an $m \times n$ matrix A and integers i and j such that $1 \leq i$ and $i - 1 \leq j \leq m$, return the $j - i + 1 \times n$ submatrix of X consisting of rows $[i \ldots j]$ inclusive. The integer j may equal $i - 1$, in which case a matrix with zero rows and n columns will be returned.

`ColumnSubmatrix(A, i, k)`

Given an $m \times n$ matrix A and integers i and k such that $1 \leq i \leq i + k \leq n + 1$, return the $m \times k$ submatrix of X consisting of columns $[i \ldots i + k - 1]$ inclusive. The integer k may be zero and i may also be $n + 1$ if k is zero, but the result will always have m rows.

`ColumnSubmatrix(A, i)`

Given an $m \times n$ matrix A and an integer i such that $0 \leq i \leq n$, return the $m \times i$ submatrix of X consisting of the first i columns. The integer i may be 0, but the result will always have m rows.

`ColumnSubmatrixRange(A, i, j)`

Given an $m \times n$ matrix A and integers i and j such that $1 \leq i$ and $i - 1 \leq j \leq n$, return the $m \times j - i + 1$ submatrix of X consisting of columns $[i \ldots j]$ inclusive. The integer j may equal $i - 1$, in which case a matrix with zero columns and n rows will be returned.

Example H26E5

The use of the submatrix operations is illustrated by applying them to a 6×6 matrix over the ring of integers **Z**.

```
> A := Matrix(6,
>     [ 9, 1, 7, -3, 2, -1,
>       3, -4, -5, 9, 2, 7,
>       7, 1, 0, 1, 8, 22,
>       -3, 3, 3, 8, 8, 37,
>       -9, 0, 7, -1, 2, 3,
>       7, 2, -2, 4, 3, 47 ]);
> A;
[ 9  1  7 -3  2 -1]
[ 3 -4 -5  9  2  7]
[ 7  1  0  1  8 22]
[-3  3  3  8  8 37]
[-9  0  7 -1  2  3]
[ 7  2 -2  4  3 47]
> Submatrix(A, 2,2, 3,3);
[-4 -5  9]
[ 1  0  1]
[ 3  3  8]
> SubmatrixRange(A, 2,2, 3,3);
[-4 -5]
[ 1  0]
> S := $1;
> InsertBlock(~A, S, 5,5);
> A;
[ 9  1  7 -3  2 -1]
[ 3 -4 -5  9  2  7]
[ 7  1  0  1  8 22]
[-3  3  3  8  8 37]
[-9  0  7 -1 -4 -5]
[ 7  2 -2  4  1  0]
> RowSubmatrix(A, 5, 2);
```

```
[-9  0  7 -1 -4 -5]
[ 7  2 -2  4  1  0]
> RowSubmatrixRange(A, 2, 3);
[ 3 -4 -5  9  2  7]
[ 7  1  0  1  8 22]
> RowSubmatrix(A, 2, 0);
Matrix with 0 rows and 6 columns
```

26.4.3 Row and Column Operations

The following functions and procedures provide elementary row or column operations on matrices. For each operation, there is a corresponding function which creates a new matrix for the result (leaving the input matrix unchanged), and a corresponding procedure which modifies the input matrix in place.

`SwapRows(A, i, j)`

`SwapRows(~A, i, j)`

Given an $m \times n$ matrix A and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq m$, swap the i-th and j-th rows of A.

`SwapColumns(A, i, j)`

`SwapColumns(~A, i, j)`

Given an $m \times n$ matrix A and integers i and j such that $1 \leq i \leq n$ and $1 \leq j \leq n$, swap the i-th and j-th columns of A.

`ReverseRows(A)`

`ReverseRows(~A)`

Given a matrix A, reverse all the rows of A.

`ReverseColumns(A)`

`ReverseColumns(~A)`

Given a matrix A, reverse all the columns of A.

`AddRow(A, c, i, j)`

`AddRow(~A, c, i, j)`

Given an $m \times n$ matrix A over a ring R, a ring element c coercible into R, and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq m$, add c times row i of A to row j of A.

`AddColumn(A, c, i, j)`

`AddColumn(~A, c, i, j)`

Given an $m \times n$ matrix A over a ring R, a ring element c coercible into R, and integers i and j such that $1 \leq i \leq n$ and $1 \leq j \leq n$, add c times column i of A to column j.

`MultiplyRow(A, c, i)`

`MultiplyRow(~A, c, i)`

Given an $m \times n$ matrix A over a ring R, a ring element c coercible into R, and an integer i such that $1 \leq i \leq m$, multiply row i of A by c (on the left).

`MultiplyColumn(A, c, i)`

`MultiplyColumn(~A, c, i)`

Given an $m \times n$ matrix A over a ring R, a ring element c coercible into R, and an integer i such that $1 \leq i \leq n$, multiply column i of A by c (on the left).

`RemoveRow(A, i)`

`RemoveRow(~A, i)`

Given an $m \times n$ matrix A and an integer i such that $1 \leq i \leq m$, remove row i from A (leaving an $(m-1) \times n$ matrix).

`RemoveColumn(A, j)`

`RemoveColumn(~A, j)`

Given an $m \times n$ matrix A and an integer j such that $1 \leq j \leq n$, remove column j from A (leaving an $m \times (n-1)$ matrix).

`RemoveRowColumn(A, i, j)`

`RemoveRowColumn(~A, i, j)`

Given an $m \times n$ matrix A and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, remove row i and column j from A (leaving an $(m-1) \times (n-1)$ matrix).

`RemoveZeroRows(A)`

`RemoveZeroRows(~A)`

Given a matrix A, remove all the zero rows of A.

Example H26E6

The use of row and column operations is illustrated by applying them to a 5×6 matrix over the ring of integers $\mathbf{Z}$.

```
> A := Matrix(5, 6,
>     [ 3, 1, 0, -4, 2, -12,
>       2, -4, -5, 5, 23, 6,
>       8, 0, 0, 1, 5, 12,
>       -2, -6, 3, 8, 9, 17,
>       11, 12, -6, 4, 2, 27 ]);
> A;
[  3   1   0  -4   2 -12]
[  2  -4  -5   5  23   6]
[  8   0   0   1   5  12]
[ -2  -6   3   8   9  17]
[ 11  12  -6   4   2  27]
> SwapColumns(~A, 1, 2);
> A;
[  1   3   0  -4   2 -12]
[ -4   2  -5   5  23   6]
[  0   8   0   1   5  12]
[ -6  -2   3   8   9  17]
[ 12  11  -6   4   2  27]
> AddRow(~A, 4, 1, 2);
> AddRow(~A, 6, 1, 4);
> AddRow(~A, -12, 1, 5);
> A;
[  1   3   0  -4   2 -12]
[  0  14  -5 -11  31 -42]
[  0   8   0   1   5  12]
[  0  16   3 -16  21 -55]
[  0 -25  -6  52 -22 171]
> RemoveRow(~A, 1);
> A;
[ 2 -4 -5  5 23  6]
[ 8  0  0  1  5 12]
[-2 -6  3  8  9 17]
[11 12 -6  4  2 27]
> RemoveRowColumn(~A, 4, 6);
> A;
[ 2 -4 -5  5 23]
[ 8  0  0  1  5]
[-2 -6  3  8  9]
```

26.5 Building Block Matrices

Block matrices can be constructed either by listing the blocks, or by joining together smaller matrices horizontally, vertically or diagonally.

`BlockMatrix(m, n, blocks)`

The matrix constructed from the given block matrices, which should all have the same dimensions, and should be given as a sequence of $m \cdot n$ block matrices (given in row major order, in other words listed across rows).

`BlockMatrix(m, n, rows)`

`BlockMatrix(rows)`

The matrix constructed from the given block matrices, which should all have the same dimensions, and should be given as a sequence of m rows, each containing n block matrices.

`HorizontalJoin(X, Y)`

Given a matrix X with r rows and c columns, and a matrix Y with r rows and d columns, both over the same coefficient ring R, return the matrix over R with r rows and $(c + d)$ columns obtained by joining X and Y horizontally (placing Y to the right of X).

`HorizontalJoin(Q)`

`HorizontalJoin(T)`

Given a sequence Q or tuple T of matrices, each having the same number of rows and being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q or T horizontally in order.

`VerticalJoin(X, Y)`

Given a matrix X with r rows and c columns, and a matrix Y with s rows and c columns, both over the same coefficient ring R, return the matrix with $(r + s)$ rows and c columns over R obtained by joining X and Y vertically (placing Y underneath X).

`VerticalJoin(Q)`

`VerticalJoin(T)`

Given a sequence Q or tuple T of matrices, each having the same number of columns and being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q or T vertically in order.

`DiagonalJoin(X, Y)`

Given matrices X with a rows and b columns and Y with c rows and d columns, both over the same coefficient ring R, return the matrix with $(a+c)$ rows and $(b+d)$ columns over R obtained by joining X and Y diagonally (placing Y diagonally to the right of and underneath X, with zero blocks above and below the diagonal).

`DiagonalJoin(Q)`

`DiagonalJoin(T)`

Given a sequence Q or tuple T of matrices, each being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q or T diagonally in order.

`KroneckerProduct(A, B)`

Given an $m \times n$ matrix A and a $p \times q$ matrix B, both over a ring R, return the Kronecker product of A and B, which is the $mp \times nq$ matrix C over R such that the $((i-1)p+r, (j-1)q+s)$-th entry of C is the (i,j)-th entry of A times the (r,s)-th entry of B, for $1 \leq i \leq m$, $1 \leq j \leq n$, $1 \leq r \leq p$ and $1 \leq s \leq q$.

26.6 Changing Ring

`ChangeRing(A, R)`

`Matrix(R, A)`

Given a matrix A over a ring S having m rows and n columns, and another ring R, return the $m \times n$ matrix over R obtained by coercing the entries of A from S into R. The argument order to `ChangeRing(A, R)` here is consistent with other forms of `ChangeRing`, while the `Matrix(R, A)` form of this function is provided to be consistent with the matrix creation functions above, for which the destination ring is the first argument, if supplied.

`ChangeRing(A, R, f)`

`ChangeRing(A, f)`

Given a matrix A over a ring S having m rows and n columns, another ring R, and a map $f : S \to R$, return the $m \times n$ matrix over R obtained by applying f to each of the entries of A. R may be omitted, in which case it is taken to be the codomain of f.

26.7 Elementary Arithmetic

`A + B`

Given $m \times n$ matrices A and B over a ring R, return $A + B$.

`A - B`

Given $m \times n$ matrices A and B over a ring R, return $A - B$.

`A * B`

Given an $m \times n$ matrix A over a ring R and an $n \times p$ matrix B over R, return the $m \times p$ matrix $A \cdot B$ over R. This function attempts to preserve the maximal amount of information in the choice of parent for the product. For example, if A and B are both square and have the same matrix algebra M as parent, then the product will also have M as parent. Similarly, if the parents of A and B are R-matrix spaces such that the codomain of B equals the domain A, then the product will have domain equal to that of A and codomain equal to that of B.

`x * A`
`A * x`

Given an $m \times n$ matrix A over a ring R and a ring element x coercible into R, return the scalar product $x \cdot A$.

`-A`

Given a matrix A, return $-A$.

`A ^ -1`

Given a invertible square matrix A over a ring R, return the inverse B of A so that $A \cdot B = B \cdot A = 1$. The coefficient ring R must be either a field, a Euclidean domain, or a ring with an exact division algorithm and having characteristic equal to zero or greater than m (this includes most commutative rings).

`A ^ n`

Given a square matrix A over a ring R and an integer n, return the matrix power A^n. A^0 is defined to be the identity matrix for any square matrix A (even if A is zero). If n is negative, A must be invertible (see the previous function), and the result is $(A^{-1})^{-n}$.

`Transpose(A)`

Given an $m \times n$ matrix A over a ring R, return the transpose of A, which is simply the $n \times m$ matrix over R whose (i, j)-th entry is the (j, i)-th entry of A.

`AddScaledMatrix(A, s, B)`

Given a matrix A over a ring R, a scalar s coercible into R, and a matrix B over R with the same shape as A, return $A + s \cdot B$. This is generally quicker than the call `A + s*B`.

`AddScaledMatrix(~A, s, B)`

Given a matrix A over a ring R, a scalar s coercible into R, and a matrix B over R with the same shape as A, set A to $A + s \cdot B$. This is generally quicker than the statement `A := A + s*B;`.

26.8 Nullspaces and Solutions of Systems

The following functions compute nullspaces of matrices (solving equations of the form $V \cdot A = 0$), or solve systems of the form $V \cdot A = W$, for given A and W.

Magma possesses a very rich suite of internal algorithms for computing nullspaces of matrices efficiently, including a fast p-adic algorithm for matrices over $\mathbf{Z}$ and $\mathbf{Q}$, and also algorithms which take advantage of sparsity if it is present.

`Nullspace(A)`

`Kernel(A)`

Given an $m \times n$ matrix A over a ring R, return the nullspace of A (or the kernel of A, considered as a linear transformation or map), which is the R-space consisting of all vectors v of length m such that $v \cdot A = 0$. If the parent of A is an R-matrix space, then the result will be the appropriate submodule of the domain of A. The function `Kernel(A)` also returns the inclusion map from the kernel into the domain of A, to be consistent with other forms of the `Kernel` function.

`NullspaceMatrix(A)`

`KernelMatrix(A)`

Given an $m \times n$ matrix A over a ring R, return a basis matrix of the nullspace of A. This is a matrix N having m columns and the maximal number of independent rows subject to the condition that $N \cdot A = 0$. This function has the advantage that the nullspace is not returned as a R-space, so echelonization of the resulting nullspace may be avoided.

`NullspaceOfTranspose(A)`

This function is equivalent to `Nullspace(Transpose(A))`, but may be more efficient in space for large matrices, since the transpose may not have to be explicitly constructed to compute the nullspace.

`IsConsistent(A, W)`

Given an $m \times n$ matrix A over a ring R, and a vector W of length n over R or a $r \times n$ matrix W over R, return `true` iff and only if the system of linear equations $V \cdot A = W$ is consistent. If the system is consistent, then the function will also return:

(a) A particular solution V so that $V \cdot A = W$;

(b) The nullspace N of A so that adding any elements of N to any rows of V will yield other solutions to the system.

`IsConsistent(A, Q)`

Given an $m \times n$ matrix A over a ring R, and a sequence Q of vectors of length n over R, return `true` if and only if the system of linear equations $V[i] * A = Q[i]$ for all i is consistent. If the system is consistent, then the function will also return:

(a) A particular solution sequence V;

(b) The nullspace N of A so that $(V[i] + u) * A = Q[i]$ for $u \in N$ for all i.

`Solution(A, W)`

Given an $m \times n$ matrix A over a ring R, and a vector W of length n over R or a $r \times n$ matrix W over R, solve the system of linear equations $V \cdot A = W$ and return:

(a) A particular solution V so that $V \cdot A = W$;

(b) The nullspace N of A so that adding any elements of N to any rows of V will yield other solutions to the system.
If there is no solution, an error results.

`Solution(A, Q)`

Given an $m \times n$ matrix A over a ring R, and a sequence Q of vectors of length n over R, solve the system of linear equations $V[i] * A = Q[i]$ for each i and return:

(a) A particular solution sequence V;

(b) The nullspace N of A so that $(V[i] + u) * A = Q[i]$ for $u \in N$ for all i.
If there is no solution, an error results.

Example H26E7

We compute the nullspace of a 301×300 matrix over $\mathbf{Z}$ with random entries in the range [0..10]. The nullity is 1 and the entries of the non-zero null vector are integers, each having about 455 decimal digits.

```
> m := 301; n := 300;
> X := Matrix(n, [Random(0, 10): i in [1 .. m*n]]);
> time N := NullspaceMatrix(X);
Time: 9.519
> Nrows(N), Ncols(N);
1 301
> time IsZero(N*X);
true
Time: 0.429
> {#Sprint(N[1,i]): i in [1..301]};
{ 452, 455, 456, 457, 458 }
```

Example H26E8

We show how one can enumerate all solutions to the system $V \cdot X = W$ for a given matrix X and vector W over a finite field. The `Solution` function gives a particular solution for V, and then adding this to every element in the nullspace N of X, we obtain all solutions.

```
> K := GF(3);
> X := Matrix(K, 4, 3, [1,2,1, 2,2,2, 1,1,1, 1,0,1]);
> X;
[1 2 1]
[2 2 2]
[1 1 1]
[1 0 1]
> W := Vector(K, [0,1,0]);
> V, N := Solution(X, W);
> V;
(1 1 0 0)
> N;
Vector space of degree 4, dimension 2 over GF(3)
Echelonized basis:
(1 0 1 1)
(0 1 1 0)
> [V + U: U in N];
[
    (1 1 0 0),
    (2 1 1 1),
    (0 1 2 2),
    (0 2 0 2),
    (1 2 1 0),
    (2 2 2 1),
    (2 0 0 1),
    (0 0 1 2),
    (1 0 2 0)
]
> [(V + U)*X eq W: U in N];
[ true, true, true, true, true, true, true, true, true ]
```

26.9 Predicates

The functions in this section test various properties of matrices. See also the Lattices chapter for a description of the function `IsPositiveDefinite` and related functions.

`IsZero(A)`

Given an $m \times n$ matrix A over the ring R, return `true` iff A is the $m \times n$ zero matrix.

`IsOne(A)`

Given a square $m \times m$ matrix A over the ring R, return `true` iff A is the $m \times m$ identity matrix.

`IsMinusOne(A)`

Given a square $m \times m$ matrix A over the ring R, return `true` iff A is the negation of the $m \times m$ identity matrix.

`IsScalar(A)`

Given a square $m \times m$ matrix A over the ring R, return `true` iff A is scalar, i.e., iff A is the product of some element of R and the $m \times m$ identity matrix.

`IsDiagonal(A)`

Given a square matrix A over the ring R, return `true` iff A is diagonal, i.e., iff the only non-zero entries of A are on the diagonal.

`IsSymmetric(A)`

Given a square matrix A over the ring R, return `true` iff A is symmetric, i.e., iff A equals its transpose.

`IsUpperTriangular(A)`

Given a matrix A over the ring R, return `true` iff A is upper triangular, i.e., iff the only non-zero entries of A are on or above the diagonal.

`IsLowerTriangular(A)`

Given a matrix A over the ring R, return `true` iff A is lower triangular, i.e., iff the only non-zero entries of A are on or below the diagonal.

`IsUnit(A)`

Given a square matrix A over the ring R, return `true` iff A is a unit, i.e., iff A has an inverse. The coefficient ring R may be any commutative ring (since the computation depends on testing if the determinant is a unit – a calculation which is supported in all commutative rings).

`IsSingular(A)`

Given a square $m \times m$ matrix A over the ring R, return `true` iff A is singular, i.e., iff the determinant of A is zero (or, equivalently, iff the rank of A is less than m). Note that `(not IsSingular(A))` is *not* equivalent to `IsUnit(A)` whenever R is not a field: if the determinant of A is non-zero but not a unit, then A is non-singular but not invertible. The coefficient ring R may be any commutative ring (since the computation involves only computing the determinant and testing whether it is zero).

`IsSymplecticMatrix(A)`

Given an $m \times m$ matrix A over the integers, return `true` if and only if A is an integer symplectic matrix, that is, $AJ^tA = J$, where $J = \begin{pmatrix} 0 & \mathbf{1}_g \\ -\mathbf{1}_g & 0 \end{pmatrix}$.

26.10 Determinant and Other Properties

`Determinant(A: parameters)`

`MonteCarloLevel`	RngIntElt	*Default :* 0
`Proof`	BoolElt	*Default :* `true`
`pAdic`	BoolElt	*Default :* `true`
`Divisor`	RngIntElt	*Default :* 0

Given a square matrix A over the ring R, return the determinant of A as an element of R. R may be any commutative ring. The determinant of the 0×0 matrix over R is defined to be `R!1`.

If the coefficient ring is the integer ring $\mathbf{Z}$ or the rational field $\mathbf{Q}$ then a modular algorithm based on that of Abbott et al. [ABM99] is used, which first computes a divisor d of the determinant D using a fast p-adic nullspace computation, and then computes the quotient D/d by computing the determinant D modulo enough small primes to cover the Hadamard bound divided by d. This always yields a correct answer.

If the parameter `MonteCarloLevel` is set to a small positive integer s, then a probabilistic Monte-Carlo modular technique is used. Rather than using sufficient primes to cover the Hadamard bound divided by the divisor d, this version of the algorithm terminates when the constructed residue remains constant for s steps. The probability of this being wrong is non-zero but extremely small, even if s is only 1 or 2. If the level is set to 0, then the normal deterministic algorithm is used. Setting the parameter `Proof` to `false` is equivalent to setting `MonteCarloLevel` to 2.

If the coefficient ring is $\mathbf{Z}$ and the parameter `Divisor` is set to an integer d, then d must be a known exact divisor of the determinant (the sign does not matter), and the algorithm may be sped up because of this knowledge.

`Trace(A)`

Given a square matrix A over the ring R, return the trace of A as an element of R, which is simply the sum of the diagonal elements of A.

`TraceOfProduct(A, B)`

Given square matrices A and B over the ring R, with the same size, return the trace of $A \cdot B$ as an element of R. This is in general much faster than the call `Trace(A*B)`.

`Rank(A)`

Given an $m \times n$ matrix A over a ring R, return the rank of A. This is defined to be the largest r such that there exists a non-zero $r \times r$ subdeterminant of A, so $r \leq m$ and $r \leq n$. The rank may have to be obtained by computing the Smith form or echelon form of A, and this computation may be quite expensive over some rings.

`Minor(M, i, j)`

The determinant of the submatrix of M (which must be square) formed by removing the i-th row and j-th column.

`Minor(M, I, J)`

The determinant of the submatrix of M given by the row indices in I and the column indices in J.

`Minors(M, r)`

Returns a sequence of all the r by r minors of the matrix M.

`Cofactor(M, i, j)`

The appropriate cofactor of M, equal to $(-1)^{i+j}$ times the corresponding minor.

`Cofactors(M)`

Returns a sequence of all the cofactors of the matrix M.

`Cofactors(M, r)`

Returns a sequence of all the r by r cofactors of the matrix M.

`Pfaffian(M)`

`Pfaffian(M, I, J)`

`Pfaffians(M, r)`

Let M be an anti-symmetric square matrix. Then its determinant is always a square and a particular square-root of this, which can be described by a universal polynomial in its entries, is called the Pfaffian of M. The first function returns this. The second function returns the Pfaffian of the submatrix of M described by the indices in I and J. The third function returns the sequence of Pfaffians of the $\binom{n}{r}$ principal r by r submatrices of M (n = the number of rows of M).

These are primarily convenience functions and are computed naively by Pfaffian row-expansion.

26.11 Minimal and Characteristic Polynomials and Eigenvalues

The functions in this section deal with minimal and characteristic polynomials.

`MinimalPolynomial(A: parameters)`

`Proof` BoolElt *Default :* `true`

Given a square matrix A over a ring R, return the minimal polynomial of A. This is defined to be the unique monic univariate polynomial $f(x)$ of minimal degree such that $f(A) = 0$, and $f(x)$ always divides the characteristic polynomial of A. The coefficient ring R is currently restricted to being a field or the integer ring $\mathbf{Z}$.

Setting the parameter `Proof` to false suppresses proof of correctness.

`CharacteristicPolynomial(A: parameters)`

`Al` MonStg *Default :* "*Modular*"

`Proof` BoolElt *Default :* `true`

Given a square matrix A over a ring R, return the characteristic polynomial of A. This is defined to be the monic univariate polynomial $\mathrm{Det}(x - A) \in R[x]$ where $R[x]$ is the univariate polynomial ring over R. R may be any commutative ring.

The parameter `Al` allows the user to specify which algorithm that is to be employed. The algorithm `"Modular"` (the default) may be used for matrices over $\mathbf{Z}$ and $\mathbf{Q}$—in such a case the parameter `Proof` can also be used to suppress proof of correctness.

The algorithm `"Hessenberg"`, available for matrices over fields, works by first reducing the matrix to Hessenberg form. The algorithm `"Interpolation"`, available for matrices over Z and Q, works by evaluating the characteristic matrix of a at various points and then interpolating. The algorithm `"Trace"`, available for matrices over fields, works by calculating the traces of powers of a.

Since V2.8, none of these algorithms are now recommended for matrices over $\mathbf{Z}$ or $\mathbf{Q}$, as the new p-adic modular algorithm over the integers is extremely fast.

`MinimalAndCharacteristicPolynomials(A: parameters)`

`MCPolynomials(A)`

`Proof` BoolElt *Default :* `true`

Given a square matrix A over a ring R, return the minimal and characteristic polynomials of A. For some rings, both polynomials can be computed at the same time, so in such cases it will be more efficient to use this function than to call `MinimalPolynomial` and `CharacteristicPolynomials` separately.

Setting the parameter `Proof` to false suppresses proof of correctness.

`FactoredMinimalPolynomial(A:` *parameters*`)`

`Proof`	BoolElt	*Default :* `true`

Given a square matrix A over a ring R, return the factorization of the minimal polynomial of A. This is equivalent to `Factorization(MinimalPolynomial(A))`, but may be faster than that for some coefficient rings (in particular, $\mathbf{Z}$ and $\mathbf{Q}$). Setting the parameter `Proof` to false suppresses proof of correctness.

`FactoredCharacteristicPolynomial(A:` *parameters*`)`

`Al`	MonStg	*Default :* "*Modular*"
`Proof`	BoolElt	*Default :* `true`

Given a square matrix A over a ring R, return the factorisation of the characteristic polynomial of A. This function returns the same result as the command `Factorisation(CharacteristicPolynomial(A))`, but may be faster than that for some coefficient rings (in particular, $\mathbf{Z}$ and $\mathbf{Q}$). The parameters are as for the function `CharacteristicPolynomial` above (setting the parameter `Proof` to false suppresses proof of correctness).

`FactoredMinimalAndCharacteristicPolynomials(A:` *parameters*`)`

`FactoredMCPolynomials(A:` *parameters*`)`

`Al`	MonStg	*Default :* "*Modular*"
`Proof`	BoolElt	*Default :* `true`

Given a square matrix A over a ring R, return the factorizations of the minimal and characteristic polynomials of A, respectively. For some rings, both polynomials can be computed and factored at the same time, so in such cases it will be more efficient to use this function than to call the above functions separately. Setting the parameter `Proof` to false suppresses proof of correctness.

`Eigenvalues(A)`

Given a square matrix A over a ring R, return the eigenvalues of A as a set of pairs, each of which gives the value of a distinct eigenvalue and its multiplicity. Factorization of polynomials over the base ring R must be possible.

`Eigenspace(A, e)`

Given a square matrix A over a ring R, and an element e of R, return the eigenspace of A corresponding to e, which is `Nullspace(A - e)`. If the ring element e is not a eigenvalue for the matrix A then the trivial nullspace is returned.

26.12 Canonical Forms

26.12.1 Canonical Forms over General Rings

The functions defined here apply to matrices defined over fields or Euclidean domains. See also the section on Reduction in the Lattices chapter for a description of the function `LLL` and related basis-reduction functions for matrices.

`EchelonForm(A)`

Given an $m \times n$ matrix A over the ring R, return the (reduced) row echelon form E of A, and also an invertible $m \times m$ transformation matrix T over R such that $T \cdot A = E$. Recall that T is a product of elementary matrices that transforms A into the echelon form E. If R is a Euclidean domain, the function `HermiteForm` (described below) is invoked. Note however, that the the user cannot set the parameters for `HermiteForm` when invoking it via `EchelonForm`.

`Adjoint(A)`

Given a square $m \times m$ matrix A over the ring R, return the adjoint of A as an $m \times m$ matrix. The base ring R must be a ring with exact division whose characteristic is zero or greater than m (this includes most commutative rings).

26.12.2 Canonical Forms over Fields

The functions described in this section apply to square matrices defined over fields which support factorization of univariate polynomials. See [Ste97] for a description of the single algorithm which is the basis of most of these functions.

`PrimaryRationalForm(A)`

Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the primary rational form of A. Each block in the form is the companion matrix of a power of an irreducible polynomial. This function returns three values:

(a) The primary rational canonical form F of A;

(b) An invertible matrix T such that $T \cdot A \cdot T^{-1} = F$;

(c) A sequence of pairs corresponding to the blocks of F where each pair consists of the irreducible polynomial and the multiplicity making up the block. This is the value returned by `PrimaryInvariantFactors(A)`.

`JordanForm(A)`

Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the generalized Jordan form of A. Each block in the form is a Jordan block (which itself is derived from a power of an irreducible polynomial), and the generalized Jordan form corresponds to the usual Jordan form if the minimal polynomial splits over K. This function returns three values:

(a) The Jordan canonical form F of A;

(b) An invertible matrix T such that $T \cdot A \cdot T^{-1} = F$;

(c) A sequence of pairs corresponding to the blocks of F where each pair consists of the irreducible polynomial and the multiplicity making up the block. This is the value returned by `PrimaryInvariantFactors(A)`.

`RationalForm(A)`

Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the rational form of A. For each block other than the final block, the polynomial corresponding to that block divides the polynomial corresponding to the next block. This function returns three values:

(a) The rational form F of A;

(b) An invertible matrix T such that $T \cdot A \cdot T^{-1} = F$;

(c) A sequence containing the polynomials corresponding to the successive blocks (where each polynomial, other than the last, divides the next polynomial). This is the value returned by `InvariantFactors(A)`.

`PrimaryInvariantFactors(A)`

Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the primary invariant factors of A. This is the same as the third return value of `PrimaryRationalForm(A)` or `JordanForm(A)`.

`InvariantFactors(A)`

Given a square matrix A over a field K such that factorization of polynomials is possible over K, return the invariant factors of A. This is the same as the third return value of `RationalForm(A)`.

`IsSimilar(A, B)`

Given square $m \times m$ matrices A and B, both over a field K such that factorization of polynomials is possible over K, return `true` iff and only if A is similar to B, and if so, return also an invertible $m \times m$ transformation matrix T such that $T \cdot A \cdot T^{-1} = B$.

`HessenbergForm(A)`

Given a square $m \times m$ matrix A over the ring R, return the Hessenberg form of A as an $m \times m$ matrix. The form has zero entries above the super-diagonal. (This form is used in one of the characteristic polynomial algorithms.) The base ring R must be a field.

`FrobeniusFormAlternating(A)`

Given an non-singular $2n \times 2n$ alternating matrix A over the integers, this function returns the (alternating) Frobenius form F of A. That is, a block matrix $F = \begin{pmatrix} 0 & D \\ -D & 0 \end{pmatrix}$, where D is a diagonal matrix with positive diagonal entries, d_i, satisfying $d_1 | d_2 | \cdots | d_n$. The second return value is the change of basis matrix B, such that $BA^tB = F$.

Example H26E9

We construct a 5×5 matrix over the finite field with 5 elements and then calculate various canonical forms. We verify the correctness of the polynomial invariant factors corresponding to the rational form by calculating the Smith form of the characteristic matrix of the original matrix (see below).

```
> K := GF(5);
> A := Matrix(K, 5,
>     [ 0, 2, 4, 2, 0,
>       2, 2, 2, 3, 3,
>       3, 4, 4, 1, 3,
>       0, 0, 0, 0, 1,
>       0, 0, 0, 1, 0 ]);
> A;
[0 2 4 2 0]
[2 2 2 3 3]
[3 4 4 1 3]
[0 0 0 0 1]
[0 0 0 1 0]
> PrimaryInvariantFactors(A);
[
    <x + 1, 1>,
    <x + 1, 1>,
    <x + 4, 1>,
    <x + 4, 1>,
    <x + 4, 1>
]
> JordanForm(A);
[4 0 0 0 0]
[0 4 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
> R, T, F := RationalForm(A);
> R;
[1 0 0 0 0]
[0 0 1 0 0]
[0 1 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
> T;
[1 3 0 2 1]
[2 1 2 2 0]
[3 4 3 4 1]
[1 0 0 0 0]
[0 2 4 2 0]
> T*A*T^-1 eq R;
true;
```

```
> F;
[
    x + 4,
    x^2 + 4,
    x^2 + 4
]
> P<x> := PolynomialRing(K);
> PM := MatrixAlgebra(P, 5);
> Ax := PM ! x - PM ! A;
> Ax;
[    x     3     1     3     0]
[    3 x + 3     3     2     2]
[    2     1 x + 1     4     2]
[    0     0     0     x     4]
[    0     0     0     4     x]
> SmithForm(Ax);
[      1       0       0       0       0]
[      0       1       0       0       0]
[      0       0   x + 4       0       0]
[      0       0       0 x^2 + 4       0]
[      0       0       0       0 x^2 + 4]
> ElementaryDivisors(Ax);
[
    1,
    1,
    x + 4,
    x^2 + 4,
    x^2 + 4
]
```

26.12.3 Canonical Forms over Euclidean Domains

The functions defined here apply to matrices defined over Euclidean domains. See also the section on Reduction in the Lattices chapter for a description of the function LLL and related functions, which are very useful for integer matrices.

`HermiteForm(A)`

`Al`	MonStg	*Default :* "*Default*"
`Optimize`	BoolElt	*Default :* `true`
`Integral`	BoolElt	*Default :* `true`

Given an $m \times n$ matrix A over the Euclidean ring R, return the Hermite form H of A, and also an invertible $m \times m$ transformation matrix T over R such that $T \cdot A = H$.

The basic algorithm used is the classical Kannan-Bachem algorithm [KB79, CC82], which has classical complexity (but does not suffer from bad coefficient growth).

Since V2.13, for matrices over the integers there is also a fast modular algorithm by Allan Steel. By default, MAGMA chooses between these two algorithms, usually favouring the new modular algorithm. But one may set the parameter `Al` to `"Modular"` to force the modular algorithm to be used, and to `"Classical"` to force the classical algorithm to be used.

If R is the ring of integers $\mathbf{Z}$ and the matrix T is requested (i.e., if an assignment statement is used with two variables on the left side), then the LLL algorithm will also be used by default to improve T (using the kernel of A) so that the size of its entries are very small. If the parameter `Optimize` is set to `false`, then this will not happen (which will be faster but the entries of T will not be as small). If the parameter `Integral` is set to `true`, then the integral (de Weger) LLL method will be used in the LLL step, instead of the default floating point method.

`SmithForm(A)`

Given an $m \times n$ matrix A over the Euclidean ring R, return the Smith normal form of A. This function returns three values:

(a) The Smith normal form S of A; and

(b) Unimodular matrices P and Q such that $P \cdot A \cdot Q = S$, i.e., P and Q are matrices which transform A into Smith normal form.

The algorithm implemented first uses the sparse techniques described in [HHR93] to reduce the matrix to a dense submatrix, then, if this is non-trivial, it either repeatedly calls the Hermite normal form algorithm (see above) and transposes until a diagonal form is obtained, or uses the modular algorithm of F. Lübeck [Lüb02].

Unless one wishes one or both of the transformation matrices, it is preferable to use the following function `ElementaryDivisors` since it gives the same information, but saves memory since the matrix S does not need to be constructed.

`ElementaryDivisors(A)`

Given an $m \times n$ matrix A over the Euclidean ring or field R, return the elementary divisors of A. These are simply the non-zero diagonal entries of the Smith form of A, in order. The divisors are returned as a sequence $[e_1, \ldots, e_r]$ of r elements of R (which may include ones), where r is the rank of A and $e_i | e_{i+1}$ for $i = 1, \ldots, r-1$. The divisors are normalized, so the result is unique. If R is a field, the result is always the sequence of r ones, where r is the rank of A.

Note that if $m = n = r$, then the determinant of A is the product of the e_i and if R is also a domain, then e_r is the lowest common denominator of the inverse of A over the field of fractions of R.

`Saturation(A)`

Given an $m \times n$ matrix A over the integer ring $\mathbf{Z}$, having rank r, return an $m \times n$ matrix S over $\mathbf{Z}$ whose first r rows form a basis of the saturation w.r.t. $\mathbf{Q}$ of the $\mathbf{Q}$-vector space spanned by the rows of A. The rows of S thus span the same space over $\mathbf{Q}$ as those of A, while the $\mathbf{Z}$-module spanned by the rows of S is the set of all

v such that for some non-zero scalar s, $s \cdot v$ is in the $\mathbf{Z}$-module spanned by the rows of A.

Example H26E10

We illustrate some of these operations for a 4×3 matrix over $\mathbf{F}_8$.

```
> K<w> := GF(8);
> A := Matrix(K, 4, 3, [1,w,w^5, 0,w^3,w^4, w,1,w^6, w^3,1,w^4]);
> A;
[  1   w w^5]
[  0 w^3 w^4]
[  w   1 w^6]
[w^3   1 w^4]
> EchelonForm(A);
[  1   0   0]
[  0   1   0]
[  0   0   1]
[  0   0   0]
```

We now illustrate some of these operations for a 4×5 matrix over $\mathbf{Z}$.

```
> A := Matrix(4, 5,
>     [ 2,-4,12,7,0,
>       3,-3,5,-1,4,
>       2,-1,-4,-5,-12,
>       0,3,6,-2,0]);
> A;
[  2  -4  12   7   0]
[  3  -3   5  -1   4]
[  2  -1  -4  -5 -12]
[  0   3   6  -2   0]
> Rank(A);
4
> HermiteForm(A);
[   1    1    1    6 -164]
[   0    3    0   16 -348]
[   0    0    2   13 -200]
[   0    0    0   19 -316]
> SmithForm(A);
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 2 0]
> ElementaryDivisors(A);
[ 1, 1, 1, 2 ]
```

26.13 Orders of Invertible Matrices

The functions defined here apply to invertible square matrices. MAGMA can efficiently compute the order of an invertible matrix over a finite field, using the Cunningham database to factorize the numbers of the form $p^n - 1$ which arise. The algorithm employed is that described in [CLG97a].

MAGMA also contains efficient algorithms for rigorously proving whether a matrix over the ring of integers $\mathbf{Z}$, the rational field $\mathbf{Q}$, an algebraic number field, a cyclotomic field or a quadratic field has finite order or not, and for determining the order if it is finite.

`HasFiniteOrder(A)`

Given a square invertible matrix A over a ring R, return `true` iff A has finite order, i.e., iff there exists a positive integer n such that $A^n = 1$. The coefficient ring R is currently restricted to being either a finite field, the ring of integers $\mathbf{Z}$, the rational field $\mathbf{Q}$, an algebraic number field, a cyclotomic field or a quadratic field. For matrices over any of these rings, the function rigorously proves its result (over other rings, an error results).

`Order(A)`

`Proof` BOOLELT *Default :* `true`

Given a square invertible matrix A over any commutative ring, return the order of A. If R is a ring for which a finite order proof exists (see `HasFiniteOrder` above), then an error results if A has infinite order. Over other rings, if A has infinite order then the function may loop indefinitely since it may not be able to prove the infinitude of the order.

`FactoredOrder(A)`

`Proof` BOOLELT *Default :* `true`

Given a square invertible matrix A over a finite field, return the order of A in factored form. This returns the same value as `Factorization(Order(A))`, but since the order computation must compute the factorization of the order anyway, it involves no more effort to have it return the factorization. The conditions on the ring are as for `Order`.

`ProjectiveOrder(A)`

`Proof` BOOLELT *Default :* `true`

Given a square invertible matrix A over a finite field K, return the projective order n of A and a scalar $s \in K$ such that $A^n = sI$. The projective order of A is the smallest n such that A^n is a scalar matrix (not just the identity matrix), and it always divides the true order of A. The parameter `Proof` is as for `Order`.

`FactoredProjectiveOrder(A)`

`Proof` BoolElt *Default :* `true`

Given a square invertible matrix A over a finite field K, return the projective order n of A in factored form and a scalar $s \in K$ such that $A^n = sI$. The parameter `Proof` is as for `FactoredOrder`.

26.14 Miscellaneous Operations on Matrices

`FrobeniusImage(A, e)`

Given a matrix A over a finite field K of characteristic p, return the matrix obtained from A by mapping each entry $A_{i,j}$ to $(A_{i,j})^{p^e}$.

`NumericalEigenvectors(M, e)`

Given a square matrix M that is coercible into the complexes, and an approximation e to an eigenvalue of it, attempt to find eigenvectors. This function is for cases for which there are no numerical worries.

26.15 Bibliography

[**ABM99**] John Abbott, Manuel Bronstein, and Thom Mulders. Fast Deterministic Computation of Determinants of Dense Matrices. In Sam Dooley, editor, *Proceedings ISSAC'99*, pages 197–204, New York, 1999. ACM Press.

[**CC82**] T.W.J. Chou and G.E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM J. Computing*, 11(4):687–708, 1982.

[**CLG97**] Frank Celler and Charles R. Leedham-Green. Calculating the Order of an Invertible Matrix. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 55–60. AMS, 1997.

[**HHR93**] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented Z-modules. *Linear Algebra and its Applications*, 192:137–164, 1993.

[**KB79**] R. Kannan and A. Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM J. Computing*, 9:499–507, 1979.

[**Lüb02**] F. Lübeck. On the computation of elementary divisors of integer matrices. *J. Symbolic Comp.*, 33:57–65, 2002.

[**Ste97**] Allan Steel. A New Algorithm for the Computation of Canonical Forms of Matrices over Fields. *J. Symbolic Comp.*, 24(3):409–432, 1997.

27 SPARSE MATRICES

Chapter 27
SPARSE MATRICES

27.1 Introduction

A separate type is provided for sparse matrices to allow the user to construct such matrices and apply algorithms which take advantage of sparsity. Sparse matrices are distinct from normal matrices in MAGMA, which have a dense representation. This chapter describes the operations available for creating and working with sparse matrices.

The operations provided for sparse matrices include dynamic construction, simple properties, and the calculation of a number of non-trivial and important invariants (such as rank, determinant, or computing a non-zero vector in the nullspace). In particular, this datatype supports the class of *index-calculus* algorithms which involve generating a very large sparse system and then solving the system or finding the elementary divisors of the corresponding matrix (for example, to compute the abelian group structure). An extended example presented at the end of the chapter (H27E3) illustrates how an index-calculus method may be implemented in practice in the MAGMA language using the sparse matrix facilities.

The type name for the category of sparse matrices is `MtrxSprs`. All sparse matrices over a given ring R lie in the same *sparse matrix structure*, whose type name is `MtrxSprsStr`. The user will, in practice, rarely need to refer explicitly to the parent structure.

27.2 Creation of Sparse Matrices

This section describes the constructs provided for creating sparse matrices.

27.2.1 Construction of Initialized Sparse Matrices

```
SparseMatrix(R, m, n, Q)
```

```
SparseMatrix(m, n, Q)
```

Given a ring R (optional), integers $m, n \geq 0$ and a sequence Q, return the $m \times n$ sparse matrix over R whose non-zero entries are those specified by Q, coerced into R. If R is not given, it is derived from the entries in Q. Either of m and n may be 0, in which case Q must have length 0 (and may be null if R is given), and the $m \times n$ zero sparse matrix over R is returned. There are two possibilities for Q:

(a) The sequence Q is a sequence of tuples, each of the form `<i, j, x>`, where $1 \leq i \leq m$, $1 \leq j \leq n$, and $x \in S$ for some ring S. Such a tuple specifies that the (i, j)-th entry of the matrix is x. If R is given, then x is coerced into R; otherwise the matrix is created over S. If an entry position is not given then its value is zero, while if an entry position is repeated then the last value overrides any previous values.

(b) The sequence Q is a "flat" sequence of integers, giving the entries of the matrix in compact form. To be precise, Q begins with the number of non-zero entries n for the first row, then $2 \cdot n$ integers giving column-entry pairs for the first row, and this format is immediately followed for the second row and so on. A zero row is specified by a zero value for n. If R is given, the integer entries are coerced into R; otherwise the matrix is defined over $\mathbf{Z}$. (Thus this method will not allow elements of R which cannot be created by coercing integers into R alone; another way of saying this is that the entries must all lie in the prime ring of R). This allows a very compact way to create (and store) sparse matrices. The examples below illustrate this format.

`SparseMatrix(R, m, n)`

Given a ring R, and integers $m, n \geq 0$, create the $m \times n$ sparse matrix over R.

`SparseMatrix(m, n)`

Given integers $m, n \geq 0$, create the $m \times n$ sparse matrix over the integer ring $\mathbf{Z}$.

27.2.2 Construction of Trivial Sparse Matrices

`SparseMatrix(R)`

`SparseMatrix()`

Create the 0×0 sparse matrix over R. If R is omitted (so there are no arguments), then R is taken to be the integer ring $\mathbf{Z}$. These functions will usually be called when the user wishes to create a sparse matrix whose final dimensions are initially unknown, and for which the SetEntry procedure below will be used to extend the matrix automatically, as needed.

Example H27E1

This example demonstrates simple ways of creating matrices using the general `SparseMatrix(R, m, n, Q)` function. Sparse matrices may be displayed in the sparse representation using the `Magma` print-format. Also, the function Matrix (described below) takes a sparse matrix and returns a normal (dense-representation) matrix, so this function provides a means of printing a small sparse matrix as a normal matrix.

(a) A sparse 2×2 matrix is defined over $\mathbf{Z}$, using the first (sequence of tuples) method. We specify that there is a 3 in the $(1, 2)$ position and a -1 in the $(2, 3)$ position. The ring need not be given since the entries are in $\mathbf{Z}$ already.

```
> A := SparseMatrix(2, 3, [<1,2,3>, <2,3,-1>]);
> A;
Sparse matrix with 2 rows and 3 columns over Integer Ring
> Matrix(A);
[ 0  3  0]
```

```
[ 0  0 -1]
```

(b) The same matrix is now defined over $\mathbf{F}_{23}$. We could also coerce the 3rd component of each tuple into $\mathbf{F}_{23}$ and thus omit the first argument.

```
> A := SparseMatrix(GF(23), 2, 3, [<1,2,3>, <2,3,-1>]);
> A;
Sparse matrix with 2 rows and 3 columns over GF(23)
> Matrix(A);
[ 0  3  0]
[ 0  0 22]
```

(c) A similar sparse matrix is defined over $\mathbf{F}_{2^4}$. When A is printed in `Magma` format, the sequence-of-tuples form is used (because the entries cannot be printed as integers).

```
> K<w> := GF(2^4);
> A := SparseMatrix(K, 2, 3, [<1,2,3>, <2,3,w>]);
> A;
Sparse matrix with 2 rows and 3 columns over GF(2^4)
> Matrix(A);
[   0    1    0]
[   0    0    w]
> A: Magma;
SparseMatrix(GF(2, 4), 2, 3, [
    <1, 2, 1>,
    <2, 3, w>
])
```

(d) A sparse 4×5 matrix A is defined over $\mathbf{Z}$, using the second (flat integer sequence) method. Here row 1 has one non-zero entry: -1 at column 3; then row 2 has three non-zero entries: 9 at column 2, 7 at column 3, and -3 at column 4; row 3 has no non-zero entries (so we give a 0 at this point); and finally row 4 has one non-zero entry 3 at column 4. Note that when A is printed in `Magma` format, this time the compact "flat" sequence of integers form is used, since this is possible.

```
> A := SparseMatrix(4,5, [1,3,-1, 3,2,9,3,7,4,-3, 0, 1,4,3]);
> A;
Sparse matrix with 4 rows and 5 columns over Integer Ring
> Matrix(A);
[ 0  0 -1  0  0]
[ 0  9  7 -3  0]
[ 0  0  0  0  0]
[ 0  0  0  3  0]
> A: Magma;
SparseMatrix(4, 5, \[
    1, 3,-1,
    3, 2,9, 3,7, 4,-3,
    0,
    1, 4,3
])
```

27.2.3 Construction of Structured Matrices

`IdentitySparseMatrix(R, n)`

Given a ring R, and an integer $n \geq 0$, return the $n \times n$ identity sparse matrix over R.

`ScalarSparseMatrix(n, s)`

Given an integer $n \geq 0$ and an element s of a ring R, return the $n \times n$ scalar sparse matrix over R which has s on the diagonal and zeros elsewhere. The argument n may be 0, in which case the 0×0 sparse matrix over R is returned.

`ScalarSparseMatrix(R, n, s)`

Given a ring R, an integer $n \geq 0$ and an element s of a ring S, return the $n \times n$ scalar sparse matrix over R which has s, coerced into R, on the diagonal and zeros elsewhere. n may be 0, in which case in which case the 0×0 sparse matrix over R is returned.

`DiagonalSparseMatrix(R, n, Q)`

Given a ring R, an integer $n \geq 0$ and a sequence Q of n ring elements, return the $n \times n$ diagonal sparse matrix over R whose diagonal entries correspond to the entries of Q, coerced into R.

`DiagonalSparseMatrix(R, Q)`

Given a ring R and a sequence Q of n ring elements, return the $n \times n$ diagonal sparse matrix over R whose diagonal entries correspond to the entries of Q, coerced into R.

`DiagonalSparseMatrix(Q)`

Given a sequence Q of n elements from a ring R, return the $n \times n$ diagonal sparse matrix over R whose diagonal entries correspond to the entries of Q.

27.2.4 Parents of Sparse Matrices

`SparseMatrixStructure(R)`

Create the structure containing all sparse matrices (of any shape) over ring R. This structure does not need to be created explicitly by the user (it will be the parent of any sparse matrix over R), but it may be useful to create it in this way occasionally.

27.3 Accessing Sparse Matrices

The following functions access basic properties of sparse matrices.

27.3.1 Elementary Properties

`BaseRing(A)`

`CoefficientRing(A)`

Given a sparse matrix A with entries lying in a ring R, return R.

`NumberOfRows(A)`

`Nrows(A)`

Given an $m \times n$ sparse matrix A, return m, the number of rows of A.

`NumberOfColumns(A)`

`Ncols(A)`

Given an $m \times n$ sparse matrix A, return n, the number of columns of A.

`ElementToSequence(A)`

`Eltseq(A)`

Given a sparse matrix A over the ring R having m rows and n columns, return the entries of A as a sequence of all tuples of the form `<i, j, x>` such that the $[i, j]$-th entry of A equals x and x is non-zero. It is always true that `SparseMatrix(Nrows(A), Ncols(A), Eltseq(A))` equals A.

`NumberOfNonZeroEntries(A)`

`NNZEntries(A)`

Given a sparse matrix A, return the number of non-zero entries in A.

`Density(A)`

Given a sparse matrix A, return the density of A as a real number, which is the number of non-zero entries in A divided by the product of the number of rows of A and the number of columns of A (or zero if A has zero rows or columns).

`Support(A, i)`

Given a sparse matrix A having r rows, and an integer i such that $1 \leq i \leq r$, return the support of row i of A; i.e., the column numbers of the non-zero entries of row i of A.

`Support(A)`

Given a sparse matrix A, return the sequence of all pairs `<i, j>` such that the $[i, j]$-th entry of A is non-zero.

27.3.2 Weights

`RowWeight(A, i)`

Given a sparse matrix A with m rows and an integer i such that $1 \leq i \leq m$, return the weight (number of non-zero entries) of the i-th row of A.

`RowWeights(A)`

Given a sparse matrix A with m rows, return the length m sequence of integers whose i-th entry is the weight of the i-th row of A.

`ColumnWeight(A, j)`

Given a sparse matrix A with n columns and an integer j such that $1 \leq j \leq n$, return the weight (number of non-zero entries) of the j-th column of A.

`ColumnWeights(A)`

Given a sparse matrix A with n columns, return the length n sequence of integers whose j-th entry is the weight of the j-th column of A.

27.4 Accessing or Modifying Entries

The following functions and procedures enable the user to access or set individual entries of sparse matrices.

`A[i]`

Given a sparse matrix A over a ring R having m rows and n columns, and an integer i such that $1 \leq i \leq m$, return the i-th row of A, as a dense vector of length n (lying in R^n).

`A[i, j]`

Given a sparse matrix A over a ring R having m rows and n columns, integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, return the (i, j)-th entry of A, as an element of the ring R.

`A[i, j] := x`

Given a sparse matrix A over a ring R having m rows and n columns, integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, and a ring element x coercible into R, modify the (i, j)-th entry of A to be x. Here i and j must be within the ranges given by the current dimensions of A; see `SetEntry` below for a procedure to automatically extend A if necessary.

`SetEntry(~A, i, j, x)`

(Procedure.) Given a sparse matrix A over a ring R, integers $i, j \geq 1$, and a ring element x coercible into R, modify the (i, j)-th entry of A to be x. The entry specified by i and j is allowed to be beyond the current dimensions of A; if so, A is automatically extended to have at least i rows and j columns.

This procedure will be commonly used in situations where the final size of the matrix is not known as an algorithm proceeds (e.g., in index-calculus methods). One can create the 0×0 sparse matrix over $\mathbf{Z}$, say, and then call `SetEntry` to build up the matrix dynamically. See the example H27E3 below, which uses this technique.

Note that extending the dimensions of A with a very large i or j will not in itself consume much memory, but if A then becomes dense or is passed to some algorithm, then the memory needed may of course be proportional to the dimensions of A.

Example H27E2

This example demonstrates simple ways of accessing the entries of sparse matrices.

```
> A := SparseMatrix(2, 3, [<1,2,3>, <2,3,-1>]);
> A;
Sparse matrix with 2 rows and 3 columns over Integer Ring
> Matrix(A);
[ 0  3  0]
[ 0  0 -1]
> A[1];
(0 3 0)
> A[1, 3]:=5;
> A[1];
(0 3 5)
```

We next extend A using the procedure `SetEntry`.

```
> SetEntry(~A, 1, 5, -7);
> A;
Sparse matrix with 2 rows and 5 columns over Integer Ring
> Matrix(A);
[ 0  3  5  0 -7]
[ 0  0 -1  0  0]
```

A common situation is to start with the empty 0×0 matrix over $\mathbf{Z}$ and then to extend it dynamically.

```
> A := SparseMatrix();
> A;
Sparse matrix with 0 rows and 0 columns over Integer Ring
> SetEntry(~A, 1, 4, -2);
> A;
Sparse matrix with 1 row and 4 columns over Integer Ring
> SetEntry(~A, 2, 3, 8);
> A;
```

```
Sparse matrix with 2 rows and 4 columns over Integer Ring
> Matrix(A);
[ 0  0  0 -2]
[ 0  0  8  0]
> SetEntry(~A, 200, 319, 1);
> SetEntry(~A, 200, 3876, 1);
> A;
Sparse matrix with 200 rows and 3876 columns over Integer Ring
> Nrows(A);
200
> Ncols(A);
3876
> NNZEntries(A);
4
> Density(A);
0.00000515995872033023735810113519 0
> Support(A, 200);
[ 319, 3876 ]
```

27.4.1 Extracting and Inserting Blocks

The following functions enable the extraction of certain rows, columns or general submatrices, or the replacement of a block by another sparse matrix.

`Submatrix(A, i, j, p, q)`

`ExtractBlock(A, i, j, p, q)`

Given an $m \times n$ sparse matrix A and integers i, j, p and q such that $1 \leq i \leq i+p \leq m+1$ and $1 \leq j \leq j+q \leq n+1$, return the $p \times q$ submatrix of A rooted at (i, j). Either or both of p and q may be zero, while i may be $m+1$ if p is zero and j may be $n+1$ if q is zero.

`SubmatrixRange(A, i, j, r, s)`

`ExtractBlockRange(A, i, j, r, s)`

Given an $m \times n$ sparse matrix A and integers i, j, r and s such that $1 \leq i$, $i-1 \leq r \leq m$, $1 \leq j$, and $j-1 \leq s \leq n$, return the $r-i+1 \times s-j+1$ submatrix of A rooted at the (i, j)-th entry and extending to the (r, s)-th entry, inclusive. r may equal $i-1$ or s may equal $j-1$, in which case a sparse matrix with zero rows or zero columns, respectively, will be returned.

`Submatrix(A, I, J)`

Given an $m \times n$ sparse matrix A and integer sequences I and J, return the submatrix of A given by the row indices in I and the column indices in J.

`InsertBlock(A, B, i, j)`

`InsertBlock(~A, B, i, j)`

Given an $m \times n$ sparse matrix A over a ring R, a $p \times q$ sparse matrix B over R, and integers i and j such that $1 \leq i \leq i + p \leq m + 1$ and $1 \leq j \leq j + q \leq n + 1$, insert B at position (i, j) in A. In the functional version (A is a value argument), this function returns the new sparse matrix and leaves A untouched, while in the procedural version ($\sim A$ is a reference argument), A is modified in place so that the $p \times q$ submatrix of A rooted at (i, j) is now equal to B.

`RowSubmatrix(A, i, k)`

Given an $m \times n$ sparse matrix A and integers i and k such that $1 \leq i \leq i+k \leq m+1$, return the $k \times n$ submatrix of X consisting of rows $[i \ldots i + k - 1]$ inclusive. The integer k may be zero and i may also be $m + 1$ if k is zero, but the result will always have n columns.

`RowSubmatrix(A, i)`

Given an $m \times n$ sparse matrix A and an integer i such that $0 \leq i \leq m$, return the $i \times n$ submatrix of X consisting of the first i rows. The integer i may be 0, but the result will always have n columns.

`RowSubmatrixRange(A, i, j)`

Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i$ and $i - 1 \leq j \leq m$, return the $j - i + 1 \times n$ submatrix of X consisting of rows $[i \ldots j]$ inclusive. The integer j may equal $i - 1$, in which case a sparse matrix with zero rows and n columns will be returned.

`ColumnSubmatrix(A, i, k)`

Given an $m \times n$ sparse matrix A and integers i and k such that $1 \leq i \leq i+k \leq n+1$, return the $m \times k$ submatrix of X consisting of columns $[i \ldots i+k-1]$ inclusive. The integer k may be zero and i may also be $n + 1$ if k is zero, but the result will always have m rows.

`ColumnSubmatrix(A, i)`

Given an $m \times n$ sparse matrix A and an integer i such that $0 \leq i \leq n$, return the $m \times i$ submatrix of X consisting of the first i columns. The integer i may be 0, but the result will always have m rows.

`ColumnSubmatrixRange(A, i, j)`

Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i$ and $i-1 \leq j \leq n$, return the $m \times j - i + 1$ submatrix of X consisting of columns $[i \ldots j]$ inclusive. The integer j may equal $i - 1$, in which case a sparse matrix with zero columns and n rows will be returned.

27.4.2 Row and Column Operations

The following functions and procedures provide elementary row or column operations on sparse matrices. For each operation, there is a corresponding function which creates a new sparse matrix for the result (leaving the input sparse matrix unchanged), and a corresponding procedure which modifies the input sparse matrix in place.

`SwapRows(A, i, j)`

`SwapRows(~A, i, j)`

Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq m$, swap the i-th and j-th rows of A.

`SwapColumns(A, i, j)`

`SwapColumns(~A, i, j)`

Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i \leq n$ and $1 \leq j \leq n$, swap the i-th and j-th columns of A.

`ReverseRows(A)`

`ReverseRows(~A)`

Given a sparse matrix A, reverse all the rows of A.

`ReverseColumns(A)`

`ReverseColumns(~A)`

Given a sparse matrix A, reverse all the columns of A.

`AddRow(A, c, i, j)`

`AddRow(~A, c, i, j)`

Given an $m \times n$ sparse matrix A over a ring R, a ring element c coercible into R, and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq m$, add c times row i of A to row j of A.

`AddColumn(A, c, i, j)`

`AddColumn(~A, c, i, j)`

Given an $m \times n$ sparse matrix A over a ring R, a ring element c coercible into R, and integers i and j such that $1 \leq i \leq n$ and $1 \leq j \leq n$, add c times column i of A to column j.

`MultiplyRow(A, c, i)`

`MultiplyRow(~A, c, i)`

Given an $m \times n$ sparse matrix A over a ring R, a ring element c coercible into R, and an integer i such that $1 \leq i \leq m$, multiply row i of A by c (on the left).

`MultiplyColumn(A, c, i)`

`MultiplyColumn(~A, c, i)`

Given an $m \times n$ sparse matrix A over a ring R, a ring element c coercible into R, and an integer i such that $1 \leq i \leq n$, multiply column i of A by c (on the left).

`RemoveRow(A, i)`

`RemoveRow(~A, i)`

Given an $m \times n$ sparse matrix A and an integer i such that $1 \leq i \leq m$, remove row i from A (leaving an $(m-1) \times n$ sparse matrix).

`RemoveColumn(A, j)`

`RemoveColumn(~A, j)`

Given an $m \times n$ sparse matrix A and an integer j such that $1 \leq j \leq n$, remove column j from A (leaving an $m \times (n-1)$ sparse matrix).

`RemoveRowColumn(A, i, j)`

`RemoveRowColumn(~A, i, j)`

Given an $m \times n$ sparse matrix A and integers i and j such that $1 \leq i \leq m$ and $1 \leq j \leq n$, remove row i and column j from A (leaving an $(m-1) \times (n-1)$ sparse matrix).

`RemoveZeroRows(A)`

`RemoveZeroRows(~A)`

Given a sparse matrix A, remove all the zero rows of A.

27.5 Building Block Matrices

`HorizontalJoin(A, B)`

Given a sparse matrix A with r rows and c columns, and a sparse matrix B with r rows and d columns, both over the same coefficient ring R, return the sparse matrix over R with r rows and $(c+d)$ columns obtained by joining A and B horizontally (placing B to the right of A).

`VerticalJoin(A, B)`

Given a sparse matrix A with r rows and c columns, and a sparse matrix B with s rows and c columns, both over the same coefficient ring R, return the sparse matrix with $(r+s)$ rows and c columns over R obtained by joining A and B vertically (placing B underneath A).

`DiagonalJoin(A, B)`

Given matrices A with a rows and b columns and B with c rows and d columns, both over the same coefficient ring R, return the sparse matrix with $(a + c)$ rows and $(b + d)$ columns over R obtained by joining A and B diagonally (placing B diagonally to the right of and underneath A, with zero blocks above and below the diagonal).

27.6 Conversion to and from Dense Matrices

The following functions convert between sparse matrices and normal (dense-representation) matrices.

`Matrix(A)`

Given a sparse matrix A, return the normal (dense-representation) matrix equal to A. This function should only be used if the size of A is reasonably small since otherwise the amount of memory needed to represent the dense-representation matrix may be huge. Printing the result of this operation is a convenient way to display A as a normal (dense) matrix if A is small.

`SparseMatrix(A)`

Given a normal (dense-representation) matrix A, return the sparse matrix equal to A. Note that if there is a fast algorithm available for the sparse matrix type, there is no need to convert a dense-representation matrix to the sparse matrix type first and then to use that algorithm, since MAGMA does this automatically.

27.7 Changing Ring

`ChangeRing(A, R)`

`SparseMatrix(R, A)`

Given a sparse matrix A over a ring S having m rows and n columns, and another ring R, return the $m \times n$ sparse matrix over R obtained by coercing the entries of A from S into R.

The argument order to `ChangeRing(A, R)` here is consistent with other forms of `ChangeRing`, while the `SparseMatrix(R, A)` form of this function is provided to be consistent with the sparse matrix creation functions above, for which the destination ring is the first argument, if supplied.

27.8 Predicates

The functions in this section test various properties of sparse matrices.

`A eq B`

Given sparse matrices A and B, return `true` if and only if A and B are equal.

`IsZero(A)`

Given an $m \times n$ sparse matrix A over the ring R, return `true` iff A is the $m \times n$ zero sparse matrix.

`IsOne(A)`

Given a square $m \times m$ sparse matrix A over the ring R, return `true` iff A is the $m \times m$ identity sparse matrix.

`IsMinusOne(A)`

Given a square $m \times m$ sparse matrix A over the ring R, return `true` iff A is the negation of the $m \times m$ identity sparse matrix.

`IsScalar(A)`

Given a square $m \times m$ sparse matrix A over the ring R, return `true` iff A is scalar, i.e., iff A is the product of some element of R and the $m \times m$ identity sparse matrix.

`IsDiagonal(A)`

Given a square sparse matrix A over the ring R, return `true` iff A is diagonal, i.e., iff the only non-zero entries of A are on the diagonal.

`IsSymmetric(A)`

Given a square sparse matrix A over the ring R, return `true` iff A is symmetric, i.e., iff A equals its transpose.

`IsUpperTriangular(A)`

Given a sparse matrix A over the ring R, return `true` iff A is upper triangular, i.e., iff the only non-zero entries of A are on or above the diagonal.

`IsLowerTriangular(A)`

Given a sparse matrix A over the ring R, return `true` iff A is lower triangular, i.e., iff the only non-zero entries of A are on or below the diagonal.

27.9 Elementary Arithmetic

`A + B`

Given $m \times n$ sparse matrices A and B over a ring R, return $A + B$.

`A - B`

Given $m \times n$ sparse matrices A and B over a ring R, return $A - B$.

`A * B`

Given an $m \times n$ sparse matrix A over a ring R and an $n \times p$ sparse matrix B over R, return the $m \times p$ sparse matrix $A \cdot B$ over R.

`x * A`
`A * x`

Given an $m \times n$ sparse matrix A over a ring R and a ring element x coercible into R, return the scalar product $x \cdot A$.

`-A`

Given a sparse matrix A, return $-A$.

`A ^-1`

Given a invertible square sparse matrix A over a ring R, return the inverse B of A so that $A \cdot B = B \cdot A = 1$. The coefficient ring R must be either a field, a Euclidean domain, or a ring with an exact division algorithm and having characteristic equal to zero or greater than m (this includes most commutative rings).

`A ^ n`

Given a square sparse matrix A over a ring R and an integer n, return the matrix power A^n. A^0 is defined to be the identity matrix for any square matrix A (even if A is zero). If n is negative, A must be invertible (see the previous function), and the result is $(A^{-1})^{-n}$.

`Transpose(A)`

Given an $m \times n$ sparse matrix A over a ring R, return the transpose of A, which is simply the $n \times m$ sparse matrix over R whose (i, j)-th entry is the (j, i)-th entry of A.

27.10 Multiplying Vectors or Matrices by Sparse Matrices

These functions allow the multiplication of a normal (dense-representation) vector by a sparse matrix.

`v * A`

`V * A`

Given a dense-representation vector v or dense-representation matrix V with c columns, together with a sparse $c \times n$ matrix A, both over a ring R, return the product $v \cdot A$ or $V \cdot A$.

This is generally fast if A is sparse and uses minimal memory.

`MultiplyByTranspose(v, A)`

`MultiplyByTranspose(V, A)`

Given a dense-representation vector v or dense-representation matrix V with c columns, together with a sparse $n \times c$ matrix A, both over a ring R, return the product of v or V by the transpose of A.

This is generally fast if A is sparse, and is much faster than computing the transpose of A first. For example, if the vector-matrix product $v \cdot A \cdot A^{tr}$ is required, then the function call `MultiplyByTranspose(v*A, A)` should be used to avoid forming the matrix $A \cdot A^{tr}$ which is usually dense. This product occurs in iterative algorithms such as Lanzcos.

27.11 Non-trivial Properties

The following functions compute non-trivial properties of sparse matrices.

27.11.1 Nullspace and Rowspace

The following functions compute nullspaces (solving equations of the form $V \cdot A = 0$) or rowspaces of sparse matrices.

`Nullspace(A)`

`Kernel(A)`

Given an $m \times n$ sparse matrix A over a ring R, return the nullspace of A (or the kernel of A, considered as a linear transformation or map), which is the R-space consisting of all vectors v of length m such that $v \cdot A = 0$. Since the result will be given in the dense representation, both the nullity of A and the number of rows of A must both be reasonably small.

The algorithm first performs sparse elimination using Markowitz pivoting ([DEJ84, Sec. 9.2]) to obtain a smaller dense matrix, then the nullspace algorithm for dense-representation matrices is applied to this matrix.

`NullspaceMatrix(A)`

`KernelMatrix(A)`

Given an $m \times n$ sparse matrix A over a ring R, return a (dense-representation) basis matrix of the nullspace of A. This is a matrix N having m columns and the maximal number of independent rows subject to the condition that $N \cdot A = 0$. This function has the advantage that the nullspace is not returned as a R-space, so echelonization of the resulting nullspace may be avoided.

`NullspaceOfTranspose(A)`

This function is equivalent to `Nullspace(Transpose(A))`, but will be more efficient in space for large matrices, since the transpose may not have to be explicitly constructed to compute the nullspace.

`Rowspace(A)`

Given an $m \times n$ sparse matrix A over a ring R, return the rowspace of A, which is the R-space generated by the rows of A. Since the result will be given in the dense representation, the rank and the number of columns of A must both be reasonably small.

27.11.2 Rank

`Rank(A)`

Given an $m \times n$ sparse matrix A over a ring R, return the rank of A. The algorithm first performs sparse elimination using Markowitz pivoting ([DEJ84, Sec. 9.2]) to obtain a smaller dense matrix, then the rank algorithm for dense-representation matrices is applied to this matrix.

27.12 Determinant and Other Properties

`Determinant(A:` *parameters*`)`

`MonteCarloSteps` RngIntElt *Default :*

Given a square sparse matrix A over the ring R, return the determinant of A as an element of R. R may be any commutative ring.

The algorithm first performs sparse elimination using Markowitz pivoting ([DEJ84, Sec. 9.2]) to obtain a smaller dense matrix, then the determinant algorithm for dense-representation matrices is applied to this matrix. If the parameter `MonteCarloSteps` is given, then this is passed to the dense algorithm for the dense matrix.

27.12.1 Elementary Divisors (Smith Form)

`ElementaryDivisors(A)`

Given an $m \times n$ matrix A over the Euclidean ring or field R, return the elementary divisors of A. These are simply the non-zero diagonal entries of the Smith form of A, in order.

The divisors are returned as a sequence $Q = [e_1, \ldots, e_d]$, $e_i | e_{i+1}$ $(i = 1, \ldots, d-1)$ of d elements of R (which may include ones), where d is the rank of A. If R is a field, the result is always a sequence of r ones, where r is the rank of A.

A function for computing the Smith normal form is not supplied for sparse matrices since the form may be trivially derived from the elementary divisors, and the sequence Q containing the divisors is often more convenient (and takes less memory). As transformation matrices are dense in general, they are not supported for the sparse representation.

The algorithm first performs sparse elimination using Markowitz pivoting to obtain a smaller dense matrix ([DEJ84, Sec. 9.2]; this is similar to the techniques described in [HHR93]). Then it invokes the dense Smith normal form algorithm for normal (dense-representation) matrices (`SmithForm`).

27.12.2 Verbosity

`SetVerbose("SparseMatrix", v)`

(Procedure.) Set the verbose printing level for all sparse matrix algorithms to be v. Currently the legal values for v are `true`, `false`, 0, 1, 2, and 3 (`false` has the same effect as 0, and `true` has the same effect as 1).

27.13 Linear Systems (Structured Gaussian Elimination)

`ModularSolution(A, M)`

`ModularSolution(A, L)`

`Lanczos` BoolElt *Default :* `false`

Given a sparse $m \times n$ matrix A, defined over the integer ring $\mathbf{Z}$, and a positive integer M, compute a vector v such that v satisfies the equation $v \cdot A^{tr} = 0$ modulo M. v will be non-zero with high probability.

This function is designed for index-calculus-type algorithms where a large sparse linear system defined by the matrix A is first constructed and then a vector satisfying the above equation modulo M is required. For this reason it is natural that the transpose of A appears in this equation. The example H27E3 below illustrates such a situation in detail.

The first version of the function takes the actual integer M as the second argument given and so must be factored as part of the calculation, while the second

version of the function takes the factorization sequence L of M. If possible, the solution vector is multiplied by a unit modulo M so that its first entry is 1.

The function uses the *Structured Gaussian Elimination* algorithm [LO91b, Sec. 5]. This reduces the linear system to be solved to a much smaller but denser system. By default, the function recurses on the smaller system until it is almost completely dense, and then this dense system is solved using the fast dense modular nullspace algorithm of MAGMA.

If the parameter `Lanczos` is set to `true`, then the *Lanczos* algorithm [LO91b, Sec. 3] will be used instead. This is generally *very much* slower than the default method (it is often 10 to 50 times slower), but it will take considerably less memory, so may be preferable in the case of extremely large matrices.

For typical matrices arising in index-calculus problems, and for most machines, the default method (reducing to a completely dense system) should solve a linear system of size roughly $100,000 \times 100,000$ using about 500MB of memory while a linear system of size roughly $200,000 \times 200,000$ should require about 1.5GB to 2.0GB of memory.

Example H27E3

In this extended example, we demonstrate the application of the function `ModularSolution` to a sparse matrix arising in an index-calculus algorithm. We present MAGMA code which performs the first stage of the basic linear sieve [COS86, LO91a] for computing discrete logarithms in a prime finite field $\mathbf{F}_p$. This first stage determines most of the logarithms of the elements of the *factor basis* (which is the set of small primes up to a given limit) by using a *sieve* to compute a sparse linear system which is then solved modulo $p-1$.

Even though the following sieving code is written in the MAGMA language and so is *much* slower than a serious C implementation, it is sufficiently powerful to be able to compute the first stage logarithms in less than a minute for fields $\mathbf{F}_p$ where p is about 10^{20} and $(p-1)/2$ is prime. In comparison, the standard Pohlig-Hellman algorithm based on the Pollard-Rho method would take many hours for such fields. This MAGMA code can also be adapted for other index-calculus methods.

Suppose p is an odd prime and let $K = \mathbf{F}_p$. Let Q be the *factor base*, an ordered set consisting of all positive primes from 2 to a given limit `qlimit`. Fix a primitive element α of K which is also *prime* as an integer, so α is in Q. We wish to compute the discrete logarithms of most of the elements of Q with respect to α; i.e., we wish to compute l_q with $\alpha^{l_q} = q$ for most $q \in Q$.

The algorithm uses a *sieve* to search for linear relations involving the logarithms of the elements of Q. Let $H = \lfloor \sqrt{p} \rfloor + 1$ and $J = H^2 - p$. We search for pairs of integers (c_1, c_2) with $1 \le c_1, c_2 \le$ `climit` (where `climit` is a given limit which is much less than H) such that

$$[(H + c_1)(H + c_2)] \bmod p = J + (c_1 + c_2)H + c_1 c_2$$

is *smooth* with respect to Q (i.e., all of its prime factors are in Q). If we include these $H + c_i$ in the factor base, then this gives a linear relation modulo $(p-1)$ among the logarithms of the elements of the extended factor base.

Fix c_1 with $1 \leq c \leq$ `climit` and suppose we initialize a sieve array (to be indexed by c_2) to have zero in each position. For each prime power q^h with $q \in Q$ and h sufficiently small, we compute

$$d = [(J + c_1 H)(H + c_1)^{-1}] \bmod q^h.$$

Then for all $c_2 \equiv d \pmod{q^h}$, it turns out that

$$(H + c_1)(H + c_2) \equiv 0 \pmod{q^h}.$$

So for each c_2 with $c_1 \leq c_2 \leq$ `climit` and $c_2 \equiv d \pmod{q^h}$, we add $\log(q)$ to the position of the sieve corresponding to c_2. (Ensuring that $c_2 \geq c_1$ avoids redundant relations.) When we have done this for each $q \in Q$, we perform trial division to obtain relations for each of the c_2 whose sieve values are beneath a suitable threshold.

We repeat this with a new c_1 value until we have more relations than elements of the factor base (typically we make the ratio be 1.1 or 1.2), then we solve the corresponding linear system modulo $M = p - 1$ to obtain the desired logarithms. We ensure that α is the first element of Q so that when the vector is normalized modulo M (so that its first entry is 1), the logarithms will be with respect to α. For derivations of the above equations and for further details concerning the sieving, see [LO91a].

In practice, one first writes $M = p - 1 = M_1 \cdot M_2$ where M_1 contains the maximal powers of all the small primes dividing M (say, for primes ≤ 10000). The solution space modulo M_1 will often have high dimension, so the logarithms modulo M_1 usually cannot be correctly computed from the linear system alone. So we only compute the solution of the linear system modulo M_2. It is still possible that some logarithms cannot be determined modulo M_2 (e.g., if 2 unknowns occur only in one equation), but usually most of the logarithms will be correctly computed modulo M_2. Then the logarithm of each factor basis element can be easily computed modulo M_1 by the Pohlig-Hellman algorithm, and the Chinese Remainder Theorem can be used to combine these with the correct modulo-M_2 logarithms to compute the logarithms modulo M of most elements of the factor basis.

Similar index-calculus-type algorithms should have techniques for handling small prime divisors of the modulus M when the solution of the linear system has high nullity modulo these small primes.

The following function `Sieve` has been developed by Allan Steel, based on code by Benjamin Costello. Its arguments are the field $K = \mathbf{F}_p$, the factor base prime limit `qlimit`, the c_1, c_2 range limit `climit`, and the desired stopping ratio of relations to unknowns. The function returns the sparse relation matrix A together with an indexed set containing the corresponding extended factor base (the small primes and the $H + c_i$ values which yield relations).

```
> function Sieve(K, qlimit, climit, ratio)
>     p := #K;
>     Z := Integers();
>     H := Iroot(p, 2) + 1;
>     J := H^2 - p;
>
>     // Get factor basis of all primes <= qlimit.
>     fb_primes := [p: p in [2 .. qlimit] | IsPrime(p)];
```

```
>
>     printf "Factor base has %o primes, climit is %o\n", #fb_primes, climit;
>
>     // Ensure that the primitive element of K is prime (as an integer).
>     a := rep{x: x in [2..qlimit] | IsPrime(x) and IsPrimitive(K!x)};
>     SetPrimitiveElement(K,K!a);
>
>     // Initialize extended factor base FB to fb_primes (starting with a).
>     FB := {@ Z!a @};
>     for x in fb_primes do
>         Include(~FB, x);
>     end for;
>
>     // Initialize A to 0 by 0 sparse matrix over Z.
>     A := SparseMatrix();
>
>     // Get logs of all factor basis primes.
>     log2 := Log(2.0);
>     logqs := [Log(q)/log2: q in fb_primes];
>
>     for c1 in [1 .. climit] do
>
>         // Stop if ratio of relations to unknowns is high enough.
>         if Nrows(A)/#FB ge ratio then break; end if;
>
>         if c1 mod 50 eq 0 then
>             printf "c1: %o, #rows: %o, #cols: %o, ratio: %o\n",
>                 c1, Nrows(A), #FB, RealField(8)!Nrows(A)/#FB;
>         end if;
>
>         // Initialize sieve.
>         sieve := [z: i in [1 .. climit]] where z := Log(1.0);
>         den := H + c1;                  // denominator of relation
>         num := -(J + c1*H);             // numerator
>
>         for i := 1 to #fb_primes do
>             // For each prime q in factor base...
>             q := fb_primes[i];
>             logq := logqs[i];
>
>             qpow := q;
>             while qpow le qlimit do
>                 // For all powers qpow of q up to qlimit...
>
>                 if den mod qpow eq 0 then break; end if;
>                 c2 := num * Modinv(den, qpow) mod qpow;
>                 if c2 eq 0 then c2 := qpow; end if;
>
```

```
>                 nextqpow := qpow*q;
>                 // Ensure c2 >= c1 to remove redundant relations.
>                 while c2 lt c1 do
>                     c2 +:= qpow;
>                 end while;
>
>                 while c2 le #sieve do
>                     // Add logq into sieve for c2.
>                     sieve[c2] +:= logq;
>
>                     // Test higher powers of q if nextqpow is too large.
>                     if nextqpow gt qlimit then
>                         prod := (J + (c1 + c2)*H + c1*c2) mod p;
>                         nextp := nextqpow;
>                         while prod mod nextp eq 0 do
>                             sieve[c2] +:= logq;
>                             nextp *:= q;
>                         end while;
>                     end if;
>                     c2 +:= qpow;
>                 end while;
>                 qpow := nextqpow;
>             end while;
>         end for;
>
>         // Check sieve for full factorizations.
>         rel := den * (H + 1);   // the relation
>         relinc := H + c1;       // add to relation to get next relation
>         count := 0;
>         for c2 in [1 .. #sieve] do
>             n := rel mod p;
>             if Abs(sieve[c2] - Ilog2(n)) lt 1 then
>                 fact, r := TrialDivision(n, qlimit);
>                 if r eq [] and (#fact eq 0 or fact[#fact][1] lt qlimit) then
>                     // Include each H + c_i in extended factor basis.
>                     Include(~FB, H + c1);
>                     Include(~FB, H + c2);
>
>                     // Include relation (H + c1)*(H + c2) = fact.
>                     row := Nrows(A) + 1;
>                     for t in fact do
>                         SetEntry(~A, row, Index(FB, t[1]), t[2]);
>                     end for;
>                     if c1 eq c2 then
>                         SetEntry(~A, row, Index(FB, H + c1), -2);
>                     else
>                         SetEntry(~A, row, Index(FB, H + c1), -1);
>                         SetEntry(~A, row, Index(FB, H + c2), -1);
```

```
>                     end if;
>                 end if;
>             end if;
>             rel +:= relinc;
>         end for;
>     end for;
>
>     // Check matrix by multiplying out relations in field.
>     assert {&*[(K!FB[j])^A[k, j]: j in Support(A, k)]: k in [1..Nrows(A)]}
>             eq {1};
>
>     return A, FB;
> end function;
```

We first apply the function to a trivial example to illustrate the main points. We let K be $\mathbf{F}_{103}$, and we make the factor basis include primes up to 35, the c_i range be up to 27, and the stopping ratio be 1.1. We first compute the relation matrix A and the extended factor basis F.

```
> K := GF(103);
> A, F := Sieve(K, 35, 27, 1.1);
Factor base has 11 primes, climit is 27
> A;
Sparse matrix with 33 rows and 30 columns over Integer Ring
```

We examine a few rows of A and the extended factor basis F. Note that 5 is the smallest prime which is primitive in K, so it has been inserted first in F.

```
> A[1]; A[2]; A[30];
(1 0 0 0 0 1 0 0 0 0 0 -1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 1 1 0 0 0 0 0 0 -1 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 0 0 -1 0 0 0 0 0)
> F;
{@ 5, 2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 12, 14, 15, 16, 21, 38,
20, 26, 35, 22, 33, 34, 24, 25, 28, 30, 37, 32, 36 @}
```

Next we compute a solution vector v such that $v \cdot A^{tr} = 0$ modulo $M = \#K - 1$.

```
> Factorization(#K-1);
[ <2, 1>, <3, 1>, <17, 1> ]
> v := ModularSolution(A, #K - 1);
> v;
(1 44 39 4 61 72 70 80 24 86 57 25 48 40 74 43 22 0 1 5 3 100 12 69 2
    92 84 93 16 64)
```

Notice that 0 occurs in v, so the corresponding logarithm is not known. The vector is normalized so that the logarithm of 5 is 1, as desired. We finally compute the powers of the primitive element of K by each element of v and check that all of these match the entries of F except for the one missed entry. Note also that because M has small prime divisors, it is fortunate that the logarithms are all correct, apart from the missed one.

```
> a := PrimitiveElement(K);
```

```
> a;
5
> Z := IntegerRing();
> [a^Z!x: x in Eltseq(v)];
[ 5, 2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 12, 14, 15, 16, 21, 38, 1,
5, 35, 22, 33, 34, 24, 25, 28, 30, 37, 32, 36 ]
> F;
{@ 5, 2, 3, 7, 11, 13, 17, 19, 23, 29, 31, 12, 14, 15, 16, 21, 38, 20,
26, 35, 22, 33, 34, 24, 25, 28, 30, 37, 32, 36 @}
```

Finally, we take $K = \mathbf{F}_p$, where p is the smallest prime above 10^{20} such that $(p-1)/2$ is prime. The sparse matrix constructed here is close to 1000×1000, but the solution vector is still found in less than a second.

```
> K := GF(100000000000000000763);
> Factorization(#K - 1);
[ <2, 1>, <50000000000000000381, 1> ]
> time A, F := Sieve(K, 2200, 800, 1.1);
Factor base has 327 primes, climit is 800
c1: 50, #rows: 222, #cols: 555, ratio: 0.4
c1: 100, #rows: 444, #cols: 738, ratio: 0.60162602
c1: 150, #rows: 595, #cols: 836, ratio: 0.71172249
c1: 200, #rows: 765, #cols: 921, ratio: 0.83061889
c1: 250, #rows: 908, #cols: 973, ratio: 0.9331963
c1: 300, #rows: 1014, #cols: 1011, ratio: 1.003
c1: 350, #rows: 1105, #cols: 1023, ratio: 1.0802
Time: 3.990
> A;
Sparse matrix with 1141 rows and 1036 columns over Integer Ring
> time v := ModularSolution(A, #K - 1);
Time: 0.170
```

We observe that the list consisting of the primitive element powered by the computed logarithms agrees with the factor basis for at least the first 30 elements.

```
> a := PrimitiveElement(K);
> a;
2
> v[1], v[2];
1 71610399209536789314
> a^71610399209536789314;
3
> P := [a^x: x in Eltseq(v)];
> [P[i]: i in [1 .. 30]];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ]
> [F[i]: i in [1 .. 30]];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
```

```
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ]
```

There are in fact 8 logarithms which could not be computed correctly.

```
> [i: i in [1..#F] | P[i] ne F[i]];
[ 671, 672, 673, 737, 738, 947, 1024, 1025 ]
```

A zero value appearing in the place of a logarithm indicates that the nullity of the system was larger than 1, even considered modulo $(\#K - 1)/2$.

```
> [v[i]: i in [1..#F] | P[i] ne F[i]];
[ 0, 22076788376647522787, 73252391663364176895, 0,
33553634905886614528, 42960107136526083388, 0, 57276316725691590267 ]
> [P[i]: i in [1..#F] | P[i] ne F[i]];
[ 1, 7480000052677, 7960000056517, 1, 5220000041437,
8938028430619715763, 1, 11360000275636 ]
```

However, we have found the correct logarithms for nearly all factor basis elements. As pointed out above, the Pollard-Rho method applied to an element of the factor basis for this field would take many hours!

27.14 Bibliography

[**COS86**] D. Coppersmith, A. M. Odlyzko, and R. Schroeppel. Discrete logarithms in GF(p). *Algorithmica*, 1:1–15, 1986.

[**DEJ84**] I.S. Duff, A.M. Erisman, and J.K.Reid. *Direct methods for sparse matrices.* Monographs on Numerical Analysis. Oxford University Press, 1984.

[**HHR93**] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented Z-modules. *Linear Algebra and its Applications*, 192:137–164, 1993.

[**LO91a**] B. A. LaMacchia and A. M. Odlyzko. Computation of Discrete Logarithms in Prime Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of *INCS*, pages 616–618. Springer-Verlag, 1991.

[**LO91b**] B. A. LaMacchia and A. M. Odlyzko. Solving Large Sparse Linear Systems over Finite Fields. In A.J. Menezes and S. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990*, volume 537 of *LNCS*, pages 109–133. Springer-Verlag, 1991.

28 VECTOR SPACES

Chapter 28
VECTOR SPACES

28.1 Introduction

In this chapter we will discuss vector spaces and their linear transformations. Let K be a field. In Magma, the standard K-vector space is taken to be the set of n-tuples over the field K, which we shall write as $K^{(n)}$.

A rectangular matrix over a field K is considered to be an element of the vector space consisting of all $m \times n$ matrices over K. This vector space will be written as $K^{(m\times n)}$. Let U and V be K-vector spaces of dimensions m and n, respectively. The set of all linear transformations with domain U and codomain V will be denoted by $\mathrm{Hom}_K(U, V)$. Once bases have been chosen for U and V, we may identify $\mathrm{Hom}_K(U, V)$ with $K^{(m\times n)}$. Thus, $K^{(m\times n)}$ is first of all a vector space and all the normal vector space operations apply. However, since it is also a set of mappings, some additional operations arising from this characterization apply.

We shall use the term *vector space* or *K-vector space* (if we wish to emphasize the coefficient field) to refer to both the space $K^{(n)}$ and the space $K^{(m\times n)}$. If we wish to differentiate between the two, we shall use the term *tuple space* when referring to $K^{(n)}$ and the term *matrix space* referring to $K^{(m\times n)}$.

28.1.1 Vector Space Categories

The family of all finite dimensional vector spaces over a given field K forms a category, while the set of all finite dimensional vector spaces forms a family of categories indexed by the field K. In this family of categories, objects are vector spaces and the morphisms are linear transformations. The (indexed family of) categories consisting of vector spaces of n-tuples has the name `ModTupFld`, while the (indexed family of) categories consisting of vector spaces of $m \times n$-matrices has the name `ModMatFld`.

28.1.2 The Construction of a Vector Space

Every vector space V defined over a field K is created either as a subspace of the row space $K^{(n)}$ (tuple spaces) or as a subspace of $K^{(m\times n)}$ (matrix modules). Thus, the construction of a general vector space is a two step process:

(i) The appropriate row space $K^{(n)}$, is constructed;

(ii) The required vector space V is then defined as a subspace or quotient space of $K^{(n)}$.

28.2 Creation of Vector Spaces and Arithmetic with Vectors

28.2.1 Construction of a Vector Space

`VectorSpace(K, n)`

`KSpace(K, n)`

Given a field K and a non-negative integer n, create the n-dimensional vector space $V = K^{(n)}$, consisting of all n-tuples over K. The vector space is created with respect to the standard basis, $e_1, \ldots, e_n$, where e_i $(i = 1, \ldots, n)$ is the vector containing a 1 in the i-th position and zeros elsewhere.

Use of the functions `VectorSpace` and `KSpace` ensures that subspaces of V will be presented in embedded form.

`KModule(K, n)`

Given a field K and a non-negative integer n, create the n-dimensional vector space $V = K^{(n)}$, consisting of all n-tuples over K. The vector space is created with respect to the standard basis, $e_1, \ldots, e_n$, where e_i $(i = 1, \ldots, n)$ is the vector containing a 1 in the i-th position and zeros elsewhere.

Use of the function `KModule` ensures that subspaces of V will be presented in reduced form. In all other respects, a vector space created by this function is identical to one created by `KSpace`.

`KMatrixSpace(K, m, n)`

Given a field K and integers m and n greater than one, create the vector space $K^{(m \times n)}$, consisting of all $m \times n$ matrices over K. The vector space is created with the standard basis, $\{E_{ij} \mid i = 1 \ldots, m, j = 1 \ldots, n\}$, where E_{ij} is the matrix having a 1 in the (i, j)-th position and zeros elsewhere.

Note that for a matrix space, subspaces will always be presented in embedded form, i.e. there is no reduced mode available for matrix spaces.

`Hom(V, W)`

If V is the vector space $K^{(m)}$ and W is the vector space $K^{(n)}$, create the matrix space $\mathrm{Hom}_K(V, W)$ as the vector space $K^{(m \times n)}$, represented as the set of all $m \times n$ matrices over K. The vector space is created with the standard basis, $\{E_{ij} \mid i = 1 \ldots, m, j = 1 \ldots, n\}$, where E_{ij} is the matrix having a 1 in the (i, j)-th position and zeros elsewhere.

Example H28E1

We construct the vector space V consisting of 6-tuples over the rational field.

```
> Q := RationalField();
> V := VectorSpace(Q, 6);
> V;
Vector space of dimension 6 over Rational Field
```

Example H28E2

We construct the matrix space M consisting of 3×5 matrices over the field $\mathbf{Q}(\sqrt{5})$.

```
> K<w> := QuadraticField(5);
> V := KMatrixSpace(K, 3, 5);
> V;
Full Vector Space of 3 by 5 matrices over Quadratic Field Q(w)
```

28.2.2 Construction of a Vector Space with Inner Product Matrix

`VectorSpace(K, n, F)`

`KSpace(K, n, F)`

Given a field K, a non-negative integer n and a square $n \times n$ symmetric matrix F, create the n-dimensional vector space $V = K^{(n)}$ (in embedded form), with inner product matrix F. This is the same as `VectorSpace(K, n)`, except that the functions `Norm` and `InnerProduct` (see below) will be with respect to the inner product matrix F.

28.2.3 Construction of a Vector

`elt< V | L >`

(1) Suppose V is a subspace of the vector space $K^{(n)}$. Given elements $a_1, \ldots, a_n$ belonging to K, construct the vector $v = (a_1, \ldots, a_n)$ as a vector of V. Note that if v is not an element of V, an error will result.

(2) Suppose V is a subspace of the matrix space $K^{(m \times n)}$. Given elements $a_1, \ldots, a_{mn}$ belonging to K, construct the matrix $m = (a_1, \ldots, a_{mn})$ as an element of V. Note that if m is not an element of V, an error will result.

`V ! Q`

(1) Suppose V is a subspace of the vector space $K^{(n)}$. Given elements $a_1, \ldots, a_n$ belonging to K, construct the vector $v = (a_1, \ldots, a_n)$ as a vector of V. Note that if v is not an element of V, an error will result.

(2) Suppose V is a subspace of the matrix space $K^{(m \times n)}$. Given elements $a_1, \ldots, a_{mn}$ belonging to K, construct the matrix $m = (a_1, \ldots, a_{mn})$ as an element of V. Note that if m is not an element of V, an error will result.

`CharacteristicVector(V, S)`

Given a subspace V of the vector space $K^{(n)}$ together with a set S of integers lying in the interval $[1, n]$, return the characteristic number of S as a vector of V.

`V ! 0`

`Zero(V)`

The zero element for the vector space V.

`Random(V)`

Given a vector space V defined over a finite field, return a random vector.

Example H28E3

We create the 5-dimensional vector space V over $\mathbf{F}_4$ and define the vector $u = (1, w, 1 + w, 0, 0)$, where w is a primitive element of $\mathbf{F}_4$.

```
> K<w> := GaloisField(4);
> V    := VectorSpace(K, 5);
> u    := V ! [1, w, 1+w, 0, 0];
> u;
(1 w w + 1 0 0)
> zero := V ! 0;
> zero;
(0 0 0 0 0)
r := Random(V);
(1 0 w 1 w + 1)
```

Example H28E4

We create an element belonging to the space of 3×4 matrices over the number field $\mathbf{Q}(w)$, where w is a root of $x^7 - 7x + 3$.

```
> R<x> := PolynomialRing(RationalField());
> L<w> := NumberField(x^7 - 7*x + 3);
> L34 := KMatrixSpace(L, 3, 4);
> a := L34 ! [ 1, w, 0, -w, 0, 1+w, 2, -w^3, w-w^3, 2*w, 1/3, 1 ];
> a;
[1    w    0    -1 * w]
```

```
[0    w + 1    2    -1 * w^3]
[-1 * w^3 + w    2 * w    1/3    1]
```

28.2.4 Deconstruction of a Vector

`ElementToSequence(u)`

`Eltseq(u)`

Given an element u belonging to the K-vector space V, return u in the form of a sequence Q of elements of V. Thus, if u is an element of $K^{(n)}$, then $Q[i] = u[i]$, $1 \leq i \leq n$.

28.2.5 Arithmetic with Vectors

For the following operations the vectors u and v must belong to the same vector space i.e. the same tuple space $K^{(n)}$ or the same matrix space $K^{(m \times n)}$. The scalar a must belong to the field K.

`u + v`

Sum of the vectors u and v, where u and v lie in the same vector space.

`-u`

Additive inverse of the vector u.

`u - v`

Difference of the vectors u and v, where u and v lie in the same vector space.

`x * u`

`u * x`

The scalar product of the vector u belonging to the K-vector space and the field element x belonging to K.

`u / x`

The scalar product of the vector u belonging to the K-vector space and the field element $1/x$ belonging to K where x is non-zero.

`NumberOfColumns(u)`

`Ncols(u)`

The number of columns in the vector u.

`Depth(u)`

The index of the first non-zero entry of the vector u (0 if none such).

`(u, v)`

`InnerProduct(u, v)`

Return the inner product of the vectors u and v with respect to the inner product defined on the space. If an inner product matrix F is given when the space is created, then this is defined to be $u \cdot F \cdot v^{tr}$. Otherwise, this is simply $u \cdot v^{tr}$.

`IsZero(u)`

Returns `true` iff the vector u belonging to a vector space is the zero element.

`Norm(u)`

Return the norm product of the vector u with respect to the inner product defined on the space. If an inner product matrix F is given when the space is created, then this is defined to be $u \cdot F \cdot u^{tr}$. Otherwise, this is simply $u \cdot u^{tr}$.

`Normalise(u)`

`Normalize(u)`

Given an element u, not the zero element, belonging to the K-vector space V, return $\frac{1}{a} * u$, where a is the first non-zero component of u. If u is the zero vector, it is returned. The net effect is that `Normalize(u)` always returns a vector v in the subspace generated by u, such that the first non-zero component of v (if existent) is $K!1$.

`Rotate(u, k)`

Given a vector u, return the vector obtained from u by rotating by k coordinate positions.

`Rotate(~u, k)`

Given a vector u, destructively rotate u by k coordinate positions.

`NumberOfRows(u)`

`Nrows(u)`

The number of rows in the vector u (1 of course; included for completeness).

`Support(u)`

A set of integers giving the positions of the non-zero components of the vector u.

`TensorProduct(u, v)`

The tensor (Kronecker) product of the vectors u and v. The resulting vector has degree equal to the product of the degrees of u and v.

`Trace(u, F)`

`Trace(u)`

Given a vector belonging to the space $K^{(n)}$, and a subfield F of K, return the vector obtained by replacing each component of u by its trace over the subfield F. If F is the prime field of K, it may be omitted.

`Weight(u)`

The number of non-zero components of the vector u.

Example H28E5

We illustrate the use of the arithmetic operators for module elements by applying them to elements of the 4-dimensional vector space over the field $\mathbf{Q}(w)$, where w is an 8-th root of unity.

```
> K<w> := CyclotomicField(8);
> V := VectorSpace(K, 4);
> x := V ! [ w, w^ 2, w^ 4, 0];
> y := V ! [1, w, w^ 2, w^ 4];
> x + y;
((1 + w) (w + w^2) (-1 + w^2) -1)
> -x;
((-w) (-w^2)         1         0)
> x - y;
((-1 + w) (-w + w^2) (-1 - w^2) 1)
> w * x;
(  (w^2) (w^3)  (-w)          0)
> y * w^ -4;
(       -1 (-w) (-w^2)         1)
> Normalize(x);
(         1   (w) (w^3)          0)
> InnerProduct(x, y);
(w - w^2 + w^3)
> z := V ! [1, 0, w, 0 ];
> z;
(       1        0 (w)        0)
> Support(z);
{ 1, 3 }
```

Example H28E6

We illustrate how one can define a non-trivial inner product on a space.

```
> Q := RationalField();
> F := SymmetricMatrix(Q, [1, 0,2, 0,0,3, 1,2,3,4]);
> F;
[1 0 0 1]
[0 2 0 2]
```

```
[0 0 3 3]
[1 2 3 4]
> V := VectorSpace(Q, 4, F);
> V;
Full Vector space of degree 4 over Rational Field
Inner Product Matrix:
[1 0 0 1]
[0 2 0 2]
[0 0 3 3]
[1 2 3 4]
> v := V![1,0,0,0];
> Norm(v);
1
> w := V![0,1,0,0];
> Norm(w);
2
> InnerProduct(v, w);
0
> z := V![0,0,0,1];
> Norm(z);
4
> InnerProduct(v, z);
1
> InnerProduct(w, z);
2
```

28.2.6 Indexing Vectors and Matrices

The indexing operations have a different meaning depending upon whether they are applied to a tuple space or a matrix space.

`u[i]`

`u[i, j]`

Given an vector u belonging to a K-vector space V, the result of this operation depends upon whether V is a tuple or matrix space.

If V is a subspace of $K^{(n)}$, and i, $1 \leq i \leq n$, is a positive integer, the i-th component of the vector u is returned (as an element of the field K).

If V is a subspace of $K^{(m \times n)}$, and i, $1 \leq i \leq m$, is a positive integer, $u[i]$ will return the i-th row of the matrix u (as an element of the vector space $K^{(n)}$). Similarly, if i and j, $1 \leq i \leq m$, $1 \leq j \leq n$, are positive integers, $u[i, j]$ will return the (i, j)-th component of the matrix u (as an element of K).

`u[i] := x`

`u[i] := x`

`u[i, j] := x`

Given an vector u belonging to a K-vector space V, and an element x of K, the result of this operation depends upon whether V is a tuple or matrix space.

If V is a subspace of $K^{(n)}$, and i, $1 \leq i \leq n$, is a positive integer, the i-th component of the vector u is redefined to be x.

If V is a subspace of $K^{(m \times n)}$ and $1 \leq i \leq m$ is a positive integer and x is an element of $K^{(n)}$, `u[i] := x` will redefine the i-th row of the matrix u to be the vector x, where x must be an element of $K^{(n)}$. Similarly, if $1 \leq i \leq m$, $1 \leq j \leq n$, are positive integers, `u[i, j] := x` will redefine the (i, j)-th component of the matrix u to be x, where x must be an element of K.

Example H28E7

We illustrate the use of the indexing operators for vector space elements by applying them to a 3-dimensional tuple space and a 2×3 matrix space over the field $\mathbf{Q}(w)$, where w is an 8-th root of unity.

```
> K<w> := CyclotomicField(8);
> V := VectorSpace(K, 3);
> u := V ! [ 1 + w, w^ 2, w^ 4];
> u;
((1 + w)     (w^2)          -1)
> u[3];
-1
> u[3] := 1 + w - w^7;
> u;
((1 + w) (w^2) (1 + w + w^3))
> // We now demonstrate indexing a matrix space
> W := KMatrixSpace(K, 2, 3);
> l := W ! [ 1 - w, 1 + w, 1 + w + w^2, 0, 1 - w^7, 1 - w^3 + w^6 ];
> l;
[(1 - w) (1 + w) (1 + w + w^2)]
[0 (1 + w^3) (1 - w^2 - w^3)]
> l[2];
(0 (1 + w^3) (1 - w^2 - w^3))
> l[2,2];
(1 + w^3)
> m := l[2];
> m;
(0 (1 + w^3) (1 - w^2 - w^3))
> l[2] := u;
> l;
[(1 - w) (1 + w) (1 + w + w^2)]
[(1 + w) (w^2) -1]
> l[2, 3] := 1 + w - w^7;
```

```
> l;
[(1 - w) (1 + w) (1 + w + w^2)]
[(1 + w) (w^2) (1 + w + w^3)]
```

28.3 Subspaces, Quotient Spaces and Homomorphisms

28.3.1 Construction of Subspaces

The conventions defining the presentations of subspaces and quotient spaces are as follows:

If V has been created using the function `VectorSpace` or `MatrixSpace`, then every subspace and quotient space of V is given in terms of a basis consisting of elements of V, i.e. by means of an embedded basis.

If V has been created using the function `RModule`, then every subspace and quotient space of V is given in terms of a reduced basis.

`sub< V | L >`

Given a K-vector space V, construct the subspace U generated by the elements of V specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of K defining an element of V;

(b) A set or sequence whose terms are elements of V;

(c) A subspace of V;

(d) A set or sequence whose terms are subspaces of V.

The generators stored for U consist of the vectors specified by terms L_i together with the stored generators for subspaces specified by terms of L_i. Repetitions of a vector and occurrences of the zero vector are removed (unless U is the trivial subspace).

The constructor returns the subspace U and the inclusion homomorphism $f : U \to V$. If V is of embedded type, the basis constructed for U consists of elements of V. If V is of standard type, a standard basis is constructed for U.

`Morphism(U, V)`

Assuming the vector space U has been created as a subspace of V, the function returns the matrix defining the embedding of U into V.

Example H28E8

The ternary Golay code is a six-dimensional subspace of the vector space $K^{(11)}$, where K is $\mathbf{F}_3$. This subspace is first constructed in the space constructed by the `VectorSpace` function.

```
> K11 := VectorSpace(FiniteField(3), 11);
> G3 := sub< K11  |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> G3;
Vector space of degree 11, dimension 6 over GF(3)
Generators:
(1 0 0 0 0 0 1 1 1 1 1)
(0 1 0 0 0 0 0 1 2 2 1)
(0 0 1 0 0 0 1 0 1 2 2)
(0 0 0 1 0 0 2 1 0 1 2)
(0 0 0 0 1 0 2 2 1 0 1)
(0 0 0 0 0 1 1 2 2 1 0)
Echelonized basis:
(1 0 0 0 0 0 1 1 1 1 1)
(0 1 0 0 0 0 0 1 2 2 1)
(0 0 1 0 0 0 1 0 1 2 2)
(0 0 0 1 0 0 2 1 0 1 2)
(0 0 0 0 1 0 2 2 1 0 1)
(0 0 0 0 0 1 1 2 2 1 0)
```

Example H28E9

We now construct the ternary Golay code starting with the vector space constructed using the `RModule` function. In this case the subspace is presented on a reduced basis.

```
> K11 := RModule(FiniteField(3), 11);
> G3 := sub< K11 |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> G3;
KModule G3 of dimension 6 with base ring GF(3)
> Basis(G3);
[
    G3: (1 0 0 0 0 0),
    G3: (0 1 0 0 0 0),
    G3: (0 0 1 0 0 0),
    G3: (0 0 0 1 0 0),
    G3: (0 0 0 0 1 0),
    G3: (0 0 0 0 0 1)
]
> f := Morphism(G3, K11);
```

```
> f;
[1 0 0 0 0 0 1 1 1 1 1]
[0 1 0 0 0 0 0 1 2 2 1]
[0 0 1 0 0 0 1 0 1 2 2]
[0 0 0 1 0 0 2 1 0 1 2]
[0 0 0 0 1 0 2 2 1 0 1]
[0 0 0 0 0 1 1 2 2 1 0]
```

28.3.2 Construction of Quotient Vector Spaces

`quo< V | L >`

Given a K-vector space V, construct the quotient vector space $W = V/U$, where U is the subspace generated by the elements of V specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of K defining an element of V;

(b) A set or sequence whose terms are elements of V;

(c) A subspace of V;

(d) A set or sequence whose terms are subspaces of V.

The generators constructed for U consist of the elements specified by terms L_i together with the stored generators for subspaces specified by terms of L_i.

The constructor returns the quotient space W and the natural homomorphism $f : V \to W$.

`V / U`

Given a subspace U of the vector space V, construct the quotient space W of V by U. If r is defined to be $\dim(V) - \dim(U)$, then W is created as an r-dimensional vector space relative to the standard basis.

The constructor returns the quotient space W and the natural homomorphism $f : V \to W$.

Example H28E10

We construct the quotient of $K^{(11)}$ by the Golay code.

```
> K11 := VectorSpace(FiniteField(3), 11);
> Q3, f := quo< K11 |
>  [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>  [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>  [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> Q3;
Full Vector space of degree 5 over GF(3)
> f;
Mapping from: ModTupFld: K11 to ModTupFld: Q3
```

Example H28E11

If we wished to construct this quotient of $K^{(11)}$ as a subspace of the original space, we could do so using the `Complement` function.

```
> K11 := VectorSpace(FiniteField(3), 11);
> S := sub< K11 |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> Complement(K11, S);
Vector space of degree 11, dimension 5 over GF(3)
Echelonized basis:
(0 0 0 0 0 0 1 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0)
(0 0 0 0 0 0 0 0 1 0 0)
(0 0 0 0 0 0 0 0 0 1 0)
(0 0 0 0 0 0 0 0 0 0 1)
```

Example H28E12

We construct a subspace and its quotient space in $\mathbf{Q}^{(3\times 4)}$.

```
> Q := RationalField();
> Q3 := VectorSpace(Q, 3);
> Q4 := VectorSpace(Q, 4);
> H34 := Hom(Q3, Q4);
> a := H34 ! [ 2, 0, 1, -1/2,  1, 0, 3/2, 4,  4/5, 6/7, 0, -1/3];
> b := H34 ! [ 1/2, -3, 0, 5,  1/3, 2, 4/5, 0,  5, -1, 5, 7];
> c := H34 ! [ -1, 4/9, 1, -4,  5, -5/6, -3/2, 0,  4/3, 7, 0, 7/9];
> d := H34 ! [ -3, 5, 1/3, -1/2,  2/3, 4, -2, 0,  0, 4, -1, 0];
> a, b, c, d;
[   2    0    1 -1/2]
[   1    0  3/2    4]
[ 4/5  6/7    0 -1/3]

[1/2  -3   0   5]
[1/3   2 4/5   0]
[  5  -1   5   7]

[  -1  4/9    1   -4]
[   5 -5/6 -3/2    0]
[ 4/3    7    0  7/9]

[  -3    5  1/3 -1/2]
[ 2/3    4   -2    0]
[   0    4   -1    0]
> U := sub< H34 | a, b, c, d >;
> U:Maximal;
```

```
KMatrixSpace of 3 by 4 GHom matrices and dimension 4 over Rational Field
Echelonized basis:

[1    0    0    0]
[-33872/30351    -5164/10117    42559/50585    11560/10117]
[-10514/10117    -121582/70819    -8476/10117    -48292/30351]

[           0            1            0            0]
[ -7797/10117   4803/10117 12861/101170   5940/10117]
[ -7818/10117 -38214/70819  -7821/10117 -10967/10117]

[           0            0            1            0]
[ 31261/10117  28101/20234  -2157/20234  18552/10117]
[161802/50585 291399/70819  20088/10117  33419/10117]

[          0           0           0           1]
[-8624/30351  7445/10117  7696/50585  2408/10117]
[32388/50585 -3562/10117  6272/10117 27580/30351]
> W := H34/U;
> W;
Full Vector space of degree 8 over Rational Field
```

28.4 Changing the Coefficient Field

The standard constructions described in section 31.5 for R-modules may be applied to vector spaces. In addition, we may extend or restrict the field of scalars, using the functions described here.

`ExtendField(V, L)`

Given a K-vector space V, with K a field and L an extension of K, construct the L-vector space $U = V \otimes_K L$. The function returns

(a) the vector space U; and

(b) the inclusion homomorphism $\phi : V \to U$.

`RestrictField(V, L)`

Given a K-vector space V, with K a field and L a subfield of K, construct the L-vector space U consisting of those vectors of V having all of their components lying in the subfield L. The function returns

(a) the vector space U; and

(b) the restriction homomorphism $\phi : V \to U$.

`VectorSpace(V, F)`

`KSpace(V, F)`

`KMatrixSpace(V, F)`

`KModule(V, F)`

Given an n-dimensional K-vector space V, and a subfield F of a finite field or cyclotomic field K such that K has degree m over F, construct a vector space U of dimension mn over the field F. The function returns

(a) the vector space U; and

(b) a mapping $\phi : V \to U$ such that a vector $(v_1, \ldots, v_i, \ldots, v_n)$ of V is mapped into the vector

$$(u_{11}, \ldots, u_{1n}, \ldots, u_{i1}, \ldots, u_{in}, \ldots, u_{n1}, \ldots u_{nn}),$$

where $(u_{i1}, \ldots, u_{in})$ is the field element v_i written as a vector over the subfield F.

28.5 Basic Operations

28.5.1 Accessing Vector Space Invariants

`V . i`

Given an vector space V and a positive integer i, return the i-th generating element of V.

`CoefficientField(V)`

`BaseField(V)`

Given a K-vector space V, return the field K.

`Degree(V)`

Given a K-vector space V which is a subspace of $K^{(n)}$, return n.

`Degree(u)`

Given an vector u belonging to a subspace of the vector space $K^{(n)}$, return n.

`Dimension(V)`

The dimension of the vector space V.

`Generators(V)`

The generators for the vector space V, returned as a set.

`NumberOfGenerators(M)`
`Ngens(M)`

The number of generators for the vector space V.

`OverDimension(V)`

Given a K-vector space V which is a subspace of $K^{(n)}$, return n.

`OverDimension(u)`

Given an vector u belonging to a subspace of the vector space $K^{(n)}$, return n.

`Generic(V)`

The generic vector space containing V, i.e. the full vector space in which V is naturally embedded.

`Parent(V)`

The power structure for the vector space V (the set consisting of all finite dimensional vector spaces).

28.5.2 Membership and Equality

`v in V`

Returns `true` if the element v lies in the vector space V, where v and V belong to a common space.

`v notin V`

Returns `true` if the element v does not lie in the vector space V, where v and V belong to a common space.

`U subset V`

Returns `true` if the K-vector space U is contained in the K-vector space V, where U and V are subspaces of some common vector space.

`U notsubset V`

Returns `true` if the K-vector space U is not contained in the K-vector space V, where U and V are subspaces of some common vector space.

`U eq V`

Returns `true` if the subspaces U and V are equal, where U and V belong to a common vector space.

`U ne V`

Returns `true` if the subspaces U and V are not equal, where U and V belong to a common vector space.

28.5.3 Operations on Subspaces

`U + V`

Sum of the subspaces U and V, where U and V must be subspaces of a common vector space.

`U meet V`

Intersection of the subspaces U and V, where U and V must be subspaces of a common vector space.

`U meet:= V`

Replace U with the intersection of the subspaces U and V, where U and V must be subspaces of a common vector space.

`&meet S`

Intersection of the subspaces of the set or sequence S, which must be subspaces of a common vector space.

`TensorProduct(U, V)`

The tensor (Kronecker) product of the vector spaces U and V, generated by all the tensor products of elements of U by elements of V. The resulting vector space has degree equal to the product of the degrees of U and V.

`Complement(V, U)`

Given a subspace U of the vector space V, construct a complement for U in V (a subspace of V).

`Transversal(V, U)`

Given a subspace U of the vector space V over a finite field, return a transversal for U in V as a set of vectors.

28.6 Reducing Vectors Relative to a Subspace

`ReduceVector(W, v)`

(Function.) Given a vector v from a tuple module V and a submodule W of V, return the reduction of v with respect to W (that is, the canonical representative of the coset $v + W$).

`ReduceVector(W, ~v)`

(Procedure.) Given a vector v from a tuple module V and a submodule W of V, replace v with its reduction of with respect to W (that is, the canonical representative of the coset $v + W$).

`DecomposeVector(U, v)`

Given a vector v from a tuple module V and a submodule U of V, return the unique u in U and w in the complement to U in $U+ < v >$ such that $v = u + w$.

28.7 Bases

This section is concerned with the construction of bases for vector spaces.

`VectorSpaceWithBasis(Q)`
`VectorSpaceWithBasis(a)`
`KSpaceWithBasis(Q)`
`KSpaceWithBasis(a)`
`KModuleWithBasis(Q)`

Create a vector space having as basis the terms of B (rows of a).

`Basis(V)`

The current basis for the vector space V, returned as a sequence of vectors.

`BasisElement(V, i)`

The i-th basis element for the vector space V.

`BasisMatrix(V)`

The current basis for the vector space V, returned as the rows of a matrix belonging to the matrix space $K^{(m\times n)}$, where m is the dimension of V and n is the over-dimension of V.

`Coordinates(V, v)`

Given a vector v belonging to the r-dimensional K-vector space V, with basis $v_1, \ldots, v_r$, return a sequence $[a_1, \ldots, a_r]$ of elements of K giving the coordinates of v relative to the V-basis: $v = a_1 * v_1 + \cdots + a_r * v_r$.

`Dimension(V)`

The dimension of the vector space V.

`ExtendBasis(Q, U)`

Given a sequence Q containing r linearly independent vectors belonging to the vector space U, extend the vectors of Q to a basis for U. The basis is returned in the form of a sequence T such that $T[i] = Q[i], i = 1, \ldots r$.

`ExtendBasis(U, V)`

Given an r-dimensional subspace U of the vector space V, return a basis for V in the form of a sequence T of elements such that the first r elements correspond to the given basis vectors for U.

`IsIndependent(S)`

Given a set S of elements belonging to the vector space V, return `true` if the elements of S are linearly independent.

`IsIndependent(Q)`

Given a sequence Q of elements belonging to the vector space V, return `true` if the terms of Q are linearly independent.

Example H28E13

These operations will be illustrated in the context of the subspace $G3$ of the 11-dimensional vector space over $\mathbf{F}_3$ defining the ternary Golay code.

```
> V11 := VectorSpace(FiniteField(3), 11);
> G3  := sub< V11 |  [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>                    [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>                    [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> Dimension(G3);
    6
> Basis(G3);
[
    (1 0 0 0 0 0 1 1 1 1 1),
    (0 1 0 0 0 0 0 1 2 2 1),
    (0 0 1 0 0 0 1 0 1 2 2),
    (0 0 0 1 0 0 2 1 0 1 2),
    (0 0 0 0 1 0 2 2 1 0 1),
    (0 0 0 0 0 1 1 2 2 1 0)
]
> S := ExtendBasis(G3, V11);
> S;
[
    (1 0 0 0 0 0 1 1 1 1 1),
    (0 1 0 0 0 0 0 1 2 2 1),
    (0 0 1 0 0 0 1 0 1 2 2),
    (0 0 0 1 0 0 2 1 0 1 2),
    (0 0 0 0 1 0 2 2 1 0 1),
    (0 0 0 0 0 1 1 2 2 1 0),
    (0 0 0 0 0 0 1 0 0 0 0),
    (0 0 0 0 0 0 0 1 0 0 0),
    (0 0 0 0 0 0 0 0 1 0 0),
    (0 0 0 0 0 0 0 0 0 1 0),
    (0 0 0 0 0 0 0 0 0 0 1)
]
> C3:= Complement(V11, G3);
> C3;
Vector space of degree 11, dimension 5 over GF(3)
Echelonized basis:
(0 0 0 0 0 0 1 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0)
(0 0 0 0 0 0 0 0 1 0 0)
(0 0 0 0 0 0 0 0 0 1 0)
(0 0 0 0 0 0 0 0 0 0 1)
```

```
> G3 + C3;
Full Vector space of degree 11 over GF(3)
> G3 meet C3;
Vector space of degree 11, dimension 0 over GF(3)
> x := Random(G3);
> x;
(1 1 2 0 0 1 1 1 1 2 0)
> c := Coordinates(G3, x);
> c;
[ 1, 1, 2, 0, 0, 1 ]
> G3 ! &+[ c[i] * G3.i : i in [1 .. Dimension(G3)]];
(1 1 2 0 0 1 1 1 1 2 0)
```

28.8 Operations with Linear Transformations

Throughout this section, V is a subspace of $K^{(m)}$, W is a subspace of $K^{(n)}$ and a is a linear transformation belonging to $\mathrm{Hom}_K(V, W)$. See also the chapter on general matrices for many other functions applicable to such matrices (e.g., `EchelonForm`).

`v * a`

`a(v)`

Given an element v belonging to the vector space V, and an element a belonging to $\mathrm{Hom}_K(V, W)$, return the image of v under the linear transformation a as an element of the vector space W.

`a * b`

Given a matrix a belonging to $K^{(m\times n)}$ and a matrix b belonging to $K^{(n\times p)}$, for some integers m, n, p, form the product of a and b as an element of $K^{(m\times p)}$.

`Domain(a)`

The domain of the linear transformation a belonging to $\mathrm{Hom}_K(V, W)$, returned as a subspace of V.

`Codomain(a)`

The codomain of the linear transformation a belonging to $\mathrm{Hom}_K(V, W)$, returned as a subspace of W.

`Image(a)`

The image of the linear transformation a belonging to $\mathrm{Hom}_K(V, W)$, returned as a subspace of W.

`Rank(a)`

The dimension of the image of the linear transformation a, i.e., the rank of the matrix a.

`Kernel(a)`

`NullSpace(a)`

The kernel of the linear transformation a belonging to $\mathrm{Hom}_K(V, W)$, returned as a subspace of V.

`Cokernel(a)`

The cokernel of the linear transformation a belonging to $\mathrm{Hom}_K(V, W)$.

Example H28E14

We illustrate the map operations for matrix spaces in the following example:

```
> Q := RationalField();
> Q2 := VectorSpace(Q, 2);
> Q3 := VectorSpace(Q, 3);
> Q4 := VectorSpace(Q, 4);
> H23 := Hom(Q2, Q3);
> H34 := Hom(Q3, Q4);
> x := Q2 ! [ -1, 2 ];
> a := H23 ! [ 1/2, 3, 0,  2/3, 4/5, -1 ];
> a;
[1/2   3   0]
[2/3 4/5  -1]
> Domain(a);
Full Vector space of degree 2 over Rational Field
> Codomain(a);
Full Vector space of degree 3 over Rational Field
> x*a;
( 5/6 -7/5   -2)
> b := H34 ! [ 2, 0, 1, -1/2,  1, 0, 3/2, 4,  4/5, 6/7, 0, -9/7];
> b;
[   2    0    1 -1/2]
[   1    0  3/2    4]
[ 4/5  6/7    0 -9/7]
> c := a*b;
> c;
[      4       0       5    47/4]
[    4/3    -6/7   28/15 436/105]
> x*c;
(     -4/3     -12/7    -19/15 -1447/420)
> Image(c);
Vector space of degree 4, dimension 2 over Rational Field
Echelonized basis:
(     1      0    5/4  47/16)
(     0      1  -7/30 -11/40)
> Kernel(c);
Vector space of degree 2, dimension 0 over Rational Field
```

```
> Rank(c);
2
> EchelonForm(c);
[     1      0    5/4  47/16]
[     0      1  -7/30 -11/40]
```

---

29 POLAR SPACES

Chapter 29
POLAR SPACES

29.1 Introduction

This chapter describes MAGMA functions for working with quadratic, bilinear and sesquilinear forms defined on vector spaces. The emphasis is on vector spaces defined over finite fields but in some instances the functions apply more widely. For quadratic forms defined on lattices see Chapter 32. For the interpretation of reflexive forms as algebras with involution see Chapter 87. General references for this material are [Bou07, Tay92].

29.2 Reflexive Forms

Let V be a vector space of dimension n over a field K. If σ is an automorphism of K, a σ-sesquilinear form on the vector space V over K is a map $\beta : V \times V \to K$ such that

$$\beta(u_1 + u_2, v) = \beta(u_1, v) + \beta(u_2, v),$$
$$\beta(u, v_1 + v_2) = \beta(u, v_1) + \beta(u, v_2)$$

and

$$\beta(au, bv) = a\sigma(b)\beta(u, v).$$

for all $u, u_1, u_2, v, v_1, v_2 \in V$ and all $a, b \in K$. If σ is the identity, the form is said to be *bilinear*.

A linear transformation g of V is an *isometry* if g preserves β; it is a *similarity* if it preserves β up to a non-zero scalar multiple.

A σ-sesquilinear form β is *reflexive* if for all $u, v \in V$, $\beta(u, v) = 0$ implies $\beta(v, u) = 0$. Any non-zero multiple of a reflexive form is again reflexive with the same group of isometries. By a theorem of Brauer [Bra36] (but sometimes referred to as the Birkhoff–von Neumann theorem), up to a non-zero scalar multiple, there are three types of *non-degenerate* reflexive forms:

Alternating. In this case σ is the identity, $\beta(u, u) = 0$ for all $u \in V$ and consequently $\beta(u, v) = -\beta(v, u)$ for all $u, v \in V$. The group of isometries is a *symplectic* group.

Symmetric. In this case σ is the identity and $\beta(u, v) = \beta(v, u)$ for all $u, v \in V$. If the characteristic of K is not two, the group of isometries is an *orthogonal* group. If the characteristic is two, the form is either alternating or pseudo-alternating (see below).

Hermitian. In this case σ is an automorphism of order two and $\beta(u, v) = \sigma\beta(v, u)$ for all $u, v \in V$. The group of isometries is a *unitary* group.

If V is a vector space V and if β is a reflexive form defined on V, the partially ordered set of totally isotropic subspaces with respect to β is often referred to as a *polar space*.

Similarly, there are polar spaces associated with quadratic forms (see Section 29.2.1). But throughout this chapter by polar space we shall simply mean a vector space furnished with either a reflexive σ-sesquilinear form or a quadratic form. See [Bue95, Chap. 2] for an account of polar spaces in a more general context.

Let K_0 be the fixed field of σ. Multiplying an alternating, symmetric or hermitian form by a non-zero element of K_0 leaves the type of the form unchanged.

However, multiplying an hermitian form by a non-zero element of K produces a sesquilinear form ξ and an element $\varepsilon \in K$ such that for all $u, v \in V$, $\xi(v, u) = \varepsilon\sigma\xi(u, v)$, where $\varepsilon\sigma(\varepsilon) = 1$. In this case ξ is said to be ε-hermitian.

Skew-hermitian. A reflexive σ-sesquilinear form is skew-hermitian if the order of σ is two and $\xi(v, u) = -\sigma\xi(v, u)$ for all $u, v \in V$. If β is hermitian and if $d \in K$ is chosen so that $d \neq \sigma(d)$, then $e = d - \sigma(d)$ satisfies $\sigma(e) = -e$. Thus $\xi(u, v) = e\beta(u, v)$ is skew-hermitian. The group of isometries of ξ coincides with the group of isometries of β and it is therefore a *unitary* group.

In the case of fields of characteristic two there is no distinction between hermitian and skew-hermitian forms and moreover, every alternating form is symmetric.

Pseudo-alternating. A symmetric form (in characteristic two) which is not alternating is said to be *pseudo-alternating*.

The three types of forms—alternating, symmetric and hermitian—correspond to the three types of classical groups of isometries: symplectic, orthogonal and unitary. But this is not quite the whole story because it does not include orthogonal groups over fields of characteristic two. In order to include these groups it is necessary to consider quadratic forms in addition to symmetric bilinear forms.

29.2.1 Quadratic Forms

If β is a bilinear form, a *quadratic form* with *polar form* β is a function $Q : V \to K$ such that

$$Q(av) = a^2 Q(v)$$

and

$$\beta(u, v) = Q(u + v) - Q(u) - Q(v)$$

for all $u, v \in V$ and all $a \in K$. We have $\beta(v, v) = 2Q(v)$ and therefore, if the characteristic of K is not two, β determines Q.

We extend the notion of polar space to include vector spaces V with an associated quadratic form Q. The pair (V, Q) is an orthogonal geometry and V is a *quadratic* space.

29.3 Inner Products

Every vector space V in MAGMA created via the `VectorSpace` intrinsic (or its synonym `KSpace`) has an associated bilinear form which is represented by a matrix and which can be accessed via `InnerProductMatrix(V)` or via the attribute `ip_form`. By default the inner product matrix is the identity. If the dimension of V is n, then any $n \times n$ matrix defined over the base field of V can serve as the inner product matrix by passing it to `VectorSpace` as an additional parameter.

If $e_1, e_2, \ldots, e_n$ is a basis for V, the matrix of the form β with respect to this basis is $J := \big(\beta(e_i, e_j)\big)$.

Example H29E1

```
> K := GF(11);
> J := Matrix(K,3,3,[1,2,3, 4,5,6, 7,8,9]);
> V := VectorSpace(K,3,J);
> InnerProductMatrix(V);
[ 1  2  3]
[ 4  5  6]
[ 7  8  9]
```

A vector space may also have an associated quadratic form. This can be assigned via the function `QuadraticSpace` described in Section 29.6.3 and, if assigned, it can be accessed as the return value of `QuadraticFormMatrix`.

In addition, in order to accommodate hermitian forms, a vector space of type `ModTupFld` has an attribute `Involution`. This attribute is intended to hold an automorphism (of order two) of the base field.

`DotProduct(u, v)`

If V is the generic space of the parent of u and v, let σ be the field automorphism `V‘Involution` if this attribute is assigned or the identity automorphism if `V‘Involution` is not assigned. If J is the inner product matrix of V, the expression `DotProduct(u,v)` evaluates to $uJ\sigma(v^{\mathrm{tr}})$. That is, it returns $\beta(u, v)$, where β is a bilinear or sesquilinear form on V.

`DotProductMatrix(W)`

The matrix of inner products of the vectors in the sequence W. The inner products are calculated using `DotProduct` and therefore take into account any field automorphism attached to the `Involution` attribute of the generic space of the universe of S.

`GramMatrix(V)`

If B is the basis matrix of V and if J is the inner product matrix, this function returns BJB^{tr}. In this case the `Involution` attribute is ignored.

`InnerProductMatrix(V)`

The inner product matrix attached to the generic space of V. This is the attribute `V‘ip_form`.

Example H29E2

This example illustrates the difference between `GramMatrix` and `InnerProductMatrix`. The function `GramMatrix` uses the echelonised basis of the subspace W. To obtain the matrix of inner products between a given list of vectors, use `DotProductMatrix`.

```
> K<a> := QuadraticField(-2);
> J := Matrix(K,3,3,[1,2,1, 2,1,0, 1,0,2]);
> V := VectorSpace(K,3,J);
> W := sub<V| [a,a,a], [1,2,3]>;
> InnerProductMatrix(W);
[1 2 1]
[2 1 0]
[1 0 2]
> GramMatrix(W);
[1 0]
[0 9]
> DotProductMatrix([W.1,W.2]);
[ -20 19*a]
[19*a   37]
```

Example H29E3

Continuing the previous example, the vector space V does not have the attribute `Involution` assigned and therefore `DotProduct` uses the *symmetric* bilinear form represented by the inner product matrix J. However, the field K has a well-defined operation of complex conjugation and so `InnerProduct` uses the *hermitian* form represented by J.

```
> u := W.1+W.2;
> DotProduct(u,u);
38*a + 17
> InnerProduct(u,u);
57
```

29.3.1 Orthogonality

If β is any bilinear or sesquilinear form, the vectors u and v are *orthogonal* if $\beta(u, v) = 0$. The *left orthogonal complement* of a subset X of V is the subspace

$$^{\perp}X := \{\, u \in V \mid \beta(u, x) = 0 \text{ for all } x \in X \,\}$$

and the *right orthogonal complement* of W is

$$X^{\perp} := \{\, u \in V \mid \beta(x, u) = 0 \text{ for all } x \in X \,\}.$$

If β is reflexive, then $^{\perp}X = X^{\perp}$.

`OrthogonalComplement(V, X : parameters)`

`Right` BoolElt *Default :* `false`

The default value is the left orthogonal complement of X in V. To obtain the right orthogonal complement set `Right` to `true`.

`Radical(V : parameters)`

`Right` BoolElt *Default :* `false`

The left radical of the inner product space V, namely $^{\perp}V$. To obtain the right radical set `Right` to `true`.

A bilinear or sesquilinear form β is *non-degenerate* if $\mathrm{rad}(V) = 0$, where V is the polar space of β.

`IsNondegenerate(V)`

Returns `true` if the determinant of the matrix of inner products of the basis vectors of V is non-zero, otherwise `false`. This function takes into account the field automorphism, if any, attached to the `Involution` attribute of the generic space of V.

If V is a quadratic space over a perfect field of characteristic 2, the restriction of the quadratic form Q to the radical is a semilinear functional (with respect to $x \mapsto x^2$) whose kernel is the *singular radical* of V. A quadratic space is *non-singular* if its singular radical is zero.

`SingularRadical(V)`

The kernel of the restriction of the quadratic form of the quadratic space V to the radical of V.

`IsNonsingular(V)`

Returns `true` if V is a non-singular quadratic space, otherwise `false`.

29.4 Isotropic and Singular Vectors and Subspaces

Let β be a reflexive bilinear or a sesquilinear form on the vector space V. A non-zero vector v is *isotropic* (with respect to β) if $\beta(v, v) = 0$. If Q is a quadratic form, a non-zero vector v is *singular* if $Q(v) = 0$.

`HasIsotropicVector(V)`

Determine whether the polar space V contains an isotropic vector; if it does, the second return value is a representative.

`HasSingularVector(V)`

Determine whether the quadratic space V contains a singular vector; if it does, the second return value is a representative.

An ordered pair of vectors (u, v) such that u and v are isotropic and $\beta(u, v) = 1$ is a *hyperbolic pair*. If V is a quadratic space, u and v are required to be singular.

`HyperbolicPair(V, u)`

Given a singular or isotropic vector u which is not in the radical, return a vector v such that (u, v) is a hyperbolic pair.

If V is the direct sum of subspaces U and W and if $\beta(u, w) = 0$ for all $u \in U$ and all $w \in W$, we write $V = U \perp W$.

A vector space V furnished with a reflexive form β has a direct sum decomposition $V = U \perp \mathrm{rad}(V)$, where U is any complement to $\mathrm{rad}(V)$ in V.

If V is a polar space, it has a *hyperbolic splitting*; namely, it is a direct sum

$$V = L_1 \perp L_2 \perp \cdots \perp L_m \perp W$$

where the L_i are 2-dimensional subspaces spanned by hyperbolic pairs and m is maximal. If the form defining the polar space is non-degenerate and not pseudo-alternating, then every isotropic (resp. singular) vector belongs to a hyperbolic pair and consequently W does not contain any isotropic (resp. singular) vectors. In this case the integer m is the *Witt index* of the form and W is called the *anisotropic component* of the splitting. A non-degenerate form on V is said to have *maximal Witt index* if $\dim V$ is $2m$ or $2m + 1$.

Example H29E4

The vector space of dimension 2 over $\mathbf{F}_2$ is pseudo-symplectic (the form is the identity matrix). It has three non-zero elements only one of which is isotropic. This confirms that not every isotropic vector in a non-degenerate pseudo-symplectic space belongs to a hyperbolic pair.

```
> V := VectorSpace(GF(2),2);
> IsPseudoSymplecticSpace(V);
true
> IsNondegenerate(V);
true
> { v : v in V | v ne V!0 and DotProduct(v,v) eq 0};
```

```
{
    (1 1)
}
```

`HyperbolicSplitting(V)`

A maximal list of pairwise orthogonal hyperbolic pairs together with a basis for the orthogonal complement of the subspace they span. This function requires the form to be non-degenerate and, except for symplectic spaces, the base ring of V must be a finite field.

Example H29E5

Find the hyperbolic splitting of a polar space defined by a symmetric bilinear form. In this example W is a non-degenerate subspace of the polar space V.

```
> K<a> := GF(7,2);
> J := Matrix(K,3,3,[1,2,1, 2,1,0, 1,0,2]);
> V := VectorSpace(K,3,J);
> W := sub<V| [a,a,a], [1,2,3]>;
> IsNondegenerate(W);
true
> HyperbolicSplitting(W);
<[
    [
        (a^20    1 a^39),
        (a^12    2    a)
    ]
], []>
```

Example H29E6

The polar space V of the previous example is degenerate and so `HyperbolicSplitting` cannot be applied directly. Instead, we first split off the radical.

```
> IsNondegenerate(V);
false
> R := Radical(V);
> H := (Dimension(R) eq 0) select V else
>   sub<V|[e : e in ExtendBasis(B,V) | e notin B] where B is Basis(R)>;
> HyperbolicSplitting(H);
<[
    [
        (   0 a^20    1),
        (   0 a^12    2)
    ]
], []>
```

A subspace W of a polar space V is *totally isotropic* if every non-zero vector of W is isotropic. If V is a quadratic space, W is *totally singular* if every non-zero vector of W is singular.

`IsTotallyIsotropic(V)`

Returns `true` if the polar space V is totally isotropic, otherwise `false`.

`IsTotallySingular(V)`

Returns `true` if the quadratic space V is totally singular, otherwise `false`.

Suppose that $V = L_1 \perp \cdots \perp L_m \perp W \perp \mathrm{rad}(V)$ where the L_i are 2-dimensional subspaces spanned by hyperbolic pairs (e_i, f_i) for $1 \leq i \leq m$. The subspaces $P = \langle e_1, \ldots, e_m \rangle$ and $N = \langle f_1, \ldots, f_m \rangle$ are totally isotropic and we call the 4-tuple $(\mathrm{rad}(V), P, N, W)$ a *Witt decomposition* of V. A polar space is *hyperbolic* if it is the direct sum of two totally isotropic (resp. totally singular) subspaces; in Bourbaki [Bou07, p. 66] the corresponding form is said to be *neutral*.

`WittDecomposition(V)`

The Witt decomposition of the space V.

`WittIndex(V)`

The Witt index of the polar space V; namely half the dimension of a maximal hyperbolic subspace.

`MaximalTotallyIsotropicSubspace(V)`

A representative maximal totally isotropic subspace of the polar space V.

`MaximalTotallySingularSubspace(V)`

A representative maximal totally singular subspace of the quadratic space V.

29.5 The Standard Forms

This section describes the "standard" alternating, hermitian, quadratic and symmetric forms defined on a finite dimensional vector space over a field. These are forms of maximal Witt index together with the quadratic forms of non-maximal Witt index over finite fields (see Section 29.4). Except for the orthogonal groups the standard forms are preserved by the MAGMA implementation of the classical groups over finite fields.

If J is the matrix of the form, then X preserves the form if $XJX^{\mathrm{tr}} = J$.

If β is a non-degenerate alternating form, then $\mathrm{rad}(V)$ and the anisotropic component of a hyperbolic splitting are zero. Thus the dimension of V must be even and V has a basis of mutually orthogonal hyperbolic pairs. In particular, up to equivalence, there is only one non-degenerate alternating form on V.

`StandardAlternatingForm(n,R)`

`StandardAlternatingForm(n,q)`

If $n = 2m$, this function returns the $n \times n$ matrix of a non-degenerate alternating form over the ring R (or the field of q elements) such that if e_1, e_2, ..., e_{2m} is the standard basis, then (e_1, e_{2m}), (e_2, e_{2m-1}), ..., (e_m, e_{m+1}) are mutually orthogonal hyperbolic pairs.

The group of isometries of this form is the symplectic group $\mathrm{Sp}(2m, R)$.

Example H29E7

Create a symplectic geometry with the standard alternating form and then check that every non-zero vector is isotropic.

```
> K := GF(5);
> J := StandardAlternatingForm(4,K);
> J;
[0 0 0 1]
[0 0 1 0]
[0 4 0 0]
[4 0 0 0]
> V := VectorSpace(K,4,J);
> forall{ v : v in V | DotProduct(v,v) eq 0 };
true
```

`StandardPseudoAlternatingForm(n,K)`

`StandardPseudoAlternatingForm(n,q)`

The matrix of the standard pseudo-alternating form of degree n over the field K (or the finite field of order q), which must have characteristic 2; that is, a symmetric form which is not alternating.

`StandardHermitianForm(n,K)`

`StandardHermitianForm(n,q)`

The first return value of this function is the $n \times n$ anti-diagonal matrix $(\delta_{i,n-i+1})$ over the field K (or the field of q^2 elements). If K is the finite field of q^2 elements, the second return value is the field involution $K \to K : x \mapsto x^q$. If K is a field which admits the operation of complex conjugation, the second return value is the field automorphism which sends each element to its complex conjugate.

If β is a non-degenerate hermitian form over a finite field, then $\mathrm{rad}(V)$ is zero and the dimension of the anisotropic component of a hyperbolic splitting is either 1 or 0.

In the finite field case, the group of isometries of this form is $\mathrm{GU}(n, q)$.

Suppose that β is the polar form of a quadratic form Q defined on the vector space V. A vector $v \in V$ is said to be *singular* if $Q(v) = 0$. A subspace W is *totally singular* is $Q(w) = 0$ for all $w \in W$. The *Witt index* of Q is the dimension of a maximal totally singular subspace. If the characteristic of the field is not 2 a subspace is totally singular if and only if it is totally isotropic with respect to β and hence in this case the Witt index of Q coincides with the Witt index of β.

`StandardQuadraticForm(n, K : parameters)`

`StandardQuadraticForm(n, q : parameters)`

`Minus`	BoolElt	*Default :* `false`
`Variant`	MonStgElt	*Default :* "Default"

An $n \times n$ upper triangular matrix representing a quadratic form over the field K (or the field of order q). The default option is to return a form of maximal Witt index, namely the upper triangular matrix whose non-zero entries are $\delta_{i,n-i+1}$, where $1 \leq i \leq (n+1)/2$.

If `Minus` is `true` and $n = 2m$, this function returns a form whose Witt index is $m - 1$. This option is available only for finite fields.

For historical reasons, over finite fields of odd characteristic, the (m, m) element in the quadratic form used by the orthogonal groups of odd degree is $1/4$.

If the K is a finite field and W is the anisotropic component of a hyperbolic splitting of a form of `Minus` type, then W has basis vectors e and f such that $Q(e) = \beta(e, f) = 1$ and $Q(f) = a$, where $x^2 + x + a$ is an irreducible polynomial in $F[x]$. This is the form returned by the `Default` option. If the characteristic of K is two, this is also returned by the `Revised` option. However, if the characteristic of K is odd, the `Revised` option returns a form corresponding to an orthonormal basis for W. The `Original` option returns the form preserved by `OldGOMinus(2*m,q)`.

Example H29E8

Construct a standard quadratic form of minus type.

```
> K<z> := GF(7,2);
```

```
> Q := StandardQuadraticForm(4,49 : Minus);
> Q;
[   0    0     0    1]
[   0    1     1    0]
[   0    0  z^23    0]
[   0    0     0    0]
> _<x> := PolynomialRing(K);
> a := Q[3,3];
> IsIrreducible(x^2+x+a);
true
```

Example H29E9

Compare the revised form with the standard form: the forms `Q` above and `QR` below have different entries in the central 2×2 block.

```
> QR := StandardQuadraticForm(4,49 : Minus, Variant := "Revised");
> QR;
[   0    0     0    1]
[   0    4     0    0]
[   0    0  z^11    0]
[   0    0     0    0]
```

`StandardSymmetricForm(n, K)`

`StandardSymmetricForm(n, q : parameters)`

`Minus`	BoolElt	*Default :* `false`
`Variant`	MonStgElt	*Default :* "Default"

In all casses this is $Q + Q^{\mathrm{tr}}$, where Q is the corresponding standard quadratic form, as defined above.

29.6 Constructing Polar Spaces

If J is an $n \times n$ matrix, the command `VectorSpace(K,n,J)` creates a vector space of dimension n over K with a bilinear form whose matrix is J.

The default form attached to every vector space in MAGMA is the symmetric form whose matrix is the identity matrix.

A vector space V is recognised as a polar space if any of the following conditions apply. (There is no check to ensure that the inner product matrix is is non-degenerate.)

1. There is a quadratic form attached to V.
2. There is a field involution attached to V and the inner product matrix of V is hermitian or skew-hermitian with respect to this involution.
3. The inner product matrix of V is symmetric or alternating.

Thus a vector space with a symmetric inner product matrix but no quadratic form attached is a polar space. If the characteristic of the field is 2 and the form is not alternating it is a pseudo-symplectic space, otherwise we shall call it an orthogonal space to distinguish it from quadratic spaces.

`IsPolarSpace(V)`

Check if the vector space V is a quadratic space or if the Gram matrix of V is a reflexive form.

`PolarSpaceType(V)`

The type of the polar space V, returned as a string.

Example H29E10

Create the standard vector space of dimension 4 over the rational field and check if it is a polar space.

```
> V := VectorSpace(Rationals(),4);
> IsPolarSpace(V);
true
> PolarSpaceType(V);
orthogonal space
```

29.6.1 Symplectic Spaces

`SymplecticSpace(J)`

The symplectic space of dimension n defined by the $n \times n$ matrix J. This function checks to ensure that J is alternating.

`IsSymplecticSpace(W)`

Returns `true` if the `Involution` attribute of the generic vector space W is not assigned and the space carries an alternating form, otherwise `false`.

Note that a quadratic space over a field of characteristic 2 satisfies these conditions and consequently this function will return `true` for these spaces.

`IsPseudoSymplecticSpace(W)`

Given a vector space W over a finite field, this intrinsic returns `true` if the base field has characteristic 2, the `Involution` attribute is not assigned to the generic space and the form is symmetric but not alternating, otherwise `false`.

29.6.2 Unitary Spaces

In order to accommodate hermitian forms it is necessary to assign a field automorphism of order two to the `Involution` attribute of the vector space.

Thus a unitary space is characterised as a vector space V whose ambient space, `Generic(V)`, has the attribute `Involution` and whose inner product matrix is either hermitian or skew hermitian.

`UnitarySpace(J, sigma)`

The n-dimensional unitary space over the base field K of J, where σ is an automorphism of K of order 2 and where J is an $n \times n$ matrix which is hermitian or skew-hermitian with respect to σ.

`IsUnitarySpace(W)`

Return `true` if the `Involution` attribute of the generic space of W is assigned and the form is either hermitian or skew-hermitian when restricted to W.

Example H29E11

Create a unitary geometry with the standard hermitian form and check that the given vector is isotropic. Note that the function `DotProduct` takes both the form and the field involution into account when calculating its values. For finite fields, the function `InnerProduct` *ignores* the field involution.

```
> K<z> := GF(25);
> J, sigma := StandardHermitianForm(5,K);
> J;
[   0    0    0    0    1]
[   0    0    0    1    0]
[   0    0    1    0    0]
```

```
[   0    1    0    0    0]
[   1    0    0    0    0]
> sigma(z);
z^5
> V := UnitarySpace(J,sigma);
> u := V![1,z,0,z^2,-1];
> DotProduct(u,u);
0
> InnerProduct(u,u);
z^20
```

`ConjugateTranspose(M, sigma)`

The transpose of the matrix $\sigma(M)$, where σ is an automorphism of the base field ofthe matrix M.

29.6.3 Quadratic Spaces

A vector space V with an attached quadratic form is called a *quadratic space*. The polar form of a quadratic space is the inner product matrix J of the space. If the characteristic of the field is not 2, the value of the quadratic form on a row vector v is $\frac{1}{2}v * J * v^{\mathrm{tr}}$.

`QuadraticSpace(Q)`

The quadratic space of dimension n defined by the quadratic form represented by an upper triangular $n \times n$ matrix Q. The inner product matrix of the space is $Q + Q^{\mathrm{tr}}$. If Q is not upper triangular, the space will be constructed but there is no guarantee that other functions in this package will return correct results.

`QuadraticSpace(f)`

The quadratic space of dimension n whose quadratic form is given by the quadratic polynomial f in n variables. If the variables are $x_1, \ldots, x_n$, then for $i \leq j$, the (i, j)-th entry of the matrix of the form is the coefficient of $x_i x_j$ in f.

`SymmetricToQuadraticForm(J)`

Provided the characteristic of the field is not two, this is the upper triangular matrix which represents the same quadratic form as the symmetric matrix J.

`QuadraticFormMatrix(V)`

The (upper triangular) matrix which represents the quadratic form of the quadratic space V.

`QuadraticNorm(v)`

The value $Q(v)$, where Q is the quadratic form attached to the generic space of the parent of the vector v.

`QuadraticFormPolynomial(V)`

The polynomial $\sum_{i \leq j} q_{ij} x_i x_j$, where $Q = (q_{ij})$ is the matrix of the quadratic form of the quadratic space V.

Example H29E12

The quadratic space defined by a polynomial.

```
> _<x,y,z> := PolynomialRing(Rationals(),3);
> f := x^2 + x*y +3*x*z - 2*y*z + y^2 +z^2;
> V := QuadraticSpace(f);
> PolarSpaceType(V);
quadratic space
> IsNonsingular(V);
true
> QuadraticFormMatrix(V);
[ 1  1  3]
[ 0  1 -2]
[ 0  0  1]
```

`OrthogonalSum(V, W)`

The orthogonal direct sum of the quadratic spaces V and W.

`TotallySingularComplement(V, U, W)`

Given totally singular subspaces U and W of the quadratic space V such that $U^\perp \cap W = 0$ this function returns a totally singular subspace X such that $V = X \oplus U^\perp$ and $W \subseteq X$.

`Discriminant(V)`

If V is a vector space over the finite field K and if J is the Gram matrix of V, the *discriminant* of V is the determinant Δ of J modulo squares. That is, the discriminant is 0 if Δ is a square in K, 1 if it is a non-square. The form J is required to be non-degenerate.

`ArfInvariant(V)`

The Arf invariant of the quadratic space V.

Currently this is available only for quadratic spaces of even dimension $2m$ over a finite field F of characteristic 2. In this case there are two possibilities: either the Witt index of the form is m and the Arf invariant is 0, or the Witt index is $m-1$ and the Arf invariant is 1.

`DicksonInvariant(V, f)`

The Dickson invariant of the isometry f of the quadratic space V is the rank (mod 2) of $1 - f$. If the polar form of Q is non-degenerate, the Dickson invariant defines a homomorphism from the orthogonal group $O(V)$ onto the additive group of order 2.

`SpinorNorm(V, f)`

The spinor norm of the isometry f of the quadratic space V. This is the discriminant of the Wall form (Section 29.8) of f.

`HyperbolicBasis(U, B, W)`

Given complementary totally singular subspaces U and W of a quadratic space and a basis B for U, return a sequence of pairwise orthogonal hyperbolic pairs whose second components form a basis for W.

`OrthogonalReflection(a)`

The reflection determined by a non-singular vector of a quadratic space.

`RootSequence(V, f)`

Given a matrix f representing an isometry of the quadratic space V, return a sequence of vectors such that the product of the corresponding orthogonal reflections is f. The empty sequence is returned if f is the identity matrix.

`ReflectionFactors(V, f)`

Given a matrix f representing an isometry of the quadratic space V, return a sequence of reflections whose product is f. The empty sequence corresponds to the identity matrix.

Given a quadratic space V defined by a quadratic form Q with polar form β and non-zero vectors $u, v \in V$ such that u is singular and $\beta(u, v) = 0$, the *Siegel transformation* (also called an Eichler transformation) is the isometry $\rho_{u,v}$ defined by

$$x\rho_{u,v} = x + \beta(x, v)u - \beta(x, u)v - Q(v)\beta(x, u)u.$$

`SiegelTransformation(u, v)`

The Siegel transformation defined by a singular vector u and a vector v orthogonal to u. The common parent of u and v must be a quadratic space.

Example H29E13

A group of isometries generated by Siegel transformations.

```
> Q := StandardQuadraticForm(4,3);
> V := QuadraticSpace(Q);
> u := V.1;
> QuadraticNorm(u);
0
> X := { v : v in V | DotProduct(u,v) eq 0 and QuadraticNorm(v) ne 0 };
> #X;
12
> H := sub< GL(V) | [SiegelTransformation(u,v) : v in X]>;
> #H;
9
```

29.7 Isometries and Similarities

If β is a bilinear or sesquilinear form on the vector space V, a linear transformation g of V is an *isometry* if g preserves β; it is a *similarity* if it preserves β up to a non-zero scalar multiple.

29.7.1 Isometries

`IsIsometry(U, V, f)`

Returns `true` if the map f is an isometry from U to V with respect to the attached forms.

`IsIsometry(f)`

Returns `true` if the map f is an isometry from its domain to its codomain.

`IsIsometry(V, g)`

Returns `true` if the matrix g is an isometry of V with respect to the attached form.

`IsIsometric(V, W)`

Determines whether the polar spaces V and W are isometric; if they are, an isometry is returned (as a map).

Example H29E14

A vector space is always equipped with a bilinear form and the default value is the identity matrix. However the "standard" symmetric form is $Q + Q^{\mathrm{tr}}$, where Q is the standard quadratic form. In the following example the polar spaces defined by these forms are similar but not isometric.

```
> F := GF(5);
> V1 := VectorSpace(F,5);
> PolarSpaceType(V1);
orthogonal space
> WittIndex(V1);
2
> J2 := StandardSymmetricForm(5,F);
> J2;
[0 0 0 0 1]
[0 0 0 1 0]
[0 0 2 0 0]
[0 1 0 0 0]
[1 0 0 0 0]
> V2 := VectorSpace(F,5,J2);
> IsIsometric(V1,V2);
false
> V3 := VectorSpace(F,5,2*J2);
> flag, f := IsIsometric(V1,V3); flag;
true
> IsIsometry(f);
true
```

If J is an $n \times n$ matrix which represents a bilinear form and if M is a non-singular $n \times n$ matrix, then J and MJM^{tr} are said to be *congruent* and they define isometric polar spaces.

Conversely, given bilinear forms J_1 and J_2 the following example shows how to use the `IsIsometric` function to determine whether J_1 and J_2 are congruent and if so how to find a matrix M such that $J_1 = MJ_2M^{\mathrm{tr}}$.

Example H29E15

Begin with alternating forms J_1 and J_2 over $\mathbf{F}_{25}$, construct the corresponding symplectic spaces and then use the isometry to define the matrix M.

```
> F<x> := GF(25);
> J1 := Matrix(F,4,4,[ 0, x^7, x^14, x^13, x^19, 0, x^8, x^5,
>   x^2, x^20, 0, x^17, x, x^17, x^5, 0 ]);
> J2 := Matrix(F,4,4,[ 0, x^17, 2, x^23, x^5, 0, x^15, x^5,
>   3, x^3, 0, 4, x^11, x^17, 1, 0 ]);
> V1 := SymplecticSpace(J1);
> V2 := SymplecticSpace(J2);
> flag, f := IsIsometric(V1,V2); assert flag;
```

```
> f;
Mapping from: ModTupFld: V1 to ModTupFld: V2 given by a rule
> M := Matrix(F,4,4,[f(V1.i) : i in [1..4]]);
> J1 eq M*J2*Transpose(M);
true
```

Example H29E16

Another way to obtain a matrix with the same effect as M is to use the function `TransformForm`.

```
> M1 := TransformForm(J1,"symplectic");
> M2 := TransformForm(J2,"symplectic");
> M_alt := M1*M2^-1;
> J1 eq M_alt*J2*Transpose(M_alt);
true
```

`CommonComplement(V, U, W)`

A common complement to the subspaces U and W in the vector space V. (The subspaces must have the same dimension.) This is used by the following function, which implements Witt's theorem.

`ExtendIsometry(V, U, f)`

An extension of the isometry $f : U \to V$ to an isometry $V \to V$, where U is a subspace of the polar space V.

This is an implementation of Witt's theorem on the extension of an isometry defined on a subspace of a symplectic, unitary or quadratic space. The isometry f must satisfy $f(U \cap \mathrm{rad}(V)) = f(U) \cap \mathrm{rad}(V)$.

If the characteristic is two and the form J of V is symmetric, then J must be alternating.

`IsometryGroup(V)`

The group of isometries of the polar space V. This includes degenerate polar spaces as well as polar spaces defined by a quadratic form over a field of characteristic two.

Given a reflexive form J, the function `IsometryGroup(J)` defined in Chapter 87 returns the isometry group of J. More generally, if S is a sequence of reflexive forms, the function `IsometryGroup(S)` returns the group of isometries of the system.

Example H29E17

We give an example of an isometry group of a degenerate quadratic space over a field of characteristic 2.

```
> F := GF(4);
> Q1 := StandardQuadraticForm(4,F : Minus);
> Q := DiagonalJoin(Q1,ZeroMatrix(F,2,2));
> V := QuadraticSpace(Q);
> G := IsometryGroup(V);
> [ IsIsometry(V,g) : g in Generators(G) ];
[ true, true, true, true, true, true, true ]
> #G;
96259276800
```

Example H29E18

The matrix M constructed in Example H29E15 can be used to conjugate the isometry group of J_1 to the isometry group of J_2.

```
> F<x> := GF(25);
> J1 := Matrix(F,4,4,[ 0, x^7, x^14, x^13, x^19, 0, x^8, x^5,
>   x^2, x^20, 0, x^17, x, x^17, x^5, 0 ]);
> J2 := Matrix(F,4,4,[ 0, x^17, 2, x^23, x^5, 0, x^15, x^5,
>   3, x^3, 0, 4, x^11, x^17, 1, 0 ]);
> V1 := SymplecticSpace(J1);
> V2 := SymplecticSpace(J2);
> flag, f := IsIsometric(V1,V2); assert flag;
> M := Matrix(F,4,4,[f(V1.i) : i in [1..4]]);
> G1 := IsometryGroup(V1);
> G2 := IsometryGroup(V2);
> M^-1*G1.1*M in G2;
true
> M^-1*G1.2*M in G2;
true
```

29.7.2 Similarities

`IsSimilarity(U, V, f)`

Returns `true` if the map f is a similarity from U to V with respect to the attached forms.

`IsSimilarity(f)`

Returns `true` if the map f is a similarity from its domain to its codomain.

`IsSimilarity(V, g)`

Returns `true` if the matrix g is a similarity of V with respect to the attached form.

`SimilarityGroup(V)`

The group of similarities of the polar space V. This includes degenerate polar spaces as well as polar spaces defined by a quadratic form over a field of characteristic two.

29.8 Wall Forms

Given an isometry f of a quadratic or symplectic space V with bilinear form β, the Wall form of f is the form θ defined on the image I of $1 - f$ by $\theta(u, v) = \beta(w, v)$, where $u = w(1 - f)$. In general, the Wall form is not reflexive.

`WallForm(V, f)`

The space of the Wall form of f and its embedding in V.

`WallIsometry(V, I, mu)`

The inverse of `WallForm`. This is an isometry corresponding to the embedding $\mu : I \to V$, where V is a quadratic or symplectic space.

`WallDecomposition(V, f)`

An isometry f of a quadratic or symplectic space V is *Wall-regular* if the restriction of $1 - f$ to the image of $1 - f$ is invertible. If f is any isometry of V this function returns a Wall-regular element f_r and a unipotent element f_u such that $f = f_r f_u = f_u f_r$.

`SemiOrthogonalBasis(V)`

If V is a vector space with a bilinear form β, a basis e_1, e_2, ..., e_n for V is semi-orthogonal if $\beta(e_i, e_j) = 0$ for $i < j$. This function returns a semi-orthogonal basis with respect to the non-degenerate, non-alternating form attached to V. If the base field is $\mathbf{F}_2$, the form should be symmetric.

29.9 Invariant Forms

Given a group G which acts on a vector space V over a finite field F, the space of all G-invariant bilinear forms is isomorphic to $\mathrm{Hom}_G(V, V^*)$, where V^* is the dual space of V. The isomorphism associates the form β to $\theta \in \mathrm{Hom}_G(V, V^*)$, where $\beta(u, v) = \langle v, u\theta \rangle$ and where $\langle v, \varphi \rangle$ denotes the action of φ on v. If $v_1, v_2, \ldots, v_n$ is a basis for V with dual basis $\omega_1, \omega_2, \ldots, \omega_n$, the matrix of θ with respect to these bases is $J = (\beta(e_i, e_j))$.

A linear transformation with matrix A preserves the form if and only if $AJA^{\mathrm{tr}} = J$.

If the characteristic of the field is not 2, then $J = \frac{1}{2}(J + J^{\mathrm{tr}}) + \frac{1}{2}(J - J^{\mathrm{tr}})$. Therefore, in this case, every G-invariant form is the sum of a G-invariant symmetric form and a G-invariant alternating form. If the characteristic of the field is 2, every alternating form is symmetric. Thus in this case the space of G-invariant alternating forms is a subspace of the space of G-invariant symmetric forms. If the G-module is irreducible these two spaces coincide.

`InvariantBilinearForms(G)`

Given a matrix group G this function returns two sequences: a basis for the space of G-invariant symmetric forms and a basis for the space of G-invariant alternating forms.

Example H29E19

In this example the group G is reducible but (up to a scalar multiple) there is a unique G-invariant bilinear form.

```
> F<x> := GF(25);
> G := MatrixGroup< 4, F |
>    [ 1, 0, 0, 0,  0, 1, 0, 0,  0, x^14, 1, 0,  0, 0, 0, 1 ],
>    [ 3, x^23, x^20, x^10,  2, 3, 0, x^13,  4, x^10, x^13, x^23,
>      x^5, x^11, x, x^17 ] >;
> IsIrreducible(G);
false
> InvariantBilinearForms(G);
[]
[
    [   0    0    0    1]
    [   0    0    1    0]
    [   0    4    0    0]
    [   4    0    0    0]
]
```

If G acts irreducibly on a vector space V of dimension n over a (finite) field F and if $\theta_0 : V \to V^*$ is a G-invariant isomorphism, then $D \to \mathrm{Hom}_G(V, V^*) : \theta \mapsto \theta\theta_0$ is an isomorphism of vector spaces, where $D = \mathrm{End}_G(V)$. The algebra D is a division ring and hence a field (since F is finite). Thus V becomes a vector space of dimension m over D, where $n = m|D : F|$ and G is isomorphic to a subgroup of $\mathrm{GL}(m, D)$.

Example H29E20

If G acts irreducibly on V, the spaces of symmetric and alternating G-invariant forms are isomorphic (as vector spaces) to subfields of $\mathrm{End}_G(V)$ and therefore their dimensions are either 0 or divide $\dim_F(V)$.

```
> F<a> := GF(25);
> G := MatrixGroup< 4, F |
>   [ a^10, a^21, a^4, 4,
>     a^16, 4, a^9, a^8,
>     a^20, 4, 4, a^13,
>     0, a^2, a^11, a ] >;
> IsIrreducible(G), #G;
true 626
> sym, alt := InvariantBilinearForms(G);
> #sym,#alt;
2 2
```

If the characteristic of the field is not two and if J is a symmetric bilinear form there is a unique upper triangular matrix Q such that $J = Q + Q^{\mathrm{tr}}$.

On the other hand, if the characteristic is two and J is alternating, the upper triangular matrices Q such that $J = Q + Q^{\mathrm{tr}}$ form an affine space of dimension $\dim V$.

Suppose that the characteristic is two. If G preserves a symmetric bilinear form which is not alternating, then G is reducible. Conversely, if G is irreducible and if J is the matrix of a symmetric form preserved by G, then the form must be alternating and there is a unique G-invariant quadratic form Q such that $J = Q + Q^{\mathrm{tr}}$.

`InvariantQuadraticForms(G)`

A basis for the space of quadratic forms preserved by the irreducible matrix group G.

Example H29E21

In the following example the quadratic forms which are invariant under the action of a cyclic group H of order 13 form a vector space of dimension 3 over $\mathbf{F}_4$.

```
> F<z> := GF(4);
> H := MatrixGroup<6,F |
>     [ z, 0, z^2, z, z, 1,
>       1, z, 0, z, z, z,
>       0, z^2, z, 1, z^2, z^2,
>       z, 1, z, 1, 1, 0,
>       1, z^2, z, z, 0, 1,
>       1, 0, 1, 0, z^2, 1 ] >;
>
> InvariantQuadraticForms(H);
[
    [  1   1   0   1   0   0]
    [  0   0   1 z^2 z^2   z]
```

```
    [  0   0   1   0 z^2   z]
    [  0   0   0   0   0   z]
    [  0   0   0   0 z^2   z]
    [  0   0   0   0   0 z^2],

    [  1   0   1 z^2   1   0]
    [  0   z   1   1   1   z]
    [  0   0   z   0   1   0]
    [  0   0   0 z^2 z^2 z^2]
    [  0   0   0   0 z^2   1]
    [  0   0   0   0   0   1],

    [  0   0   0   0   0   1]
    [  0   0   0   0   1   0]
    [  0   0   1   1   0   0]
    [  0   0   0   z   0   0]
    [  0   0   0   0   0   0]
    [  0   0   0   0   0   0]
]
```

Given a group G which acts on a vector space V over a finite field F with an automorphism $F \to F : a \mapsto \overline{a}$ of order 2, the space of G-invariant sesquilinear forms is isomorphic to the space of G-invariant semilinear maps from V to V^*; equivalently it is isomorphic to $\mathrm{Hom}_G(V, \overline{V}^*)$, where $\overline{V}^*$ is the semilinear dual of V, namely the space of all semilinear maps from V to F.

If $\theta \in \mathrm{Hom}_G(V, \overline{V}^*)$, the corresponding sesquilinear form β is defined by $\beta(u, v) = \langle v, u\theta \rangle$ where, as before, $\langle v, \varphi \rangle$ denotes the action of φ on v.

`SemilinearDual(M, mu)`

The semilinear dual of the G-module M with respect to the field automorphism `mu`.

`InvariantSesquilinearForms(G)`

A basis for the space of *hermitian* forms preserved by the matrix group G.

Example H29E22

Let F_0 be the fixed field of the involution. The set $\mathcal{H}$ of G-invariant hermitian forms is a vector space over F_0 and if the characteristic of F is not 2, then $\mathrm{Hom}_G(V, \overline{V}^*) \simeq \mathcal{H} \otimes_{F_0} F$.

```
> F<x> := GF(5,2);
> mu := hom< F->F | x :-> x^5 >;
> H := MatrixGroup< 5, F |
>    [ 0, x^3, 0, 1, x^9, x^8, 1, 0, x^11, x^7, x^20, x^16, 1,
>     x^11, x^3, x^21, 4, 1, x^3, x^23, x^4, x^3, x, x^3, 2 ] >;
> M := GModule(H);
> D := SemilinearDual(M,mu);
> E := AHom(M,D);
> Dimension(E);
```

```
5
> herm := InvariantSesquilinearForms(H);
> #herm;
5
```

Example H29E23

If an irreducible group preserves both a bilinear and a sesquilinear form then it is realisable over a subfield of its base field. Conversely, this observation can be used to construct an example:

```
> F<x> := GF(81);
> H := MatrixGroup< 4, F | [ChangeRing(g,F) : g in Generators(Sp(4,9))]>;
> InvariantBilinearForms(H);
[]
[
    [   0    0    0    1]
    [   0    0    1    0]
    [   0    2    0    0]
    [   2    0    0    0]
]
> InvariantSesquilinearForms(H);
[
    [   0    0    0 x^45]
    [   0    0 x^45    0]
    [   0  x^5    0    0]
    [ x^5    0    0    0]
]
```

`InvariantFormBases(G)`

This function returns four sequences: bases for the spaces of symmetric, alternating, hermitian and quadratic forms preserved by the matrix group G.

29.9.1 Semi-invariant Forms

Given a vector space V over a finite field F and a group G which acts on V, a bilinear form $\beta : V \times V \to F$ is *semi-invariant* if for all $g \in G$ there is a scalar $\lambda(g)$ such that $\beta(ug, vg) = \lambda(g)\beta(u, v)$ for all $u, v \in V$. The function $\lambda : G \to F^\times$ is a homomorphism and its kernel contains the derived group of G. The *twisted dual* V^*_λ of the G-module V is the dual space of V with G-action given by $\langle v, \varphi g\rangle = \lambda(g)\langle vg^{-1}, \varphi\rangle$; thus if A is the matrix of g acting on V the matrix of the action on V^*_λ with respect to the dual basis is $\lambda(g)A^{-\mathrm{tr}}$.

The space of all semi-invariant bilinear forms is isomorphic to $\mathrm{Hom}_G(V, V^*_\lambda)$. The isomorphism associates the form β to $\theta \in \mathrm{Hom}_G(V, V^*_\lambda)$, where $\beta(u, v) = \langle v, u\theta\rangle$.

If β is a bilinear form with matrix J, then the linear transformation g with matrix A preserves the form up to multiplication by $\lambda(g)$ if and only if $AJA^{\mathrm{tr}} = \lambda(g)J$

`TwistedDual(M, lambda)`

The twisted dual of the G-module M with respect to the linear character `lambda`.

`SemiInvariantBilinearForms(G)`

A sequence of triples $\langle L, S, A\rangle$ where L is a sequence of field elements (one for each generator) which define a homomorphism from the matrix group G to its base field and S and A are bases for the spaces of symmetric and alternating forms preserved by G (up to multiplication by scalars).

`SemiInvariantQuadraticForms(G)`

A sequence of pairs $\langle L, Q\rangle$ where L is a sequence of field elements (one for each generator) which define a homomorphism from the matrix group G to its base field and Q is a basis for the space of quadratic forms preserved by G (up to multiplication by scalars).

`TwistedSemilinearDual(M, lambda, mu)`

The twisted semilinear dual of the G-module M with respect to the linear character `lambda` and the field automorphism `mu`.

`SemiInvariantSesquilinearForms(G)`

A sequence of pairs $\langle L, H\rangle$ where L is a sequence of field elements (one for each generator) which define a homomorphism from the matrix group G to the field F_0, where the base field of G is a quadratic extension of F_0, and H is a basis for the space of hermitian forms preserved by G (up to multiplication by scalars).

Example H29E24

In this example H is a normal subgroup of the absolutely irreducible group N an H is irreducible but not absolutely irreducible.

```
> F<x> := GF(3,2);
> H := MatrixGroup<3,F|
>   [x^2,x^7,x^3, x,0,1, x^3,x^6,2],
>   [x^3, 0, 0, 0, x^3, 0, 0, 0, x^3 ] >;
> N := MatrixGroup<3,F|H.1,H.2,[x^5,x^5,2, 0,x^2,x^6, x^7,x^7,2]>;
> IsNormal(N,H);
true
> IsIrreducible(H), IsAbsolutelyIrreducible(H);
true false
> IsIrreducible(N), IsAbsolutelyIrreducible(N);
true true
> SemiInvariantSesquilinearForms(H);
[
    <[ 1, 2 ],
    [
        [  1   x   x]
        [x^3   0 x^3]
        [x^3   x   1],
```

```
        [  0 x^3   x]
        [  x   0 x^5]
        [x^3 x^7   0],

        [  0   0   1]
        [  0   1   0]
        [  1   0   0]
    ]>
]
> SemiInvariantSesquilinearForms(N);
[
    <[ 1, 2, 1 ],
    [
        [  0   0   1]
        [  0   1   0]
        [  1   0   0]
    ]>
]
```

29.10 Bibliography

[**Bou07**] N. Bourbaki. *Éléments de mathématique. Algèbre. Chapitre 9.* Springer-Verlag, Berlin, 2007. Reprint of the 1959 original.

[**Bra36**] Richard Brauer. A characterization of null systems in projective space. *Bull. Amer. Math. Soc.*, 42(4):247–254, 1936.

[**Bue95**] Francis Buekenhout. *Handbook of incidence geometry.* North-Holland, Amsterdam, 1995.

[**Tay92**] Donald E. Taylor. *The geometry of the classical groups*, volume 9 of *Sigma Series in Pure Mathematics.* Heldermann Verlag, Berlin, 1992.

PART V
LATTICES AND QUADRATIC FORMS

30 LATTICES

Chapter 30
LATTICES

30.1 Introduction

Lattices play an important role in various areas, e.g., representation theory, coding theory, geometry and algebraic number theory.

A lattice in MAGMA is a free $\mathbf{Z}$-module contained in $\mathbf{Q}^n$ or $\mathbf{R}^n$, together with a positive definite inner product having image in $\mathbf{Q}$ or $\mathbf{R}$. The information specifying a lattice is a basis, given by a sequence of elements in $\mathbf{Q}^n$ or $\mathbf{R}^n$, and a bilinear product $(\cdot, \cdot)$, given by $(v, w) = vMw^{tr}$ for a positive definite matrix M.

Central to the lattice machinery in MAGMA is a fast implementation of the *Lenstra-Lenstra-Lovász lattice reduction* algorithm [LLL82]. The Lenstra-Lenstra-Lovász algorithm (LLL for short) takes a basis of a lattice and returns a new basis of the lattice which is *LLL-reduced* which usually means that the vectors of the new basis have small norms. The MAGMA algorithm is based on both the floating-point LLL algorithm of Nguyen and Stehlé [NS09] and the exact integral algorithm as described by de Weger in [dW87], but includes many optimisations, with particular attention to different kinds of input matrices. The algorithm can operate on either a basis matrix or a Gram matrix and can be controlled by many parameters (selection of methods, LLL reduction constants, step and time limits, etc.).

For each lattice, a LLL-reduced basis for the lattice is computed and stored internally when necessary and subsequently used for many operations. This gives maximum efficiency for the operations, yet all the operations are presented using the external ("user") basis of the lattice. Lattices are thus a more efficient alternative to R-spaces where R is the integer ring $\mathbf{Z}$ since lattices use a LLL-reduced form of the basis matrix extensively instead of the Hermite form of a basis matrix used by R-spaces which may have very large entries.

Another important component of the lattice machinery is a very efficient algorithm for enumerating all vectors of a lattice with norms in a given range. This algorithm is used for computing the minimum, the shortest vectors, vectors up to a chosen length, and vectors close to or closest to a given vector (possibly) outside a lattice.

Several interesting lattices are directly accessible inside MAGMA using standard constructions (e.g., root lattices and preimages of linear codes). Additionally, an interface has been provided to convert lattices in the database of G. Nebe and N.J.A. Sloane [NS01a] into MAGMA format.

30.2 Presentation of Lattices

A lattice in MAGMA is a free **Z**-module of rank m inside $\mathbf{Z}^n$, $\mathbf{Q}^n$, or $\mathbf{R}^n$, where $m \leq n$. We call m the rank or dimension of the lattice and n its degree. A lattice with $m = n$ is called a *full* lattice. There are two pieces of information which completely determine a lattice L: the basis matrix B and the inner product matrix M. M is always a positive definite matrix (i.e., a matrix M such that $vMv^{tr} > 0$ for all non-zero vectors $v \in \mathbf{R}^n$).

All other information associated with a lattice L is derived from B and M. For example, the inner product $(\cdot, \cdot)$ for L is given by $(v, w) = vMw^{tr}$ and thus the Gram matrix F for L giving the inner products of the basis vectors is BMB^{tr}.

Lattices differ from R-spaces in that a lattice is always a **Z**-module even if entries of some elements of the lattice are not integers. For R-spaces the base ring R affects the module structure but for lattices the "base ring" R is just defined to be the smallest ring over which the basis and inner product matrices can be represented. MAGMA basically has two kinds of lattices: *exact* lattices for which all of the entries of B and M (and thus F) lie in either **Z** or **Q**, and *non-exact* lattices for which all of the entries of B and M (and thus F) lie in a particular approximate real field **R**.

For exact lattices, the "base ring" R may be **Z** or **Q** as mentioned above so that elements are represented over R, but otherwise there is nothing in the lattice which distinguishes between the **Z** and **Q** cases. When exact lattices are printed, the basis matrix B is presented uniquely as an integral matrix divided by a minimal positive integral denominator. The inner product matrix M is presented similarly. This tends to be a more natural presentation which reflects the **Z**-module structure. Another useful feature of this presentation is that the inner product can be efficiently defined to be the identity matrix divided by an appropriate integral scale factor. In this way, the basis can be kept rational while still obtaining an integral primitive Gram matrix (a matrix with integral entries which are coprime), thus avoiding the need for rescaling the basis by an irrational square root. This method is followed in the construction of special lattices provided within MAGMA.

For non-exact lattices, the basis matrix B is simply presented as it is represented over the ring R which is an approximate real field. Some operations applicable for exact lattices are not possible for non-exact lattices. Thus if in the description of a function below it is said that the function can only be applied to an exact lattice it means that the function cannot be applied to a lattice over an approximate real field.

Two lattices are said to be *compatible* if their inner product matrices are equal. Many operations on lattices assume that their arguments are compatible. Intuitively, two lattices and their elements are not comparable if their inner product matrices are different.

Associated with a lattice L is its coordinate lattice L' which has the standard basis (i.e., its basis matrix is the identity matrix) but has the same Gram matrix as L. Some operations, e.g., automorphism group, always work with the coordinates of elements of the lattice, so it is necessary to move to the coordinate lattice.

The category of lattices is `Lat`.

30.3 Creation of Lattices

Lattices can be created in elementary ways whereby the basis matrix or a generating matrix and possibly also the inner product matrix are specified. MAGMA also provides functions for creating lattices from linear codes or algebraic number fields and for creating some special lattices.

30.3.1 Elementary Creation of Lattices

This subsection describes the elementary ways of creating lattices by supplying the basis or the Gram matrix.

There are two ways of specifying the basis of a lattice at creation. First, a generating matrix can be specified whose rows need not be independent; the matrix is reduced to a LLL-reduced form and zero rows are removed to yield a basis matrix. Secondly, a basis matrix (with independent rows) can be specified; this matrix is used for the basis matrix without any changes being made to it. As well as the basis, a particular inner product can also be specified by a symmetric positive definite matrix. By default (when a particular inner product matrix is not given), the inner product is taken to be the standard Euclidean product.

Instead of giving the basis one can also define a lattice by its Gram matrix F which prescribes the inner products of the basis vectors. The basis is taken to be the standard basis and the inner product matrix is taken as F so that the Gram matrix for the lattice is also F.

```
Lattice(X, M)
```

```
Lattice(n, Q, M)
```

```
Lattice(S, M)
```

`CheckPositive` BOOLELT *Default :* `true`

Construct the lattice L specified by the given generating matrix and with inner product given by the $n \times n$ matrix M. The generating matrix can be specified by an $r \times n$ matrix X, an integer n with a sequence Q of ring elements of length rn (interpreted as a $r \times n$ matrix), or an R-space S of dimension r and degree n. In each case, the generating matrix need not have independent rows (though it always will in the R-space case). The matrix is reduced to a LLL-reduced form and zero rows are removed to yield a basis matrix B of rank m (so $m \leq r$). The lattice L is then constructed to have rank m, degree n, basis matrix B and inner product matrix M. By default MAGMA checks that M is positive definite but by setting `CheckPositive := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

```
Lattice(X)
```

```
Lattice(n, Q)
```

```
Lattice(S)
```

Construct the lattice L specified by the given generating matrix and with standard Euclidean inner product. The generating matrix can be specified by an $r \times n$ matrix X, an integer n with a sequence Q of ring elements of length rn (interpreted as a $r \times n$ matrix), or an R-space S of dimension r and degree n. In each case, the generating matrix need not have independent rows (though it always will in the R-space case). The matrix is reduced to a LLL-reduced form and zero rows are removed to yield a basis matrix B of rank m (so $m \leq r$). The lattice L is then constructed to have rank m, degree n, basis matrix B and standard Euclidean inner product.

`LatticeWithBasis(B, M)`

`LatticeWithBasis(n, Q, M)`

`LatticeWithBasis(S, M)`

`CheckIndependent`	BoolElt	*Default :* `true`
`CheckPositive`	BoolElt	*Default :* `true`

Construct the lattice L with the specified $m \times n$ basis matrix and with inner product given by the $n \times n$ matrix M. The basis matrix B can be specified by an $m \times n$ matrix B, an integer n with a sequence Q of ring elements of length mn (interpreted as an $m \times n$ matrix B), or an R-space S of dimension m and degree n (whose basis matrix is taken to be B). The rows of B must be independent (i.e., form a basis). The lattice L is then constructed to have rank m, degree n, basis matrix B and inner product matrix M. (Note that the basis matrix B is *not* reduced to a LLL-reduced form as in the `Lattice` function.) By default Magma checks that the rows of the matrix B specifying the basis are independent but by setting `CheckIndependent := false` this check will be suppressed. By default Magma also checks that M is positive definite but by setting `CheckPositive := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

`LatticeWithBasis(B)`

`LatticeWithBasis(n, Q)`

`LatticeWithBasis(S)`

`CheckIndependent`	BoolElt	*Default :* `true`

Construct the lattice L with the specified $m \times n$ basis matrix and with standard Euclidean inner product. The basis matrix B can be specified by an $m \times n$ matrix B, an integer n with a sequence Q of ring elements of length mn (interpreted as an $m \times n$ matrix B), or an R-space S of dimension m and degree n (whose basis matrix is taken to be B). The rows of B must be independent (i.e., form a basis). The lattice L is then constructed to have rank m, degree n, basis matrix B and standard Euclidean product. (Note that the basis matrix B is *not* reduced to a LLL-reduced form as in the `Lattice` function.) By default Magma checks that the rows of the matrix B specifying the basis are independent but by setting `CheckIndependent`

`:= false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

`LatticeWithGram(F)`

`LatticeWithGram(n, Q)`

`CheckPositive` BOOLELT *Default :* `true`

Construct a lattice with the given Gram matrix. The Gram matrix F can be specified as a symmetric $n \times n$ matrix F, or an integer n and a sequence Q of ring elements of length n^2 or $\binom{n+1}{2}$. In the latter case, a sequence of length n^2 is regarded as an $n \times n$ matrix and a sequence of length $\binom{n+1}{2}$ as a lower triangular matrix determining a symmetric matrix F.

This function constructs the lattice L of degree n having the standard basis and inner product matrix F, therefore having Gram matrix F as well. By default MAGMA checks that F is positive definite but by setting `CheckPositive := false` this check will be suppressed. (Unpredictable behaviour will follow if unchecked incorrect input is given.)

`StandardLattice(n)`

Create the standard lattice $\mathbf{Z}^n$ with standard Euclidean inner product.

`CoordinateLattice(L)`

Constructs the lattice with the same Gram matrix as the lattice L, but with basis equal to the canonical basis of the free module of Rank(L) over the base ring of L. The identification of basis elements thus determines an isometry of lattices.

`ScaledLattice(L,n)`

Constructs the coordinate lattice with Gram matrix equal to that of the lattice L scaled by the integer or rational n. The identification of basis elements thus determines an similitude of lattices.

Example H30E1

We create the lattice in $\mathbf{Z}^3$ of degree 3 and rank 2 generated by the vectors $(1, 2, 3)$ and $(3, 2, 1)$ in four different ways. The first two (identical) lattices are represented by LLL-reduced bases since the `Lattice` function is used to create them, while the latter two (identical) lattices remain represented by the original basis since the `LatticeWithBasis` function is used to create them.

```
> B := RMatrixSpace(IntegerRing(), 2, 3) ! [1,2,3, 3,2,1];
> B;
[1 2 3]
[3 2 1]
> L1 := Lattice(B);
> L1;
Lattice of rank 2 and degree 3
Basis:
( 2  0 -2)
```

```
( 1  2  3)
> L2 := Lattice(3, [1,2,3, 3,2,1]);
> L2 eq L1;
true
> L3 := LatticeWithBasis(B);
> L3;
Lattice of rank 2 and degree 3
Basis:
(1 2 3)
(3 2 1)
> L4 := LatticeWithBasis(3, [1,2,3, 3,2,1]);
> L4 eq L3, L1 eq L3;
true true
```

We next create a 3×3 positive definite matrix M and create the lattice L with basis the same as above but also with inner product matrix M. We note the Gram matrix of the lattice which equals BMB^{tr} where B is the basis matrix.

```
> B := RMatrixSpace(IntegerRing(), 2, 3) ! [1,2,3, 3,2,1];
> M := MatrixRing(IntegerRing(), 3) ! [3,-1,1, -1,3,1, 1,1,3];
> M;
[ 3 -1  1]
[-1  3  1]
[ 1  1  3]
> IsPositiveDefinite(M);
true
> L := LatticeWithBasis(B, M);
> L;
Lattice of rank 2 and degree 3
Basis:
(1 2 3)
(3 2 1)
Inner Product Matrix:
[ 3 -1  1]
[-1  3  1]
[ 1  1  3]
> GramMatrix(L);
[56 40]
[40 40]
```

Finally, we create a lattice C with the same Gram matrix as the last lattice, but with standard basis. To do this we use the `LatticeWithGram` function. This lattice C is the *coordinate* lattice of L: it has the same Gram matrix as L but is not compatible with L since its inner product matrix is different to that of L.

```
> F := MatrixRing(IntegerRing(), 2) ! [56,40, 40,40];
> C := LatticeWithGram(F);
> C;
Standard Lattice of rank 2 and degree 2
Inner Product Matrix:
```

```
[56 40]
[40 40]
> GramMatrix(C);
[56 40]
[40 40]
> C eq CoordinateLattice(L);
true
> GramMatrix(C) eq GramMatrix(L);
true
>  C eq L;
Runtime error in 'eq': Arguments are not compatible
```

30.3.2 Lattices from Linear Codes

This subsection presents some standard constructions (known as constructions 'A' and 'B') to obtain lattices from linear codes. In some cases the structural invariants of these lattices (e.g., minimum and kissing number, hence the centre density) can be deduced from invariants of the codes; in general one still gets estimates for the invariants of the lattices.

`Lattice(C, "A")`

Let C be a linear code of length n over a prime field $K := \mathbf{F}_p$. Construct the lattice $L \subseteq \mathbf{Z}^n$ which is the full preimage of C under the natural epimorphism from $\mathbf{Z}^n$ onto K^n.

`Lattice(C, "B")`

Let C be a linear code of length n over a prime field $K := \mathbf{F}_p$ such that all code words have coordinate sum 0. Construct the lattice $L \subseteq \mathbf{Z}^n$ which consists of all vectors reducing modulo p to a code word in C and whose coordinate sum is 0 modulo p^2. The inner product matrix is set to the appropriate scalar matrix so that the Gram matrix is integral and primitive (its entries are coprime).

Example H30E2

The 16-dimensional Barnes-Wall lattice Λ_{16} can be constructed from the first order Reed-Muller code of length 16 using construction 'B'. Note that the inner product matrix is the identity matrix divided by 2 so that the Gram matrix is integral and primitive.

```
> C := ReedMullerCode(1, 4);
> C: Minimal;
[16, 5, 8] Reed-Muller Code (r = 1, m = 4) over GF(2)
> L := Lattice(C, "B");
> L;
Lattice of rank 16 and degree 16
Basis:
(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)
(0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1)
```

```
(0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1)
(0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 2)
(0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1)
(0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 2)
(0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2)
(0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 2)
(0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1)
(0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 2)
(0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 2)
(0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 2)
(0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2)
(0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4)
Inner Product denominator: 2
```

30.3.3 Lattices from Algebraic Number Fields

Let K be an algebraic number field of degree n over $\mathbf{Q}$. Then K has n embeddings into the complex numbers, denoted by $\sigma_1, \ldots, \sigma_n$. By convention the first s of these are embeddings into the real numbers and the last $2t$ are pairs of complex embeddings such that $\sigma_{s+t+i} = \overline{\sigma_{s+i}}$ for $1 \le i \le t$. The T_2-norm $K \to \mathbf{R} : \alpha \mapsto \sum_{i=1}^{n} |\sigma_i(\alpha)|^2$ defines a positive definite norm on K in which an order or ideal is a positive definite lattice. We restrict to the r real embeddings and the first s complex embeddings to obtain an embedding of a rank n order or ideal $\mathbf{R}^s \times \mathbf{C}^s = \mathbf{R}^n$. By rescaling the complex coordinates by $\sqrt{2}$, we recover the T_2-norm of the element as its Euclidean norm under the embedding in $\mathbf{R}^n$. This embedding, given by

$$\alpha \mapsto \Big(\sigma_1(\alpha), \ldots, \sigma_s(\alpha), \sqrt{2}\,\mathrm{Re}(\sigma_{s+1}(\alpha)), \sqrt{2}\,\mathrm{Im}(\sigma_{s+1}(\alpha)), \ldots, \sqrt{2}\,\mathrm{Im}(\sigma_{s+t}(\alpha))\Big)\,,$$

is called the *Minkowski map* on K.

`MinkowskiLattice(O)`

`Lattice(O)`

Given an order O in a number field of degree n, create the lattice L in $\mathbf{R}^n$ generated by the images of the basis of O under the Minkowski map, followed by the isomorphism $O \to L$.

`MinkowskiLattice(I)`

`Lattice(I)`

Given an ideal I of an order O in a number field of degree n, create the lattice in $\mathbf{R}^n$ generated by the images of the basis of I under the Minkowski map, followed by the isomorphism $O \to L$.

`MinkowskiSpace(K)`

Construct the real inner product space V defined with respect to the inner product matrix of the T_2-norm on the number field K, followed by the Minkowski map $K \to V$.

Example H30E3

In this example we create the lattice corresponding to the equation order of the number field $\mathbf{Q}(\sqrt[3]{15})$. For comparison we apply the Minkowski map to the elements of the basis for the order.

```
> P<x> := PolynomialRing(Integers());
> R := EquationOrder(NumberField(x^3-15));
> w := R![0,1,0];
> L, f := MinkowskiLattice(R);
> L;
Lattice of rank 3 and degree 3 over Real Field
Basis:
Lattice of rank 3 and degree 3 over Real field of precision 30
Basis:
(1.0000000000000000000000000000 1.4142135623730950488016887242l
    0.0000000000000000000000000000)
(2.4662120743304701014916113231 5 -1.7438752816032172007962969421 6
    3.0204805898002556568833586998 8)
(6.0822019955734001848984449977 4 -4.3007662756163029791484373981 2
    -7.4491457008462102763572729513 9)
> B := Basis(R);
> f(B[2]);
> f(B[2]);
(2.4662120743304701014916113231 5 -1.7438752816032172007962969421 6
    3.0204805898002556568833586998 8)
```

Similarly, we create the lattice defined by the ideal generated by $w^2 + 1$, and verify the that the T_2-norm, given by `Length` on the number field, agrees with the norm of the lattice image.

```
> I := ideal< R | w^2+1 >;
> L, f := Lattice(I);
> B := Basis(I);
> [ Length(B[k]) : k in [1..3] ];
[ 113.97954334487115456712250954195650692309680189826241920804543
3346320196288153038584473344161989557225281060164140817511930289
0493325999786309897500571170938924477237 24,
693.2466059867202005546953349932134072596330121315299369603182 55
6345313709169384387254042929336529027528589276741384648056766427
8812020993502245102110970371559009076524,
4216.4658903569162793735728830149731403405245314924982352796529 5
1232760419192679099060699351691798691646069328322562756396547517
8203732351663613772502545069466948994156 ]
> [ Norm(f(B[k])) : k in [1..3] ];
[ 113.97954334487115456712250954 2,
```

```
693.24660598672020055469533 4993, 4216.46589035691627937357288302
]
```

30.3.4 Special Lattices

This subsection presents functions to construct some special lattices, namely root lattices, laminated lattices and Kappa-lattices.

A much wider variety of lattices can be found in a database provided by G. Nebe and N.J.A. Sloane at the web-site [NS01a]. These lattices can easily be made accessible to MAGMA by a conversion script also available at this web-site. The database currently contains (at least):

- The root lattices A_n and their duals for $1 \leq n \leq 24$
- The root lattices D_n and their duals for $1 \leq n \leq 24$
- The root lattices E_n and their duals for $6 \leq n \leq 8$
- The laminated lattices Λ_n for $1 \leq n \leq 24$ including the 16-dimensional Barnes-Wall lattice Λ_{16} and the Leech lattice Λ_{24}
- The Kappa-lattices K_n and their duals for $7 \leq n \leq 13$ including the Coxeter-Todd lattice K_{12}
- The perfect lattices up to dimension 7
- The 3-dimensional Bravais lattices
- Various interesting lattices in dimensions 20, 24, 28, 32, 40, 80, 105 including, e.g., some of the densest known lattices in dimension 32.

`Lattice(X, n)`

Given a family name X as a string which is one of `"A"`, `"B"`, `"C"`, `"D"`, `"E"`, `"F"`, `"G"`, `"Kappa"` or `"Lambda"`, together with an integer n, construct a lattice subject to the following specifications:

`A`: The root lattice A_n which is the zero-sum lattice in $\mathbf{Q}^{n+1}$.

`B`: $n \geq 2$: The root lattice B_n which is the standard lattice of dimension n.

`C`: $n \geq 3$: The root lattice C_n which is the even sublattice of $\mathbf{Z}^n$ and is equal to D_n.

`D`: $n \geq 3$: The root lattice D_n which is the even sublattice of $\mathbf{Z}^n$, also called the checkerboard lattice.

`E`: $6 \leq n \leq 8$: The root lattice E_n, also called Gosset lattice.

`F`: $n = 4$: The root lattice F_4 which is equal to D_4.

`G`: $n = 2$: The root lattice G_2 which is equal to A_2.

`Kappa`: $1 \leq n \leq 13$: The Kappa-lattice K_n. For $n = 12$ this is the Coxeter-Todd lattice.

`Lambda`: $1 \leq n \leq 31$: The laminated lattice Λ_n. For $n = 16$ this is the Barnes-Wall lattice, for $n = 24$ the Leech lattice.

To avoid irrational entries, each lattice is presented with inner product matrix taken to be the identity matrix divided by a suitable scale factor so that the Gram matrix is integral and primitive (its entries are coprime).

30.4 Lattice Elements

The following functions allow basic operations on elements of lattices. The elements of lattices are simply (row) vectors, just as for R-spaces. Most of the operations for R-space elements are also applicable to lattice elements.

30.4.1 Creation of Lattice Elements

`L . i`

Return the i-th basis element of the current basis of the lattice L.

`L ! Q`

`elt< L | Q >`

Given a lattice L of degree n and a sequence Q of length n, create the lattice element with the corresponding sequence elements as entries. The sequence must consist of elements coercible into the base ring of L. The resulting vector must lie within L.

`CoordinatesToElement(L, C)`

`Coordelt(L, C)`

Given a lattice L of rank m and a sequence or vector $C = [c_1, \ldots, c_m]$ of length m of *integers*, create the lattice element $c_1 \cdot b_1 + \ldots + c_m \cdot b_m$, where $[b_1, \ldots, b_m]$ is the basis of L.

`L ! 0`

`Zero(L)`

Return the zero element of the lattice L.

30.4.2 Operations on Lattice Elements

`-v`

Given an element v in a lattice L, return its negation $-v$ in L.

`v + w`

Given elements v and w in a lattice L, return the sum $v + w$ in L.

`v - w`

Given elements v and w in a lattice L, return the difference $v - w$ in L.

`v * s`
`s * v`

Given an element v in a lattice L and a scalar s of the ring S, return the product $s \cdot v$ (scalar multiplication of v by s). If s is an integer, the resulting vector will lie in L; otherwise the resulting vector will lie in the R-space of the appropriate degree whose coefficient ring is the parent S of s.

`v / s`

Given an element v in a lattice L and a scalar s of the ring S, return the product $(1/s) \cdot v$ (scalar multiplication of v by $1/s$). The resulting vector will always lie in the R-space of the appropriate degree whose coefficient ring is the field of fractions of the parent S of s.

`v div d`

Given an element v in a lattice L and an *integer* d, return the vector $(1/d) \cdot v$ (scalar multiplication of v by $1/d$) as an element of L if the scaled vector lies in L. If the scaled vector does not lie in L, an error ensues. Note that this is different from v/d.

`v +:= w`

(Assignment statement.) Replace lattice element v by the sum $v + w$.

`v -:= w`

(Assignment statement.) Replace lattice element v by the difference $v - w$.

`v *:= n`

(Assignment statement.) Replace lattice element v by the scalar product $n \cdot v$, where n is an integer.

`v * T`

Given an element v in a lattice L of degree n, return the result of multiplying v from the right by the $n \times n$ matrix T. The matrix T may be any matrix which is n by n and over the base ring of L. The resulting product must lie in the lattice L.

`InnerProduct(v, w)`
`(v, w)`

Given elements v and w of a lattice L, return their inner product (v, w) with respect to the inner product of L. This is vMw^{tr} where M is the inner product matrix of L.

`Norm(v)`

Given an element v of a lattice L, return its norm (v, v) with respect to the inner product of L. This is vMv^{tr} where M is the inner product matrix of L. Note that in the case of a lattice with standard Euclidean inner product this is the square of the usual Euclidean length.

`Length(v, K)`

`Length(v)`

Given an element v of a lattice L, return its length $\sqrt{(v,v)}$ with respect to the inner product of L as an element of the real field K. This is $\sqrt{vMv^{tr}}$ where M is the inner product matrix of L. The argument for the real field K may be omitted, in which case K is taken to be the current default real field. In the case of a lattice with standard Euclidean inner product this is the usual Euclidean length.

`Support(v)`

Given an element v of a lattice L, return its support, i.e., the numbers of the columns at which v has non-zero entries.

30.4.3 Predicates and Boolean Operations

`v in L`

Given an element v of a lattice which is compatible with the lattice L, return `true` if and only if v is in L.

`v eq w`

Given elements v and w of a lattice L, return `true` if and only if lattice elements v and w of lattice L are equal.

`v ne w`

Given elements v and w of a lattice L, return `false` if and only if lattice elements v and w of lattice L are equal.

`IsZero(v)`

Given an element v of a lattice L, return `true` if and only if v is the zero element L.

30.4.4 Access Operations

`ElementToSequence(v)`

`Eltseq(v)`

Given an element v of a lattice L of degree n, return the sequence of entries of v of length n.

`Coordinates(v)`

Given an element v of a lattice L having $\mathbf{Z}$-basis $[b_1, \ldots, b_m]$, return a sequence $[c_1, \ldots, c_m]$ of elements of $\mathbf{Z}$ giving the (unique) coordinates of v relative to the $\mathbf{Z}$-basis, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

`Coordinates(L, v)`

Given a lattice L of degree n having $\mathbf{Z}$-basis $[b_1, \ldots, b_m]$, together with an element v of a lattice L' (also of degree n), return a sequence $[c_1, \ldots, c_m]$ of elements of $\mathbf{Z}$ giving the (unique) coordinates of v relative to the $\mathbf{Z}$-basis of L, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

`CoordinateVector(v)`

Given an element v of a lattice L having $\mathbf{Z}$-basis $[b_1, \ldots, b_m]$, return the vector $c = (c_1, \ldots, c_m)$ of the coordinate lattice C of L (see the function `CoordinateLattice`) giving the (unique) coordinates of v relative to the $\mathbf{Z}$-basis, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

`CoordinateVector(L, v)`

Given a lattice L of degree n having $\mathbf{Z}$-basis $[b_1, \ldots, b_m]$, together with an element v of a lattice L' (also of degree n), return the vector $c = (c_1, \ldots, c_m)$ of the coordinate lattice C of L (see the function `CoordinateLattice`) giving the (unique) coordinates of v relative to the $\mathbf{Z}$-basis of L, so that $v = c_1 \cdot b_1 + \ldots + c_m \cdot b_m$.

Example H30E4

This example demonstrates simple uses of the operations on lattice elements.

```
> L := LatticeWithBasis(3, [1,0,0, 1,2,3, 3,6,2]);
> L;
Lattice of rank 3 and degree 3
Basis:
(1 0 0)
(1 2 3)
(3 6 2)
> Coordelt(L, [1, 2, 1]);
( 6 10  8)
> v := L.2;
> w := L ! [2, 4, 6];
> Eltseq(v);
[ 1, 2, 3 ]
> Coordinates(w);
[ 0, 2, 0 ]
> Coordelt(L, [1, 1, 1]);
(5 8 5)
> Norm(v);
14
> InnerProduct(v, w);
28
> A := MatrixRing(Integers(), 3);
> X := A ! [0,-1,0, 1,0,0, 0,1,2];
> X;
[ 0 -1  0]
```

```
[ 1  0  0]
[ 0  1  2]
> u := L.1 + L.3;
> Determinant(X);
2
> Norm(u);
56
> u * X;
( 6 -2  4)
> Norm(u * X);
56
```

30.5 Properties of Lattices

The following functions provide access to the elementary attributes and properties of lattices. Other attributes and invariants of a lattice, e.g., successive minimum, kissing number, and theta series are described in subsequent sections.

30.5.1 Associated Structures

`AmbientSpace(L)`

The ambient rational or real vector space in which the lattice L embeds, followed by the embedding map.

`CoordinateSpace(L)`

The ambient vector space of the coordinate lattice, i.e., the vector space of dimension equal to the rank of the lattice L, with inner product matrix equal to the Gram matrix of L. The embedding map is returned as the second return value.

`Category(L)`

`Type(L)`

Returns the category `Lat` of lattices.

30.5.2 Attributes of Lattices

`Dimension(L)`

`Rank(L)`

Return the rank of the lattice L, which equals the number of basis elements in L. Note that the rank of the lattice may be smaller than its degree n, which is the dimension of the real space $\mathbf{R}^n$ in which L is defined.

`Degree(L)`

Return the degree of the lattice L, which is the dimension n of the real space $\mathbf{R}^n$ in which L is defined.

`Degree(v)`

Return the degree of the lattice element v, which the degree of the lattice to which it belongs.

`Content(L)`

Given an exact lattice L, return the largest rational number c such that $(u, v) \in c\mathbf{Z}$ for all $u, v \in L$.

`Level(L)`

Given an integral lattice L, return the smallest integer k such that $k(v, v) \in 2\mathbf{Z}$ for all v in the dual of L.

`Determinant(L)`

Returns the determinant of the lattice L, which is defined to be the determinant of the Gram matrix F of L. For a full rank lattice the square root of `Determinant(L)` is the volume of a fundamental parallelotope of the lattice.

`GramMatrix(L)`

Return the Gram matrix for the lattice L of rank m, which is the $m \times m$ matrix $F = BMB^{tr}$, where B is the basis matrix of L and M is the inner product matrix of L. Thus the (i, j)-th entry of F equals the inner product of the basis vectors b_i and b_j of L.

`GramMatrix(X)`

Given a matrix X, return XX^{tr}. Note that this function will take half the time as would be taken for the invocation `X*Transpose(X)` since the symmetry of the result is taken advantage of.

`InnerProductMatrix(L)`

The inner product matrix M of the lattice L, which is an $n \times n$ matrix, where n is the degree of L. If L has the standard Euclidean product, M is the identity matrix.

`Basis(L)`

Return the basis of the lattice L as a sequence $[b_1, \ldots, b_m]$ of elements of L.

`BasisMatrix(L)`

Return the $m \times n$ matrix having the basis elements of L as rows, where m is the rank and n the degree of L. The coefficient ring of the matrix is the same as the base ring of L.

`BasisDenominator(L)`

Given an exact lattice L, return the common denominator of the entries of the current basis of L.

`QuadraticForm(L)`

The quadratic form of the lattice L as a multivariate polynomial.

30.5.3 Predicates and Booleans on Lattices

`L eq M`

Given lattices L and M, return `true` if and only if the lattices have the same basis matrix and inner product matrix.

`L ne M`

The logical negation of `eq`.

`L subset M`

Return `true` if and only if L is a sublattice of M, i.e., both L and M are lattices, and L is a subset of M.

`IsExact(L)`

Return `true` if and only if L is an exact lattice, i.e., the coefficient ring of L is $\mathbf{Z}$ or $\mathbf{Q}$ and $(v, w) \in \mathbf{Q}$ for all $v, w \in L$.

`IsIntegral(L)`

Return `true` if and only if L is an integral lattice, i.e., if and only if $(v, w) \in \mathbf{Z}$ for all $v, w \in L$.

`IsEven(L)`

Return `true` if and only if L is an even lattice, i.e., if and only if L is integral and $(v, v) \in 2\mathbf{Z}$ for all $v \in L$.

30.5.4 Base Ring and Base Change

`BaseRing(L)`

`CoefficientRing(L)`

Return the base ring of the lattice L, which is the ring over which the elements of L are represented. Note that lattices are always **Z**-modules even if the base ring is not **Z**; the base ring R is just defined to be the smallest ring over which the basis and inner product matrices can be represented. See the section on presentation of lattices at the beginning of the chapter for further discussion.

`CoordinateRing(L)`

Return the ring of coordinate coefficients for the lattice L. This currently will always return the integer ring **Z**.

`ChangeRing(L, S)`

`BaseChange(L, S)`

`BaseExtend(L, S)`

Given a lattice L, return the lattice L' obtained from coercing the entries of the basis and of the matrix for the inner product into the ring S, together with the homomorphism from L to L'. This will result in an error if any of the entries is not coercible into S. This function is only really useful for moving between real fields of varying precision (it is unnecessary and ineffectual to change a lattice from **Z** to **Q**; see the section on presentation of lattices).

30.6 Construction of New Lattices

30.6.1 Sub- and Superlattices and Quotients

`sub< L | S >`

Given a lattice L and a list S, construct the sublattice L' of L generated by the elements specified by the list S. Each term S_i of the list S must be an expression defining an object of one of the following types:

(a) an element of L or coercible into L,

(b) a set or sequence of elements coercible into L,

(c) a sublattice of L,

(d) a set or sequence of sublattices of L.

The constructor returns the sublattice L' and the inclusion homomorphism from L' into L.

`ext< L | S >`

Given a lattice L lying inside $V = \mathbf{R}^n$ and a list S, construct the superlattice L' of L generated by L together with the elements specified by the list S. Each term S_i of the list S must be an expression defining an object of one of the following types:

(a) an element of V or coercible into V,

(b) a set or sequence of elements coercible into V,

(c) a sublattice of V,

(d) a set or sequence of sublattices of V.

The constructor returns the superlattice L' and the inclusion homomorphism from L into L'.

`T * L`

Given a lattice L of rank m and an $l \times m$ integer matrix, construct the sublattice of L defined by the transformation matrix T, i.e., the lattice generated by the rows of the matrix obtained by multiplying the basis matrix of L from the left by T. The resulting lattice will have rank less than or equal to l.

`s * L`
`L * s`

Given a lattice L and a scalar s, construct the sublattice or superlattice of L obtained by multiplying the basis matrix of L by the scalar s.

`L / s`

Given a lattice L and a scalar s, construct the sublattice or superlattice of L obtained by multiplying the basis matrix of L by the scalar $1/s$.

`quo< L | S >`

Given a lattice L and a list S, construct the quotient L/L', where L' is the sublattice of L generated by the elements of the list S. The elements of S must be the same as for the `sub<>` constructor. The quotient $Q := L/L'$ is constructed as an abelian group. As a second value the function returns the natural epimorphism $L \to Q$.

`L / S`

Given two lattices L and S such that S is a sublattice of L, construct the quotient $Q := L/S$ as an abelian group. As a second value the function returns the natural epimorphism $L \to Q$.

`Index(L, S)`

Given a lattice L and a sublattice S of L, return the index of S in L. This is the cardinality of the quotient L/S. If the index is infinite, zero is returned.

Example H30E5

We demonstrate simple uses of the `sub-`, `ext-` and `quo`-constructors.

```
> L := LatticeWithBasis(4, [1,2,3,4, 0,1,1,1, 0,1,3,5]);
> L;
Lattice of rank 3 and degree 4
Basis:
(1 2 3 4)
(0 1 1 1)
(0 1 3 5)
> E := ext<L | [1,0,0,0]>;
> E;
Lattice of rank 3 and degree 4
Basis:
( 1  0  0  0)
( 0  1  0 -1)
( 0  1  1  1)
> Index(E, L);
2
> Q, f := quo<E | L>;
> Q;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*Q.1 = 0
> f(E.1);
Q.1
```

30.6.2 Standard Constructions of New Lattices

The functions in this section enable one to construct new lattices from old ones using standard operations. Note that the functions will preserve the basis matrices of their arguments if possible (e.g., `DirectSum`), but if this is not possible the basis of the resulting lattice will be LLL-reduced (e.g., +, `meet`).

`Dual(L)`

`Rescale` BoolElt *Default :* `true`

Let L be a lattice in $\mathbf{R}^n$ and let $V := \mathbf{R} \otimes_{\mathbf{Z}} L$ be the real vector space generated by L. Then the dual lattice $L^{\#}$ of L is defined by $L^{\#} := \{v \in V | (v, l) \in \mathbf{Z}\ \forall l \in L\}$. For an integral lattice L one always has $L \subseteq L^{\#}$. This function returns a rescaled version L' of the dual $L^{\#}$ by default so that the basis of L' is LLL-reduced and L' is an integral lattice and its Gram matrix is primitive (its entries are coprime). By setting the parameter `Rescale` to `false`, the rescaling can be suppressed and the proper dual lattice is returned.

`PartialDual(L, n)`

`Rescale` BoolElt *Default :* `true`

Given an integral lattice L and a positive integer n that divides the exponent e of the `DualQuotient` group of L, this function computes the nth partial dual of L. This is defined by pulling back $(e/n) \cdot g$ for generators g of the `DualQuotient` and intersecting with the lattice itself.

`DualBasisLattice(L)`

Let L be a lattice in $\mathbf{R}^n$ and let $V := \mathbf{R} \otimes_{\mathbf{Z}} L$ be the real vector space generated by L, then $L^{\#} := \{v \in V | (v, l) \in \mathbf{Z}\ \forall l \in L\}$ is the dual lattice of L. Let B be the basis matrix of L, M the inner product matrix of L, and F the Gram matrix of L. This function returns the dual $L^{\#}$ as the lattice with basis matrix $F^{-1}B$ and inner product matrix M (so that its Gram matrix is F^{-1}).

`DualQuotient(L)`

Given an integral lattice L, construct the dual quotient Q of L, which is defined to be the finite abelian group $L^{\#}/L$ of order `Determinant(L)`, where $L^{\#}$ is the unscaled dual lattice of L (the lattice returned by `Dual` with the `Rescale` parameter set to `false`). This function returns three values:

(a) The dual quotient Q.

(b) The unscaled dual lattice $L^{\#}$.

(c) The natural epimorphism $\phi : L^{\#} \to Q$ whose kernel is L.

`EvenSublattice(L)`

Given an integral lattice L, construct its maximal even sublattice together with the natural embedding into L.

Example H30E6

```
> L := Lattice(LatticeDatabase(), 541);
> Dimension(L);
29
> IsEven(L);
true
> IsEven(Dual(L));
false;
> G := DualQuotient(L);
> Exponent(G);
8
> Factorization(Determinant(L));
[ <2, 31> ]
> PartialDual(L,1) eq L;
true
> Factorization(Determinant(PartialDual(L, 2)));
```

```
[ <2, 54> ]
> Factorization(Determinant(PartialDual(L, 4)));
[ <2, 73> ]
> Factorization(Determinant(PartialDual(L, 8)));
[ <2, 56> ]
> Factorization(Determinant(Dual(L)));
[ <2, 56> ]
```

`L + M`

Given compatible lattices L and M, construct the lattice generated by their union.

`L meet M`

Given compatible lattices L and M, construct their intersection $L \cap M$.

`DirectSum(L, M)`

`OrthogonalSum(L, M)`

Given lattices L and M, construct their orthogonal sum which is their direct sum with inner product being the orthogonal sum of the inner products of L and M.

`OrthogonalDecomposition(L)`

Given a lattice L, construct the sequence of indecomposable orthogonal summands composing L. Additional bilinear forms can be given in F. In this case the decomposition will be orthogonal wrt. these forms as well.

`OrthogonalDecomposition(F)`

`Optimize` BoolElt *Default :* `false`

Given a sequence of bilinear forms F, where the first form is positive definite, returns the basis matrices $B_1, \ldots, B_s$ of the indecomposable orthogonal summands of the standard lattice $\mathbf{Z}^n$ wrt. the forms in F. The second return value is a list of s sequences. The i-th sequence contains the forms of F wrt. to basis described by B_i. If `Optimize` is set, then the basis matrices will be LLL reduced wrt. the first form in F.

`TensorProduct(L, M)`

Given two lattices L and M, construct their tensor product with inner product given by the Kronecker product of the matrices defining the inner products of L and M.

`ExteriorSquare(L)`

Given a lattice L, construct its exterior square, generated by the skew tensors in $L \otimes L$. The inner product is inherited from the inner product of the tensor square of the vector space containing L and the exterior square lattice lies inside the tensor square of the lattice.

`SymmetricSquare(L)`

Given a lattice L, construct its symmetric square, generated by the symmetric tensors in $L \otimes L$. The inner product is inherited from the inner product of the tensor square of the vector space containing L and the symmetric square lattice lies inside the tensor square of the lattice.

`PureLattice(L)`

Given a lattice L of degree n with integral or rational entries, return the pure lattice $P = (\mathbf{Q} \otimes L) \cap \mathbf{Z}^n$ of L. The pure lattice P generates the same subspace in $\mathbf{Q}^n$ over $\mathbf{Q}$ that L does but the elementary divisors of its basis matrix are trivial.

`IntegralBasisLattice(L)`

Given an exact lattice L, return the lattice obtained from L by multiplying the basis by the smallest positive scalar S so that the resulting basis is integral, and S.

30.7 Reduction of Matrices and Lattices

The functions in this section perform reduction of lattice bases. For each reduction algorithm there are three functions: a function which takes a basis matrix, a function which takes a Gram matrix and a function which takes a lattice.

30.7.1 LLL Reduction

The Lenstra-Lenstra-Lovász algorithm [LLL82] was first described in 1982 and was immediately used by the authors to provide a polynomial-time algorithm for factoring integer polynomials, for solving simultaneous diophantine equations and for solving the integer programming problem. It very quickly received much attention and in the last 25 years has found an incredible range of applications, in such areas as computer algebra, cryptography, optimisation, algorithmic number theory, group theory, etc. However, the original LLL algorithm is mainly of theoretical interest, since, although it has polynomial time complexity, it remains quite slow in practice. Rather, floating-point variants are used, where the underlying Gram-Schmidt orthogonalisation is performed with floating-point arithmetic instead of rational arithmetic (which produces huge numerators and denominators). Most of these floating-point variants are heuristic, but here we use the provable Nguyen-Stehlé algorithm [NS09] (see also [Ste09] for more details about the implementation).

Let $\delta \in (1/4, 1]$ and $\eta \in [1/2, \sqrt{\delta})$. An ordered set of d linearly independent vectors $b_1, b_2, \ldots, b_d \in \mathbf{R}^n$ is said to be (δ, η)-LLL-reduced if the two following conditions are satisfied:

(a) For any $i > j$, we have $|\mu_{i,j}| \leq \eta$,

(b) For any $i < d$, we have $\delta \|b_i^*\|^2 \leq \|b_{i+1}^* + \mu_{i+1,i} b_{i+1}^*\|^2$,

where $\mu_{i,j} = (b_i, b_j^*)/\|b_j^*\|^2$ and b_i^* is the i-th vector of the Gram-Schmidt orthogonalisation of $(b_1, b_2, \ldots, b_d)$. LLL-reduction classically refers to the case where $\eta = 1/2$ and $\delta = 3/4$ since it was the choice of parameters originally made in [LLL82], but the closer that η

and δ are to $1/2$ and 1, respectively, the more reduced the lattice basis should be. In the classical LLL algorithm, the polynomial-time complexity is guaranteed as long as $\delta < 1$, whereas the floating-point LLL additionally requires that $\eta > 1/2$.

A (δ, η)-LLL-reduced basis $(b_1, b_2, \ldots, b_d)$ of a lattice L has the following useful properties:

(a) $\|b_1\| \leq \left(\delta - \eta^2\right)^{-(d-1)/4} (\det L)^{1/d}$,

(b) $\|b_1\| \leq (\delta - \eta^2)^{-(d-1)/2} \min_{b \in L \setminus \{0\}} \|b\|$,

(c) $\prod_{i=1}^{d} \|b_i\| \leq (\delta - \eta^2)^{-d(d-1)/4} (\det L)$,

(d) For any $j < i$, $\|b_j^*\| \leq (\delta - \eta^2)^{(j-i)/2} \|b_i^*\|$.

The default Magma parameters are $\delta = 0.75$ and $\eta = 0.501$, so that $(\delta - \eta^2)^{-1/4} < 1.190$ and $(\delta - \eta^2)^{-1/2} < 1.416$. The four previous bounds can be reached, but in practice one usually obtains better bases. It is possible to obtain lattice bases satisfying these four conditions without being LLL-reduced, by using the so-called Siegel condition [Akh02]; this useful variant, though available, is not the default one.

Internally, the LLL routine may use up to eight different LLL variants, one of them being de Weger's exact integer method [dW87], and the seven others relying upon diverse kinds of floating-point arithmetic. All but one of these variants are heuristic and implement diverse ideas from [SE94], Victor Shoup's NTL [Sho] and [NS06]. The heuristic variants possibly loop forever, but when that happens it is hopefully detected and a more reliable variant is used. For a given input, the LLL routine tries to find out which variant should be the fastest, and may eventually call other variants before the end of the execution: either because it deems that the current variant loops forever or that a faster variant could be used with the thus-far reduced basis. This machinery remains unknown to the user (except when the LLL verbose mode is activated) and makes the Magma LLL routine the fastest currently available, with the additional advantage of being fairly reliable.

The floating-point variants essentially differ on the two following points:

(a) whether or not the matrix basis computations are performed with the help of a complete knowledge of the Gram matrix of the vectors (the matrix of the pairwise inner products),

(b) the underlying floating-point arithmetic.

In theory, to get a provable variant, one needs the Gram matrix and arbitrary precision floating-point numbers (as provided by the MPFR-based [Pro] Magma real numbers). In practice, to get an efficient variant, it is not desirable not to maintain the exact knowledge of the Gram matrix (maintaining the Gram matrix represents asymptotically a large constant fraction of the total computational cost), while it is desirable to use the machine processor floating-point arithmetic (for instance, C doubles). Three of the seven variants use the Gram matrix, while the four others do not. In each group (either the three or the remaining four), one variant relies on arbitrary precision floating-point arithmetic, another on C doubles and another on what we call doubles with extended exponent. These last variants are important because they are almost as fast as the variants based on C doubles, and can be used for much wider families of inputs: when the initial basis is made of integers larger than approximately 500 bits, overflows can occur with the C doubles. To be

precise, a "double with extended exponent" is a C double with a C integer, the last one being the extended exponent. So far, we have described six variants. The seventh does not rely on the Gram matrix and is based on an idea similar to "doubles with extended exponents". The difference is that for a given vector of dimension n, instead of having n pairs of doubles and integers, one has n doubles and only one integer: the extended exponent is "factorised". This variant is quite often the one to use in practice, because it is reasonably efficient and not too unreliable.

Important warning. By default, the `LLL` routine is entirely reliable. This implies the default provable variant can be *much* slower than the heuristic one. By setting `Proof` to `false`, a significant run-time improvement can be gained without taking much risk: even in that case, the `LLL` routine remains more reliable than any other implementation based on floating-point arithmetic. For any application where guaranteed LLL-reduction is not needed, we recommend setting `Proof` to `false`.

Recommended usage. The formal description of the function `LLL` below explains all the parameters in detail, but we first note here some common situations which arise. The desired variant of LLL is often not the default one. Since the LLL algorithm is used in many different contexts and with different output requirements, it seems impossible to define a natural default variant, and so using the LLL routine efficiently often requires some tuning. Here we consider three main-stream situations.

- It may be desired to obtain the main LLL inequalities (see the introduction of this subsection), without paying much attention to the δ and η reduction parameters. In this case one should activate the `Fast` parameter, possibly with the verbose mode to know afterwards which parameters have been used.
- It may be desired to have the main LLL inequalities for a given pair of parameters (δ, η). In this case one should set the parameters `Delta` and `Eta` to the desired values, and set `SwappingCondition` to `"Siegel"`.
- It may be desired to compute a very well LLL-reduced basis. In this case one should set `Delta` to 0.9999, `Eta` to 0.5001, and possibly also activate the `DeepInsertion` option.

In any case, if you want to be absolutely sure of the quality of the output, you need to keep the `Proof` option activated.

Example H30E7

This example is meant to show the differences between the three main recommended usages above. Assume that we are interested in the lattice generated by the rows of the matrix B defined as follows.

```
> R:=RealField(5);
> B:=RMatrixSpace(IntegerRing(), 50, 51) ! 0;
> for i := 1 to 50 do B[i][1] := RandomBits(10000); end for;
> for i := 1 to 50 do B[i][i+1] := 1; end for;
```

The matrix B is made of 100 vectors of length 101. Each entry of the first column is approximately 10000 bits long. Suppose first that we use the default variant.

```
> time C:=LLL(B:Proof:=false);
```

```
Time: 11.300
> R!(Norm (C[1]));
5.1959E121
```

The output basis is $(0.75, 0.501)$-reduced. Suppose now that we only wanted to have a basis that satisfies the main LLL properties with $\delta = 0.75$ and $\eta = 0.501$. Then we could have used the Siegel swapping condition.

```
> time C:=LLL(B:Proof:=false, SwapCondition:="Siegel");
Time: 10.740
> R!(Norm (C[1]));
6.6311E122
```

Notice that the quality of the output is quite often worse with the Siegel condition than with the Lovász condition, but the main LLL properties are satisfied anyway. Suppose now that we want a very well reduced basis. Then we fix δ and η close to 1 and 0.5 respectively.

```
> time C:=LLL(B:Proof:=false, Delta:=0.9999, Eta:=0.5001);
Time: 19.220
> R!(Norm (C[1]));
1.8056E121
```

This is of course more expensive, but the first output vector is significantly shorter. Finally, suppose that we only wanted to "shorten" the entries of the input basis, very efficiently and without any particular preference for the reduction factors δ and η. Then we could have used the `Fast` option, that tries to choose such parameters in order to optimize the running-time.

```
> time C:=LLL(B:Proof:=false, Fast:=1);
Time: 8.500
> R!(Norm (C[1]));
6.2746E121
```

By activating the verbose mode, one can know for which parameters the output basis is reduced.

```
> SetVerbose ("LLL", 1);
> C:=LLL(B:Proof:=false, Fast:=1);
[...]
The output basis is (0.830,0.670)-reduced
```

`LLL(X)`

`Al`	MonStgElt	*Default :* "*New*"
`Proof`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*FP*"
`Delta`	RngElt	*Default :* 0.75
`Eta`	RngElt	*Default :* 0.501
`InitialSort`	BoolElt	*Default :* `false`
`FinalSort`	BoolElt	*Default :* `false`

`StepLimit`	RngIntElt	*Default :* 0
`TimeLimit`	RngElt	*Default :* 0.0
`NormLimit`	RngIntElt	*Default :*
`UseGram`	BoolElt	*Default :* `false`
`DeepInsertions`	BoolElt	*Default :* `false`
`EarlyReduction`	BoolElt	*Default :* `false`
`SwapCondition`	MonStgElt	*Default :* "*Lovasz*"
`Fast`	RngIntElt	*Default :* 0
`Weight`	SeqEnum	*Default :* $[0, \ldots, 0]$

Given a matrix X belonging to the matrix module $S = \mathrm{Hom}_R(M, N)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, compute a matrix Y whose non-zero rows are a LLL-reduced basis for the **Z**-lattice spanned by the rows of X (which need not be **Z**-linearly independent). The LLL function returns three values:

(a) A LLL-reduced matrix Y in S whose rows span the same lattice (**Z**-module) as that spanned by the rows of X.

(b) A unimodular matrix T in the matrix ring over **Z** whose degree is the number of rows of X such that $TX = Y$;

(c) The rank of X.

Note that the returned transformation matrix T is *always* over the ring of integers **Z**, even if R is not **Z**.

The input matrix X does not need to have linearly independent rows: this function performs a floating-point variant of what is usually known as the MLLL algorithm of M. Pohst [Poh87]. By default the rows of the returned matrix Y are sorted so that all the zero rows are at the bottom. The non-zero vectors are sorted so that they form a LLL-reduced basis (except when `FinalSort` is `true`; see below).

A detailed description of the parameters now follows. See the discussion and the example above this function for recommended parameters for common cases.

By default, the `Proof` option is set to `true`. It means that the result is guaranteed. It is possible, and usually faster, to switch off this option: it will perform the same calculations, without checking that the output is indeed reduced by using the L^2 algorithm. For the vast majority of cases, we recommend setting `Proof` to `false`.

By default the δ and η constants used in the LLL conditions are set respectively to 0.75 and 0.501 (except when `Method` is `"Integral"`, see below). The closer δ and η are to 1 and 0.5, respectively, the shorter the output basis will be. In the original description of the LLL algorithm, δ is 0.75 and η is 0.5. Making δ and η vary can have a significant influence on the running time. If `Method` is `"Integral"`, then by default η is set to 0.5 and δ to 0.99. With this value of `Method`, the parameter δ can be chosen arbitrarily in $(0.25, 1]$, but when δ is 1, the running time

may increase dramatically. If `Method` is not `"Integral"`, then δ may be chosen arbitrarily in $(0.25, 1)$, and η may be chosen arbitrarily in $(0.5, \sqrt{\delta})$. Notice that the parameters δ and η used internally will be slightly larger and smaller than the given ones, respectively, to take floating-point inaccuracies into account and to ensure that the output basis will indeed be reduced for the expected parameters. In all cases, the output basis will be (δ, η)-LLL reduced.

For matrices over $\mathbf{Z}$ or $\mathbf{Q}$, there are two main methods for handling the Gram-Schmidt orthogonalisation variables used in the algorithm: the Nguyen-Stehlé floating-point method (called L^2) and the exact integral method described by de Weger in [dW87]. The Nguyen-Stehlé algorithm is implemented as described in the article, along with a few faster heuristic variants. When `Method` is `"FP"`, these faster variants will be tried first, and when they are considered as misbehaving, they will be followed by the proved variant automatically. When `Method` is `"L2"`, the provable L^2 algorithm is used directly. Finally, when `Method` is `"Integral"`, the De Weger integral method is used. In this case, the `DeepInsertions`, `EarlyReduction`, `Siegel SwapCondition`, `NormLimit`, `Fast` and `Weight` options are unavailable (the code is older and has not been updated for these). Furthermore, the default value for `Eta` is then 0.5. The default `FP` method is nearly always *very* much faster than the other two methods.

By default, it is possible (though it happens very infrequently) that the returned basis is not of the expected quality. In order to be sure that the output is indeed correct, `Method` can be set to either `"Integral"` or `"L2"`.

The parameter `InitialSort` specifies whether the vectors should be sorted by length before the LLL reduction. When `FinalSort` is `true`, the vectors are sorted by increasing length (and alphabetical order in case of equality) after the LLL reduction (this parameter was called `Sort` before V2.13). The resulting vectors may therefore not be strictly LLL-reduced, because of the permutation of the rows.

The parameter `UseGram` specifies that for the floating-point methods (`L2` and `FP`) the computation should be performed by computing the Gram matrix F, using the `LLLGram` algorithm below, and by updating the basis matrix correspondingly. MAGMA will automatically do this internally for the floating-point method if it deems that it is more efficient to do so, so this parameter just allows one to stipulate that this method should or should not be used. For the integral method, using the Gram matrix never improves the computation, so the value of this parameter is simply ignored in this case.

Setting the parameter `StepLimit` to a positive integer s will cause the function to terminate after s steps of the main loop. Setting the parameter `TimeLimit` to a positive real number t will cause the function to terminate after the algorithm has been executed for t seconds (process user) time. Similarly, setting the parameter `NormLimit` to a non-negative integer N will cause the algorithm to terminate after a vector of norm less or equal to N has been found. In such cases, the current reduced basis is returned, together with the appropriate transformation matrix and an upper bound for the rank of the matrix. Nothing precise can then be said exactly about the reduced basis (it will not be LLL-reduced in general of course) but will at least

be reduced to a certain extent.

When the value of the parameter `DeepInsertions` is `true`, Schnorr-Euchner's deep insertion algorithm is used [SE94]. It usually provides better bases, but can take significantly more time. In practice, one may first LLL-reduce the basis and then use the deep insertion algorithm on the computed basis.

When the parameter `EarlyReduction` is `true` and when any of the two floating-point methods are used, some early reduction steps are inserted inside the execution of the LLL algorithm. This sometimes makes the entries of the basis smaller very quickly. It occurs in particular for lattice bases built from minimal polynomial or integer relation detection problems. The speed-up is sometimes dramatic, especially if the reduction in length makes it possible to use C integers for the basis matrix (instead of multiprecision integers) or C doubles for the floating-point calculations (instead of multiprecision floating-point numbers or doubles with additional exponent).

The parameter `SwapCondition` can be set either to `"Lovasz"` (default) or to `"Siegel"`. When its value is `"Lovasz"`, the classical Lovász swapping condition is used, whereas otherwise the so-called *Siegel* condition is used. The Lovász condition tests whether the inequality $\|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2 \geq \delta \|b_{i-1}^*\|^2$ is satisfied or not, whereas in the Siegel case the inequality $\|b_i^*\|^2 \geq (\delta - \eta^2)\|b_{i-1}^*\|^2$ is used (see [Akh02] for more details). In the Siegel case, the execution may be significantly faster. Though the output basis may not be LLL-reduced, the classical LLL inequalities (see the introduction of this subsection) will be fulfilled.

When the option `Fast` is set to 1 or 2, all the other parameters may be changed internally to end the execution as fast as possible. Even if the other parameters are not set to the default values, they may be ignored. This includes the parameters `Delta`, `Eta` and `SwapCondition`, so that the output basis may not be LLL-reduced. However, if the verbose mode is activated, the chosen values of these parameters will be printed, so that the classical LLL inequalities (see the introduction of this subsection) may be used afterwards. When `Fast` is set to 2, the chosen parameters are such that the classical LLL inequalities will be at least as strong as for the default parameters `Delta` and`Eta`.

If `Weight` is the list of integers $[x_1, \ldots, x_n]$ (where n is the degree of the lattice), then the LLL algorithm uses a weighted Euclidean inner product: the inner product of the vectors $(v_1, v_2, \ldots, v_n)$ and $(w_1, w_2, \ldots, w_n)$ is defined to be $\sum_{i=1}^{n} \left(v_i \cdot w_i \cdot 2^{2x_i}\right)$.

By default, the `Al` parameter is set to `New`. If it is set to `Old`, then the former MAGMA LLL is used (prior to V2.13); notice that in this case, the execution is not guaranteed to finish and that the output basis may not be reduced for the given LLL parameters. This variant is not maintained anymore.

Note 1: If the input basis has many zero entries, then try to place them in the last columns; this will slightly improve the running time.
Note 2: If the elementary divisors of the input matrix are fairly small, relative to the size of the matrix and the size of its entries, then it may be very much quicker

to reduce the matrix to Hermite form first (using the function `HermiteForm`) and then to LLL-reduce this matrix. This case can arise: for example, when one has a "very messy" matrix for which it is known that the lattice described by it has a relatively short basis.
Note 3: Sometimes, the `EarlyReduction` variant can significantly decrease the running time.

`BasisReduction(X)`

`BasisReduction(X)`

This is a shortcut for `LLL(X:Proof:=false)`.

`LLLGram(F)`

`Isotropic` BoolElt *Default :* `false`

Given a symmetric matrix F belonging to the the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to a LLL-reduced form of the matrix X. The rows of the corresponding generator matrix X need not be **Z**-linearly independent in which case F will be singular. This function returns three values:

(a) A LLL-reduced Gram matrix G in S of the Gram matrix F;

(b) A unimodular matrix T in the matrix ring over **Z** whose degree is the number of rows of F such that $G = TFT^{tr}$.

(c) The rank of F (which equals the dimension of the lattice generated by X).

The options available are the same as for the `LLL` routine, except of course the `UseGram` option that has no sense here. The `Weight` parameter should not be used either. The routine can be used with any symmetric matrix (possibly not definite nor positive): Simon's indefinite LLL variant (which puts absolute values on the Lovász condition) is used (see [Sim05]).

When the option `Isotropic` is true, some attempt to deal with isotropic vectors is made. (This is only relevant when working on an indefinite Gram matrix). In particular, if the determinant is squarefree, hyperbolic planes are split off iteratively.

`LLLBasisMatrix(L)`

Given a lattice L with basis matrix B, return *the LLL basis matrix* B' of L, together with the transformation matrix T such that $B' = TB$. The LLL basis matrix B' is simply defined to be a LLL-reduced form of B; it is stored in L when computed and subsequently used internally by many lattice functions. The LLL basis matrix will be created automatically internally as needed with $\delta = 0.999$ by default (note that this is different from the usual default of 0.75); by the use of parameters to this function one can ensure that the LLL basis matrix is created in a way which is different to the default. The parameters (not listed here again) are the same as for the function `LLL` (q.v.), except that the limit parameters are illegal.

`LLLGramMatrix(L)`

Given a lattice L with Gram matrix F, return the *LLL Gram matrix* F' of L, together with the transformation matrix T such that $F' = TFT^{tr}$. F' is simply defined to be $B'(B')^{tr}$, where B' is the LLL basis matrix of L—see the function `LLLBasisMatrix`. The parameters (not listed here again) are the same as for the function `LLL` (q.v.), except that the limit parameters are illegal.

`LLL(L)`

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is LLL-reduced and $B' = TB$. Note that the inner product used in the LLL computation is that given by the inner product matrix of L (so, for example, the resulting basis may not be LLL-reduced with respect to the standard Euclidean norm). The LLL basis matrix of L is used, so calling this function with argument L is completely equivalent (ignoring the second return value) to the invocation `LatticeWithBasis(LLLBasisMatrix(L), InnerProductMatrix(L))`. The parameters (not listed here again) are the same as for the function `LLL` (q.v.), except that the limit parameters are illegal. The `BasisReduction` shortcut to turn off the `Proof` option is also available.

`BasisReduction(L)`

This is a shortcut for `LLL(L:Proof:=false)`.

`SetVerbose("LLL", v)`

(Procedure.) Set the verbose printing level for the LLL algorithm to be v. Currently the legal values for v are `true`, `false`, 0, 1, 2 and 3 (`false` is the same as 0, and `true` is the same as 1). The three non-zero levels notify when the maximum LLL-reduced rank of the LLL algorithm increases, level 2 also prints the norms of the reduced vectors at such points, and level 3 additionally gives some current status information every 15 seconds.

Example H30E8

We demonstrate how the LLL algorithm can be used to find very good multipliers for the extended GCD of a sequence of integers. Given a sequence $Q = [x_1, \ldots, x_n]$ of integers we wish to find the GCD g of Q and integers m_i for $1 \leq i \leq m$ such that $g = m_1 \cdot x_1 + \ldots + m_n \cdot x_m$.
For this example we set Q to a sequence of $n = 10$ integers of varying bit lengths (to make the problem a bit harder).

```
> Q := [ 67015143, 248934363018, 109210, 25590011055, 74631449,
>        10230248, 709487, 68965012139, 972065, 864972271 ];
> n := #Q;
> n;
10
```

We next choose a scale factor S large enough and then create the $n \times (n+1)$ matrix $X = [I_n|C]$ where C is the column vector whose i-th entry is $S \cdot Q[i]$.

```
> S := 100;
```

```
> X := RMatrixSpace(IntegerRing(), n, n + 1) ! 0;
> for i := 1 to n do X[i][i + 1] := 1; end for;
> for i := 1 to n do X[i][1] := S * Q[i]; end for;
> X;
[6701514300     1 0 0 0 0 0 0 0 0 0]
[24893436301800 0 1 0 0 0 0 0 0 0 0]
[10921000       0 0 1 0 0 0 0 0 0 0]
[2559001105500  0 0 0 1 0 0 0 0 0 0]
[7463144900     0 0 0 0 1 0 0 0 0 0]
[1023024800     0 0 0 0 0 1 0 0 0 0]
[70948700       0 0 0 0 0 0 1 0 0 0]
[6896501213900  0 0 0 0 0 0 0 1 0 0]
[97206500       0 0 0 0 0 0 0 0 1 0]
[86497227100    0 0 0 0 0 0 0 0 0 1]
```

Finally, we compute a LLL-reduced form L of X.

```
> L := LLL(X);
> L;
[   0    0    1    0  -15   -6    3    1    2   -3   -3]
[   0   -3    5   -3  -11   -1    0    8  -14    4    3]
[   0   -5   -2    2   -2   14    8   -6    8    1   -4]
[   0    6    1   -3  -10   -3  -14    5    0    8    8]
[   0   -1   -2    0   -2  -13    8    6    8   10   -2]
[   0   -9   -3  -11    5    7   -1    1    9  -11   -2]
[   0   16    0    3    3    9   -3    0   -1   -4  -11]
[   0   -6    1  -16    4   -1    6   -6   -5    7   -7]
[   0    9   -3  -10    7   -3    1   -7    8    0   18]
[-100   -3   -1   13   -1   -4    2    3    4    5   -1]
```

Notice that the large weighting on the first column forces the LLL algorithm to produce as many vectors as possible at the top of the matrix with a zero entry in the first column (since such vectors will have shorter norms than any vector with a non-zero entry in the first column). Thus the GCD of the entries in Q is the entry in the bottom left corner of L divided by S, viz. 1. The last n entries of the last row of L gives a sequence of multipliers M for the extended GCD algorithm. Also, taking the last n entries of each of the first $n-1$ rows of L gives independent null-relations for the entries of Q (i.e., the kernel of the corresponding column matrix). We check that M gives a correct sequence of multipliers.

```
> M := Eltseq(L[10])[2 .. n+1]; M;
[ 3, 1, -13, 1, 4, -2, -3, -4, -5, 1 ]
> &+[Q[i]*M[i]: i in [1 .. n]];
-1
```

30.7.2 Pair Reduction

`PairReduce(X)`

Given a matrix X belonging to the the matrix module $S = \mathrm{Hom}_R(M, N)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbf{Z}$ or $\mathbf{Q}$, compute a matrix Y whose rows form a pairwise reduced basis for the $\mathbf{Z}$-lattice spanned by the rows of X. Being pairwise reduced (i.e., $2|(v, w)| \leq \min(\|v\|, \|w\|)$ for all pairs of basis vectors) is a much simpler criterion than being LLL-reduced, but often yields sufficiently good results very quickly. It can also be used as a preprocessing for LLL or in alternation with LLL to obtain better bases. The rows of X need not be Z-linearly independent. This function returns two values:

(a) The pairwise reduced matrix Y row-equivalent to X as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbf{Z}$ whose degree is the number of rows of X such that $TX = Y$.

`PairReduceGram(F)`

`Check` BoolElt *Default :* `false`

Given a symmetric positive semidefinite matrix F belonging to the the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbf{Z}$ or $\mathbf{Q}$, so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to a pairwise reduced form of the matrix X. The rows of the corresponding matrix X need not be Z-linearly independent. This function returns two values:

(a) The pairwise reduced Gram matrix G of F as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbf{Z}$ whose degree is the number of rows of F which gives the corresponding transformation: $G = TFT^{tr}$.

The matrix F must be a symmetric positive semidefinite matrix; if it is not the results are unpredictable. By default, Magma does not check this since it may be expensive in higher dimensions and in many applications will be known a priori. The checking can be invoked by setting `Check := true`.

`PairReduce(L)`

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is pairwise reduced and $B' = TB$. Note that the inner product used in the pairwise reduction computation is that given by the inner product matrix of L (so, for example, the resulting basis may not be pairwise reduced with respect to the standard Euclidean norm).

30.7.3 Seysen Reduction

`Seysen(X)`

Given a matrix X belonging to the the matrix module $S = \mathrm{Hom}_R(M, N)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbf{Z}$ or $\mathbf{Q}$, compute a matrix Y whose rows form a Seysen-reduced basis for the $\mathbf{Z}$-lattice spanned by the rows of X. The rows of X need not be Z-linearly independent. The Seysen-reduced matrix Y is such that the entries of the corresponding Gram matrix $G = YY^{tr}$ and its inverse G^{-1} are simultaneously reduced. This function returns two values:

(a) The Seysen-reduced matrix Y corresponding to X as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbf{Z}$ whose degree is the number of rows of X such that $TX = Y$.

`SeysenGram(F)`

`Check` BoolElt *Default :* `false`

Given a symmetric positive semidefinite matrix F belonging to the the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is $\mathbf{Z}$ or $\mathbf{Q}$, so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to a Seysen-reduced form of the matrix X. The rows of the corresponding matrix X need not be Z-linearly independent. The Seysen-reduced Gram matrix G is such that the entries of G and its inverse G^{-1} are simultaneously reduced. This function returns two values:

(a) The Seysen-reduced Gram matrix G of F as a matrix in S;

(b) A unimodular matrix T in the matrix ring over $\mathbf{Z}$ whose degree is the number of rows of F which gives the corresponding transformation: $G = TFT^{tr}$.

The matrix F must be a symmetric positive semidefinite matrix; if it is not the results are unpredictable. By default, Magma does not check this since it may be expensive in higher dimensions and in many applications will be known a priori. The checking can be invoked by setting `Check := true`.

`Seysen(L)`

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is Seysen-reduced and $B' = TB$. Note that the inner product used in the Seysen-reduction computation is that given by the inner product matrix of L (so, for example, the resulting basis may not be Seysen-reduced with respect to the standard Euclidean norm). The effect of the reduction is that the basis of L' and the dual basis of L' will be simultaneously reduced.

Example H30E9

We demonstrate how the function `Seysen` can be used on the Gram matrix of the Leech lattice to obtain a gram matrix S for the lattice so that both S and S^{-1} are simultaneously reduced. Note that all three reduction methods yield a basis of vectors of minimal length, but the Seysen reduction also has a dual basis of vectors of norm 4. This is of interest in representation theory, for example, as the entries of the inverse of the Gram matrix control the size of the entries in a representation on the lattice.

```
> F := GramMatrix(Lattice("Lambda", 24));
> [ F[i][i] : i in [1..24] ];
[ 8, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ (F^-1)[i][i] : i in [1..24] ];
[ 72, 8, 12, 8, 10, 4, 4, 8, 8, 4, 4, 8, 4, 4, 4, 4, 4, 4, 4, 4, 8, 4, 4, 8 ]
> L := LLLGram(F);
> P := PairReduceGram(F);
> S := SeysenGram(F);
> [ L[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ P[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ S[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> [ (L^-1)[i][i] : i in [1..24] ];
[ 6, 4, 8, 4, 32, 4, 4, 8, 18, 4, 4, 8, 8, 8, 12, 4, 6, 4, 20, 4, 8, 4, 14, 4 ]
> [ (P^-1)[i][i] : i in [1..24] ];
[ 72, 40, 12, 8, 10, 4, 4, 8, 8, 4, 4, 8, 4, 4, 4, 4, 4, 4, 4, 4, 8, 4, 4, 8 ]
> [ (S^-1)[i][i] : i in [1..24] ];
[ 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
```

30.7.4 HKZ Reduction

An ordered set of d linearly independent vectors $b_1, b_2, \ldots, b_d \in \mathbf{R}^n$ is said to be Hermite-Korkine-Zolotarev-reduced (HKZ-reduced for short) if the three following conditions are satisfied:

(a) For any $i > j$, we have $|\mu_{i,j}| \leq 0.501$,

(b) The vector b_1 is a shortest non-zero vector in the lattice spanned by $b_1, b_2, \ldots, b_d$,

(c) The vectors $b_2 - \mu_{2,1}b_1, \ldots, b_d - \mu_{d,1}b_1$ are themselves HKZ-reduced,

where $\mu_{i,j} = (b_i, b_j^*)/\|b_j^*\|^2$ and b_i^* is the i-th vector of the Gram-Schmidt orthogonalisation of $(b_1, b_2, \ldots, b_d)$.

`HKZ(X)`

`Proof`	BoolElt	*Default :* `true`
`Unique`	BoolElt	*Default :* `false`
`Prune`	SeqEnum	*Default :* $[\ldots, [1.0, \ldots, 1.0], \ldots]$

Given a matrix X over a subring of the real field, compute a matrix Y whose non-zero rows are an HKZ-reduced basis for the **Z**-lattice spanned by the rows of X (which need not be **Z**-linearly independent). The HKZ function returns an HKZ-reduced matrix Y whose rows span the same lattice (**Z**-module) as that spanned by the rows of X, and a unimodular matrix T in the matrix ring over **Z** whose degree is the number of rows of X such that $TX = Y$.

Although the implementation relies on floating-point arithmetic, the result can be guaranteed to be correct. By default, the `Proof` option is set to `true`, thus guaranteeing the output. The run-time might be improved by switching off this option.

A given lattice may have many HKZ-reduced bases. If the `Unique` option is turned on, a uniquely determined HKZ-reduced basis is computed. This basis is chosen so that: for any $i > j$, we have $\mu_{i,j} \in [-1/2, 1/2)$; for any i, the first non-zero coordinate of the vector b_i^* is positive; and for any i, the vector b_i^* is the shortest among possible vectors for the lexicographical order.

The implementation solves the Shortest Vector Problem for lattices of dimensions 1 to d, where d is the number of rows of the input matrix X. The latter instances of the Shortest Vector Problem are solved with the same enumeration algorithm as is used for computing the minimum of a lattice, which can be expressed as a search within a large tree. The i-th table of the `Prune` optional parameter is used to prune the i-dimensional enumeration. The default value is:

$[[1.0 : j \texttt{ in } [1..i]] : i \texttt{ in } [1..\texttt{NumberOfRows}(X)]]$.

See the introduction of Section 30.8 for more details on the `Prune` option.

`HKZGram(F)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[\ldots, [1.0, \ldots, 1.0], \ldots]$

Given a symmetric positive semidefinite matrix F over a subring of the real field so that $F = XX^{tr}$ for some matrix X over the real field, compute a matrix G which is the Gram matrix corresponding to an HKZ-reduced form of the matrix X. The function returns the HKZ-reduced Gram matrix G of F, and a unimodular matrix T in the matrix ring over **Z** whose degree is the number of rows of F which gives the corresponding transformation: $G = TFT^{tr}$.

HKZ(L)

Proof	BoolElt	*Default :* true
Prune	SeqEnum	*Default :* $[\ldots, [1.0, \ldots, 1.0], \ldots]$

Given a lattice L with basis matrix B, return a new lattice L' with basis matrix B' and a transformation matrix T so that L' is equal to L but B' is HKZ-reduced and $B' = TB$.

SetVerbose("HKZ", v)

(Procedure.) Set the verbose printing level for the HKZ algorithm to be v. Currently the legal values for v are true, false, 0 and 1 (false is the same as 0, and true is the same as 1). More information on the progress of the computation can be obtained by setting the "Enum" verbose on.

GaussReduce(X)

GaussReduceGram(F)

GaussReduce(L)

Restrictions of the HKZ functions to lattices of rank 2.

Example H30E10

HKZ-reduced bases are much harder to compute than LLL-reduced bases, but provide a significantly more powerful representation of the spanned lattice. For example, computing all short lattice vectors is more efficient if one starts from an HKZ-reduced basis.

```
> d:=60;
> B:=RMatrixSpace(IntegerRing(), d, d)!0;
> for i:=1 to d do for j:=1 to d do B[i][j]:=Random(100*d); end for; end for;
> time C1 := LLL(B);
Time: 0.020
> time C2 := HKZ(B);
Time: 1.380
> m := Norm(C2[1]);
> time _:=ShortVectors(Lattice(C1), 11/10*m);
Time: 1.750
> time _:=ShortVectors(Lattice(C2), 11/10*m);
Time: 0.850
> time _:=ShortVectors(Lattice(C1), 3/2*m);
Time: 73.800
> time _:=ShortVectors(Lattice(C2), 3/2*m);
Time: 32.220
```

30.7.5 Recovering a Short Basis from Short Lattice Vectors

`ReconstructLatticeBasis(S, B)`

Given a basis S of a finite index sublattice of the lattice L spanned by the rows of the integral matrix B, return a matrix C whose rows span L and are not much longer than those of S. Specifically, the algorithm described in Lemma 7.1 of [MG02] is implemented, and the rows of the output matrix satisfy the following properties:

(a) For any i, $\|c_i\| \leq i^{1/2}\|s_i\|$,

(b) For any i, $\|c_i^*\| \leq \|s_i^*\|$,

where c_i^* (respectively s_i^*) denotes the i-th vector of the Gram-Schmidt orthogonalisation of $(c_1, c_2, \ldots, c_d)$ (respectively $(s_1, s_2, \ldots, s_d)$) .

30.8 Minima and Element Enumeration

The functions in this section are all based on one algorithm which enumerates all vectors of a lattice in a specified hyperball [FP83, Kan83, SE94]. As the underlying computational problems (the Shortest and Closest Lattice Vector Problems) are hard [Ajt98, vEB81], the general application of this algorithm is restricted to lattices of moderate dimension (up to 50 or 60). However, some tasks like finding a couple of short vectors or finding the minimum of a lattice without symmetry may still be feasible in higher dimensions. The function `EnumerationCost` provides an estimate of the cost of running the enumeration algorithm on a specified input. This allows to know beforehand if the computation is likely to terminate within a reasonable amount of time.

For each function which enumerates short vectors of a lattice L, there is a corresponding function which enumerates vectors of L which are close to a given vector w (which usually lies outside L). Note that if one wishes to enumerate short vectors of the coset $L+w$ of the lattice L, where w is any vector of degree compatible with L, one can simply enumerate vectors $v \in L$ close to $-w$ and then just take the vectors $v + w$ for each v as the short vectors of the coset.

The enumeration routine underlying all the functions described below relies on floating-point approximations. However, it can be run in a rigorous way in some cases, see [PS08]. By default, the outputs of the functions `Minimum`, `PackingRadius`, `HermiteNumber`, `CentreDensity`, `Density`, `KissingNumber`, `ShortestVectors`, `ShortestVectorsMatrix`, `ShortVectors`, `ShortVectorsMatrix` and `ThetaSeries` are guaranteed to be correct. This correctness guarantee can be turned off by setting the optional parameter `Proof` to `false`. However, this comes virtually for free, the additional computations being most often negligible. In the present version of Magma, the outputs of all other functions are only likely to be correct.

The enumeration algorithm from [FP83, Kan83, SE94] can be interpreted as a search within a large tree. This can be extremely time-consuming. Schnorr, Euchner and Hörner [SE94, SH95] introduced techniques to prune the latter tree: the correct output might be missed, but the execution of the algorithm is likely to terminate faster. A new pruning strategy is available for the functions `Minimum`, `PackingRadius`, `HermiteNumber`,

`CentreDensity`, `Density`, `KissingNumber`, `ShortestVectors`, `ShortestVectorsMatrix`, `ShortVectors`, `ShortVectorsMatrix` and `ThetaSeries`. Naturally, if the pruning strategy is used, the result cannot be guaranteed to be correct anymore. Let $(b_1, b_2, \ldots, b_d)$ be a basis of a lattice L, let $(b_1^*, b_2^*, \ldots, b_d^*)$ denote its Gram-Schmidt orthogonalisation, and let $\mu_{i,j} = (b_i, b_j^*)/\|b_j^*\|^2$ for $i \geq j$. Suppose we are interested in finding all vectors in L of norm $\leq u$. The enumeration algorithm considers the equations:

$$\| \sum_{j=i}^{d} \left(\sum_{k=j}^{d} \mu_{k,j} x_k \right) b_j^* \|^2 \leq u, \text{ for } i = d, d-1, \ldots, 1,$$

where the x_k's are integers. If the `Prune` optional parameter is set to $[p_1, \ldots, p_d]$, then the equations above will be replaced by:

$$\| \sum_{j=i}^{d} \left(\sum_{k=j}^{d} \mu_{k,j} x_k \right) b_j^* \|^2 \leq p_i u, \text{ for } i = d, d-1, \ldots, 1.$$

The p_i's must belong to the interval $[0, 1]$. For a given input $((b_1, \ldots, b_d), u)$ to the enumeration procedure, it is possible to heuristically estimate both the running-time gain and the probability of missing a solution. See Subsection 30.8.6 for more details.

30.8.1 Minimum, Density and Kissing Number

The functions in this subsection compute invariants of a lattice which are all related to its minimum. See [JC98] for background about minimum, density and kissing numbers.

`Minimum(L)`

`Min(L)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the minimum of the lattice L, i.e., the minimal norm of a non-zero vector in the lattice. Note that this is in general a hard problem and may be very time consuming. See also the attributes section below for how to assert the minimum of a lattice.

`PackingRadius(L)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

The packing radius is half the square root of the minimum of the non-zero lattice L.

`HermiteConstant(n)`

Return the n-th Hermite constant raised to the power of n. The exact value is provided if $n \leq 8$ or $n = 24$, and otherwise an upper bound is returned. The n-th Hermite constant is defined as the maximum of `Min`(L)/`Determinant`(L) over all n-dimensional lattices L.

`HermiteNumber(L)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the Hermite number of the non-zero lattice L, i.e., $\texttt{Min}(L)/\texttt{Determinant}(L)$.

`CentreDensity(L)`
`CenterDensity(L)`
`CentreDensity(L, K)`
`CenterDensity(L, K)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

The center density of the lattice L, as an element of the real field K. This is defined to be the square root of $(\texttt{Min}(L)/4)^{\texttt{Rank(L)}}/\texttt{Determinant(L)}$. The argument for the real field K may be omitted, in which case K is taken to be the current default real field.

The product of the centre density by the volume of a sphere of radius 1 in n-dimensional space gives the density of the lattice-centered sphere packing of L, called the density of the lattice.

`Density(L)`
`Density(L, K)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

The density of the lattice L, as an element of the real field K, i.e., the density of the lattice-centered sphere packing. If the argument for the real field K is omitted, the field K is taken to be the default real field.

`KissingNumber(L)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the kissing number of the lattice L, which equals the number of vectors of minimal non-zero norm (twice the length of the sequence returned by `ShortestVectors(L)` since that returns only normalized vectors). This is the maximum number of nonoverlapping spheres of diameter `Minimum(L)` with centres at lattice points which touch a fixed sphere of the same diameter with centre at a lattice point.

Example H30E11

We create the Leech lattice Λ_{24} and compute its minimum (4), density, and kissing number (196560). Note that the computation of the minimum is very fast since the lattice is even and at least one basis element has norm 4; one has only to prove that there are no vectors of norm 2.

```
> L := Lattice("Lambda", 24);
> IsEven(L), Norm(L.2);
true 4
> time Minimum(L);
4
Time: 0.020
> Density(L);
0.0019295743094039230479033455 6369
> time KissingNumber(L);
196560
Time: 0.180
```

30.8.2 Shortest and Closest Vectors

`ShortestVectors(L)`

`Max`	RngIntElt	*Default :* ∞
`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the shortest non-zero vectors of the lattice L, i.e., the vectors $v \in L$ such that (v, v) is minimal, as a sorted sequence Q. The vectors are computed up to sign (so only one of v and $-v$ appears in Q) and normalized so that the first non-zero entry in each vector is positive, and Q is sorted with respect to lexicographic order. By default, all the (normalized) vectors are computed; the optional parameter `Max` allows the user to specify the maximal number of computed vectors. Note that unless the minimum of the lattice is already known, it has to be computed by this function, which may be very time consuming.

`ShortestVectorsMatrix(L)`

`Max`	RngIntElt	*Default :* ∞
`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the shortest non-zero vectors of the lattice L as the rows of a matrix. This is more efficient or convenient for some applications than forming the sequence of vectors. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space). By default, all the (normalized) vectors are computed; the optional parameter `Max` allows the user to specify the maximal number of computed vectors.

`ClosestVectors(L, w)`

`Max` RngIntElt *Default :* ∞

Return the vectors of the lattice L which are closest to the vector w, together with the minimal squared distance d. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. The closest vectors are those vectors $v \in L$ such that the squared distance $(v - w, v - w)$ between v and w (and thus the distance between v and w) is minimal; this minimal squared distance is the second return value d. The vectors are returned as a sequence Q, sorted with respect to lexicographic order. Note that the closest vectors are *not* symmetrical with respect to sign (while the shortest vectors are) so the returned closest vectors are not normalized. By default, all the closest vectors are computed; the optional parameter `Max` allows the user to specify the maximal number of computed vectors.

`ClosestVectorsMatrix(L, w)`

`Max` RngIntElt *Default :* ∞

Return the vectors of the lattice L which are closest to the vector w as a matrix, together with the squared distance d. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space). The vectors are returned as sequence Q, sorted with respect to lexicographic order. Note that the closest vectors are *not* symmetrical with respect to sign (while the shortest vectors are) so the returned closest vectors are not normalized. By default, all the closest vectors are computed; the optional parameter `Max` allows the user to specify the maximal number of computed vectors.

Example H30E12

We create the Gosset lattice $L = E_8$ and find the shortest vectors of L. There are 120 normalized vectors so the kissing number is 240, and the minimum is 2.

```
> L := Lattice("E", 8);
> S := ShortestVectors(L);
> #S;
120
> KissingNumber(L);
240
> { Norm(v): v in S };
{ 2 }
> Minimum(L);
2
```

We note that the rank of the space generated by the shortest vectors is 8 so that the successive minima of L are $[2, 2, 2, 2, 2, 2, 2, 2]$ (see the function `SuccessiveMinima` below).

```
> Rank(ShortestVectorsMatrix(L));
```

```
8
```

We next find the vectors in L which are closest to a certain vector in the Q-span of L. The vector is an actual hole of L and the square of its distance from L is $8/9$.

```
> w := RSpace(RationalField(), 8) !
>   [ -1/6, 1/6, -1/2, -1/6, 1/6, -1/2, 1/6, -1/2 ];
> C, d := ClosestVectors(L, w);
> C;
[
    (-1/2 -1/2 -1/2 -1/2  1/2 -1/2  1/2 -1/2),
    (-1/2  1/2 -1/2 -1/2 -1/2 -1/2  1/2 -1/2),
    (-1/2  1/2 -1/2 -1/2  1/2 -1/2 -1/2 -1/2),
    (-1/2  1/2 -1/2  1/2  1/2 -1/2  1/2 -1/2),
    ( 1/2  1/2 -1/2 -1/2  1/2 -1/2  1/2 -1/2),
    ( 0  0 -1  0  0 -1  0  0),
    ( 0  0 -1  0  0  0  0 -1),
    ( 0  0  0  0  0 -1  0 -1),
    (0 0 0 0 0 0 0 0)
]
> d;
8/9
> { Norm(v): v in C };
{ 0, 2 }
```

We verify that the squared distance of the vectors in C from w is $8/9$.

```
> { Norm(v - w): v in C };
{ 8/9 }
```

We finally notice that these closest vectors are in fact amongst the shortest vectors of the lattice (together with the zero vector).

```
> Set(C) subset (Set(S) join {-v: v in S} join { L!0 });
true
```

30.8.3 Short and Close Vectors

`ShortVectors(L, u)`

`ShortVectors(L, l, u)`

`Max`	RngIntElt	*Default :* ∞
`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Return the vectors of the lattice L with norm within the prescribed range, together with their norms, as a sorted sequence Q. The sequence Q contains tuples of the form $< v, r >$ where v is a vector and r its norm. Either one positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the

range $[l, u]$. The vectors are computed up to sign (so only one of v and $-v$ appears in Q) and normalized so that the first non-zero entry in each vector is positive, and Q is sorted with respect to first the norms and then the lexicographic order for the vectors. By default, all the (normalized) vectors with norm in the prescribed range are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors.

`ShortVectorsMatrix(L, u)`

`ShortVectorsMatrix(L, l, u)`

`Max`	RngIntElt	*Default :* ∞
`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

This is very similar to `ShortVectors`, but returns the vectors of the lattice L with norm within the prescribed range as the rows of a matrix S rather than as a sequence of tuples. This is more efficient or convenient for some applications than forming the sequence. By default, all the (normalized) vectors with norm in the prescribed range are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space).

`CloseVectors(L, w, u)`

`CloseVectors(L, w, l, u)`

`Max`	RngIntElt	*Default :* ∞

Return the vectors of the lattice L whose squared distance from the vector w is within the prescribed range, together with their squared distances, as a sorted sequence Q. The returned sequence Q contains tuples of the form $< v, d >$ where v is a vector from L and $d = (v - w, v - w)$ is its squared distance from w. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. Either one positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. Note that close vectors are *not* symmetrical with respect to sign (while short vectors are) so the returned close vectors are not normalized. Q is sorted with respect to first the squared distances and then the lexicographic order for the vectors. By default, all the vectors with squared distance in the prescribed range are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors.

`CloseVectorsMatrix(L, w, u)`

`CloseVectorsMatrix(L, w, l, u)`

`Max` RngIntElt *Default :* ∞

This is very similar to `CloseVectors`, but returns the vectors of the lattice L whose squared distance from the vector w is within the prescribed range as the rows of a matrix C rather than as a sequence of tuples. This is more efficient or convenient for some applications than forming the sequence. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. By default, all the close vectors are computed. The optional parameter `Max` allows the user to specify the maximal number of computed vectors. Note that the matrix will lie in the appropriate matrix space and the inner product for L will not be related to the rows of the matrix (which will lie in the appropriate R-space).

Example H30E13

Let $Q = [a_1, \ldots, a_n]$ be a sequence of (not necessarily distinct) positive integers and let s be a positive integer. We wish to find all solutions to the equation $\sum_{i=1}^{n} x_i a_i = s$ with $x_i \in \{0, 1\}$. This is known as the *Knapsack* problem. The following lattice-based solution is due to Schnorr and Euchner (op. cit., at the beginning of this chapter). To solve the problem, we create the lattice L of rank $n + 1$ and degree $n + 2$ with the following basis:

$$\begin{aligned} b_1 &= (2, 0, \ldots, 0, na_1, 0) \\ b_2 &= (0, 2, \ldots, 0, na_2, 0) \\ &\vdots \\ b_n &= (0, 0, \ldots, 2, na_n, 0) \\ b_{n+1} &= (1, 1, \ldots, 1, ns, 1). \end{aligned}$$

Then every vector $v = (v_1, \ldots, v_{n+2}) \in L$ such that the norm of v is $n + 1$ and

$$v_1, \ldots, v_n, v_{n+2} \in \{\pm 1\}, v_{n+1} = 0,$$

yields the solution $x_i = |v_i - v_{n+2}|/2$ for $i = 1, \ldots, n$ to the original equation.

We first write a function `KnapsackLattice` which, given the sequence Q and sum s, creates a matrix X representing the above basis and returns the lattice generated by the rows of X. Note that the `Lattice` creation function will automatically LLL-reduce the matrix X as it creates the lattice.

```
> function KnapsackLattice(Q, s)
>     n := #Q;
>     X := RMatrixSpace(IntegerRing(), n + 1, n + 2) ! 0;
>     for i := 1 to n do
>         X[i][i] := 2;
>         X[i][n + 1] := n * Q[i];
>         X[n + 1][i] := 1;
```

```
>     end for;
>     X[n + 1][n + 1] := n * s;
>     X[n + 1][n + 2] := 1;
>     return Lattice(X);
> end function;
```

We next write a function `Solutions` which uses the function `ShortVectors` to enumerate all vectors of the lattice L having norm exactly $n + 1$ and thus to find all solutions to the Knapsack problem associated with L. (Note that the minimum of the lattice may be less than $n + 1$.) The function returns each solution as a sequence of indices for Q.

```
> function KnapsackSolutions(L)
>     n := Rank(L) - 1;
>     M := n + 1;
>     S := ShortVectors(L, M, M);
>     return [
>         [i: i in [1 .. n] | v[i] ne v[n + 2]]: t in S |
>             forall{i: i in [1 .. n] cat [n + 2] | Abs(v[i]) eq 1} and
>                 v[n + 1] eq 0 where v is t[1]
>     ];
> end function;
```

We now apply our functions to a sequence Q of 12 integers each less than 1000 and the sum 2676. There are actually 4 solutions. We verify that each gives the original sum.

```
> Q := [ 52, 218, 755, 221, 574, 593, 172, 771, 183, 810, 437, 137 ];
> s := 2676;
> L := KnapsackLattice(Q, s);
> L;
Lattice of rank 13 and degree 14
Determinant: 1846735827632128
Basis:
( 0  0  0  0 -2  0  0  0  0  0  2  2  0  0)
( 1  1 -1 -1 -1  1 -1 -1 -1  1  1  1  0  1)
( 1 -1 -1 -1 -1  1  1 -1  1  1  1 -1  0  1)
( 1  1  1  1 -3  1  1 -1 -1  1 -1 -1  0  1)
( 1 -1 -3  1  1 -1 -1  1 -1  1  1  1  0  1)
( 2  2  0  0  0 -2  2  2 -2  0 -2  0  0  0)
( 3 -1  1  1 -1  1 -1 -1 -1 -1  1  1  0  1)
( 1 -1  1  1  1  1  3 -1 -1 -1 -1  1  0 -1)
(-1 -1 -1 -1  1  1  1  1 -1 -1 -1  1  0  1)
( 2  0  0  2  0  2  0  0  0  0 -2  0  0 -2)
(-1 -1 -1  1  1 -1 -1  1 -1  1  1 -3  0 -1)
(-1 -1  1 -1 -1  1 -1 -1 -1  3 -1  1  0 -3)
( 0 -2  0  0  2  0 -2  0 -2  0  0  0 12  0)
> S := KnapsackSolutions(L);
> S;
[
    [ 2, 3, 4, 5, 8, 12 ],
    [ 3, 4, 5, 7, 8, 9 ],
```

```
    [ 3, 4, 7, 8, 9, 11, 12 ],
    [ 1, 2, 3, 4, 9, 10, 11 ]
]
> [&+[Q[i]: i in s]: s in S];
[ 2676, 2676, 2676, 2676 ]
```

Finally, we apply our method to a larger example. We let Q be a sequence consisting of 50 random integers in the range $[1, 2^{1000}]$. We let I be a random subset of $\{1 \dots 50\}$ and let s be the sum of the elements of Q indexed by I. We then solve the Knapsack problem with input (Q, s) and this time obtain I as the only answer.

```
> b := 1000;
> n := 50;
> SetSeed(1);
> Q := [Random(1, 2^b): i in [1 .. n]];
> I := { };
> while #I lt n div 2 do
>     Include(~I, Random(1, n));
> end while;
> I := Sort(Setseq(I)); I;
[ 1, 3, 4, 7, 10, 11, 13, 14, 18, 20, 22, 23, 26, 28, 29, 34, 35, 37, 40, 41,
42, 45, 48, 49, 50 ]
> s := &+[Q[i]: i in I]; Ilog2(s);
1003
> time L := KnapsackLattice(Q, s);
Time: 0.570
> [Ilog2(Norm(b)): b in Basis(L)];
[ 5, 46, 46, 45, 47, 47, 46, 46, 46, 46, 46, 46, 46, 45, 47, 46, 46, 46, 46,
47,46, 46, 45, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 46, 45, 46, 46,
45, 45, 46, 46, 46, 46, 46, 45, 46, 46, 46, 46, 46 ]
> time KnapsackSolutions(L);
[
    [ 1, 3, 4, 7, 10, 11, 13, 14, 18, 20, 22, 23, 26, 28, 29, 34, 35, 37, 40,
    41, 42, 45, 48, 49, 50 ]
]
Time: 0.040
```

Example H30E14

In this example we demonstrate how short vectors of lattices can be used to split homogeneous components of integral representations. We first define an integral matrix group of degree 8. The group is isomorphic to A_5 and the representation contains the 4-dimensional irreducible representation of A_5 with multiplicity 2.

```
> G := MatrixGroup< 8, Integers() |
>   [ -673,  -291,  -225,  -316,   250,   -32,    70,  -100,
>      252,   274,   349,   272,  -156,   -94,  -296,   218,
>     2532,  1159,   609,  5164, -1450,  -181,   188,   742,
>     -551,   163,   629, -1763,   285,  -162,  -873,   219,
```

```
>    -2701,  -492,   411, -3182,  1062,  -397, -1195,   151,
>    -5018, -1112,  1044,-12958,  2898,  -153, -2870,  -454,
>     2581,    90, -1490,  8197, -1553,   261,  2556,  -149,
>    -3495, -2776, -3218, -2776,  1773,   652,  2459, -1910],
>   [ 2615,  1314,  1633,   400,  -950, -1000, -2480,  1049,
>      161,   159,   347,  -657,     2,  -385,  -889,   230,
>    -2445, -1062, -1147,   269,   744,  1075,  2441,  -795,
>     1591,   925,  1454, -1851,  -350, -1525, -3498,   982,
>    10655,  5587,  7476, -1751, -3514, -5575,-13389,  4873,
>     6271,  3253,  4653, -6126, -1390, -5274,-11904,  3219,
>    -3058, -1749, -2860,  5627,   392,  3730,  8322, -2009,
>     4875,  1851,  1170,  5989, -2239,   625,  1031,   692] >;
> Order(G);
60
```

Since the group is small enough we can generate elements in the endomorphism ring by averaging over the group elements.

```
> M := MatrixRing(Integers(), 8);
> e := [ &+[ M!g * MatrixUnit(M, i, i) * M!(g^-1) : g in G ] : i in [1..4] ];
> E := sub<M | e>;
> Dimension(E);
4
```

We now transform E into a lattice of dimension 4 and degree 64 and rescale the basis vectors so that they have lengths of the same order of magnitude.

```
> L := Lattice(64, &cat[ Eltseq(b) : b in Basis(E) ]);
> LL := sub<L | [ Round( Norm(L.4)/Norm(L.i) ) * L.i : i in [1..4] ]>;
> Minimum(LL);
910870284600
> SV := ShortVectors(LL, 100*Minimum(LL));
> #SV;
46
> Sing := [ X : v in SV | Determinant(X) eq 0 where X is M!Eltseq(v[1]) ];
> #Sing;
3
```

We thus have found three singular elements amongst the 46 shortest vectors and use the kernel of the first of these to get the representation on a subspace of dimension 4.

```
> ker := LLL( KernelMatrix(Sing[1]) );
> ker;
[ 10   0  -9   5  -2   0   5  -4]
[ -3  -4   1 -16   3   5   4  -3]
[ -8 -11 -10  -1   4   0   6  -5]
[-13   3   4  10   1   5   8   2]
> H := MatrixGroup<4, Integers() | [Solution(ker, ker*g) : g in Generators(G)]>;
> H;
MatrixGroup(4, Integer Ring)
Generators:
```

```
    [ 1  0  0  0]
    [-3 -2 -4 -5]
    [-2 -3 -3 -4]
    [ 2  3  4  5]
    [-1 -1 -1 -1]
    [ 2  1  2  2]
    [-4 -2 -1 -2]
    [ 3  2  1  2]
> #H;
60
```

30.8.4 Short and Close Vector Processes

`ShortVectorsProcess(L, u)`

`ShortVectorsProcess(L, l, u)`

Given a lattice L and a range, create a corresponding short vectors lattice enumeration process P. This process provides the environment for enumerating each vector $v \in L$ with norm within the range. Either a positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. Successive calls to `NextVector` (see below) will result in the enumeration of the vectors.

`CloseVectorsProcess(L, w, u)`

`CloseVectorsProcess(L, w, l, u)`

Given a lattice L, a vector w and a range, create a corresponding close vectors lattice enumeration process P. This process provides the environment for enumerating each vector $v \in L$ such its squared distance $(v - w, v - w)$ from w is within the range. w may be any element of a lattice of degree n or an R-space of degree n compatible with L, where n is the degree of L. Either a positive number u can be given, specifying the range $(0, u]$, or a pair l, u of positive numbers, specifying the range $[l, u]$. Successive calls to `NextVector` (see below) will result in the enumeration of the vectors.

`NextVector(P)`

Given a lattice enumeration process P as created by `ShortVectorsProcess` or `CloseVectorsProcess`, return the next element found in the enumeration.

If the process is for short vectors, the next short vector of the specified region of the lattice is returned together with its norm, or the zero vector together with -1, indicating that the enumeration process has been completed. The vectors are computed up to sign (so that only one of v and $-v$ will be enumerated) and normalized so that the first non-zero entry in each vector is positive.

If the process is for the vectors close to w, the next close vector v whose squared distance $d = (v - w, v - w)$ from w is in the specified range is returned together with

d, or the zero vector together with -1, indicating that the enumeration process has been completed. The close vectors are *not* symmetrical with respect to sign (while short vectors are) so the returned close vectors are not normalized. Note also that to test for completion the second return value should be tested for equality with -1 (or, preferably, the next function `IsEmpty` should be used) since the zero vector could be a valid close vector.

Note that the order of the vectors returned by this function in each case is arbitrary, unlike the previous functions where the resulting sequence or matrix is sorted.

`IsEmpty(P)`

Given a lattice enumeration process P, return whether the process P has found all short or close vectors.

30.8.5 Successive Minima and Theta Series

`SuccessiveMinima(L)`

`SuccessiveMinima(L, k)`

Return the first k successive minima of lattice L, or all of the m successive minima of L if k is omitted, where m is the rank of L. The first k successive minima $M_1, \ldots, M_k$ of a lattice L are defined by the property that $M_1, \ldots, M_k$ are minimal such that there exist linearly independent vectors $l_1, \ldots, l_k$ in L with $(l_i, l_i) = M_i$ for $1 \leq i \leq k$. The function returns a sequence containing the minima M_i and a sequence containing the vectors l_i. The lattice L must be an exact lattice (over $\mathbf{Z}$ or $\mathbf{Q}$). Note that the minima are unique but the vectors are not.

`ThetaSeries(L, n)`

`Proof`	BoolElt	*Default :* `true`
`Prune`	SeqEnum	*Default :* $[1.0, \ldots, 1.0]$

Given an integral lattice L and a small positive integer n, return the Theta series $\Theta_L(q)$ of L as a formal power series in q to precision n (i.e., up to and including the coefficient of q^n). The coefficient of q^k in $\Theta_L(q)$ is defined to be the number of vectors of norm k in L. Note that this function needs to enumerate all vectors of L having norm up to and including n, so its application is restricted to lattices where the number of these vectors is reasonably small. The lattice L must be an exact lattice (over $\mathbf{Z}$ or $\mathbf{Q}$). Note that the angle bracket notation should be used to assign a name to the indeterminate for the returned Theta series (e.g., `T<q> := ThetaSeries(L);`).

Example H30E15

We show how a lattice enumeration process can be used to compute the Theta series of a lattice. We write a simple function `Theta` which takes a lattice L and precision n and returns the Theta series of L up to the term q^n just as in the function `ThetaSeries`. The function assumes that the

lattice L is integral. We simply loop over the non-zero vectors of norm up to n and count the number of vectors for each norm.

```
> function Theta(L, n)
>     Z := IntegerRing();
>     P := ShortVectorsProcess(L, n);
>     C := [1] cat [0: i in [1 .. n]];
>     while not IsEmpty(P) do
>         v, norm := NextVector(P);
>         C[Z!norm + 1] +:= 2;
>     end while;
>     return PowerSeriesRing(IntegerRing()) ! C;
> end function;
```

We now compute the Theta series up to norm 10 of the Gosset lattice E_8 using the function `Theta`. We compare this MAGMA-language version with the builtin function `ThetaSeries` (which is much faster of course because of the lack of interpreter overhead etc.).

```
> L := Lattice("E", 8);
> time T<q> := Theta(L, 10);
Time: 0.050
> T;
1 + 240*q^2 + 2160*q^4 + 6720*q^6 + 17520*q^8 + 30240*q^10
> time TT<r> := ThetaSeries(L, 10);
Time: 0.000
> TT;
1 + 240*q^2 + 2160*q^4 + 6720*q^6 + 17520*q^8 + 30240*q^10 + O(q^11)
```

`ThetaSeriesIntegral(L, n)`

`Proof`	BOOLELT	*Default :* `true`
`Prune`	SEQENUM	*Default :* $[1.0, \ldots, 1.0]$

Restriction of `ThetaSeries` to integral lattices.

30.8.6 Lattice Enumeration Utilities

`SetVerbose("Enum", v)`

(Procedure.) Set the verbose printing level for the lattice enumeration algorithm to be v. Currently the legal values for v are `true`, `false`, 0, and 1 (`false` is the same as 0, and `true` is the same as 1). When the verbose level is non-zero, some information about the current status of the enumeration algorithm is printed out every 15 seconds. This concerns the functions `Minimum`, `CentreDensity`, `CenterDensity`, `Density`, `KissingNumber`, `ShortestVectors`, `ShortestVectorsMatrix`, `ShortVectors`, `ShortVectorsMatrix` and `ThetaSeries`.

`EnumerationCost(L)`

`EnumerationCost(L, u)`

`Prune` SeqEnum *Default :* $[1.0, \ldots, 1.0]$

Estimate the number of nodes in the tree to be visited during the execution of the enumeration algorithm, for the lattice L and an hyperball of squared radius u. If u is not provided, an upper bound to the lattice minimum is used. Since the cost per tree node is relatively constant, this function can be used to obtain a heuristic estimate of the running-time of the enumeration algorithm. Note that in some cases the enumeration may run faster than expected, because the tree to be visited may be shrunk during the execution, as described in [SE94]. Contrary to the enumeration itself, this function runs in time polynomial in the input bit-size (if the value of the `Prune` optional parameter is the default one). It relies on the Gaussian heuristic, which consists in estimating the number of integral points within an n-dimensional body by its volume (see [HS07] for more details).

If the `Prune` parameter is set to $[p_1, \ldots, p_d]$, then `EnumerationCost` estimates the cost of the tree pruned with the strategy described in the introduction of the present section, with pruning coefficients $p_1, \ldots, p_d$. Although the enumeration itself is likely to terminate faster, the estimation of the cost may be significantly more time-consuming.

`EnumerationCostArray(L)`

`EnumerationCostArray(L, u)`

`Prune` SeqEnum *Default :* $[1.0, \ldots, 1.0]$

Estimate the number of nodes in each layer of the tree to be visited during the execution of the enumeration algorithm, for the lattice L and an hyperball of squared radius u. If u is not provided, an upper bound to the lattice minimum is used.

Example H30E16

The algorithm that enumerates short lattice vectors may be very time-consuming. The `EnumerationCost` and `EnumerationCostArray` functions allow one to estimate the running-time on a given enumeration instantiation, before actually running the algorithm. If the running-time estimate is too high, then it is unlikely that the execution will terminate within a reasonable amount of time. A good strategy then consists in reducing the input lattice basis further. If the running-time estimate remains too high, then pruning the enumeration tree should be considered. Both the running-time gain and the "probability" of missing a solution can be estimated.

```
> B:=RMatrixSpace(IntegerRing(), 50, 51) ! 0;
> for i := 1 to 50 do B[i][1] := RandomBits(1000); end for;
> for i := 1 to 50 do B[i][i+1] := 1; end for;
> B := LLL(B);
> EnumerationCost (Lattice(B));
4.06E18
```

The value above is an estimate of the number of nodes in the enumeration tree that corresponds to the computation of the shortest non-zero vectors in the lattice spanned by the rows of B. This

is much too high to have a chance to terminate. Reducing B further allows us to compute these shortest non-zero vectors.

```
> B:=LLL(B:Delta:=0.999);
> EnumerationCost(Lattice(B));
1.03E13
> B:=LLL(B:Delta:=0.999, DeepInsertions);
> EnumerationCost(Lattice(B));
7.43E7
> time _:=ShortestVectors(Lattice(B));
3.660
```

In larger dimensions, reducing the lattice further may not prove sufficient to make the enumeration reasonably tractable. Then pruning the enumeration tree can be considered.

```
>  B:=RMatrixSpace(IntegerRing(), 65, 66) ! 0;
>  for i := 1 to 65 do B[i][1] := RandomBits(1000); end for;
>  for i := 1 to 65 do B[i][i+1] := 1; end for;
>  B := LLL(B:Delta:=0.999);
>  B := LLL(B:Delta:=0.999, DeepInsertions);
>  EnumerationCost(Lattice(B));
2.77E12
> p:=[1.0: i in [1..65]];
> for i:=10 to 55 do p[i] := (100-i)/90.; end for;
> for i:=56 to 65 do p[i]:=0.5; end for;
> EnumerationCost(Lattice(B):Prune:=p);
1.75E10
> time _:=ShortestVectors(Lattice(B):Prune:=p);
Time: 422.120
```

Assuming that the number of visited tree nodes per time unit remains roughly constant, the execution would have taken around 18 hours without pruning. It is possible to estimate the likeliness of not missing the optimal solution, by looking at the first coefficient returned by `EnumerationCostArray`. Assuming the Gaussian heuristic, this is the ratio between the first coefficients of `EnumerationCostArray` with and without the pruning table.

```
>  t1:= EnumerationCostArray(Lattice(B):Prune:=p)[1];
>  t2:= EnumerationCostArray(Lattice(B))[1];
>  t1/t2;
0.992
```

Experimentally, it seems that pruning coefficients that decrease linearly (except for the first and last indices) provide good trade-offs between efficiency gain and success likeliness.

30.9 Theta Series as Modular Forms

The theta series of an integral lattice L is (the q-expansion of) a modular form whose weight is half the dimension of the lattice, and whose level and nebentypus are determined by the quotient $L^{\#}/L$. The space of forms with given weight, level and character is finite dimensional, which means the theta series is uniquely characterised as an element of that space from knowledge of finitely many of its coefficients.

The routine described below carries this out explicitly: given an integral lattice, it returns an element of a MAGMA space of modular forms, Moreover this is done with the least possible effort spent determining coefficients via lattice enumeration. Several ideas are used here, the most important of which is that linear constraints can be obtained from knowing coefficients of the theta series of L *and* of its partial dual lattices; in practice, one needs a certain number of coefficients in total, and the `EnumerationCost` functionality is useful for balancing how many to compute for each of the duals.

We say L is *q-modular* if it is isomorphic to its qth partial dual L_q. Knowledge that modularities exist is of some use in the algorithm, because q-modularity clearly implies that L and L_q have the same theta series.

Normalisation of theta series: We use a normalisation that is most natural in this context, in order that the theta series is a modular form on $\Gamma_0(N)$ with the level N as small as possible. In this section, the theta series of an even integral lattice L will mean $\Sigma_{v\in L} q^{1/2|v|^2}$, while the theta series of an odd integral lattice L will mean $\Sigma_{v\in L} q^{|v|^2}$, where $q = e^{2i\pi z}$.

`ThetaSeriesModularFormSpace(L)`

Given an integral lattice L, this returns the space of modular forms which contains the theta series of L. Note: the theta series is normalised as described above.

`ThetaSeriesModularForm(L)`

`KnownTheta`	RNGSERPOWELT	*Default :*
`KnownDualThetas`	SEQENUM[TUP]	*Default :*
`KnownModularities`	SET[RNGINTELT]	*Default :*
`ComputeModularities`	SET[RNGINTELT]	*Default :*

Given an integral lattice L, this returns the theta series of L as a modular form in the appropriate space: more precisely, in `ThetaSeriesModularFormSpace(L)`. Note: the theta series is normalised as described above.

If some coefficients of the theta series of L are already known, this information may be specified by setting the optional argument `KnownTheta` to be a power series f, indicating that L has theta series equal to f up to the precision of f.

More generally, if coefficients are known for any partial dual lattices, these may be specified by setting `KnownDualThetas`. This argument must be a sequence of tuples of the form $< q, f_q >$, indicating that the qth partial dual L_q has theta series equal to f_q up to the precision of f_q.

If L is known to possess modularities, the optional argument `KnownModularities` may be set equal to any set of integers q such that L is q-modular (as defined above).

In addition, `ComputeModularities` may be specified to control whether the function checks for q-modularities (for possible q, that are not listed as `KnownModularities`); its value may be a boolean, or a set of integers.

30.10 Voronoi Cells, Holes and Covering Radius

The functions in this section compute the Voronoi cell of a lattice around the origin and associated information. Note that the computation of the Voronoi cell is of truly exponential complexity and is therefore the use of these functions is restricted to small dimensions (up to about 10). See [JC98] for the relevant definitions.

A lattice to which any of these functions are applied must be an exact lattice (over **Z** or **Q**).

`VoronoiCell(L)`

Return the Voronoi cell of the lattice L around the origin which is the convex polytope consisting of all the points closer to the origin than to any other lattice point of L. This function returns three values:

(a) A sequence V of vectors which are the vertices of the Voronoi cell.

(b) A set E of pairs, where each pair $\{i, j\}$ represents an edge connecting $V[i]$ and $V[j]$.

(c) A sequence P of vectors defining the relevant hyperplanes. A vector p corresponds to the hyperplane given by $(x, p) = \mathrm{Norm}(p)/2$.

`VoronoiGraph(L)`

Return a graph having the vertices and edges of the Voronoi cell of the lattice L as vertices and edges, respectively.

`Holes(L)`

Return a sequence of vectors which are the holes of the lattice L. The holes are defined to be the vertices of the Voronoi cell around the origin. Note that this involves computing the Voronoi cell of L around the origin.

`DeepHoles(L)`

Return a sequence of vectors which are the deep holes of the lattice L. The deep holes are defined to be the holes of maximum norm and are points of maximum distance to all lattice points. Note that this involves computing the Voronoi cell of L around the origin.

`CoveringRadius(L)`

Return the squared covering radius of the lattice L, which is the norm of the deep holes of L. Note that this involves computing the Voronoi cell of L around the origin.

`VoronoiRelevantVectors(L)`

Return the Voronoi relevant hyperplanes (as a set of vectors) of the Voronoi cell of L around the origin. Note that this is the same as the third return value of the `VoronoiCell` intrinsic. However, it is usually much faster since it does not compute the Voronoi cell of L. The algorithm employed is [AEVZ02, Section C].

Example H30E17

We compute the Voronoi cell of a perfect lattice of dimension 6.

```
> L := LatticeWithGram(6, [4, 1,4, 2,2,4, 2,2,1,4, 2,2,1,1,4, 2,2,2,2,2,4]);
> L;
Standard Lattice of rank 6 and degree 6
Inner Product Matrix:
[4 1 2 2 2 2]
[1 4 2 2 2 2]
[2 2 4 1 1 2]
[2 2 1 4 1 2]
[2 2 1 1 4 2]
[2 2 2 2 2 4]
> time V, E, P := VoronoiCell(L);
Time: 1.740
> #Holes(L), #DeepHoles(L), CoveringRadius(L);
782 28 5/2
```

The Voronoi cell has 782 vertices, but only 28 of these are of maximal norm 5/2 and therefore deep holes. We now compute the norms and cardinalities for the shallow holes.

```
> M := MatrixRing(Rationals(), 6) ! InnerProductMatrix(L);
> N := [ (v*M, v) : v in V ];
> norms := Sort(Setseq(Set(N))); norms;
[ 17/9, 2, 37/18, 20/9, 7/3, 5/2 ]
> card := [ #[ x : x in N | x eq n ] : n in norms ]; card;
[ 126, 16, 288, 180, 144, 28 ]
```

So there are 126 holes of norm 17/9, 16 holes of norm 2, etc. We now investigate the Voronoi cell as a polyhedron.

```
> #V, #E, #P;
782 4074 104
> { Norm(L!p) : p in P };
{ 4, 6 }
> #ShortVectors(L, 6);
52
```

The polyhedron which is the convex closure of the holes has 782 vertices, 4074 edges and 104 faces. The faces are defined by vectors of length up to 6 and all such vectors are relevant (since there are only 104). We finally look at the graph defined by the vertices and edges of the Voronoi cell.

```
> G := VoronoiGraph(L);
```

```
> IsConnected(G);
true
> Diameter(G);
8
> Maxdeg(G);
20 ( -1   0 1/2 1/2 1/2   0)
> v := RSpace(Rationals(), 6) ! [ -1, 0, 1/2, 1/2, 1/2, 0 ]; (v*M, v);
5/2
```

The graph is (of course) connected, its diameter is 8 and the vertices of maximal degree 20 are exactly the deep holes.

30.11 Orthogonalization

The functions in this section perform orthogonalization and orthonormalization of lattice bases over the field of fractions of the base ring. Note that this yields a basis orthogonalization of the space in which a lattice embeds; in contrast `OrthogonalDecomposition` returns a decomposition into orthogonal components over the base ring. Basis orthogonalization is equivalent to diagonalization of the inner product matrix of a space.

`Orthogonalize(M)`

Given a basis matrix M over a subring R of the real field, compute a matrix N which is row-equivalent over to M over the field of fractions K of R, but whose rows are orthogonal (i.e., NN^{tr} is a diagonal matrix). This function returns three values:

(a) An orthogonalized matrix N in row-equivalent to X over K;

(b) An invertible matrix T in the matrix ring over K whose degree is the number of rows of M such that $TM = N$;

(c) The rank of M.

`Diagonalization(F)`

`OrthogonalizeGram(F)`

Given a symmetric $n\times$n matrix F over R, where R is a subring of the real field, compute a *diagonal* matrix G such that $G = TFT^{tr}$ for some invertible matrix T over K, where K is the field of fractions of R. F need not have rank n. This function returns three values:

(a) A diagonal matrix G defined over R;

(b) An invertible $n\times$n matrix T over K such that $G = TFT^{tr}$;

(c) The rank of F.

`Orthogonalize(L)`

For a lattice L, return a new lattice having the same Gram matrix as L but embedded in an ambient space with diagonal inner product matrix.

`Orthonormalize(M, K)`
`Cholesky(M, K)`
`Orthonormalize(M)`
`Cholesky(M)`

For a symmetric, positive definite matrix M, and a real field K, return a lower triangular matrix T over K such that $M = TT^{tr}$. The algorithm must take square roots so the result is returned as a matrix over the real field K. If the real field K is omitted, K is taken to be the default real field. Note that this function takes a Gram matrix M, *not* a basis matrix as in the previous functions.

`Orthonormalize(L, K)`
`Cholesky(L, K)`
`Orthonormalize(L)`
`Cholesky(L)`

Given a lattice L with Gram matrix F, together with a real field K, return a new lattice over K which has the same Gram matrix F as L but has the standard Euclidean inner product. (This will involve taking square roots so that is why the result must be over a real field.) The argument for the real field K may be omitted, in which case K is taken to be the current default real field. This function is equivalent to the invocation `LatticeWithBasis(Orthonormalize(GramMatrix(L), K))`. It is sometimes more convenient to work with the resulting lattice since it has the standard Euclidean inner product.

Example H30E18

As an example for a lattice with non-trivial basis and inner product matrices we choose the dual lattice of the 12-dimensional Coxeter-Todd lattice. We compute the inner products of all pairs of shortest vectors and notice that this gets faster after changing to an isomorphic lattice with weighted standard Euclidean inner product.

```
> L := Dual(CoordinateLattice(Lattice("Kappa", 12)));
> SL := ShortestVectors(L);
> SL := SL cat [ -v : v in SL ]; #SL;
756
> time { (v,w) : v,w in SL };
{ -4, -2, -1, 0, 1, 2, 4 }
Time: 7.120
> M := Orthogonalize(L);
> SM := ShortestVectors(M);
> SM := SM cat [ -v : v in SM ]; #SM;
756
> time { (v,w) : v,w in SM };
{ -4, -2, -1, 0, 1, 2, 4 }
```

```
Time: 1.300
```

30.12 Testing Matrices for Definiteness

The functions in this section test matrices for positive definiteness, etc. They may applied to any symmetric matrix over a real subring (i.e., **Z**, **Q**, or a real field). Each function works by calling the function `OrthogonalizeGram` on its argument and then determining whether the resulting diagonal matrix has the appropriate form.

`IsPositiveDefinite(F)`

Given a symmetric matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, return whether F is positive definite, i.e., whether $vFv^{tr} > 0$ for all non-zero vectors $v \in \mathbf{R}^n$.

`IsPositiveSemiDefinite(F)`

Given a symmetric matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, return whether F is positive semi-definite, i.e., whether $vFv^{tr} \geq 0$ for all non-zero vectors $v \in \mathbf{R}^n$.

`IsNegativeDefinite(F)`

Given a symmetric matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, return whether F is negative definite, i.e., whether $vFv^{tr} < 0$ for all non-zero vectors $v \in \mathbf{R}^n$.

`IsNegativeSemiDefinite(F)`

Given a symmetric matrix F belonging to the matrix module $S = \mathrm{Hom}_R(M, M)$ or the matrix algebra $S = M_n(R)$, where R is a subring of the real field, return whether F is negative semi-definite, i.e., whether $vFv^{tr} \leq 0$ for all non-zero vectors $v \in \mathbf{R}^n$.

30.13 Genera and Spinor Genera

The genus of an exact lattice has a distinct type `SymGen` which holds a representative lattice, and the local data defining the genus. Each genus consists of 2^n spinor genera, for some integer n, typically 1. The spinor genera share the same type `SymGen`. Unlike the genus, the spinor genus is not determined solely by the local data of the genus, so the cached representative is necessary to define the spinor class.

Equality testing of genera is fast, since this requires only a comparison of the canonical local information. It is also possible to enumerate representatives of all equivalences classes in a genus or spinor genus. This is done by a process of exploration of the p-neighbour graph, for an appropriate prime p. The neighbouring functions can be applied to individual lattices to find p-neighbours or the closure under the p-neighbour process. Functions for computing and comparing the local p-adic equivalence classes of lattices, mediated by the type `SymGenLoc`.

30.13.1 Genus Constructions

`Genus(L)`
`Genus(G)`

Given an exact lattice L or a spinor genus G this function returns the genus of L. If given a genus the function returns G itself.

`SpinorGenus(L)`

Given an exact lattice L, returns the spinor genus of L.

`SpinorGenera(G)`

Given a genus G, returns the sequence of spinor genera. If G is a spinor genus, then this function returns the sequence consisting of G itself.

30.13.2 Invariants of Genera and Spinor Genera

`Representative(G)`

Returns a representative lattice for the genus symbol G.

`IsSpinorGenus(G)`

Returns `true` if and only if G is a spinor genus. This is the negation of `IsGenus(G)`.

`IsGenus(G)`

Returns `true` if and only if G is a genus. This is the negation of `IsSpinorGenus(G)`.

`Determinant(G)`

Returns the determinant of the genus symbol G.

`LocalGenera(G)`

Returns the sequence of p-adic genera of the genus symbol G.

`Representative(G)`

Returns a representative lattice for the genus symbol G.

`G1 eq G2`

Given two genus symbols, return `true` if and only if they represent the same genus. This computation is fast for genera, but currently for spinor genera invokes a call to `Representatives`.

`#G`

The number of isometry classes in the genus or spinor genus G.

Enumeration of isometry classes is done by an explicit call to `Representatives`, so that `#G` is an expensive computation.

`SpinorCharacters(G)`

Return the spinor characters of the genus symbol G as a sequence of Dirichlet characters whose kernels intersect exactly in the group of automorphous numbers. Consult Conway and Sloane [JC98] for precise definitions and significance of the spinor kernel and automorphous numbers.

`SpinorGenerators(G)`

Return the spinor generators of the genus symbol G as a sequence of primes which generate the group of spinor norms. The primes generate a group dual to that generated by the spinor characters.

`AutomorphousClasses(L,p)`

`AutomorphousClasses(G,p)`

A set of integer representatives of the p-adic square classes in the image of the spinor norm of the lattice L (respectively the genus symbol G).

`IsSpinorNorm(G,p)`

Returns `true` if and only if p is coprime to 2 and the determinant, and p is the norm of an element of the spinor kernel of G.

30.13.3 Invariants of p-adic Genera

`Prime(G)`

Return the prime p for which G represents the p-adic genus.

`Representative(G)`

Returns a canonical representative lattice of the p-adic genus G, with Gram matrix in Jordan form. For odd p the Jordan form is diagonalized.

`Determinant(G)`

This function returns a canonical p-adic representative of the determinant of the p-adic genus G. The determinant is well-defined only up to squares.

`Dimension(G)`

Return the dimension of the p-adic genus G.

`G1 eq G2`

Given local genus symbols $G1$ and $G2$, return `true` if and only if they have the same prime and the same canonical Jordan form.

30.13.4 Neighbour Relations and Graphs

`Neighbour(L, v, p)`

`Neighbor(L, v, p)`

Let L be an integral lattice, p a prime which does not divide Determinant(L) and v a vector in $L \setminus pL$ with $(v, v) \in p^2\mathbf{Z}$. The p-neighbour of L with respect to v is the lattice generated by L_v and $p^{-1}v$, where $L_v := \{x \in L | (x, v) \in p\mathbf{Z}\}$.

See [Kne57] for the original definition and [SP91] for a generalization of the neighbouring method.

`Neighbours(L, p)`

`Neighbors(L, p)`

For an integral lattice L and prime p, returns the sequence of p-neighbours of L.

`NeighbourClosure(L, p)`

`NeighborClosure(L, p)`

`Bound` RngIntElt *Default :* 2^{32}

For an integral lattice L and prime p, returns the sequence of lattices obtained by transitive closure of the p-neighbours of L.

Note that neighbours with respect to two vectors v_1, v_2 whose images in L/pL lie in the same projective orbit of Aut(L) on L/pL are isometric. Therefore only projective orbit representatives of the action of Aut(L) on L/pL are used. The large number of orbits restricts the complexity of this algorithm, hence the function gives an error if $p^{\mathrm{Rank}(L)}$ is greater than `Bound`, by default set to 2^{32}.

`GenusRepresentatives(L)`

`SpinorRepresentatives(L)`

`Representatives(G)`

`Bound`	RngIntElt	*Default :* 2^{32}
`Depth`	RngIntElt	*Default :*

For an exact lattice L with genus or spinor genus G, this function enumerates the isometry classes in G by constructing the p-neighbour closure (up to isometry). This construction used using an appropriate prime or primes p not dividing the determinant of L. For the genus, sufficiently many primes p are chosen to generate the full image, modulo the spinor kernel, of each character defining the spinor kernel. The parameters are exactly as for the `NeighbourClosure` function.

`AdjacencyMatrix(G,p)`

For a genus or spinor genus G, this function determines the adjacency matrix of the p-neighbour graph on the representative classes for G. The integer p must be prime, and if G is a spinor genus, then an error ensues if p is not an automorphous number for G.

Example H30E19

We construct the root lattice E_8 (the unique even unimodular lattice of dimension 8) as a 2-neighbour of the 8-dimensional standard lattice.

```
> Z8 := StandardLattice(8);
> v := Z8 ! [1,1,1,1,1,1,1,1];
> E8 := Neighbour(Z8, v, 2);
> E8;
Lattice of rank 8 and degree 8
Basis:
( 2  0  0  0  0  0  0  2)
( 2  0  0  0  0  0  0 -2)
( 1  1 -1  1  1 -1  1  1)
( 1  1 -1 -1 -1 -1 -1 -1)
( 1 -1 -1 -1 -1 -1  1 -1)
( 0  0  2  0  0  0  0  2)
( 0  0  0  0  2  0  0  2)
( 0  0  0  0  0  2  0  2)
Basis denominator: 2
```

The so-obtained lattice is in fact identical to the one returned by the standard construction.

```
> L := Lattice("E", 8);
> L;
Lattice of rank 8 and degree 8
Basis:
( 4  0  0  0  0  0  0  0)
```

```
(-2  2  0  0  0  0  0  0)
( 0 -2  2  0  0  0  0  0)
( 0  0 -2  2  0  0  0  0)
( 0  0  0 -2  2  0  0  0)
( 0  0  0  0 -2  2  0  0)
( 0  0  0  0  0 -2  2  0)
( 1  1  1  1  1  1  1  1)
Basis Denominator: 2
> E8 eq L;
true
```

Example H30E20

In this example we enumerate representatives for the genus of the Coxeter-Todd lattice, performing the major steps manually. The whole computation can be done simply by calling the `GenusRepresentatives` function but the example illustrates how the function actually works.

We use a combination of the automorphism group, isometry and neighbouring functions. The idea is that the neighbouring graph spans the full genus which therefore can be computed by successively generating neighbours and checking them for isometry with already known ones. The automorphism group comes into play, since neighbours with respect to vectors in the same projective orbit under the automorphism group are isometric.

```
> L := CoordinateLattice(Lattice("Kappa", 12));
> G := AutomorphismGroup(L);
> G2 := ChangeRing(G, GF(2));
> O := LineOrbits(G2);
> [ Norm(L!Rep(o).1) : o in O ];
[ 4, 8, 10 ]
```

Hence only the first and second orbits give rise to a 2-neighbour. To obtain an even neighbour, the second vector has to be adjusted by an element of $2 * L$ such that it has norm divisible by 8.

```
> v1 := L ! Rep(O[1]).1;
> v1 +:= 2 * Rep({ u : u in Basis(L) | (v1,u) mod 2 eq 1 });
> v2 := L ! Rep(O[2]).1;
> Norm(v1), Norm(v2);
16 8
> L1 := Neighbour(L, v1, 2);
> L2 := Neighbour(L, v2, 2);
> bool := IsIsometric(L, L1); bool;
true
> bool := IsIsometric(L, L2); bool;
false
```

So we obtain only one non-isometric even neighbour of L. To obtain the full genus we can now proceed with $L2$ in the same way, and do this with the following function `EvenGenus`. Note that this function is simply one component of the function `GenusRepresentatives`.

```
> function EvenGenus(L)
```

```
>      // Start with the lattice L
>      Lambda := [ CoordinateLattice(LLL(L)) ];
>      cand := 1;
>      while cand le #Lambda do
>          L := Lambda[cand];
>          G := ChangeRing( AutomorphismGroup(L), GF(2) );
>          // Get the projective orbits on L/2L
>          O := LineOrbits(G);
>          for o in O do
>              v := L ! Rep(o).1;
>              if Norm(v) mod 4 eq 0 then
>                  // Adjust the vector such that its norm is divisible by 8
>                  if not Norm(v) mod 8 eq 0 then
>                      v +:= 2 * Rep({ u : u in Basis(L) | (v,u) mod 2 eq 1 });
>                  end if;
>                  N := LLL(Neighbour(L, v, 2));
>                  new := true;
>                  for i in [1..#Lambda] do
>                      if IsIsometric(Lambda[i], N) then
>                          new := false;
>                          break i;
>                      end if;
>                  end for;
>                  if new then
>                      Append(~Lambda, CoordinateLattice(N));
>                  end if;
>              end if;
>          end for;
>          cand +:= 1;
>      end while;
>      return Lambda;
> end function;
>
> time Lambda := EvenGenus(L);
Time: 9.300
> #Lambda;
10
> [ Minimum(L) : L in Lambda ];
[ 4, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
> &+[ 1/#AutomorphismGroup(L) : L in Lambda ];
4649359/4213820620800
```

We see that the genus consists of 10 classes of lattices where only the Coxeter-Todd lattice has minimum 4 and get the mass of the genus as 4649359/4213820620800.

30.14 Attributes of Lattices

This section lists various attributes of lattices which can be examined and set by the user. This allows low-level control of information stored in lattices. Note that when an attribute is set, only minimal testing can be done on the value so if an incorrect value is set, unpredictable results may occur. Note also that if an attribute is not set, referring to it in an expression (using the ‘ operator) will *not* trigger the calculation of it (while intrinsic functions do); rather an error will ensue. Use the `assigned` operator to test whether an attribute is set.

`L‘Minimum`

The attribute for the minimum of a rational or integer lattice L. If the attribute `L‘Minimum` is examined, either the minimum is known so it is returned or an error results. If the attribute `L‘Minimum` is set by assignment, it must be a positive number equal to the minimum of the lattice. MAGMA will not check that this is correct since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value.

`L‘MinimumBound`

The attribute for an upper bound to the minimum of a rational or integer lattice L. If the attribute `L‘MinimumBound` is examined, either an upper bound to the minimum is known so it is returned or an error results. If the attribute `L‘MinimumBound` is set by assignment, it must be a positive number greater than or equal to the minimum of the lattice. MAGMA will not check that this is correct since that may be very time-consuming. If the attribute is already set, the new value must be the same as or smaller than the old value.

30.15 Database of Lattices

MAGMA includes a database containing most of the lattices explicitly presented in the Catalogue of Lattices maintained by Neil J.A. Sloane and Gabriele Nebe [NS01b].

Many standard lattices included in the Sloane & Nebe catalogue are not in the database as they may be obtained by applying MAGMA's standard lattice creation functions. Also omitted from the database are a small number of catalogued lattices defined over rings other than $\mathbf{Z}$ or $\mathbf{Q}$.

The information available for any given lattice in the catalogue varies considerably. A similar variety is found in the Magma database version, although some data (generally either easily computable or rarely available in the catalogue) is omitted.

Where the Magma database does retain data, it is not altered from the data in the catalogue. Thus the caveat which comes with that catalogue remains relevant: “Warning! Not all the entries have been checked!”

A second version of the Lattice Database has been made available with Version 2.16 of MAGMA. It adds a few more lattices, contains more information about automorphism groups, and adds Θ-series as attributes. Furthermore, it removes some duplicates. The user should be warned that the numbering of lattices (and naming in some cases) differs

between the two versions. The newer version, however, does not contain information about any Hermitian structure at the current time.

The entries of the database can be accessed in three ways:

(i) the i-th entry of the database can be requested;

(ii) the i-th entry of a particular dimension d can be specified;

(iii) the desired entry can be denoted by its name N. This name is specified exactly as in the catalogue, including all punctuation and whitespace. In the rare event that two or more entries share a single name, particular entries may be distinguished by supplying an integer i in addition to N, to denote the i-th entry with name N.

30.15.1 Creating the Database

`LatticeDatabase()`

This function returns a database object which contains information about the database.

30.15.2 Database Information

This section gives the functions that enable the user to find out what is in the database.

`#D`

`NumberOfLattices(D)`

Returns the number of lattices stored in the database.

`LargestDimension(D)`

Returns the largest dimension of any lattice in the database.

`NumberOfLattices(D, d)`

Returns the number of lattices of dimension d stored in the database.

`NumberOfLattices(D, N)`

Returns the number of lattices named N stored in the database.

`LatticeName(D, i)`

Return the name and dimension of the i-th entry of the database D.

`LatticeName(D, d, i)`

Return the name and dimension of the i-th entry of dimension d of the database D.

`LatticeName(D, N)`

Return the name and dimension of the first entry of the database with name N.

`LatticeName(D, N, i)`

Return the name and dimension of the i-th entry of the database with name N.

Example H30E21

We find out the names of the database entries.

```
> D := LatticeDatabase();
> NumberOfLattices(D);
700
```

The database contains 700 lattices. We get the set of all names in the database.

```
> names := {LatticeName(D,i): i in [1..#D]};
> #names;
673;
> Random(names);
S4(5):2
> NumberOfLattices(D, "S4(5):2");
1
```

There are 673 names, so 27 repeated names in the database, but only one with name "S4(5):2".

30.15.3 Accessing the Database

The following functions retrieve lattice information from the database.

`Lattice(D, i: parameters)`

`Lattice(D, d, i: parameters)`

`Lattice(D, N: parameters)`

`Lattice(D, N, i: parameters)`

`TrustAutomorphismGroup`

BOOL *Default :* `true`

Returns the i-th entry (of dimension d or name N) from the database D as a lattice L.

If the `TrustAutomorphismGroup` parameter is assigned `false`, then any data which claims to be the automorphism group will not be stored in L.

`LatticeData(D, i)`

`LatticeData(D, d, i)`

`LatticeData(D, N)`

`LatticeData(D, N, i)`

Returns a record which contains all the information about the i-th lattice stored in the database D (of dimension d or name N). The automorphism group is returned separately from the lattice and not stored in it.

Example H30E22

We look up a lattice in the database. There are 19 lattices of dimension 6 in the database. We get the 10th.

```
> D := LatticeDatabase();
> NumberOfLattices(D, 6);
19
> L := Lattice(D, 6, 10);
> L;
Standard Lattice of rank 6 and degree 6
Minimum: 4
Inner Product Matrix:
[4 1 2 2 2 2]
[1 4 2 2 2 2]
[2 2 4 1 2 2]
[2 2 1 4 2 2]
[2 2 2 2 4 1]
[2 2 2 2 1 4]
```

There may be more information stored than just what is returned by the `Lattice` function. We get the record containing all the stored lattice data.

```
> R := LatticeData(D, 6, 10);
> Format(R);
recformat<name, dim, lattice, minimum, kissing_number,
is_integral, is_even, is_unimodular, is_unimodular_hermitian,
modularity, group_names, group, group_order,
hermitian_group_names, hermitian_group, hermitian_group_order,
hermitian_structure>
```

This lists all possible fields in the record. They may or may not be assigned for any particular lattice.

```
> R`lattice eq L;
true
> R`name;
A6,1
> assigned R`kissing_number;
true
> R`kissing_number;
42
> assigned R`group;
false
> A := AutomorphismGroup(L);
> A : Minimal;
MatrixGroup(6, Integer Ring) of order 96 = 2^5 * 3
```

The result of the `Lattice` call is equal to the `lattice` field of the data record. The kissing number was stored, but the automorphism group wasn't. We computed the group (as a matrix group over the integers) and found it has order 96.

30.15.4 Hermitian Lattices

There are a few facilities for computing with Hermitian lattices over an imaginary quadratic field or a quaternion algebra. However, these functions apply to a Gram matrix, and not a lattice *per se*. The main application is for automorphism groups that preserve a structure.

`HermitianTranspose(M)`

Given a matrix over an imaginary quadratic field or a quaternion algebra, return the conjugate transpose.

`ExpandBasis(M)`

Given a matrix over an imaginary quadratic field or a quaternion algebra, expand it to a basis over the rationals.

`HermitianAutomorphismGroup(M)`
`QuaternionicAutomorphismGroup(M)`

Given a conjugate symmetric Gram matrix, compute the automorphism group.

Various functions for matrix groups over associative algebras are available here, such as `CharacterTable` and `IsConjugate` which simply use a re-writing over the rationals, and `InvariantForms` which after using `GHom` needs to restrict to elements fixed by the quaterionic structure. Finally, there is `QuaternionicGModule` which will split a G-module over a quaternionic structure.

`InvariantForms(G)`

Given a matrix group over an associative algebra or an imaginary quadratic field, return a basis for the forms fixed by it.

`QuaternionicGModule(M, I, J)`

Given a G-module M and I and J in the endomorphism algebra that anti-commute and whose squares are scalars, write G over the quaternionic structure given by I and J.

`MooreDeterminant(M)`

Given a conjugate-symmetric matrix over a quaternion algebra, compute the Moore determinant. This is the "normal" determinant, which is well-defined here because all the diagonal elements are rational, and thus there is no ambiguity between left/right division.

Example H30E23

We construct the Coxeter-Todd lattice over $Q_{3,\infty}$ starting with the group $SU(3,3)$.

```
> G := SU(3, 3);
> chi := CharacterTable(G)[2];
> M := GModule(chi,Integers());
> E := EndomorphismAlgebra (M);
> while true do r:=&+[Random([-2..2])*E.i : i in [1..4]];
>                if r^2 eq -1 then break; end if; end while;
> while true do s:=&+[Random([-2..2])*E.i : i in [1..4]];
>                if s^2 eq -3 and r*s eq -s*r then break; end if; end while;
> MM := QuaternionicGModule(M, r, s);
> Discriminant(BaseRing(MM));
3
> MG := MatrixGroup(MM);
> IF := InvariantForms(MG); IF;
[
    [1 -1/2*i + 1/6*k 1/3*k]
    [1/2*i - 1/6*k 1 -1/3*j]
    [-1/3*k 1/3*j 1]
]
> assert IsIsomorphic(G, MG);
```

Example H30E24

We compute the quaternionic automorphism group for the Leech lattice.

```
> A<i,j,k> := QuaternionAlgebra<Rationals()|-1,-1>;
> v := [];
> v[1] := [2+2*i,0,0,0,0,0]; /* from Wilson's paper */
> v[2] := [2,2,0,0,0,0];
> v[3] := [0,2,2,0,0,0];
> v[4] := [i+j+k,1,1,1,1,1];
> v[5] := [0,0,1+k,1+j,1+j,1+k];
> v[6] := [0,1+j,1+j,1+k,0,1+k];
> V := [Vector(x) : x in v];
> W := [Vector([Conjugate(x) : x in Eltseq(v)]): v in V];
> M6 := Matrix(6,6,[(V[i],W[j])/2 : i,j in [1..6]]); /* 6-dim over A */
> time Q := QuaternionicAutomorphismGroup(M6);
> assert #Q eq 503193600;
```

The same can be done for the Coxeter-Todd lattice.

```
> A<i,j,k> := QuaternionAlgebra<Rationals()|-1,-3>;
> a := (1+i+j+k)/2;
> M3 := Matrix(3,3,[2,a,-1, Conjugate(a),2,a, -1,Conjugate(a),2]);
> time Q := QuaternionicAutomorphismGroup(M3);
```

```
> assert #Q eq 12096;
```

One can also compute the automorphism group over the Eisenstein field, using `InvariantForms` on the realisation of this group as `ShephardTodd(34)`.

```
> G := ShephardTodd(34);
> IF := InvariantForms(G); // scaled Coxeter-Todd over Q(sqrt(-3))
> A := HermitianAutomorphismGroup(IF[1]);
> assert IsIsomorphic(A,G);
```

30.16 Bibliography

[**AEVZ02**] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.

[**Ajt98**] Miklós Ajtai. The Shortest Vector Problem in L2 is NP-hard for Randomized Reductions (Extended Abstract). In *Proceedings of the 30th Symposium on the Theory of Computing (STOC 1998)*, pages 10–19. ACM, 1998.

[**Akh02**] Ali Akhavi. Random lattices, threshold phenomena and efficient reduction algorithms. *Theoretical Computer Science*, 287(2):359–385, 2002.

[**dW87**] Benne M.M. de Weger. Solving exponential Diophantine equations using lattice basis reduction algorithms. *J. Number Th.*, 26:325–367, 1987.

[**FP83**] U. Fincke and M. Pohst. A procedure for determining algebraic integers of given norm. In *EUROCAL*, volume 162 of *LNCS*, pages 194–202. Springer, 1983.

[**HPP06**] F. Hess, S. Pauli, and M. Pohst, editors. *ANTS VII*, volume 4076 of *LNCS*. Springer-Verlag, 2006.

[**HS07**] Guillaume Hanrot and Damien Stehlé. Improved Analysis of Kannan's Shortest Lattice Vector Algorithm (Extended Abstract). In *Advances in cryptology—CRYPTO 2007*, volume 4622 of *LNCS*, pages 170–186. Springer, 2007.

[**JC98**] N.J.A. Sloane J.H. Conway. *Sphere Packings, Lattices and Groups*, volume 290 of *Grundlehren der Mathematischen Wissenschaften*. Springer, New York–Berlin–Heidelberg, 3rd edition, 1998.

[**Kan83**] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the 15th Symposium on the Theory of Computing (STOC 1983)*, pages 99–108. ACM, 1983.

[**Kne57**] M. Kneser. Klassenzahlen indefiniter quadratischer Formen. *Archiv Math.*, 8:241–250, 1957.

[**LLL82**] Arjen K. Lenstra, Hendrik W. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[**MG02**] Daniele Micciancio and Shafi Goldwasser. *Complexity of lattice problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 2002.

[**NS01a**] G. Nebe and N.J.A. Sloane. The Catalogue of Lattices. URL:http://www.research.att.com/∼njas/lattices/, 2001.

[**NS01b**] Gabriele Nebe and Neil J.A. Sloane. A Catalogue of Lattices. URL:http://akpublic.research.att.com/∼njas/lattices/index.html, 2001.

[**NS06**] Phong Nguyen and Damien Stehlé. LLL on the Average. In Hess et al. [HPP06], pages 238–256.

[**NS09**] Phong Nguyen and Damien Stehlé. An LLL Algorithm with Quadratic Complexity. *SIAM Journal on Computing*, 39(3):874–903, 2009.

[**Poh87**] Michael Pohst. A Modification of the LLL Reduction Algorithm. *J. Symbolic Comp.*, 4(1):123–127, 1987.

[**Pro**] The SPACES Project. MPFR, a LGPL-library for multiple-precision floating-point computations with exact rounding. URL:http://www.mpfr.org/.

[**PS08**] Xavier Pujol and Damien Stehlé. Rigorous and efficient short lattice vectors enumeration. In *Advances in Cryptology—AsiaCrypt 2008*, LNCS. Springer, 2008.

[**SE94**] Claus-Peter Schnorr and Michael Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Mathematics of Programming*, 66:181–199, 1994.

[**SH95**] Claus-Peter Schnorr and Horst Helmut Hörner. Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction. In *Advances in Cryptology—EuroCrypt 1995*, volume 921 of *LNCS*, pages 1–12. Springer-Verlag, 1995.

[**Sho**] Victor Shoup. NTL, Number Theory C++ Library. URL:http://www.shoup.net/ntl/.

[**Sim05**] Denis Simon. Solving quadratic equations using reduced unimodular quadratic forms. *Math. Comp.*, 74(251):1531–1543 (electronic), 2005.

[**SP91**] Rainer Schulze-Pillot. An algorithm for computing genera of ternary and quaternary quadratic forms. In Stephen M. Watt, editor, *Proceedings ISSAC'91*, pages 134–143, Bonn, 1991.

[**Ste09**] Damien Stehlé. *Floating-point LLL: theoretical and practical aspects.* Springer-Verlag, 2009. To appear.

[**vEB81**] Peter van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical report 81-04, Mathematisch Instituut, Universiteit van Amsterdam, 1981.

31 LATTICES WITH GROUP ACTION

Chapter 31
LATTICES WITH GROUP ACTION

31.1 Introduction

In MAGMA, a G-lattice L is a lattice upon which a finite integral matrix group G acts by right multiplication. MAGMA allows various computations with lattices associated with finite integral matrix groups by use of G-lattices.

The computation of the automorphism group of a lattice (i.e. the largest matrix group that acts on the lattice) and the testing of lattices for isometry is performed within MAGMA by a search designed by Bill Unger, which is based on the Plesken-Souvignier backtrack algorithm [PS97], together with ordered partition methods. Optionally, this may be combined with orthogonal decomposition code of Gabi Nebe.

If G is a finite integral matrix group, then MAGMA uses Plesken's centering algorithm ([Ple74]) to construct all G-invariant sublattices of a given G-lattice L. The lattice of G-invariant sublattices of L can be explored much like the lattice of submodules over finite fields.

31.2 Automorphism Group and Isometry Testing

The functions in this section compute the automorphism group of a lattice and test lattices for isometry. Currently the lattices to which these functions are applied must be exact (over **Z** or **Q**).

`AutomorphismGroup(L)`

`Stabilizer`	RNGINTELT	*Default :* 0
`BacherDepth`	RNGINTELT	*Default :* −1
`Generators`	[GRPMATELT]	*Default :*
`NaturalAction`	BOOL	*Default :* `false`
`Decomposition`	BOOL	*Default :* `false`
`Vectors`	MTRX	*Default :*

This function computes the automorphism group G of a lattice L which is defined to be the group of those automorphisms of the **Z**-module underlying L which preserve the inner product of L. L must be an exact lattice (over **Z** or **Q**). The group G is returned as an integral matrix group of degree m acting on the *coordinate* vectors of L by multiplication where m is the rank of L. The coordinate vectors of L are the elements of the coordinate lattice C of L which has the same Gram matrix as L, but standard basis of degree m (C can be created using the function `CoordinateLattice`). G does not act on the elements of L, since there is no natural

matrix action of the automorphism group on L in the case that the rank of L is less than its degree. However, if the rank of L equals its degree, then the parameter `NaturalAction` may be set to `true`, in which case the resulting group has the natural action on the basis vectors (not the coordinate vectors); note that in this case the resulting matrix group will have (non-integral) rational entries in general, even though the image under the group of an integral basis vector will always be integral.

The algorithm uses a backtrack search to find images of the basis vectors. A vector is a possible image if it has the correct inner product with the images chosen so far. Additional invariants which have to be respected by automorphisms are used by default and are usually sufficient for satisfactory performance. For difficult examples the parameters described below allow to consider further invariants which are more sophisticated and harder to compute, but often find dead ends in the backtrack at an early stage.

The algorithm computes and stores a set of short vectors in the lattice that spans the lattice and is guaranteed closed under the action of the automorphism group. This restricts its general application, as for high dimensional lattices the number of vectors of minimal length may be too large to work with. However, the function can of course be applied to lattices in higher dimensions with a reasonable number of short vectors.

Setting the parameter `Stabilizer := i` will cause the function to compute only the point stabilizer of i basis vectors. These will in general not be the first i basis vectors, as the function chooses the basis to speed up the computation.

The parameter `Depth` is retained for compatibility with previous versions of MAGMA(which used Souvignier's AUTO program) but it is now ignored. The improved backtrack search achieved by using ordered partition methods has made the `Depth` concept unnecessary.

In some hard examples one may want to use Bacher polynomials, which are a combinatorial invariant that usually separates the orbits of the automorphism group on the short vectors. However, these are expensive to calculate and should only be used if one suspects that the automorphism group has many orbits on the short vectors. The parameter `BacherDepth` specifies to which depth the Bacher polynomials may used and should usually be chosen to be 1, since even small automorphism groups will have only very few (most likely 1) orbits on the vectors having correct scalar product with the first image. Setting `BacherDepth` to zero forbids the use of Bacher polynomial invariant. Setting it to anything else allows the algorithm to use this invariant at level 1. Bacher polynomials are computed by counting pairs of vectors having a certain scalar product with other vectors. This scalar product is by default chosen to be half the norm of the vector, since this will usually be the value which occurs least frequent.

In some situations one may already know a subgroup of the full automorphism group, either by the construction of the lattice or an earlier stabilizer computation. This subgroup can be made available by setting `Generators := Q`, where Q is a set or sequence containing the generators of the subgroup.

Since V2.13, the algorithm has had the ability to first attempt to compute an orthogonal decomposition of L by considering the sublattices spanned by the set of shortest vectors, and if there is a non-trivial decomposition the automorphism group is computed via an algorithm of G. Nebe which computes the automorphism groups for the components and combines these. This decomposition method can be invoked by setting `Decomposition` to `true`.

When decomposition is suppressed, it is possible to supply the backtrack algorithm with the set of vectors to be used as domain for the search. To do this set the parameter `Vectors` to be a matrix so that the rows of the matrix give the lattice elements to be used. (Only one of a vector and its negative should be given.) To guarantee correctness of the result, the rows of the matrix should satisfy two conditions:

The rows, together with their negations, must be closed under the action of the full automorphism group, and

The sublattice of L generated by the rows of the matrix must equal L.

These conditions are not checked by the code, and it is up to the user of this parameter to ensure the correctness of their input. For example, the function `ShortVectorsMatrix` returns a matrix which satisfies the first condition. If the first condition is not satisfied, there are no guarantees about what will happen. The second condition may be violated and still give a useful result. In particular, if the sublattice generated by the vectors given has finite index in the full lattice, then the final result will be the correct automorphism group. The problem is that the backtrack search may not be particularly efficient. This may still be better than working with a very large set of vectors satisfying the second condition.

`AutomorphismGroup(L, F)`

`Stabilizer`	RngIntElt	*Default :* 0
`BacherDepth`	RngIntElt	*Default :* 0
`Generators`	[GrpMatElt]	*Default :*
`Vectors`	Mtrx	*Default :*

This function computes the subgroup of the automorphism group of the lattice L which fixes also the forms given in the set or sequence F. The matrices in F are not required to be positive definite or even symmetric. This is for example useful to compute automorphism groups of lattices over algebraic number fields. The parameters are as above.

`AutomorphismGroup(F)`

`Stabilizer`	RngIntElt	*Default :* 0
`BacherDepth`	RngIntElt	*Default :* 0
`Generators`	[GrpMatElt]	*Default :*

This function computes the matrix group fixing all forms given as matrices in the sequence F. The first form in F must be symmetric and positive definite, while the

others are arbitrary. The call of this function is equivalent to `AutomorphismGroup( LatticeWithGram(F[1]), [ F[i] : i in [2..#F] ] )`. This function can be used to compute the Bravais group of a matrix group G which is defined to be the full automorphism group of the forms fixed by G. The parameters are as above.

Example H31E1

In this example we compute the automorphism group of the root lattice E_8 and manually transform the action on the coordinates into an action on the lattice vectors. Note that this exactly the same as using the `NaturalAction` parameter for the function `AutomorphismGroup`.

```
> L := Lattice("E", 8);
> G := AutomorphismGroup(L);
> #G; FactoredOrder(G);
696729600
[ <2, 14>, <3, 5>, <5, 2>, <7, 1> ]
> M := MatrixRing(Rationals(), 8);
> B := BasisMatrix(L);
> A := MatrixGroup<8, Rationals() | [B^-1 * M!G.i * B : i in [1 .. Ngens(G)]]>;
> A;
MatrixGroup(8, Rational Field)
Generators:
    [   0    0 -1/2  1/2 -1/2  1/2    0    0]
    [   0    0  1/2  1/2  1/2  1/2    0    0]
    [   0    0 -1/2  1/2  1/2 -1/2    0    0]
    [-1/2  1/2    0    0    0    0 -1/2  1/2]
    [   0    0 -1/2 -1/2  1/2  1/2    0    0]
    [-1/2 -1/2    0    0    0    0 -1/2 -1/2]
    [-1/2 -1/2    0    0    0    0  1/2  1/2]
    [ 1/2 -1/2    0    0    0    0 -1/2  1/2]

    [ 1/4  1/4  1/4 -1/4 -3/4 -1/4 -1/4 -1/4]
    [-1/4 -1/4  3/4  1/4 -1/4  1/4  1/4  1/4]
    [-1/4 -1/4 -1/4  1/4 -1/4  1/4  1/4 -3/4]
    [-1/4 -1/4 -1/4  1/4 -1/4  1/4 -3/4  1/4]
    [ 1/4 -3/4  1/4 -1/4  1/4 -1/4 -1/4 -1/4]
    [ 3/4 -1/4 -1/4  1/4 -1/4  1/4  1/4  1/4]
    [-1/4 -1/4 -1/4  1/4 -1/4 -3/4  1/4  1/4]
    [-1/4 -1/4 -1/4 -3/4 -1/4  1/4  1/4  1/4]

    [1 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0]
    [0 0 1 0 0 0 0 0]
    [0 0 0 1 0 0 0 0]
    [0 0 0 0 1 0 0 0]
    [0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 0 1]
    [0 0 0 0 0 0 1 0]
```

```
> [ #Orbit(A, b) : b in Basis(L) ];
[ 2160, 240, 240, 240, 240, 240, 240, 240 ]
> AutomorphismGroup(L: NaturalAction) eq A;
true
```

Example H31E2

In this example we demonstrate how stabilizers can be used to obtain a big subgroup of the full automorphism group fairly quickly.

```
> L := Lattice("Kappa", 13);
> LL, T := PairReduce(L);
> time G2 := AutomorphismGroup(L : Stabilizer := 2);
Time: 0.030
> #G2;
48
> time GG2 := AutomorphismGroup(LL : Stabilizer := 2);
Time: 0.020
> #GG2;
48
> Z := IntegerRing();
> H := MatrixGroup< 13, Z | G2, [ T^-1*g*T : g in Generators(GG2) ] >;
> #H;
311040
> MatrixRing(Z, 13) ! -1 in H;
false
```

The pair reduction yields a different basis for the same lattice and both 2-point stabilizers have order 48. After transforming the automorphisms of LL back to automorphisms of L, the two stabilizers generate a group of reasonable size which still can be enlarged by an index of 2 by adding $-I$. We thus obtain a group of order 622080 which turns out to be the full automorphism group of the lattice L.

```
> time G := AutomorphismGroup(L);
Time: 0.020
> #G;
622080
```

Example H31E3

```
> L := Lattice("Lambda", 19);
> time G := AutomorphismGroup(L);
Time: 0.300
> #G;
23592960
> DS := DerivedSeries(G);
> [ #DS[i]/#DS[i+1] : i in [1..#DS-1] ];
[ 4, 3, 4 ]
```

```
> [ IsElementaryAbelian(DS[i]/DS[i+1]) : i in [1..#DS-1] ];
[ true, true, true ]
> H := DS[#DS];
> C := Core(G, Sylow(H, 2));
> Q := H/C; #Q, IsSimple(Q);
60 true
> LS := LowerCentralSeries(C);
> [ #LS[i]/#LS[i+1] : i in [1..#LS-1] ];
[ 256, 16, 2 ]
```

Hence, $G := \mathrm{Aut}(\Lambda_{19})$ has a series of normal subgroups with factors $2^2, 3, 2^2, A_5, 2^8, 2^4, 2$.

IsIsometric(L, M)

IsIsomorphic(L, M)

BacherDepth	RngIntElt	*Default :* 0
LeftGenerators	[GrpMatElt]	*Default :*
RightGenerators	[GrpMatElt]	*Default :*
LeftVectors	Mtrx	*Default :*
RightVectors	Mtrx	*Default :*

This function determines whether the lattices L and M are isometric. The method is a backtrack search analogous to the one used to compute the automorphism group of a lattice. If the lattices are isometric, the function returns a transformation matrix T as a second return value such that $F_2 = TF_1T^{tr}$, where F_1 and F_2 are the Gram matrices of L and M, respectively.

For isometric lattices the cost of finding an isometry is roughly the cost of finding one automorphism of the lattice. Again, the computation may be sped up by using the additional invariants described for the automorphism group computation.

In many applications one will check whether a lattice is isometric to one for which the automorphism group is already known. In this situation the automorphism group of the second lattice can be made available by setting `RightGenerators := Q`, where Q is a set or sequence containing the generators of the group. Note however, that for isometric lattices this may slow down the computation, since generators for stabilizers have to be recomputed. Similarly, generators for an automorphism group of the first lattice may be supplied as `LeftGenerators`.

Corresponding to the `Vectors` parameter for automorphism group calculation, the parameters `LeftVectors` and `RightVectors` allow the user to. As above, left refers to the first lattice, right to the second. Either both of these parameters can be set, or neither, an error results if just one is set. The restrictions on what constitutes correct values for these parameters are as for the `Vectors` parameter above. For correct isometry testing, the vectors given must be such that any isometry will map the left vectors into the union of the right vectors and their negatives. These conditions are not checked in the code, they are the responsibility of the user.

IsIsometric(L, F_1, M, F_2)

IsIsomorphic(L, F_1, M, F_2)

IsIsometric(L, M)

IsIsomorphic(L, M)

BacherDepth	RngIntElt	*Default :* 0
LeftGenerators	[GrpMatElt]	*Default :*
RightGenerators	[GrpMatElt]	*Default :*
LeftVectors	Mtrx	*Default :*
RightVectors	Mtrx	*Default :*

This function determines whether the lattices L and M are isometric with an isometry respecting also additional bilinear forms given by the sequences of Gram matrices F_1 and F_2. The return values and parameters are as above.

IsIsometric(F_1, F_2)

IsIsomorphic(F_1, F_2)

BacherDepth	RngIntElt	*Default :* 0
LeftGenerators	[GrpMatElt]	*Default :*
RightGenerators	[GrpMatElt]	*Default :*
LeftVectors	Mtrx	*Default :*
RightVectors	Mtrx	*Default :*

For two sequences of F_1 and F_2 of Gram matrices, determine whether a simultaneous isometry exists, i.e., a matrix T such that $TF_1[i]T^{tr} = F_2[i]$ for i in $[1..\#F_1]$. The first form in both sequences must be positive definite. The return values and parameters are as above.

Example H31E4

We construct the 16-dimensional Barnes-Wall lattice in two different ways and show that the so-obtained lattices are isometric.

```
> L := Lattice("Lambda", 16);
> LL := Lattice(ReedMullerCode(1, 4), "B");
> time bool, T := IsIsometric(L, LL : Depth := 4);
Time: 2.029
> bool;
true
> T * GramMatrix(L) * Transpose(T) eq GramMatrix(LL);
true
```

We can also show that L is a 2-modular lattice (i.e., isometric to its rescaled dual).

```
> IsIsometric(L, Dual(L));
true
```

```
[ 0  1  1 -1  1 -1  0  0 -1  1 -1  0  0  0  0  0]
[-2 -3 -4  1 -2  3 -1 -2  0  1 -1 -1  1  1  1 -1]
[-1 -1 -1  1  0 -1  0 -1  0  2 -1 -1  1  1  1 -1]
[ 0  1  1 -1  1  0  0  0 -1  0  0  0  0  0  0  0]
[ 0 -1 -2  0 -1  2 -1 -1  0  1 -1  0  1  0  0  0]
[ 1  2  2  0  2 -3  0  0 -2  4 -2 -1  1  1 -1 -1]
[-1 -1 -2  0  0  1  0 -1  0  0  0 -1  1  1  1 -1]
[ 1  2  3 -1  2 -3  1  1 -1  0  0  0  0  0 -1  1]
[ 0  1  1  0  2 -3  0 -1 -2  4 -2 -2  1  1  1 -1]
[ 0 -1 -2  0 -2  3 -1  0  1 -1  0  1  0  0 -1  0]
[ 0  0  1  1  0 -1  1  1  1 -1  0  1  0  0 -1  0]
[ 0 -1 -1  1 -2  2 -1  0  1  0  0  1  0 -1 -1  0]
[ 0  0  0  0  0  0  0  1  1 -1  0  1  0  0 -1  0]
[ 0 -1 -2  0 -2  3  0  0  1 -2  0  1  1  0 -2  1]
[ 0  1  1 -1  1 -1  0  0 -1  1  0 -1  0  0  1  0]
[ 0  0 -1 -1  0  1  0 -1 -1  1 -1 -1  1  1  0  0]
```

31.2.1 Automorphism Group and Isometry Testing over $\mathbf{F}_q[t]$

Let q be some power of an odd prime. A bilinear form b over $\mathbf{F}_q[t]$ is said to be definte if the corresponding quadratic form is anisotropic over the completion of $\mathbf{F}_q(t)$ at the infinite place $(1/t)$.

The functions in this section compute automorphism groups and isometries of definite bilinear forms over $\mathbf{F}_q[t]$.

`DominantDiagonalForm(X)`

`Canonical`	Bool	*Default :* `false`
`ExtensionField`	FldFin	*Default :*

Let X be a symmetric $n \times n$-matrix of rank n over a polynomial ring $K[t]$ where K denotes a field of characteristic different from 2. The function returns a symmetric matrix G and some $T \in \mathrm{GL}(n, K[t])$ such that $G = TXT^{tr}$ has dominant diagonal. I.e. the degrees of the diagonal entries of G are ascending and the degree of a non-diagonal entry is less than the degrees of the corresponding diagonal entries (see [Ger03]).

If K is a finite field and X represents a definite form and `Canonical` is set to `true`, then the form G will be unique and the third return value will be the automorphism group of G i.e. the stabilizer of G in $\mathrm{GL}(n, K[t])$. The algorithm employed is [Kir12]. Note however, the uniqueness depends on some internal choices being made. Thus the fourth return value is a finite field E which must be given as the optional argument `ExtensionField` in subsequent runs over K to ensure that results are compatible (c.f. the following example). In particular, the defining polynomial and the primitive element of E are important for the uniqueness.

Example H31E5

We test whether two definite forms over $\mathbf{F}_q[t]$ are isometric.

```
> R<t> := PolynomialRing( GF(5) );
> X1:= SymmetricMatrix([ t^3, t+1, 2*t^2+2*t+2 ] );
> X2:= SymmetricMatrix([ t^3, t^4+2*t+2, t^5+2*t^2+2*t+3 ]);
> G1, T1, Aut, E:= DominantDiagonalForm(X1 : Canonical);
> T1 * X1 * Transpose(T1) eq G1;
true
> GG:= [ Matrix(g) : g in Generators(Aut) ];
> forall{g : g in GG | g * G1 * Transpose(g) eq G1 };
true
```

So the form G_1 is invariant under Aut. Now we reduce the second form X_2. To be able to compare the results, we have to provide the field E from above.

```
> G2, T2 := DominantDiagonalForm(X2 : Canonical, ExtensionField:= E);
> G1 eq G2;
true
```

Thus the two forms X_1 and X_2 are isometric and $T_1^{-1}T_2$ is an isometry.

`AutomorphismGroup(G)`

`ExtensionField` FldFin *Default :*

Computes the automorphism group of the definite bilinear form given by the symmetric matrix G over $\mathbf{F}_q[t]$.

The second return value is a finite field as explained in `DominantDiagonalForm` above. It may be supplied for later calls over the same ground field $\mathbf{F}_q$ via the optional argument `ExtensionField` to safe some time if q is large. The correctness of the algorithm does not depend on it.

`IsIsometric(G1, G2)`

`ExtensionField` FldFin *Default :*

Tests whether two definite bilinear forms over $\mathbf{F}_q[t]$ are isometric. If so, the second return value is a matrix $T \in \mathrm{GL}(n, q)$ such that $TG1T^{tr} = G2$.

The third return value is a finite field as explained in `DominantDiagonalForm` above. It may be supplied for later calls over the same ground field $\mathbf{F}_q$ via the optional argument `ExtensionField` to safe some time if q is large. The correctness of the algorithm does not depend on it.

`ShortestVectors(G)`

Returns a sequence Q which contains the shortest non-zero vectors with respect to a given definite bilinear form G over $\mathbf{F}_q[t]$ where q is odd. The sequence Q contains tuples $< v, r >$ where v is a shortest vector and r denotes its norm with respect to G.

`ShortVectors(G, B)`

Let G be a definite bilinear form of rank n over $\mathbf{F}_q[t]$ for some odd q. The function returns a sequence Q which contains all vectors in $\mathbf{F}_q[t]^n$ whose norm with respect to G is at most B. The sequence Q contains tuples $< v, r >$ where v is such a short vector and r denotes its norm with respect to G.

31.3 Lattices from Matrix Groups

In MAGMA a G-lattice L is a lattice upon which a finite integral matrix group G acts by right multiplication.

Each G-lattice L has references to both the original ("natural") group G which acts on the standard lattice in which L is embedded and also the reduced group of L which is the reduced representation of G on the basis of L.

31.3.1 Creation of G-Lattices

The following functions create G-lattices. Note that the group G must be a finite integral matrix group.

`Lattice(G)`

Given a finite integral matrix group G, return the standard G-lattice (with standard basis and rank equal to the degree of G).

`LatticeWithBasis(G, B)`

Given a finite integral matrix group G and a non-singular matrix B whose row space is invariant under G (i.e., $Bg = T_g B$ for each $g \in G$ where T_g is a unimodular integral matrix depending on g), return the G-lattice with basis matrix B. (The number of columns of B must equal the degree of G; G acts naturally on the lattice spanned by B.)

`LatticeWithBasis(G, B, M)`

Given a finite integral matrix group G, a non-singular matrix B whose row space is invariant under G (i.e., $Bg = T_g B$ for all $g \in G$ where T_g is a unimodular integral matrix depending on g) and a positive definite matrix M invariant under G (i.e., $gMg^{tr} = M$ for all $g \in G$) return the G-lattice with basis matrix B and inner product matrix M. (The number of columns of B must equal the degree of G and both the number of rows and the number of columns of M must equal the degree of G; G acts naturally on the lattice spanned by B and fixes the Gram matrix of the lattice).

`LatticeWithGram(G, F)`

Given a finite integral matrix group G and a positive definite matrix F invariant under G (i.e., $gFg^{tr} = F$ for all $g \in G$) return the G-lattice with standard basis and inner product matrix F (and thus Gram matrix F). (Both the number of rows and the number of columns of M must equal the degree of G; G fixes the Gram matrix of the returned lattice).

31.3.2 Operations on G-Lattices

The following functions provide basic operations on G-lattices.

`IsGLattice(L)`

Given a lattice L, return whether L is a G-lattice (i.e., there is a group associated with L).

`Group(L)`

Given a G-lattice L, return the matrix group of the (reduced) action of G on L. The resulting group thus acts on the coordinate lattice of L (like the automorphism group).

`NumberOfActionGenerators(L)`

`Nagens(L)`

Given a G-lattice L, return the number of generators of G.

`ActionGenerator(L, i)`

Given a G-lattice L, return the i-th generator of the (reduced) action of G on L. This is the reduced action of the i-th generator of the original group G (which may be the identity matrix).

`NaturalGroup(L)`

Given a G-lattice L, return the matrix group of the (natural) action of G on L. The resulting group thus acts on L naturally.

`NaturalActionGenerator(L, i)`

Given a G-lattice L, return the i-th generator of the natural action of G on L. This is simply the i-th generator of the original group G.

31.3.3 Invariant Forms

The functions in this section compute invariant forms for G-lattices.

`InvariantForms(L)`

For a G-lattice L, return a basis for the space of invariant bilinear forms for G (represented by their Gram matrices) as a sequence of matrices. The first entry of the sequence is a positive definite symmetric form for G.

`InvariantForms(L, n)`

For a G-lattice L, return a sequence consisting of $n \geq 0$ invariant bilinear forms for G.

`SymmetricForms(L)`

For a G-lattice L, return a basis for the space of symmetric invariant bilinear forms for G. The first entry of the sequence is a positive definite symmetric form of G.

`SymmetricForms(L, n)`

For a G-lattice L, return a sequence of $n \geq 0$ independent symmetric invariant bilinear forms for G. The first entry of the first sequence (if $n > 0$) is a positive definite symmetric form for G.

`AntisymmetricForms(L)`

For a G-lattice L, return a basis for the space of antisymmetric invariant bilinear forms for G.

`AntisymmetricForms(L, n)`

For a G-lattice L, return a sequence of $n \geq 0$ independent antisymmetric invariant bilinear forms for G.

`NumberOfInvariantForms(L)`

For a G-lattice L, return the dimension of the space of (symmetric and antisymmetric) invariant bilinear forms for G. The algorithm uses a modular method which is always correct and is faster than the actual computation of the forms.

`NumberOfSymmetricForms(L)`

For a G-lattice L, return the dimension of the space of symmetric invariant bilinear forms for G.

`NumberOfAntisymmetricForms(L)`

For a G-lattice L, return the dimension of the space of antisymmetric invariant bilinear forms for G.

`PositiveDefiniteForm(L)`

For a G-lattice L, return a positive definite symmetric form for G. This is a positive definite matrix F such that $gFg^{tr} = F$ for all $g \in G$.

31.3.4 Endomorphisms

The functions in this subsection compute endomorphisms of G-lattices. This is done by approximating the averaging operator over the group and applying it to random elements.

`EndomorphismRing(L)`

For a G-lattice L, return the endomorphism ring of L as a matrix algebra over $\mathbf{Q}$.

`Endomorphisms(L, n)`

For a G-lattice L, return a sequence containing n independent endomorphisms of L as elements of the corresponding matrix algebra over $\mathbf{Q}$. n must be in the range $[0..d]$, where d is the dimension of the endomorphism ring of L. This function may be useful in situations where the full endomorphism algebra is not required, e.g., to split a reducible lattice.

`DimensionOfEndomorphismRing(L)`

Return the dimension of the endomorphism algebra of the G-lattice L by a modular method (which always yields a correct answer).

`CentreOfEndomorphismRing(L)`

For a G-lattice L, return the centre of the endomorphism ring of L as a matrix algebra over $\mathbf{Q}$.

This function can be used to split a reducible lattice into its homogeneous components.

`CentralEndomorphisms(L, n)`

For a G-lattice L, return a sequence containing n independent central endomorphisms of L as elements of the corresponding matrix algebra over $\mathbf{Q}$. n must be in the range $[0..d]$, where d is the dimension of the centre of the endomorphism ring of L.

`DimensionOfCentreOfEndomorphismRing(L)`

Return the dimension of the centre of the endomorphism algebra of the G-lattice L by a modular method (which always yields a correct answer).

31.3.5 G-invariant Sublattices

The functions in this section compute G-invariant sublattices of a given G-lattice L.

For a fixed prime p, the algorithm constructs the maximal G-invariant sublattices of L as kernels of $\mathbf{F}_pG$-epimorphisms $L/pL \to S$ for some simple $\mathbf{F}_pG$-module S as described in [Ple74].

Iterating this process yields all G-invariant sublattices of L whose index in L is a p-power. Finally, intersecting lattices of coprime index yields all sublattices of L.

`Sublattices(G, Q)`

`Sublattices(L, Q)`

`Limit`	RngIntElt	*Default :* ∞
`Levels`	RngIntElt	*Default :* ∞
`Projections`	[Mtrx]	*Default :* []

Given either

(a) an integral matrix group G with natural lattice $L = \mathbf{Z}^n$

(b) a sequence G of integral matrices generating a $\mathbf{Z}$-order in $\mathbf{Q}^{n\times n}$ with natural lattice $L = \mathbf{Z}^n$

(c) a G-lattice L in $\mathbf{Q}^n$.

together with a set or sequence Q of primes, compute the G-invariant sublattices of L (as a sequence) which are not contained in pL for any $p \in Q$ and whose index in L is a product of elements of Q.

This set of G-invariant sublattices of L is finite if and only if $\mathbf{Q}_p \otimes L$ is irreducible as a $\mathbf{Q}_p G$-module for all $p \in Q$.

Setting the parameter `Limit := n` will terminate the computation after n sublattices have been found.

Setting the parameter `Levels := n` will only compute sublattices M such that L/M has at most n composition factors.

The optional parameter `Projections` can be a sequence of n by n matrices that describe projections on $\mathbf{Q}^n$ that map L to itself. In this case, MAGMA will only compute those sublattices of L which have the same images under the projections as L does.

The second return value indicates whether the returned sequence contains all such sublattices or not.

`Sublattices(G, p)`

`Sublattices(L, p)`

`Limit`	RNGINTELT	*Default :* ∞
`Levels`	RNGINTELT	*Default :* ∞
`Projections`	[MTRX]	*Default :* []

The same as the above where the set Q consists only of the given ptime p.

`Sublattices(G)`

`Sublattices(L)`

`Limit`	RNGINTELT	*Default :* ∞
`Levels`	RNGINTELT	*Default :* ∞
`Projections`	[MTRX]	*Default :* []

For an integral matrix group G or a G-lattice L this intrinsic equals the one above with Q taken to be the prime divisors of the order of G.

`SublatticeClasses(G)`

`MaximalOrders`	BOOLELT	*Default :* `false`

For an integral matrix group G returns representatives for the isomorphism classes of G-invariant lattices (i.e. the orbits under the unit group of the endomorphism ring E of G).

If `MaximalOrders` is set to `true`, only sublattice classes which are invariant under some maximal order of E are considered.

Currently the function requires E to be a field.

Example H31E6

We construct sublattices of the standard G-lattice where G is an absolutely irreducible degree-8 integral matrix representation of the group $\mathrm{GL}(2,3) \times \mathrm{S}_3$.

We first define the group G.

```
> G := MatrixGroup<8, IntegerRing() |
>    [-1,  0,  0,  0,  0,  0,  0,  0,
>      0,  0, -1,  0,  0,  0,  0,  0,
>      0,  0,  0,  1,  0,  0,  0,  0,
>      0,  1,  0,  0,  0,  0,  0,  0,
>     -1,  0,  0,  0,  1,  0,  0,  0,
>      0,  0, -1,  0,  0,  0,  1,  0,
>      0,  0,  0,  1,  0,  0,  0, -1,
>      0,  1,  0,  0,  0, -1,  0,  0],
>
>    [ 0,  0,  0,  0,  0,  0,  0,  1,
>      0,  0,  0,  0,  0,  0,  1,  0,
>      0,  0,  0,  0, -1,  0,  0,  0,
>      0,  0,  0,  0,  0,  1,  0,  0,
>      0,  0,  0, -1,  0,  0,  0,  1,
>      0,  0, -1,  0,  0,  0,  1,  0,
>      1,  0,  0,  0, -1,  0,  0,  0,
>      0, -1,  0,  0,  0,  1,  0,  0]>;
```

We next compute the unique positive definite form F fixed by G.

```
> time F := PositiveDefiniteForm(G);
Time: 0.050
> F;
[2 0 0 0 1 0 0 0]
[0 2 0 0 0 1 0 0]
[0 0 2 0 0 0 1 0]
[0 0 0 2 0 0 0 1]
[1 0 0 0 2 0 0 0]
[0 1 0 0 0 2 0 0]
[0 0 1 0 0 0 2 0]
[0 0 0 1 0 0 0 2]
```

We now compute all sublattices of the standard G-lattice.

```
> time Sub := Sublattices(G);
Time: 0.370
> #Sub;
18
```

For each sublattice we compute the invariant positive definite form for the group given by the action of G on the sublattice.

```
> PrimitiveMatrix := func<X |
>     P ! ((ChangeRing(P, RationalField()) ! X) / GCD(Eltseq(X)))
```

```
>         where P is Parent(X)>;
> FF := [PrimitiveMatrix(B * F * Transpose(B))
>             where B is BasisMatrix(L): L in Sub];
```

We next create the sequence of all the lattices whose Gram matrices are given by the (LLL-reduced) forms.

```
> Sub := [LatticeWithGram(LLLGram(F)) : F in FF];
> #Sub;
18
```

We now compute representatives for the **Z**-isomorphism classes of the sequence of lattices.

```
> Rep := [];
> for L in Sub do
>     if forall{LL: LL in Rep | not IsIsometric(L, LL)} then
>         Append(~Rep, L);
>     end if;
> end for;
> #Rep;
4
```

Thus there are 4 non-isomorphic sublattices. We note the size of the automorphism group, the determinant, the minimum and the kissing number of each lattice. (In fact, the automorphism groups of these 4 lattices happen to be maximal finite subgroups of $\mathrm{GL}(8, \mathbf{Q})$ and all have $\mathrm{GL}(2,3) \times \mathrm{S}_3$ as a common irreducible subgroup.)

```
> time A := [AutomorphismGroup(L) : L in Rep];
Time: 0.240
> [#G: G in A];
[ 497664, 6912, 696729600, 2654208 ]
> [Determinant(L): L in Rep];
[ 81, 1296, 1, 16 ]
> [Minimum(L): L in Rep];
[ 2, 4, 2, 2 ]
> [KissingNumber(L): L in Rep];
[ 24, 72, 240, 48 ]
```

Finally, we note that each lattice is isomorphic to a standard construction based on root lattices.

```
> l := IsIsometric(Rep[1],
>        TensorProduct(Lattice("A", 2), StandardLattice(4))); l;
true
> l := IsIsometric(Rep[2],
>        TensorProduct(Lattice("A", 2), Lattice("F", 4))); l;
true
> l := IsIsometric(Rep[3], Lattice("E", 8)); l;
true
> l := IsIsometric(Rep[4],
>        TensorProduct(Lattice("F", 4), StandardLattice(2))); l;
```

Example H31E7

This example illustrates the optional argument `Projections`.

```
> G := MatrixGroup<4, IntegerRing() |
> [ -1, 0, 1, 0, 0, -1, 1, -3, -1, 0, 0, 0, 0, 0, 0, 1 ],
> [ -1, 0, 0, 0, -3, 2, 0, 3, 0, 0, -1, 0, 1, -1, 0, -1 ] >;
> E := EndomorphismRing(G);
> I := CentralIdempotents(ChangeRing(E, RationalField())); I;
[
    [ 0  0  0  0]
    [-1  1  0  0]
    [ 0  0  0  0]
    [ 0  0  0  1],
    [1 0 0 0]
    [1 0 0 0]
    [0 0 1 0]
    [0 0 0 0]
]
```

Since the central idempotents are all integral, they map the standard lattice $\mathbf{Z}^n$ to itself. Even though this group G fixes infinitely many sublattices of Z^n (even up to scalar multiples), there can only be finitely many which have the same images under the central idempotents as $\mathbf{Z}^n$.

```
> S := Sublattices(G : Projections:= I); #S;
3
```

So in this case there are only three such lattices. To check that the lattices do project correctly, we can use

```
> I := [ Matrix(Integers(), i) : i in I ];
> Images := [ [Image(BasisMatrix(s) * i) : i in I] : s in S ];
> #Set(Images) eq 1;
true
```

31.3.6 Lattice of Sublattices

MAGMA can construct the lattice V of all G-invariant sublattices of the standard lattice $L = \mathbf{Z}^n$. Various properties of the lattice V may then be examined. MAGMA only stores the primitive sublattices of L, i.e. those sublattices that are not contained in kL for some $k > 1$.

In general, G fixes infinitely many primitive lattices. Thus one has to limit the number of sublattices to be constructed just as in the `Sublattice` intrinsic. In this case, all operations on V like coercions, intersections, sums etc. assume that the result of the operation is again is a scalar multiple of some element stored in V.

The lattice V has type `LatLat` and elements of V have type `LatLatElt` and are numbered from 1 to n where n is the number of primitive sublattices of L that have been constructed in the beginning.

31.3.6.1 Creating the lattice of sublattices

`SublatticeLattice(G, Q)`

`Limit`	RngIntElt	*Default :* ∞
`Levels`	RngIntElt	*Default :* ∞
`Projections`	[Mtrx]	*Default :* []

Given either an integral matrix group G of degree n or a sequence G of integral matrices generating a $\mathbf{Z}$-order in $\mathbf{Q}^{n\times n}$ together with a set or sequence Q of primes, compute the G-invariant sublattices of $\mathbf{Z}^n$ (as a sequence) which are not contained in $p\mathbf{Z}^n$ for any $p \in Q$ and whose index in $\mathbf{Z}^n$ is a product of elements of Q.

The second return value indicates whether all G-invariant lattices have been constructed.

The optional parameters are the same as for the `Sublattices` intrinsic.

`SublatticeLattice(G, p)`

`Limit`	RngIntElt	*Default :* ∞
`Levels`	RngIntElt	*Default :* ∞
`Projections`	[Mtrx]	*Default :* []

Same as above where the set Q consists only of the given prime p.

`SublatticeLattice(G)`

`Limit`	RngIntElt	*Default :* ∞
`Levels`	RngIntElt	*Default :* ∞
`Projections`	[Mtrx]	*Default :* []

Same as above where the set Q is taken to be set of prime divisors of the order of the group G.

Example H31E8

This example shows how to create a lattice of sublattices.

```
> G:= sub< GL(2, Integers()) | [0,1,-1,0] >;
> V:= SublatticeLattice(G); V;
Lattice of 2 sublattices
```

31.3.6.2 Operations on the Lattice of Sublattices

In the following, V is a lattice of G-invariant lattices for some group or **Z**-order G and Q denotes the set of primes that where used to create V.

`#V`

The number of (primitive) lattices stored in V.

`V ! i`

The i-th element of the lattice V with respect to the internal labeling.

`V ! M`

Given a (basis matrix of some) G-invariant lattice M, create the element of the lattice V corresponding to M.

`NumberOfLevels( V )`

The number of different levels (layers) stored in V. Note that levels are counted starting from 0.

`Level(V, i)`

The primitive lattices stored at the i-th level (layer). Note that levels are counted starting from 0.

`Levels(v)`

The i-th entry of the result is a sequence of the primitive lattice elements lying on the $i-1$-th level.

`Primes(V)`

The primes that where used to create V.

`Constituents(V)`

A sequence containing the constituents (simple $\mathbf{F}_pG$ - modules) that where used during the construction of the G-lattices in V.

`IntegerRing() ! e`

The integer corresponding to lattice element e.

`e + f`

The sum of the lattice elements e and f.

`e meet f`

The intersection of the lattice elements e and f.

`e eq f`

Tests whether e and f are equal.

`MaximalSublattices(e)`

The sequence S of maximal sublattices of e having index p for some $p \in Q$. The second return value is a list C of integers such that $S[i]/e$ is isomorphic to the $C[i]$-th constituent of V. The ordering of the constituents is the same as in the `Constituents` intrinsic.

`MinimalSuperlattices(e)`

The sequence S of minimal superlattices of e in which e has index p for some $p \in Q$. The second return value is a list C of integers such that $e/S[i]$ is isomorphic to the $C[i]$-th constituent of V. The ordering of the constituents is the same as in the `Constituents` intrinsic.

`Lattice(e)`

The G-lattice corresponding to e.

`BasisMatrix(e)`

`Morphism(e)`

The basis matrix of the G-lattice corresponding to e.

Example H31E9

Let G be the automorphism group of the root lattice A_5. Since G is absolutely irreducible, it fixes only finitely many lattices up to scalars. We explore them.

```
> G:= AutomorphismGroup(Lattice("A", 5));
> FactoredOrder(G);
[ <2, 5>, <3, 2>, <5, 1> ]
> #SublatticeLattice(G, 5);
1
```

Hence there are no primitive sublattices between L and $5L$. Hence it suffices to check only the lattices at 2 and 3 the two remaining prime divisors of the order of G.

```
> V:= SublatticeLattice(G, {2,3}); #V;
4
> M:= MaximalSublattices(V ! 1); M;
[
    sublattice number 2,
    sublattice number 3
]
> V ! 2 meet V ! 3;
sublattice number 4
```

Moreover, the second and third lattice are (up to rescaling) dual to each other with respect to some G-invariant form.

```
> F:= PositiveDefiniteForm(G);
> L:= Dual(Lattice(BasisMatrix(V ! 2), F) : Rescale:= false);
> V ! L;
```

```
sublattice number 3 times 1/6
```

In particular, every G-invariant lattice can be constructed from lattice number 2 by taking scalar multiples, duals, sums and intersections. For example the standard lattice can be written as:

```
> (V ! 2) + (V ! (6*L));
sublattice number 1
```

Example H31E10

Let G be the 8-dimensional (faithful) rational representation of $\mathrm{SL}(2,7)$. Its endomorphism ring E is isomorphic to $\mathbf{Q}(\sqrt{-7})$. We find all G-invariant lattices of G that are invariant under the maximal order M of E up to multiplication with elements in E. After this is done, we quickly obtain all finite subgroups of $\mathrm{GL}(8,\mathbf{Q})$ (up to conjugacy) that include a normal subgroup conjugate to G.
To shorten the example, we choose G such that the standard lattice L is already invariant under M.

```
> SetSeed(1);
> G:= MatrixGroup<8, IntegerRing() |
>    [ 0, 1, 0, 0, 0, 0, -1, 0, -1, 0, 0, 0, 0, 0, -1, 1,
>      0, 0, 0, 1, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, -1, 1,
>      0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, -1, 0, -1, 1,
>      0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 0, 0, 0, 0, -2, 1 ],
>    [ 0, -1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0,
>     -1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, -1, 1,
>      0, 0, 0, -1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
>      0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ] >; #G;
336
> E:= EndomorphismRing(G);
> M:= MaximalOrder(ChangeRing(E, RationalField()));
> ok, M:= CanChangeUniverse(Basis(M), MatrixRing(Integers(), 8)); ok;
true
```

So L is M-invariant. The lattices at the primes 3 and 7 are multiples of L as we can see as follows:

```
> w7:= IntegralMatrix(E.2 - Trace(E.2)/8);
> w7 div:= GCD( Eltseq(w7) );                    // a square root of 7
> V:= SublatticeLattice([ Matrix(G.i) : i in [1..Ngens(G)] ] cat M, [3,7]); #V;
2
> V ! w7;
sublattice number 2
```

So it remains to check the lattices at 2. The two prime ideals in M over 2 are generated by `p` and `q` where

```
> p:= 1 - (w7+1) div 2;
> q:= (w7+1) div 2;
> Gens:= [ Matrix(G.i) : i in [1..Ngens(G)] ];
> V:= SublatticeLattice(Gens cat M, 2: Levels:= 3);
> Levels(V);
```

```
[
    [
        sublattice number 1
    ],
    [
        sublattice number 2,
        sublattice number 3
    ],
    [
        sublattice number 4,
        sublattice number 5,
        sublattice number 6
    ],
    [
        sublattice number 7,
        sublattice number 8,
        sublattice number 9,
        sublattice number 10
    ]
]
> [ V | BasisMatrix(V ! i)*x : i in [1..3], x in [p,q] ];
[
    sublattice number 4,
    sublattice number 7,
    sublattice number 8,
    sublattice number 6,
    sublattice number 9,
    sublattice number 10
]
```

So the lattice numbers $1, 2, 3$ and 5 represent the orbits of the action of E on the set of all MG-invariant lattices. Moreover, every matrix group N normalizing G acts on the MG-invariant lattices and (up to conjugacy) thus fixes one of these four lattices. If it fixes L, it also fixes $V!2 + V!3 = V!5$ and vice versa. Similarly, it fixes $V!2$ if and only if it fixes $V!3$.

```
> F:= PositiveDefiniteForm(G);
> N1:= Normalizer(AutomorphismGroup(LatticeWithGram(F)), G); #N1;
672
> A:= AutomorphismGroup(Lattice(BasisMatrix(V ! 2), F) : NaturalAction);
> N2:= Normalizer(A, ChangeRing(G, Rationals())); #N2;
336
```

So `N1` (which is isomorphic to $2.L(2,7):2$) is up to conjugacy the only proper finite extension of G in $\mathrm{GL}(8, \mathbf{Q})$.

31.4 Bibliography

[**Ger03**] Larry J. Gerstein. Definite quadratic forms over $\mathbf{F}_q[X]$. *J. Algebra*, 268(1):252–263, 2003.

[**Kir12**] M. Kirschmer. A normal form for definite quadratic forms over $\mathbf{F}_q[t]$. *Math. Comp.*, 81:1619–1634, 2012.

[**Ple74**] Wilhelm Plesken. *Beiträge zur Bestimmung der endlichen irreduziblen Untergruppen von GL(n,Z) und ihrer ganzzahligen Darstellungen.* PhD thesis, RWTH Aachen, 1974.

[**PS97**] Wilhelm Plesken and Bernd Souvignier. Computing Isometries of Lattices. *J. Symbolic Comp.*, 24(3):327–334, 1997.

32 QUADRATIC FORMS

Chapter 32
QUADRATIC FORMS

32.1 Introduction

This chapter describes miscellaneous functionality for fairly general quadratic forms. The main feature currently here is an implementation of Simon's algorithm for finding isotropic subspaces of integral forms.

In MAGMA, quadratic forms are generally represented either as multivariate polynomials, or as symmetric matrices.

32.2 Constructions and Conversions

`SymmetricMatrix(f)`

Given a multivariate polynomial that is homogeneous of degree 2, this returns a symmetric matrix representing the same quadratic form.

`GramMatrix(L)`

The symmetric matrix giving the quadratic form on the lattice L.

`QuadraticForm(L)`

The quadratic form associated to the lattice L, as a multivariate polynomial.

`QuadraticForm(M)`

The quadratic form for a symmetric matrix M, as a multivariate polynomial.

32.3 Local Invariants

These commands calculate the standard invariants that characterize a quadratic form over the rationals. Definitions of the invariants may be found in Conway-Sloane [JC98], Chapter 15, Section 5.1.

```
pSignature(f,p)
pSignature(M,p)
pSignature(L,p)
```

The p-signature of the specified quadratic form over the rationals, where p is a prime number or -1 (designating the real place).

For odd primes p, this is defined by diagonalizing the form, and adding p-parts of these entries to 4 times the number of anti-squares (mod p) amongst these entries. The term "anti-square" modulo p denotes something that has: odd valuation at p; and the prime-to-p part, called u, has Kronecker symbol $(\frac{u}{p}) = -1$.

At $p = 2$ it is the sum of the odd parts of the diagonalized entries plus 4 times the number of anti-squares. In either case, the final answer is really only defined modulo 8 (this is so that p-signatures are invariant under rational equivalence).

At the real place, it is the difference between the number of positive and negative eigenvalues (the terminology here can be murky).

```
Oddity(f)
Oddity(L)
Oddity(M)
```

This returns the 2-signature of the given quadratic form over the rationals.

```
pExcess(f, p)
pExcess(M, p)
pExcess(L, p)
```

The p-excess of the specified quadratic form over the rationals, where p is a prime number or -1 (designating the real place). The p-excess is the difference between the p-signature and dimension for odd primes (including -1), and is the negation of this for $p = 2$. The sum of p-excesses over all primes should be 0 modulo 8.

```
WittInvariant(f, p)
WittInvariant(M, p)
WittInvariant(L, p)
HasseMinkowskiInvariant(f, p)
HasseMinkowskiInvariant(M, p)
HasseMinkowskiInvariant(L, p)
```

Calculates the Witt invariant (sometimes called the Hasse-Minkowski invariant) over $\mathbf{Q}_p$ of the given quadratic form. Again the form must be defined over either the rationals or the integers. The result is returned as something in the set $\{-1, +1\}$. One definition of this invariant is to diagonalize the form and then take the product (in our multiplicative notation) of the Hilbert symbols of the $\binom{n}{2}$ pairs of distinct nonzero diagonal entries, as in §5.3 of Chapter 15 of Conway-Sloane [JC98], which is what is implemented here. Another method would be to use a comparison of p-excesses with the standard form (also in Conway-Sloane). Starting with a p-adic input could lead to precision problems at the diagonalization step, and so is not allowed. Can also be called via `HasseMinkowskiInvariant`.

`WittInvariants(f)`
`WittInvariants(M)`
`WittInvariants(L)`
`HasseMinkowskiInvariants(f)`
`HasseMinkowskiInvariants(M)`
`HasseMinkowskiInvariants(L)`

Compute `WittInvariant(f,p)` for all bad primes p, and return the result of a sequence of tuples, each entry given by $\langle p, W_p(f)\rangle$. The set of bad primes includes the real place, the prime $p = 2$, and all primes that divide either the numerator or the denominator of the determinant of symmetric matrix associated to f. Can also be called via `HasseMinkowskiInvariants`.

32.4 Isotropic Subspaces

`IsotropicSubspace(f)`
`IsotropicSubspace(M)`

This returns an isotropic subspace for the given quadratic form (which must be either integral or rational), which may be given either as a multivariate polynomial f or as a symmetric matrix M. The subspace returned is in many cases guaranteed to be a maximal totally isotropic subspace. More precisely (assuming that the form is nonsingular), upon writing (r, s) for the signature of f with $r \geq s$, the dimension of the space returned is at least $\min(r, s + 2) - 2$, which is maximum possible when $s \leq r + 2$. Since version 2.18, an improvement to the original algorithm has been appended, which typically (subject to a solvability criterion for a 4-dimensional subspace) enlarges the dimension of space by 1 when $r + s$ is even and $s \leq r + 2$.

The algorithm used is due to Simon (see [Sim05]), and uses on the Bosma-Stevenhagen algorithm for the 2-part of the class groups of a quadratic field (see [BS96]). There is no corresponding intrinsic for a lattice: since the associated form is definite, there are no isotropic vectors.

Example H32E1

```
> v :=  [ 6, -2, -7, 5, -2, -10, -3, 5, -7, -3, 10, -8, 5, 5, -8, 0 ];
> M := Matrix(4, 4, v); M;
[  6  -2  -7   5]
[ -2 -10  -3   5]
[ -7  -3  10  -8]
[  5   5  -8   0]
> Determinant(M); Factorization(Determinant(M));
1936
[ <2, 4>, <11, 2> ]
> WittInvariant(M, 2);
-1
> WittInvariant(M, 11);
1
> D := Diagonalization(M); D; // signature (2,2)
[     6      0      0      0]
[     0    -96      0      0]
[     0      0     18      0]
[     0      0      0 -34848]
> pSignature(M, -1); // should be the difference of 2 and 2
0
> n := Degree(Parent(M));
> Q := Rationals();
> E := [ Q ! D[i][i] : i in [1..n]];
> &*[ &*[ HilbertSymbol(E[i], E[j], 2) : i in [j+1..n]] : j in [1..n-1]];
-1
> &*[ &*[ HilbertSymbol(E[i], E[j], 11) : i in [j+1..n]] : j in [1..n-1]];
1
> IsotropicSubspace(M);
RSpace of degree 4, dimension 2 over Integer Ring
Generators:
( 3 -3  0  4)
( 4 -4  0 -2)
Echelonized basis:
( 1 -1  0 16)
( 0  0  0 22)
> pSignature(M, 2) mod 8;
0
> pSignature(M, 11) mod 8;
4
> pSignature(M, 5) mod 8; // equals Dimension at good primes
4
```

Example H32E2

```
> SetSeed(12345);
> n := 20;
> P := PolynomialRing(Integers(),n);
> f:=&+[&+[Random([-10..10])*P.i*P.j : i in [j..n]] : j in [1..n]];
> M := ChangeRing(2*SymmetricMatrix(f), Integers()); M;
[12 -2 -8 1 -6 0 3 -5 -6 -5 10 7 -1 -3 -7 -5 -6 -3 -3 -8]
[-2 2 10 -10 3 6 -3 -9 0 -5 7 -5 -5 -4 1 -5 1 -6 7 -8]
[-8 10 14 1 6 -8 -3 4 -2 3 -4 7 0 -8 -6 -4 -5 7 -4 8]
[1 -10 1 6 -5 0 8 1 7 -1 7 -7 6 7 -1 -9 -8 1 6 6]
[-6 3 6 -5 12 7 -3 -10 -5 -4 -6 1 -5 -9 4 -8 2 5 -4 -10]
[0 6 -8 0 7 -6 -4 3 0 2 -3 0 4 10 5 -8 6 1 -7 2]
[3 -3 -3 8 -3 -4 16 -5 -10 7 -9 9 -6 -2 0 -1 1 -4 6 2]
[-5 -9 4 1 -10 3 -5 2 -4 -3 0 -3 9 -1 3 -5 2 2 -1 -10]
[-6 0 -2 7 -5 0 -10 -4 0 -1 5 1 3 5 3 0 1 -7 5 10]
[-5 -5 3 -1 -4 2 7 -3 -1 10 -9 5 2 -4 8 6 -4 9 -3 5]
[10 7 -4 7 -6 -3 -9 0 5 -9 2 2 -7 3 8 -4 7 -3 -3 6]
[7 -5 7 -7 1 0 9 -3 1 5 2 6 -10 -6 0 -2 -3 8 2 -9]
[-1 -5 0 6 -5 4 -6 9 3 2 -7 -10 -10 7 0 8 -1 -2 9 3]
[-3 -4 -8 7 -9 10 -2 -1 5 -4 3 -6 7 -6 -2 -3 3 -9 9 6]
[-7 1 -6 -1 4 5 0 3 3 8 8 0 0 -2 2 -8 5 2 3 -4]
[-5 -5 -4 -9 -8 -8 -1 -5 0 6 -4 -2 8 -3 -8 -18 5 4 6 4]
[-6 1 -5 -8 2 6 1 2 1 -4 7 -3 -1 3 5 5 6 -3 7 -7]
[-3 -6 7 1 5 1 -4 2 -7 9 -3 8 -2 -9 2 4 -3 14 5 2]
[-3 7 -4 6 -4 -7 6 -1 5 -3 -3 2 9 9 3 6 7 5 16 -3]
[-8 -8 8 6 -10 2 2 -10 10 5 6 -9 3 6 -4 4 -7 2 -3 -4]
> D := Integers() ! Determinant(M); D;
276132003816103322711292
> [ <u[1], WittInvariant(f, u[1])> : u in Factorization(D)];
[ <2, -1>, <3, -1>, <7, 1>, <199, 1>, <879089, -1>, <6263690372711, -1> ]
> &+[ pExcess(f, u[1]) : u in Factorization(D) cat [<-1, 0>]] mod 8;
0
> time S := IsotropicSubspace(f);
Time: 0.640
> Dimension(S);
8
> pSignature(f, -1); // difference of 12 and 8
4
> B := Basis(S);
> InnerProduct(B[1], B[1]*M);
0
> IP := InnerProduct;
> d := Dimension(S);
> &and [&and [ IP(B[i], B[j]*M) eq 0 : i in [1..d]] : j in [1..d]];
true
```

32.5 Bibliography

[**BS96**] W. Bosma and P. Stevenhagen. On the computation of quadratic 2-class groups. *Journal de théorie des nombres de Bordeaux*, 8(2):283–313, 1996.

[**JC98**] N.J.A. Sloane J.H. Conway. *Sphere Packings, Lattices and Groups*, volume 290 of *Grundlehren der Mathematischen Wissenschaften*. Springer, New York–Berlin–Heidelberg, 3rd edition, 1998.

[**Sim05**] Denis Simon. Quadratic equations in dimensions 4, 5 and more. Preprint, URL:http://www.math.unicaen.fr/~simon/, 2005.

33 BINARY QUADRATIC FORMS

Chapter 33
BINARY QUADRATIC FORMS

33.1 Introduction

A binary quadratic form is an integral form $ax^2+bxy+cy^2$ which is represented in MAGMA by a tuple $\langle a, b, c\rangle$. Binary quadratic forms play an central role in the ideal theory of quadratic fields, the classical theory of complex multiplication, and the theory of modular forms. Algorithms for binary quadratic forms provide efficient means of computing in the ideal class group of orders in a quadratic field. By using the explicit relation of definite quadratic forms with lattices with nontrivial endomorphism ring in the complex plane, one can apply modular and elliptic functions to forms, and exploit the analytic theory of complex multiplication.

The structures of quadratic forms of a given discriminant D correspond to ordered bases of ideals in an order in a quadratic number field, defined up to scaling by the rationals. A form is primitive if the coefficients a, b, and c are coprime. For negative discriminants the primitive reduced forms in this structure are in bijection with the class group of projective or invertible ideals. For positive discriminants, the reduced orbits of forms are used for this purpose. MAGMA holds efficient algorithms for composition, enumeration of reduced forms, class group computations, and discrete logarithms. A significant novel feature is the treatment of nonfundamental discriminants, corresponding to nonmaximal orders, and the collections of homomorphisms between different class groups coming from the inclusions of these orders.

The functionality for binary quadratic forms is rounded out with various functions for applying modular and elliptic functions to forms, and for class polynomials associated to class groups of definite forms.

33.2 Creation Functions

33.2.1 Creation of Structures

For any integer D congruent to 0 or 1 modulo 4, it is possible to create the parent structure of binary quadratic forms of discriminant D.

```
BinaryQuadraticForms(D)
```

```
QuadraticForms(D)
```

Create the structure of integral binary quadratic forms of discriminant D.

33.2.2 Creation of Forms

Binary quadratic forms may be created by coercing a triple $[a, b, c]$ of integer coefficients into the parent structure of forms of discriminant $D = b^2 - 4ac$. Other constructors are provided for constructing the group identity, prime forms, or allowing the omission of third element c of the sequence.

```
Identity(Q)
```

```
Q ! 1
```

Create the principal form in the structure Q of binary quadratic forms of discriminant D. The principal form is either $X^2 - D/4Y^2$ if $D \bmod 4$ is 0 and $X^2 + XY + (D-1)/4Y^2$ if it is 1. The principal form is a reduced form representing the identity element of the class group of Q.

```
Q ! [a, b, c]
```

```
elt< Q | a, b, c >
```

```
elt< Q | a, b >
```

Returns the binary quadratic form $aX^2 + bXY + cY^2$ in the magma of forms Q of discriminant D. Here c is determined by the solution of the equality $D = b^2 - 4ac$; if no integer c exists satisfying this, an error will occur.

```
PrimeForm(Q, p)
```

If p is a split prime or a ramified prime not dividing the conductor of the magma of quadratic forms Q, returns a quadratic form $pX^2 + bXY + cY^2$ in Q.

33.3 Basic Invariants

Structures of binary quadratic forms are defined in terms of a discriminant, and membership in a structure determined by this invariant. To aid in the construction of forms, additional elementary functions are provided to test integer inputs to determine if they define valid discriminants of quadratic forms.

```
Discriminant(f)
```

The discriminant $b^2 - 4ac$ of a quadratic form $f = aX^2 + bXY + cY^2$.

```
Discriminant(Q)
```

The discriminant of the quadratic forms belonging to the magma of quadratic forms Q.

```
IsDiscriminant(D)
```

Return `true` if the integer D is the discriminant of some quadratic form; `false` otherwise.

`FundamentalDiscriminant(D)`

The fundamental discriminant corresponding to the integer D.

`IsFundamental(D)`

`IsFundamentalDiscriminant(D)`

Return `true` if D is an integer other than 0 or 1 congruent to 0 or 1 modulo 4, which is not of the form $m^2 D_K$ for $m > 1$ and any other such integer D_K.

`Conductor(Q)`

The conductor of quadratic forms whose discriminant is that of the magma of quadratic forms Q.

33.4 Operations on Forms

33.4.1 Arithmetic

`Conjugate(f)`

Given a form $f = ax^2 + bxy + cy^2$, returns the conjugate form $ax^2 - bxy + cy^2$.

`f * g`

`Composition(f, g)`

`Al`	MonStgElt	*Default :* "*Gauss*"
`Reduction`	BoolElt	*Default :* `false`

Returns the composition of two binary quadratic forms f and g. The operator `'*'` returns a reduced representative of the product using a fast composition algorithm of Shanks. In contrast, the default for `Composition` is `Reduction := false`, so that one can work in the group of forms, rather in the set of class group representatives. The function `Composition` takes a further parameter `Al` which specifies whether the algorithm of Gauss or Shanks, set to `"Gauss"` by default. The algorithm of Shanks performs partial intermediate reductions, so the combination `Reduction := false` and `Al := "Shanks"` are incompatible and returns a runtime error.

`f ^ n`

`Power(f, n)`

`Al`	MonStgElt	*Default :* "*Gauss*"
`Reduction`	BoolElt	*Default :* `false`

Returns the n-th power of a form f. The operator `'^'` returns a reduced representative, using the fast composition algorithm of Shanks. In contrast, the default for `Power` is `Reduction := false`, so that one can work in the group of forms rather than in the class group. The function `Power` takes the further parameter `Al` in order to specify whether the algorithm of Gauss or Shanks is used, set to `"Gauss"` by default. The algorithm of Shanks performs partial intermediate reductions, so

the combination `Reduction := false` and `Al := "Shanks"` are incompatible and returns a runtime error.

`Reduction(f)`
`ReducedForm(f)`

Returns a reduced quadratic form equivalent to f, and the transformation matrix.

`ReductionStep(f)`

Returns the result of applying one reduction step to the quadratic form f.

`ReductionOrbit(f)`

The cycle of reduced forms equivalent to f (and each other) where f has positive discriminant.

`Order(f)`

For a binary quadratic form f, returns its order as an element of the class group $Cl(Q)$ where Q is the parent of f.

33.4.2 Attribute Access

The coefficient sequence can be accessed as a sequence of integers, providing the inverse operation to the forms coercion constructor.

`f[i]`

The i-th coefficient of f, where $1 \leq i \leq 3$.

`Eltseq(f)`
`ElementToSequence(f)`

The sequence $[a, b, c]$ where f is the form $ax^2 + bxy + cy^2$.

33.4.3 Boolean Operations

Several boolean operators apply to quadratic forms.

`f in Q`

Return `true` if and only if f is in Q, that is f and Q have the same discriminant.

`f eq g`

Return `true` if the quadratic form f and g are equal and `false` otherwise.

`IsIdentity(f)`

Return `true` if and only if f is the principal form in its parent structure.

`IsReduced(f)`

Return `true` if the quadratic form f is reduced; `false` otherwise.

`IsEquivalent(f, g)`

Return `true` if the quadratic forms f and g reduce to the same form and `false` otherwise. If `true` and the discriminant is negative, then the transformation matrix is also returned. An error is returned if the forms are not of the same discriminant.

33.4.4 Related Structures

In addition to the Parent and Category structures of binary quadratic forms of discriminant D, the quadratic forms map to the ideals of a fixed order or discriminant D in a quadratic number field.

`Parent(f)`

`Category(Q)`

`QuadraticOrder(Q)`

Given a structure of quadratic forms of discriminant D, returns the associated order of discriminant D in a quadratic field.

`Ideal(f)`

Given a quadratic form $f = ax^2 + bxy + cy^2$, returns the ideal $(a, (-b + \sqrt{D})/2)$ in the quadratic order $\mathbf{Z}[(t + \sqrt{D})/2]$, where t equals 0 or 1.

33.5 Class Group

`ReducedForms(Q)`

Given the structure of quadratic forms of negative discriminant D, returns the sequence of all primitive reduced forms of discriminant D.

`ReducedOrbits(Q)`

Given the structure of quadratic forms of positive discriminant D, returns the sequence of all reduced orbits of primitive forms of discriminant D, as an indexed set.

`ClassNumber(Q: parameters)`

`ClassNumber(D: parameters)`

`Al`	MonStgElt	*Default :* "*Automatic*"
`FactorBasisBound`	FldReElt	*Default :* 0.1
`ProofBound`	FldReElt	*Default :* 6
`ExtraRelations`	RngIntElt	*Default :* 1

The class number of binary quadratic forms Q of discriminant D. The parameter `Al` may be supplied to select the method used to calculate the class number. The possible values are `"ReducedForms"` (enumerating all reduced forms), `"Shanks"`

(using a Shanks-based algorithm in the class group), "Sieve" or "NoSieve". The default is to use reduced form enumeration for small discriminants, the Shanks algorithm for the middle range, a class group index-calculus for large discriminants and the sieve method described in [Jac99] for very large discriminants.

The remaining parameters apply to the index-calculus computations only; for details about the parameters FactorBasisBound, ProofBound and ExtraRelations see the description of ClassGroup.

ClassGroup(Q: *parameters*)

FactorBasisBound	FldReElt	*Default :* 0.1
ProofBound	FldReElt	*Default :* 6
ExtraRelations	RngIntElt	*Default :* 1
Al	MonStgElt	*Default :* "*Automatic*"

The class group of the binary quadratic forms Q of discriminant D. The function also returns a map from the abelian group to the structure of quadratic forms.

Depending on the size of D, either a Shanks-based method is used ([Tes98a, BJT97]) an index calculus variant ([CDO93, HM89, Coh93]) or a sieving method. The parameters FactorBasisBound, ProofBound and ExtraRelations apply to the index calculus method only. This method performs in two steps: In the first step a factor basis containing prime forms of norm $< B_1$ is build. Next, one looks for generators for the full lattice of relations between the forms in the factor basis. The determinant of this lattice will be the class number and the Smith-form of the relation matrix gives the structure of the class group. Once the matrix is of full rank, the algorithm will look for ExtraRelations more relations to potentially decrease the discriminant.

In the second stage, for all prime forms of norm $< B_2$ it is verified that they are in the class group generated by the forms of the first step. The bounds are chosen to be $B_1 :=$ FactorBasisBound$\cdot \log^2 |D|$ and $B_2 :=$ ProofBound$\cdot \log^2 |D|$, so the result is correct under the assumption of GRH.

The final result is then checked against the Euler product over the first 30.000 primes. If the quotient becomes to large, a warning is issued. In this case one should increase the ExtraRelations parameter.

If the parameter Al is set to "Sieve" (regardless of the size of D) or D is larger than 10^{20} then the sieving method described in [Jac99] which uses the multiple polynomial quadratic sieve (MPQS) will be used. This is currently only proven under GRH. If the parameter Al is set to "NoSieve" then the sieving method will not be used regardless of the size of D.

ClassGroupStructure(Q: *parameters*)

The structure of the class group of the binary quadratic forms Q of discriminant D returned as a sequence of integers giving the abelian invariants.

For details about the parameters Al, FactorBasisBound, ProofBound and Extra Relations, see ClassGroup.

`AmbiguousForms(Q)`

Enumerates the ambiguous forms of negative discriminant D, where D is the discriminant of the magma of binary quadratic forms Q.

`TwoTorsionSubgroup(Q)`

The subgroup of 2–torsion elements of in the class group of Q.

Example H33E1

We give an example of some computations in the class group of a magma of quadratic forms. Elements in the class group are not really represented by single reduced forms but by cycles of equivalent reduced forms.

```
> Q<z> := QuadraticField(7537543);                        // arbitrary choice
> Q := QuadraticForms(Discriminant(Q));
> C, m := ClassGroup(Q);
> C;
Abelian Group isomorphic to Z/2 + Z/76
Defined on 2 generators
Relations:
    76*C.1 = 0
    2*C.2 = 0
>                                    // get the generators as quadratic forms:
> f := m(C.1);
> g := m(C.2);
> h := g^2;
> g, h;
<-1038,4894,1493> <-887,4340,3189>
> c := [];                      // create the cycle of forms equivalent to g^2:
> repeat
>     h := ReductionStep(h);
>     Append(~c, h);
> until h eq g^2;
> P := Parent(g);
> Identity(P) in c;
true                 // this proves that the second class has order dividing 2
> for d in Divisors(76) do
>     c := [];
>     h := f^d;
>     repeat h := ReductionStep(h); Append(~c, h);
>     until h eq f^d;
>     d, Identity(P) in c, #c;
> end for;
1 false 16
2 false 14
4 false 12                           // the cycle lengths vary
19 false 18
38 false 16
```

```
76 true 14                          // so the true order of this class is 76
```

33.6 Class Group Coercions

The class group of a nonmaximal quadratic order R of discriminant $m^2 D_K$, are related to the class group of the maximal order O_K of fundamental discriminant D_K by an exact sequence.

$$1 \to \frac{(O_K/mO_K)^*}{O_K^*\,(\mathbf{Z}/m\mathbf{Z})^*} \to Cl(O) \to Cl(O_K) \to 1$$

Similar maps exist between quadratic orders O_1 and O_2 in a field K, with conductors m_1 and m_2, respectively, such that $m_1 \mid m_2$. The corresponding maps on quadratic forms are implemented on quadratic forms. The homomorphism is returned as a map object, or can be called directly via the coercion operator.

`FundamentalQuotient(Q)`

The quotient homomorphism from the class group of Q to the class group of fundamental discriminant.

`QuotientMap(Q1, Q2)`

Given two structures of quadratic forms Q_1 and Q_2, such that the discriminant of Q_2 equals a square times the discriminant of Q_1, the quotient homomorphism from Q_1 to Q_2 is returned as a map object.

`Q ! f`

The `!` operator applies the quotient homomorphism for automatic coercion of forms f of discriminant $m^2 D$ into the structure Q of forms of discriminant D.

33.7 Discrete Logarithms

`Log(b, x)`

The discrete logarithm of binary quadratic form x with respect to base b, or -1 if MAGMA can that determine no solution exists. This function exists only for negative discriminant forms. Computation is done via the Pohlig-Hellman algorithm along with a collision search subroutine (a variant of Pollard's rho method). If the user is unsure whether a solution exists, it is safest to use `Log` with a time limit (see below) to prevent an infinite loop in the collision search.

`Log(b, x, t)`

Searches for up to t seconds for the discrete logarithm of binary quadratic form x with respect to base b. This function exists only for negative discriminant forms. If Magma is able to determine no solution exists, then -1 will be returned. If no solution is found within the given time frame, then -2 will be returned. Computation is done via the Pohlig-Hellman algorithm along with a collision search subroutine (a variant of Pollard's rho method).

33.8 Elliptic and Modular Invariants

Binary quadratic forms of negative discriminant describe positive definite lattices in the complex plane, with integral-valued inner product. As such, it is possible to apply modular and elliptic functions to the form, interpreting this as an element of the upper half plane.

`Lattice(f)`

Given a binary quadratic form $f = ax^2 + bxy + cy^2$ of negative discriminant, returns the rank two lattice of f having Gram matrix

$$\begin{pmatrix} a & b/2 \\ b/2 & c \end{pmatrix}.$$

Note that the lattice L is the half-integral lattice such that integral representations $f(x, y) = n$ are in bijection with vectors (x, y) of norm n, which will be a rational number.

`GramMatrix(f)`

Returns the Gram matrix of the binary quadratic form f, which need not be of negative discriminant. The matrix will be half-integral and defined over the rationals.

`ThetaSeries(f, n)`

The integral theta series of the binary quadratic form f to precision n.

`RepresentationNumber(f, n)`

The nth representation number of the form f of negative discriminant.

`jInvariant(f)`

For a binary quadratic form $f = ax^2 + bxy + cy^2$ with negative discriminant, return the j–invariant of f, equal to the j–invariant of $\tau = (-b + \sqrt{b^2 - 4ac})/2a$.

`Eisenstein(k, f)`

Given a positive even integer $k = 2n$ and a binary quadratic form $f = ax^2 + bxy + cy^2$, return the value of the Eisenstein series $E_k(L)$ at the complex lattice $L = \langle a, (-b + \sqrt{b^2 - 4ac})/2 \rangle$.

`WeierstrassSeries(z, f)`

Given a complex power series z with positive valuation and a binary quadratic form $f = ax^2 + bxy + cy^2$, returns the q–expansion of the Weierstrass $\wp$-function at the complex lattice $L = \langle a, (-b + \sqrt{b^2 - 4ac})/2 \rangle$.

Example H33E2

```
> Q := QuadraticForms(-163);
> f := PrimeForm(Q,41);
> CC<i> := ComplexField();
> PC<z> := LaurentSeriesRing(CC);
> x := WeierstrassSeries(z,f);
> y := -Derivative(x)/2;
> A := -Eisenstein(4,f)/48;
> B := Eisenstein(6,f)/864;
> Evaluate(y^2 - (x^3 + A*x + B),1/2);
1.384608660824596881000000000 E-26 - 1.305091481190174818000000000 E-26*i
```

33.9 Class Invariants

`HilbertClassPolynomial(D)`

Given a negative discriminant D, returns the Hilbert class polynomial, defined as the minimal polynomial of $j(\tau)$, where $\mathbf{Z}[\tau]$ is an imaginary quadratic order of discriminant D.

`WeberClassPolynomial(D)`

Given a negative discriminant D congruent to 1 modulo 8, returns the Weber class polynomial, defined as the minimal polynomial of $f(\tau)$, where $\mathbf{Z}[\tau]$ is an imaginary quadratic order of discriminant D and f is a particular normalized Weber function generating the same class field as $j(\tau)$. A root $f(\tau)$ of the Weber class polynomial is an integral unit generating the ring class field related to the corresponding root $j(\tau)$ of the Hilbert class polynomial by the expression

$$j(\tau) = \frac{(f(\tau)^{24} - 16)^3}{f(\tau)^{24}},$$

where $\mathrm{GCD}(D, 3) = 1$, and

$$j(\tau) = \frac{(f(\tau)^{8} - 16)^3}{f(\tau)^{8}},$$

if 3 divides D. For further details, consult Yui and Zagier [YZ97].

33.10 Matrix Action on Forms

A matrix in $\mathrm{SL}(2, \mathbf{Z})$ acts on the right on quadratic forms by the rule

$$f(x, y) \begin{pmatrix} r & s \\ t & u \end{pmatrix} = f(rx + sy, tx + uy),$$

which is provided in MAGMA by the operator $*$.

`f * M`

The action of $\mathrm{SL}(2, \mathbf{Z})$ on forms.

33.11 Bibliography

[**BJT97**] J. Buchmann, M. J. Jacobson, Jr., and E. Teske. On Some Computational Problems in Finite Abelian Groups. *Mathematics of Computation*, 66:1663–1687, 1997.

[**CDO93**] H. Cohen, F. Diaz y Diaz, and M. Olivier. Calculs de nombres de classes et de régulateurs de corps quadratiques en temps sous-exponentiel. In *Séeminaire de Théorie des Nombres, Paris, 1990–91*, volume 108 of *Progr. Math.*, pages 35–46. Birkhäuser Boston, Boston, MA, 1993.

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**HM89**] J. Hafner and K. McCurley. A rigorous subexponential algorithm for computation of class groups. *Jornal American Math. Soc.*, 2:837 – 850, 1989.

[**Jac99**] M. J. Jacobson, Jr. Applying sieving to the computation of quadratic class groups. *Math. Comp.*, 68(226):859–867, 1999.

[**Tes98**] E. Teske. A Space Efficient Algorithm for Group Structure Computation. *Mathematics of Computation*, 67:1637–1663, 1998.

[**YZ97**] N. Yui and D. Zagier. On the singular values of Weber modular functions. *Mathematics of Computation*, 66(220):1645–1662, 1997.

PART VI
GLOBAL ARITHMETIC FIELDS

34 NUMBER FIELDS

Chapter 34
NUMBER FIELDS

34.1 Introduction

The number field module in MAGMA is based on the Kant/Kash system (Kant-V4) [KAN97], [KAN00], developed by the group of M. Pohst in Berlin.

This chapter deals only with the basic operations possible with number fields (objects of type `FldNum`) and their elements `FldNumElt`. Apart from material covered here, there is a wealth of other functions implemented and documented in detail in chapters dealing with

* Cyclotomic fields, Chapter 36
* Quadratic fields, Chapter 35
* Orders in number fields, including ideal theory, Chapter 37
* Galois Theory, Chapter 38
* Class Field Theory, Chapter 39

Furthermore, a lot of functionality of number fields is also shared by function fields of transcendence degree 1 (Chapters 42, 41) and functionality is imported from other areas, in particular, finite fields and polynomial rings.

Number fields in MAGMA are finite extensions of the field $\mathbf{Q}$ of rational numbers or of another number field. Number fields of the first kind, ie. number fields that are created as extensions of $\mathbf{Q}$ are referred to as absolute fields, while extensions of number fields are called relative fields.

Number fields support extended types (Section 1.14), they can be indexed by the type of the coefficient ring: `FldNum[FldRat]` refers to an absolute extension over $\mathbf{Q}$, while `FldNum[FldNum]` refers to a relative extension.

Formally, in MAGMA, an object K of type `FldNum` is an algebraic extension of finite degree over a number field k or $\mathbf{Q}$. It should be thought of as constructed as a quotient ring of a univariate polynomial ring over the base field modulo some irreducible polynomial: $K = k[t]/(f(t)k[t])$ Or, the field may be constructed as a multivariate quotient: $K = k[s_1, \ldots, s_n]/(f_1(s_1), \ldots, f_n(s_n))$ where all the polynomials are univariate. However, a slightly different representation is used internally.

An important consequence of this representation as a quotient of a polynomial ring is that one cannot distinguish between e.g. $\mathbf{Q}[2^{(1/3)}]$ and $\mathbf{Q}[\zeta_3 2^{(1/3)}]$ – both of them are generated using a root of $t^3 - 2$. Therefore every non trivial extension generates a **new** object – even if the same polynomial is used repeatedly, except when the user explicitly tells MAGMA to check whether the polynomial has been used before. This also implies that number fields are not automatically embedded into $\mathbf{C}$ the field of complex numbers, in contrast to both quadratic and cyclotomic fields that are. Using `Conjugates` or the language of places in section 34.8 all embeddings into $\mathbf{C}$ or $\mathbf{R}$ can be found and used.

It is important to remember that **Q** is **not** a number field. However, it is possible to create an extension of degree 1 over **Q** that is a number field using, for example

Example H34E1

```
> K := ext<Rationals()|Polynomial([0,1]):DoLinearExtension>;
```

One has to distinguish between number fields with a (known) primitive element α`:= K.1` which is a zero of f and number fields where no primitive element is known. In this case α_i`:= K.i` will be a zero of f_i.

Number fields always have a 'power' basis, i.e. a basis containing only powers of the zero(s) of the defining polynomial(s) and product of those powers. This allows for example to define homomorphisms from number fields by specifying images of the generators only.

An absolute extension is always an extension of **Q**. An arbitrary number field K can always be converted into an isomorphic extension of **Q** using a constructive variant of the primitive element theorem, `AbsoluteField`. Similarly, a number field defined by multiple polynomials can be converted into a field defined by a single polynomial using `SimpleExtension`.

Likewise, if a subfield k of K is known, an isomorphic field as an extension of k can be computed.

The most important facts about the different representations are the following:

* Arithmetic is fastest in absolute simple extensions. Thus, if one wants to do lots of basic arithmetic with the elements the transformation to an absolute representation is advisable. However, typically the operations are fastest when the elements are "small" in size.
* Invariants (like `Degree`, `Discriminant`, `Norm`, `Trace` etc.) are always relative to the current representation.
* Conversions of fields tend to be time consuming thus should be avoided if possible. However, once the different field representations are computed, the conversion of elements is not too time consuming.
* Some operations and invariants can (currently) only be done for absolute representations. Essentially, these are computations involving subfields and class and unit group computations.

Number fields support only arithmetic with their elements and the computation of some invariants (`GaloisGroup`, `Subfields`, `AutomorphismGroup`). Although invariants like the class group can be computed for `FldNum`s this is only a shortcut for the corresponding computations for the maximal orders so e.g. `ClassGroup(K)` is expanded to `ClassGroup(MaximalOrder(K))`.

34.2 Creation Functions

The following describes how number fields may be created. It also shows some ways of creating elements of these rings and homomorphisms from these rings into an arbitrary ring.

34.2.1 Creation of Number Fields

Algebraic Number Fields can be created in a various ways, most of which involve polynomials. The fields can be created as absolute extensions, i.e. an extension of **Q** by one or more irreducible polynomial(s), or as a relative extension which is an extension of an algebraic field by one or more polynomial(s) irreducible over that field.

`NumberField(f)`

`Check`	BoolElt	*Default :* `true`
`DoLinearExtension`	BoolElt	*Default :* `false`
`Global`	BoolElt	*Default :* `false`

Given an irreducible polynomial f of degree $n \geq 1$ over $K = \mathbf{Q}$ or some number field K, create the number field $L = K(\alpha)$ obtained by adjoining a root α of f to K.

The polynomial f is allowed to have either integer coefficients, coefficients in an order of K, coefficients from the rational field or some algebraic field K. The field K will be referred to as `CoefficientField`. If the polynomial is defined over a field and the coefficients have denominators greater than 1, an equivalent polynomial $df(x)$ is used to define L, where d is the least common multiple of the denominators of the coefficients of f.

If the optional parameter `Check` is set to `false` then the polynomial is not checked for irreducibility. This is useful when building relative extensions where factoring can be time consuming.

If `DoLinearExtension` is `true` and the degree of f is 1 a trivial extension is returned. This is an object of type `FldNum` but of degree 1. Otherwise (or by default), the coefficient field of f is returned. (This is important in situations where the number of extensions matters.) Furthermore, a degree 1 extension of **Q** is a field isomorphic to **Q**, but regarded by Magma as a number field (while **Q** itself is not, since `FldRat` is not a subtype of `FldNum`). This then supports all of the number field functions (including for instance fractional ideals) while the `Rationals()` do not. On the other hand, arithmetic will be slower.

If `Global` is `true`, then Magma checks if this polynomial is the defining polynomial of some other field created using `Global := true`. In this case, the old field will be returned.

The angle bracket notation may be used to assign the root α to an identifier e.g. `L<y> := NumberField(f)` where y will be a root of f.

`RationalsAsNumberField()`

`QNF()`

This creates a number field isomorphic to **Q**. It is equivalent to `NumberField(x-1 : DoLinearExtension)`, where x is `PolynomialRing(Rationals()).1`.

The result is a field isomorphic to **Q**, but regarded by MAGMA as a number field (while **Q** itself is not, since `FldRat` is not a subtype of `FldNum`). It therefore supports all of the number field functions, while the `Rationals()` do not. On the other hand, arithmetic will be slower.

Coercion can be used to convert to and from the `Rationals()`.

`NumberField(s)`

`Check`	BOOLELT	*Default :* `true`
`DoLinearExtension`	BOOLELT	*Default :* `false`
`Abs`	BOOLELT	*Default :* `false`

Let K be a possibly trivial algebraic extension of **Q**. K will be referred to as the `CoefficientField`.

Given a sequence s of nonconstant polynomials $s_1, \ldots, s_m$, that are irreducible over K, create the number field $L = K(\alpha_1, \ldots, \alpha_m)$ obtained by adjoining a root α_i of each s_i to K. The polynomials s_i are allowed to have coefficients in an order of K (or **Z**) or in K or a suitable field of fractions, but if in the latter cases denominators occur in the coefficients of s_i, an integral polynomial is used instead of s_i, as in the case of the definition of a number field by a single polynomial.

If $m > 1$ and `Abs` is `false`, a tower of extension fields

$$L_0 = K \subset L_1 = K(\alpha_m) \subset L_2 = K(\alpha_{m-1}, \alpha_m) \subset \cdots \subset L_m = K(\alpha_1, \ldots, \alpha_m) = L$$

is created, and L is a relative extension by s_1 over its ground field $L_{m-1} = K(\alpha_2, \ldots, \alpha_m)$. Thus, this construction has the same effect as m applications of the `ext` constructor. The angle bracket notation may be used to assign the m generators α_i to identifiers: `L<a`$_1$`, ..., a`$_m$`> := NumberField([ s`$_1$`, ..., s`$_m$ `]);` thus the first generator a_1, which corresponds to `L.1`, generates L over its ground field.

Note that it is important to ensure that in each of the above steps the polynomial s_i is irreducible over L_{i-1}; by default MAGMA will check that this is the case. If the optional parameter `Check` is set to `false` then this checking will not be done.

If the optional parameter `Abs` is changed to `true`, then a non-simple extension will be returned. This is a extension of the coefficient field of the f_i but such that the base field of L will be K. The ith generator will be a root of the ith polynomial in this case, but all of the generators will have L as parent. In this case, a sparse representation of number field elements will be used (based on multivariate polynomial rings). As a consequence, costs for arithmetic operations will (mainly) depend on the number of non-zero coefficients of the elements involved rather than the field

degree. This allows to define and work in fields of degree $< 10^6$. However, for general elements this representation is slower than the dense (default) representation.

If the optional parameter `DoLinearExtension` is set to `true`, linear polynomials will not be removed from the list.

```
ext<  F | s1, ..., sn  >
```

```
ext< F | s >
```

`Check`	BoolElt	*Default :* `true`
`Global`	BoolElt	*Default :* `false`
`Abs`	BoolElt	*Default :* `false`
`DoLinearExtension`	BoolElt	*Default :* `false`

Construct the number field defined by extending the number field F by the polynomials s_i or the polynomials in the sequence s. Similar as for `NumberField(S)` described above, F may be $\mathbf{Q}$. A tower of fields similar to that of `NumberField` is created and the same restrictions as for that function apply to the polynomials that can be used in the constructor.

Example H34E2

To create the number field $\mathbf{Q}(\alpha)$, where α is a zero of the integer polynomial $x^4 - 420x^2 + 40000$, one may proceed as follows:

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> Degree(K);
> K;
Number Field with defining polynomial x^4 - 420*x^2 + 40000 over the Rational
Field
> y^4 - 420*y^2;
-40000
```

By assigning the generating element to y, we can from here on specify elements in the field as polynomials in y. The elements will always be printed as polynomials in $\mathbf{Q}[y]/f$:

```
> z := y^5/11;
> z;
1/11*(420*y^3 - 40000*y)
```

K can be further extended by the use of either `ext` or `NumberField`.

```
> R<y> := PolynomialRing(K);
> f := y^2 + y + 1;
> L := ext<K | f>;
> L;
Number Field with defining polynomial y^2 + y + 1 over K
```

This is equivalent to

```
> KL := NumberField([x^2 + x + 1, x^4 - 420*x^2 + 40000]);
```

```
> KL;
Number Field with defining polynomial $.1^2 + $.1 + 1 over its ground field
```

but different to

```
> LK := NumberField([x^4 - 420*x^2 + 40000, x^2 + x + 1]);
> LK;
Number Field with defining polynomial $.1^4 - 420*$.1^2 + 40000 over its ground
field
```

To illustrate the use of `Global`:

```
> K1 := NumberField(x^3-2 : Global);
> K2 := NumberField(x^3-2 : Global);
> L1 := NumberField(x^3-2);
> L2 := NumberField(x^3-2);
> K1 eq K2;
true
> K1 eq L1;
false
> L1 eq L2;
false;
> K1!K2.1;
K1.1;
>  K2!K1.1;
K1.1
>> L1!L2.1;
     ^
Runtime error in '!': Arguments are not compatible
LHS: FldNum
RHS: FldNumElt
```

A typical application of `DoLinearExtension` is as follows. To construct a Kummer extension of degree p, one has to start with a field containing the p-th roots of unity. In most situation this will be a field extension of degree $p-1$, but what happens if ζ_p is already in the base field?

```
> AdjoinRoot := function(K, p: DoLinearExtension := false)
>   f := CyclotomicPolynomial(p);
>   f := Polynomial(K, f);
>   f := Factorisation(f)[1][1];
>   return ext<K|f : DoLinearExtension := DoLinearExtension>;
> end function;
> K := NumberField(x^2+x+1);
> E1 := AdjoinRoot(K, 3);
> E1;
Number Field with defining polynomial x^2 + x + 1 over the
Rational Field
> E2 := AdjoinRoot(K, 3 : DoLinearExtension);
> E2;
Number Field with defining polynomial ext<K|>.1 - K.1 over
K
```

```
> Norm(E1.1);
1
> Norm(E2.1);
K.1
> Norm($1);
1
```

`RadicalExtension(F, d, a)`

`Check`	BoolElt	*Default :* `true`

Let F be a number field. Let a be an integral element of F chosen such that a is not an n-th power for any n dividing d. Returns the number field obtained by adjoining the d-th root of a to F.

`SplittingField(F)`

`Abs`	BoolElt	*Default :* `true`
`Opt`	BoolElt	*Default :* `true`

Given a number field F, return the splitting field of its defining polynomial. The roots of the defining polynomial in the splitting field are also returned.

If `Abs` is `true`, the resulting field will be an absolute extension, otherwise a tower is returned.

If `Opt` is `true`, an attempt of using `OptimizedRepresentation` is done. If successful, the resulting field will have a much nicer representation. On the other hand, computing the intermediate maximal orders can be extremely time consuming.

`SplittingField(f)`

Given an irreducible polynomial f over $\mathbf{Z}$, return its splitting field.

`SplittingField(L)`

`Abs`	BoolElt	*Default :* `false`
`Opt`	BoolElt	*Default :* `false`

Given a sequence L of polynomials over a number field or the rational numbers, compute a common splitting field, ie. a field K such that every polynomial in L splits into linear factors over K. The roots of the polynomials are returned as the second return value.

If the optional parameter `Abs` is `true`, then a primitive element for the splitting field is computed and the field returned will be generated by this primitive element over $\mathbf{Q}$. If in addition `Opt` is also `true`, then an optimized representation of K is computed as well.

`sub< F | e1, ..., en >`

Given a number field F with coefficient field G and n elements $e_i \in F$, return the number field $H = G(e_1, \ldots, e_n)$ generated by the e_i (over G), as well as the embedding homomorphism from H to F.

`MergeFields(F, L)`

`CompositeFields(F, L)`

Let F and L be absolute number fields. Returns a sequence of fields $[M_1, \ldots, M_r]$ such that each field M_i contains both a root of the generating polynomial of F and a root of the generating polynomial of L.

In detail: Suppose that F is the smaller field (wrt. the degree). As a first step we factorise the defining polynomial of L over F. For each factor obtained, an extension of F is constructed and then transformed into an absolute extension. The sequence of extension fields is returned to the user.

`Compositum(K, L)`

For absolute number fields K and L, at least one of which must be normal, find a smallest common over field. Note that in contrast to `CompositeFields` above the result here is essentially unique since one field was normal.

`quo< FldNum : R | f >`

`Check` BoolElt *Default :* `true`

Given a ring of polynomials R in one variable over a number field K, create the number field $K(\alpha)$ obtained by adjoining a root α of f to K. Here the coefficient ring K of R is allowed to be the rational field $\mathbf{Q}$. The polynomial f is allowed to have coefficients in K, but if coefficients occur in f which require denominator greater than 1 when expressed on the basis of K, the polynomial will be replaced by an equivalent one requiring no such denominators: $\tilde{f}(x) = df(x)$, where d is a common denominator. The parameter `Check` determines whether the polynomial is checked for irreducibility.

The angle bracket notation may be used to assign the root α to an identifier: `K<y> := quo< FldNum : R | f >`.

If the category `FldNum` is not specified, `quo< R | f >` creates the quotient ring R/f as a generic ring (not as a number field), in which only elementary arithmetic is possible.

Example H34E3

To illustrate the use of `CompositeFields` we will use this function to compute the normal closure of $\mathbf{Q}(\alpha)$ where α is a zero of the integer polynomial $x^3 - 2$:

```
> K := RadicalExtension(Rationals(), 3, 2);
> l := CompositeFields(K, K);
> l;
[
```

```
    Number Field with defining polynomial $.1^3 - 2 over the Rational
    Field,
    Number Field with defining polynomial $.1^6 + 108 over the Rational
    Field
]
```

The second element of l corresponds to the smallest field L_2 containing two distinct roots of x^3-2. Since the degree of K is 3, L_2 is the splitting field of f and therefore the normal closure of K.

`OptimizedRepresentation(F)`

`OptimisedRepresentation(F)`

Given a number field F with ground field $\mathbf{Q}$, this function will attempt to find an isomorphic field L with a better defining polynomial than the one used to define F. If such a polynomial is found then L is returned; otherwise F will be returned. For more details, please refer to `OptimizedRepresentation`.

Example H34E4

Some results of `OptimizedRepresentation` are shown.

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^4-420*x^2+40000);
> L := OptimizedRepresentation(K);
> L ne K;
true
> L;
Number Field with defining polynomial x^4 - 4*x^3 -
    17*x^2 + 42*x + 59 over the Rational Field
> L eq OptimizedRepresentation(L);
```

34.2.2 Maximal Orders

The maximal order $\mathcal{O}_K$ is the ring of integers of an algebraic field consisting of all integral elements of the field; that is, elements which are roots of monic integer polynomials. It may also be called the number ring of a number field. It is arguably the single most important invariant of a number field, in fact in number theory when one talks about units, ideals, etc. of number fields, it is typically implied that the maximal order is the underlying ring.

Maximal orders and orders in general are explained in detail in the chapter 37, here we only give a very brief overview.

There are a number of algorithms which MAGMA uses whilst computing maximal orders. The main ones are the Round–2 and the Round–4 methods ([Coh93, Bai96, Poh93, PZ89] for absolute extensions and [Coh00, Fri97, Pau01b] for relative extensions).

`MaximalOrder(F)`
`IntegerRing(F)`
`Integers(F)`
`RingOfIntegers(F)`

`Al`	MonStgElt	*Default :* "*Auto*"
Verbose	`MaximalOrder`	*Maximum :* 5

Create the ring of integers of the algebraic number field F. An integral basis for F can be found as the basis of the maximal order.

For information on the parameters, see Section 37.2.3.

34.2.3 Creation of Elements

Since number fields are though of as quotients of (multivariate) polynomial rings, elements in those fields are represented as (multivariate) polynomials in the generator(s) of the field.

`F ! a`
`elt< F | a >`

Coerce a into the number field F. Here a may be an integer or a rational field element, or an element from a subfield of F, or from an order in such or any other field related to F through chains of subfields, optimised representation, absolute fields, etc.

`F ! [`a_0`, `a_1`, ..., `a_{m-1}`]`
`elt< F | [ `a_0`, `a_1`, ..., `a_{m-1}` ] >`
`elt< F | `a_0`, `a_1`, ..., `a_{m-1}` >`

Given the number field, F of degree m over its ground field G and a sequence $[a_0, \ldots, a_{m-1}]$ of elements of G, construct the element $a_0\alpha_0 + a_1\alpha_1 + \cdots a_{m-1}\alpha_{m-1}$ of F where the α_i are the basis elements of F. In case F was generated by a root of a single polynomial, we will always have $\alpha_i = \mathbf{F}.\mathbf{1}^i$. If F was defined using multiple polynomials and the `Abs` parameter, the basis will consist of products of powers of the generators.

`Random(F, m)`

A random element of the number field F. The maximal size of the coefficients is determined by the integer m.

Example H34E5

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> y^6;
136400*y^2 - 16800000
```

```
> K![-16800000, 0, 136400, 0];
136400*y^2 - 16800000
> K := NumberField([x^3-2, x^2-5]:Abs);
> Basis(K);
[
    1,
    K.1,
    K.1^2,
    K.2,
    K.1*K.2,
    K.1^2*K.2
]
> K![1,2,3,4,5,6];
6*K.1^2*K.2 + 3*K.1^2 + 5*K.1*K.2 + 2*K.1 + 4*K.2 + 1
```

One(K)	Identity(K)
Zero(K)	Representative(K)

34.2.4 Creation of Homomorphisms

To specify homomorphisms from number fields, it is necessary to specify the image of the generating elements, and possible to specify a map on the coefficient field.

hom< F -> R | r >
hom< F -> R | h, r >
hom< F -> R | r >
hom< F -> R | h, r >

Given an algebraic number field F, defined as an extension of the coefficient field G, as well as some ring R, build the homomorphism ϕ obtained by sending the defining primitive element α of F to the element $r \in R$.

In case the field F was defined using multiple polynomials, instead of an image for the primitive element, one has to give images for each of the generators.

It is possible (if $G = \mathbf{Q}$) and sometimes necessary (if $G \neq \mathbf{Q}$) to specify a homomorphism ϕ on F by specifying its action on G by providing a homomorphism h with G as its domain and R its codomain together with the image of α. If R does not cover G then the homomorphism h from G into R is necessary to ensure that the ground field can be mapped into R.

Example H34E6

We show a way to embed the field $\mathbf{Q}(\sqrt{2})$ in $\mathbf{Q}(\sqrt{2}+\sqrt{3})$. The application of the homomorphism suggests how the image could have been chosen.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^2-2);
> KL<w> := NumberField(x^4-10*x^2+1);
> H := hom< K -> KL | (9*w-w^3)/2 >;
> H(y);
1/2*(-w^3 + 9*w)
> H(y)^2;
2
```

34.3 Structure Operations

In the lists below K always denotes a number field.

34.3.1 General Functions

Number fields form the MAGMA category FldNum. The notional power structures exist as parents of algebraic fields with no operations are allowed.

`Category(K)` `Type(K)` `ExtendedType(K)` `Parent(K)`

`AssignNames(~K, s)`

Procedure to change the names of the generating elements in the number field K to the contents of the sequence of strings s.

The i-th sequence element will be the name used for the generator of the $(i-1)$-st subfield down from K as determined by the creation of K, the first element being used as the name for the generator of K. In the case where K is defined by more than one polynomial as an absolute extension, the ith sequence element will be the name used for the root of the ith polynomial used in the creation of K.

This procedure only changes the names used in printing the elements of K. It does *not* assign to any identifiers the value of a generator in K; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies K, it is necessary to have a reference ~K to K in the call to this function.

`Name(K, i)`

`K . i`

Given a number field K, return the element which has the i-th name attached to it, that is, the generator of the $(i-1)$-st subfield down from K as determined by the creation of K. Here i must be in the range $1 \leq i \leq m$, where m is the number of polynomials used in creating K. If K was created using multiple polynomials as an absolute extension, K.i will be a root of the ith polynomial used in creating K.

34.3.2 Related Structures

Each number field has other structures related to it in various ways.

`GroundField(F)`
`BaseField(F)`
`CoefficientField(F)`
`CoefficientRing(F)`

Given a number field F, return the number field over which F was defined. For an absolute number field F, the function returns the rational field $\mathbf{Q}$.

`AbsoluteField(F)`

Given a number field F, this returns an isomorphic number field L defined as an absolute extension (i.e. over $\mathbf{Q}$). (For algorithm, see [Tra76])

`SimpleExtension(F)`

Given a number field F or an order O, this returns an isomorphic field L defined as an absolute simple extension. (For algorithm, see [Tra76])

`RelativeField(F, L)`

Given number fields L and F such that MAGMA knows that F is a subfield of L, return an isomorphic number field M defined as an extension over F.

Example H34E7

It is often desirable to build up a number field by adjoining several algebraic numbers to $\mathbf{Q}$. The following function returns a number field that is the composite field of two given number fields K and L, provided that $K \cap L = \mathbf{Q}$; if K and L have a common subfield larger than $\mathbf{Q}$ the function returns a field with the property that it contains a subfield isomorphic to K as well as a subfield isomorphic to L.

```
> R<x> := PolynomialRing(Integers());
> Composite := function( K, L )
>     T<y> := PolynomialRing( K );
>     f := T!DefiningPolynomial( L );
>     ff := Factorization(f);
>     LKM := NumberField(ff[1][1]);
>     return AbsoluteField(LKM);
> end function;
```

To create, for example, the field $\mathbf{Q}(\sqrt{2}, \sqrt{3}, \sqrt{5})$, the above function should be applied twice:

```
> K := NumberField(x^2-3);
> L := NumberField(x^2-2);
> M := NumberField(x^2-5);
> KL := Composite(K, L);
> S<s> := PolynomialRing(BaseField(KL));
> KLM<w> := Composite(KL, M);
```

```
> KLM;
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

Note, that the same field may be constructed with just one call to NumberField followed by AbsoluteField:

```
> KLM2 := AbsoluteField(NumberField([x^2-3, x^2-2, x^2-5]));
> KLM2;
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

or by

```
> AbsoluteField(ext<Rationals() | [x^2-3, x^2-2, x^2-5]>);
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

In general, however, the resulting polynomials of KLM and $KLM2$ will differ. To see the difference between SimpleExtension and AbsoluteField, we will create KLM2 again:

```
> KLM3 := NumberField([x^2-3, x^2-2, x^2-5]: Abs);
> AbsoluteField(KLM3);
Number Field with defining polynomials [ x^2 - 3, x^2 - 2,
    x^2 - 5] over the Rational Field
> SimpleExtension(KLM3);
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

PrimeRing(F) PrimeField(F)

Centre(F)

Embed(F, L, a)

Install the embedding of a simple number field F in L where the image of the primitive element of F is the element a of L. This embedding will be used in coercing from F into L.

Embed(F, L, a)

Install the embedding of the non-simple number field F in L where the image of the generating elements of F are in the sequence a of elements of L. This embedding will be used in coercing from F into L.

EmbeddingMap(F, L)

Returns the embedding map of the number field F in L if an embedding is known.

Example H34E8

MAGMA does not recognize two independently created number fields as equal since more than one embedding of a field in a larger field may be possible. To coerce between them, it is convenient to be able to embed them in each other.

```
> k := NumberField(x^2-2);
> l := NumberField(x^2-2);
>  l!k.1;
>> l!k.1;
    ^
Runtime error in '!': Arguments are not compatible
LHS: FldNum
RHS: FldNumElt
> l eq k;
false
> Embed(k, l, l.1);
> l!k.1;
l.1
> Embed(l, k, k.1);
> k!l.1;
k.1
```

`Embed` is useful in specifying the embedding of a field in a larger field.

```
> l<a> := NumberField(x^3-2);
> L<b> := NumberField(x^6+108);
> Root(L!2, 3);
1/18*b^4
> Embed(l, L, $1);
> L!l.1;
1/18*b^4
```

Another embedding would be

```
> Roots(PolynomialRing(L)!DefiningPolynomial(l));
[
    <1/36*(-b^4 - 18*b), 1>,
    <1/36*(-b^4 + 18*b), 1>,
    <1/18*b^4, 1>
]
> Embed(l, L, $1[1][1]);
> L!l.1;
1/36*(-b^4 - 18*b)
```

`MinkowskiSpace(F)`

The Minkowski vector space V of the absolute number field F as a real vector space, with inner product given by the T_2-norm (`Length`) on F, and by the embedding $F \to V$.

```
Completion(K, P)
comp< K|P >
```

Precision RngIntElt *Default :* 20

For an absolute extension K of $\mathbf{Q}$, compute the completion at a prime ideal P which must be either a prime ideal of the maximal order or unramified. The result will be a local field or ring with relative precision `Precision`.

The returned map is the canonical injection into the completion. It allows pointwise inverse operations.

```
Completion(K, P)
```

Precision RngIntElt *Default :* 20

For an absolute extension K over $\mathbf{Q}$ and a (finite) place P, compute the completion at P. The precision and the map are as described for `Completion`.

34.3.3 Representing Fields as Vector Spaces

It is possible to express a number field as a vector space of any subfield using the intrinsics below. Such a construction also allows one to find properties of elements over these subfields.

```
Algebra(K, J)
Algebra(K, J, S)
```

Returns the associative structure constant algebra which is isomorphic to the number field K as an algebra over J. Also returns the isomorphism from K to the algebra mapping w^i to the $i+1$st unit vector of the algebra where w is a primitive element of K.

If a sequence S is given it is taken to be a basis of K over J and the isomorphism will map the ith element of S to the ith unit vector of the algebra.

```
VectorSpace(K, J)
KSpace(K, J)
VectorSpace(K, J, S)
KSpace(K, J, S)
```

The vector space isomorphic to the number field K as a vector space over J and the isomorphism from K to the vector space. The isomorphism maps w^i to the $i+1$st unit vector of the vector space where w is a primitive element of K.

If S is given, the isomorphism will map the ith element of S to the ith unit vector of the vector space.

Example H34E9

We use the `Algebra` of a relative number field to obtain the minimal polynomial of an element over a subfield which is not in its coefficient field tower.

```
> K := NumberField([x^2 - 2, x^2 - 3, x^2 - 7]);
> J := AbsoluteField(NumberField([x^2 - 2, x^2 - 7]));
> A, m := Algebra(K, J);
> A;
Associative Algebra of dimension 2 with base ring J
> m;
Mapping from: RngOrd: K to AlgAss: A
> m(K.1);
(1/10*(J.1^3 - 13*J.1)                        0)
> m(K.1^2);
(2 0)
> m(K.2);
(1/470*(83*J.1^3 + 125*J.1^2 - 1419*J.1 - 1735) 1/940*(-24*J.1^3 - 5*J.1^2 +
    382*J.1 + 295))
> m(K.2^2);
(3 0)
> m(K.3);
(1/10*(-J.1^3 + 23*J.1)                         0)
> m(K.3^2);
(7 0)
> A.1 @@ m;
1
> A.2 @@ m;
(($.1 - 1)*$.1 - $.1 - 1)*K.1 + ($.1 + 1)*$.1 + $.1 + 1
>
> r := 5*K.1 - 8*K.2 + K.3;
> m(r);
(1/235*(-238*J.1^3 - 500*J.1^2 + 4689*J.1 + 6940) 1/235*(48*J.1^3 + 10*J.1^2 -
    764*J.1 - 590))
> MinimalPolynomial($1);
$.1^2 + 1/5*(-4*J.1^3 + 42*J.1)*$.1 + 5*J.1^2 - 180
> Evaluate($1, r);
0
> K:Maximal;
  K
  |
  |
  $1
  |
  |
  $2
  |
  |
  Q
```

```
K  : $.1^2 - 2
$1 : $.1^2 - 3
$2 : x^2 - 7
> Parent($3);
Univariate Polynomial Ring over J
> J;
Number Field with defining polynomial $.1^4 - 18*$.1^2 + 25 over the Rational
Field
```

34.3.4 Invariants

Some information describing a number field can be retrieved.

`Characteristic(F)`

`Degree(F)`

Given a number field F, return the degree $[F : G]$ of F over its ground field G.

`AbsoluteDegree(F)`

Given a number field F, return the absolute degree of F over $\mathbf{Q}$.

`Discriminant(F)`

Given an extension F of $\mathbf{Q}$, return the discriminant of F. This discriminant is defined to be the discriminant of the defining polynomial, **not** as the discriminant of the maximal order.

The discriminant in a relative extension F is the ideal in the base ring generated by the discriminant of the defining polynomial.

`AbsoluteDiscriminant(K)`

Given a number field K, return the absolute value of the discriminant of K regarded as an extension of $\mathbf{Q}$.

`Regulator(K)`

Given a number field K, return the regulator of K as a real number. Note that this will trigger the computation of the maximal order and its unit group if they are not known yet. This only works in an absolute extension.

`RegulatorLowerBound(K)`

Given a number field K, return a lower bound on the regulator of O or K. This only works in an absolute extension.

`Signature(F)`

Given an absolute number field F, returns two integers, one being the number of real embeddings, the other the number of pairs of complex embeddings of F.

`UnitRank(K)`

The unit rank of the number field K (one less than the number of real embeddings plus number of pairs of complex embeddings).

`DefiningPolynomial(F)`

Given a number field F, the polynomial defining F as an extension of its ground field G is returned.

For non simple extensions, this will return a list of polynomials.

`Zeroes(F, n)`

Given an absolute number field F, and an integer n, return the zeroes of the defining polynomial of F with a precision of exactly n decimal digits. The function returns a sequence of length the degree of F; all of the real zeroes appear before the complex zeroes.

Example H34E10

The information provided by `Zeros` and `DefiningPolynomial` is illustrated below.

```
> L := NumberField(x^6+108);
> DefiningPolynomial(L);
x^6 + 108
> Zeros(L, 30);
[ 1.889881574842309747150815910899999999994 +
1.091123635971721403560072614199999999977*i,
1.889881574842309747150815910899999999994 -
1.091123635971721403560072614199999999977*i,  0.E-29 +
2.182247271943442807120145228399999999955*i,  0.E-29 -
2.182247271943442807120145228399999999955*i,
-1.889881574842309747150815910899999999994 +
1.091123635971721403560072614199999999977*i,
-1.889881574842309747150815910899999999994 -
1.091123635971721403560072614199999999977*i ]
> l := NumberField(x^3 - 2);
> DefiningPolynomial(l);
x^3 - 2
> Zeros(l, 30);
[ 1.259921049894873164767210607299999999994,
-0.629960524947436582383605303639109999999 +
1.091123635971721403560072614199999999977*i,
-0.629960524947436582383605303639109999999 -
1.091123635971721403560072614199999999977*i ]
```

34.3.5 Basis Representation

The basis of a number field can be expressed using elements from any compatible ring.

`Basis(F)`

`Basis(F, R)`

Return the current basis for the number field F over its ground ring as a sequence of elements of F or as a sequence of elements of R.

`IntegralBasis(F)`

`IntegralBasis(F, R)`

An integral basis for the algebraic number field F is returned as a sequence of elements of F or R if given. This is the same as the basis for the maximal order. Note that the maximal order will be determined (and stored) if necessary.

Example H34E11

The following illustrates how a basis can look different when expressed in a different ring.

```
> f := x^5 + 5*x^4 - 75*x^3 + 250*x^2 + 65625;
> N := NumberField(f);
> N;
Number Field with defining polynomial x^5 + 5*x^4 - 75*x^3 + 250*x^2 + 65625
over the Rational Field
> Basis(N);
[
    1,
    N.1,
    N.1^2,
    N.1^3,
    N.1^4
]
> IntegralBasis(N);
[
    1,
    1/5*N.1,
    1/25*N.1^2,
    1/125*N.1^3,
    1/625*N.1^4
]
> IntegralBasis(N, MaximalOrder(N));
[
    [1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 1]
```

```
]
```

`AbsoluteBasis(K)`

Returns an absolute basis for the number field K, i.e. a basis for K as a $\mathbf{Q}$ vector space. The basis will consist of the products of the basis elements of the intermediate fields. The expansion is done depth-first.

Example H34E12

We continue our example of a field of degree 4.
The functions `Basis` and `IntegralBasis` both return a sequence of elements, that can be accessed using the operators for enumerated sequences. Note that if, as in our example, O is the maximal order of K, both functions produce the same output:

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> O := MaximalOrder(K);
> I := IntegralBasis(K);
> B := Basis(O);
> I, B;
[
    1,
    1/2*y,
    1/40*(y^2 + 10*y),
    1/800*(y^3 + 180*y + 400)
]
[
    0.1,
    0.2,
    0.3,
    0.4
]
> Basis(O, K);
[
    1,
    1/2*y,
    1/40*(y^2 + 10*y),
    1/800*(y^3 + 180*y + 400)
]
```

34.3.6 Ring Predicates

Number fields can be tested for having several properties that may hold for general rings.

`F eq L`

Returns `true` if and only if the number fields F and L are indentical.

No two number fields which have been created independently of each other will be considered equal since it is possible that they can be embedded into a larger field in more than one way.

`IsCommutative(R)` `IsUnitary(R)` `IsFinite(R)`
`IsOrdered(R)` `IsField(R)`
`IsNumberField(R)` `IsAlgebraicField(R)`

`IsEuclideanDomain(F)`

This is not a check for euclidean number fields. This function will always return `true`, as all number fields are euclidean domains.

`IsSimple(F)`

Checks if the number field F is defined as a simple extension over the base ring.

`IsPID(F)` `IsUFD(F)`

`IsPrincipalIdealRing(F)`

Always `true` for number fields.

`IsDomain(R)`
`F ne L` `K subset L`

`HasComplexConjugate(K)`

This function returns `true` if there is an automorphism in the number field K that acts like complex conjugation.

`ComplexConjugate(x)`

For an element x of a number field K where `HasComplexConjugate` returns `true` (in particular this includes totally real fields, cyclotomic and quadratic fields and CM-extensions), the conjugate of x is returned.

34.3.7 Field Predicates

Here all the predicates that are specific to number fields are listed.

`IsIsomorphic(F, L)`

Given two number fields F and L, this returns `true` as well as an isomorphism $F \to L$, if F and L are isomorphic, and it returns `false` otherwise.

`IsSubfield(F, L)`

Given two number fields F and L, this returns `true` as well as an embedding $F \hookrightarrow L$, if F is a subfield of L, and it returns `false` otherwise.

`IsNormal(F)`

Returns `true` if and only if the number field F is a normal extension. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois group computation.

`IsAbelian(F)`

Returns `true` if and only if the number field F is a normal extension with abelian Galois group. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois Group computation.

`IsCyclic(F)`

Returns `true` if and only if the number field F is a normal extension with cyclic Galois group. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois and automorphism group.

`IsAbsoluteField(K)`

Returns `true` iff the number field K is a constructed as an absolute extension of $\mathbf{Q}$.

34.4 Element Operations

34.4.1 Parent and Category

`Parent(a)` `Category(a)` `Type(a)` `ExtendedType(a)`

34.4.2 Arithmetic

The table below lists the generic arithmetic functions on number field elements. Note that automatic coercion ensures that the binary operations `+`, `-`, `*`, and `/` may be applied to an element of a number field and an element of one of its orders; the result will be a number field element.

`+ a` `- a`

`a + b` `a - b` `a * b` `a / b` `a ^ k`

`Sqrt(a)`

Returns the square root of the number field element a if it exists in the field containing a.

`Root(a, n)`

Returns the n-th root of the number field element a if it exists in the field containing a.

`IsSquare(a)`

Return `true` if the number field element a is a kth power, (respectively square) and the root if so.

`Denominator(a)`

Returns the denominator of the number field element a, that is the least common multiple of the denominators of the coefficients of a.

`Numerator(a)`

Returns the numerator of the number field element a, that is the element multiplied by its denominator.

`Qround(E, M)`

`ContFrac` BoolElt *Default :* `true`

Finds an approximation of the number field element E where the denominator is bounded by the integer M. If `ContFrac` is `true`, the approximation is computed by applying the continued fraction algorithm to the coefficients of E viewed over Q.

34.4.3 Equality and Membership

Elements may also be tested for whether they lie in an ideal of an order. See Section 37.9.5.

`a eq b` `a ne b`

`a in F`

34.4.4 Predicates on Elements

In addition to the generic predicates `IsMinusOne`, `IsZero` and `IsOne`, the predicates `IsIntegral` and `IsPrimitive` are defined on elements of number fields.

`IsIntegral(a)`

Returns `true` if the element a of a number field F is contained in the ring of integers of F, `false` otherwise. We use the minimal polynomial to determine the answer, which means that the calculation of the maximal order is *not* triggered if it is not known yet. A denominator d such that $d * a$ is integral is also returned on request.

`IsPrimitive(a)`

Returns `true` if the element a of the number field F generates F over its coefficient field.

`IsTotallyPositive(a)`

Returnes `true` iff all real embeddings of the number field element a are positive. For elements in absolute fields this is equivalent to all real conjugates being positive.

`IsZero(a)` `IsOne(a)`
`IsMinusOne(a)` `IsUnit(a)`
`IsNilpotent(a)` `IsIdempotent(a)`
`IsZeroDivisor(a)` `IsRegular(a)`
`IsIrreducible(a)` `IsPrime(a)`

34.4.5 Finding Special Elements

Generators of fields can be retrieved.

`K . 1`

Return the image α of x in $G[x]/f$ where f is the first defining polynomial of the number field K and G is the base field of K.

In case of simple extensions this will be a primitive element.

`PrimitiveElement(K)`

Returns a primitive element for the simple number field K, that is an element whose minimal polynomial has the same degree as the field. For a simple number field K this is $K.1$, while for non-simple fields a random element with this property is returned.

`Generators(K)`

The set of generators of the number field K over its coefficient field, that is a set containing a root of each defining polynomial is returned.

`GeneratorsOverBaseRing(K)`

A set of generators of the number field K over $\mathbf{Q}$.

`GeneratorsSequence(K)`

The sequence of generators of the number field K over its coefficient field, that is a sequence containing a root of each defining polynomial is returned.

`GeneratorsSequenceOverBaseRing(K)`

A sequence of generators of the number field K over $\mathbf{Q}$.

`Generators(K, k)`

A sequence of generators of the number field K over k is returned. That is a sequence containing a root of each defining polynomial for K and its subfield down to the level of k is returned.

34.4.6 Real and Complex Valued Functions

The functions here return (sequences of) real or complex numbers. The precision of these numbers is governed by the appropriate or field's internal precision. See Section 37.3 for more information.

`AbsoluteValues(a)`

Return a sequence of length $r_1 + r_2$ of the real absolute values of the conjugates of the number field element a. The first r_1 values are the absolute values of the real embeddings of the element, the next r_2 are the lengths of the complex embeddings with their weight factors. That is, if the real conjugates of a are w_i, for $1 \leq i \leq r_1$, and the complex conjugates of a are $x_i \pm iy_i$ (for $1 \leq i \leq r_2$), then `AbsoluteValues` returns $[|w_1|, \ldots, |w_{r_1}|, \sqrt{\frac{x_{r_1+1}^2+y_{r_1+1}^2}{2}}, \ldots \sqrt{\frac{x_{r_1+r_2}^2+y_{r_1+r_2}^2}{2}}]$.

`AbsoluteLogarithmicHeight(a)`

Let P be the minimal polynomial of the number field element a over $\mathbf{Z}$, with leading coefficient a_0 and roots $\alpha_1, \ldots, \alpha_n$. Then the absolute logarithmic height is defined to be

$$h(\alpha) = \frac{1}{n} \log(a_0 \prod_{j=1}^{n} \max(1, |\alpha_j|)).$$

`Conjugates(a)`

The real and complex conjugates of the given algebraic number a, as a sequence of n complex numbers. The r_1 real conjugates appear first, and are followed by r_2 pairs of complex conjugates. The field should be an absolute extension. The ordering of the conjugates is consistent for elements of the same field (or even for elements of different fields that have the same defining polynomial).

`Conjugate(a, k)`

Equivalent to `Conjugates(a)[k]`.

`Conjugate(a, l)`

Let $l := [l_1, \ldots, l_n]$ be a sequence of positive integers and assume that the number field K, the parent of a is given as a tower with n steps, $\mathbf{Q} \subseteq K_1 \subseteq \ldots \subseteq K_n = K$. This function computes the image of a in $\mathbf{C}$ or $\mathbf{R}$ under the embedding determined by l, that is under embedding obtained by extending the l_1 embedding of K_1 to K_2, then extending the l_2nd of those embeddings to K_3 and so on.

`Length(a)`

Return the T_2-norm of the number field element a, which is a real number. This equals the sum of the (complex) norms of the conjugates of a.

`Logs(a)`

Return the sequence of length $r_1 + r_2$ of logarithms of the absolute values of the conjugates of a number field element $a \neq 0$.

`CoefficientHeight(E)`

Computes the coefficient height of the number field element E, that is for an element of an absolute field it returns the maximum of the denominator and the largest coefficient wrt. to the basis of the parent. For elements in relative extensions, it returns the maximal coefficient height of all the coefficients wrt. the basis of the parent.

This function indicates in some way the difficulty of operations involving this element.

`CoefficientLength(E)`

Computes the coefficient length of the number field element E, that is for an element of an absolute field it returns the sum of the denominator and the absolute values of all coefficients wrt. to the basis of the parent. For elements in relative extensions, it returns the sum of the coefficient length of all the coefficients wrt. the basis of the parent.

This function gives an indication on the amount of memory occupied by this element.

34.4.7 Norm, Trace, and Minimal Polynomial

The norm, trace and minimal polynomial of number field elements can be calculated both with respect to the coefficient ring and to **Z** or **Q**.

`Norm(a)`

`Norm(a, R)`

The relative norm $\mathrm{N}_{\mathrm{L/F}}(\mathrm{a})$ over F of the element a of the number field L where F is the field over which L is defined as an extension. If R is given the norm is calculated over R. In this case, R must occur as a coefficient ring somewhere in the tower under L.

`AbsoluteNorm(a)`

`NormAbs(a)`

The absolute norm $\mathrm{N}_{\mathrm{L}/\mathbf{Q}}(\mathrm{a})$ over **Q** of the element a of the number field L.

`Trace(a)`

`Trace(a, R)`

The relative trace $\mathrm{Tr}_{L/F}(a)$ over F of the element a of the number field L where F is the field over which L is defined as an extension. If R is given the trace is computed over R. In this case, R must occur as a coefficient ring somewhere in the tower under L.

`AbsoluteTrace(a)`

`TraceAbs(a)`

The absolute trace $\mathrm{Tr}_{L/\mathbf{Q}}(a)$ over **Q** of the element a of the number field L.

`CharacteristicPolynomial(a)`

`CharacteristicPolynomial(a, R)`

Given an element a from a number field L, returns the characteristic polynomial of the element over R if given or the subfield F otherwise where F is the field over which L is defined as an extension.

`AbsoluteCharacteristicPolynomial(a)`

Given an element a from a number field, this function returns the characteristic polynomial of a over **Q**.

`MinimalPolynomial(a)`

`MinimalPolynomial(a, R)`

Given an element a from a number field L, returns the minimal polynomial of the element over R if given otherwise the subfield F where F is the field over which L is defined as an extension.

`AbsoluteMinimalPolynomial(a)`

Given an element a from a number field, this function returns the minimal polynomial of the element as a polynomial over $\mathbf{Q}$.

`RepresentationMatrix(a)`

`RepresentationMatrix(a, R)`

Return the representation matrix of the number field element a, that is, the matrix which represents the linear map wrt to the field basis, given by multiplication by a. The ith row of the representation matrix gives the coefficients of aw_i with respect to the basis $w_1, \ldots, w_n$.

If R is given the matrix is over R and with respect to the basis of the order or field over R.

`AbsoluteRepresentationMatrix(a)`

Return the representation matrix of the number field element a relative to the $\mathbf{Q}$-basis of the field constructed using products of the basis elements, where a is an element of the relative number field L.

Let $L_i := \sum L_{i-1}\omega_{i,j}$, $L := L_n$ and $L_0 := \mathbf{Q}$. Then the representation matrix is computed with respect to the $\mathbf{Q}$-basis $(\prod_j \omega_{i_j,j})_{i\in I}$ consisting of products of basis elements of the different levels.

Example H34E13

We create the norm, trace, minimal polynomial and representation matrix of the element $\alpha/2$ in the quartic field $\mathbf{Q}(\alpha)$.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> z := y/2;
> Norm(z), Trace(z);
2500 0
> MinimalPolynomial(z);
ext<Q|>.1^4 - 105*ext<Q|>.1^2 + 2500
> RepresentationMatrix(z);
[     0    1/2      0      0]
[     0      0    1/2      0]
[     0      0      0    1/2]
[-20000      0    210      0]
```

The awkwardness of the printing of the minimal polynomial above can be overcome by providing a parent for the polynomial, keeping in mind that it is a univariate polynomial over the rationals:

```
> P<t> := PolynomialRing(RationalField());
> MinimalPolynomial(z);
t^4 - 105*t^2 + 2500
```

34.4.8 Other Functions

Elements can be represented by sequences and have a product representation.

`Eltseq(a)`

For an element a of a number field F, a sequence of coefficients of length degree of F with respect to the basis is returned.

`Eltseq(E, k)`

For an algebraic number $E \in K$ and a ring k which occurs somewhere in the defining tower for K, return the list of coefficients of E over k, that is, apply `Eltseq` to E and to its coefficients until the list is over k.

`Flat(e)`

The coefficients of the number field element e wrt. to the canonical Q basis for its field. This is performed by iterating `Eltseq` until the coefficients are rational numbers. The coefficients obtained match the coefficients wrt. to `AbsoluteBasis`.

`a[i]`

The coefficient of the ith basis element in the number field element a.

`ProductRepresentation(a)`

Return sequences P and E such that the product of elements in P to the corresponding exponents in E is the algebraic number a.

`ProductRepresentation(P, E)`

`PowerProduct(P, E)`

Return the number field element a of the universe of the sequence P such that a is the product of elements of P to the corresponding exponents in the sequence E.

34.5 Class and Unit Groups

This section briefly describes the functions available to compute in and with the ideal and unit group of the number field, that is with the corresponding group in the maximal order. For more details and an idea of the algorithms involved, see Sections 37.6 and 37.7.

All functions mentioned in this section support the verbose flag `ClassGroup` up to a maximum value of 5.

Ideals in MAGMA are discussed in Section 37.9.

`ClassGroup(K: parameters)`

`Bound`	RngIntElt	*Default :* MinkowskiBound
`Proof`	MonStgElt	*Default :* "*Full*"
`Enum`	BoolElt	*Default :* `true`
`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`ClassGroup`	*Maximum :* 5

The group of ideal classes for the ring of integers O of the number field K is returned as an abelian group, together with a map from this abstract group to O. The map admits inverses and can therefore be used to compute "discrete logarithms" for the class group.

With the default values for the optional parameters the Minkowski bound is used and the last step of the algorithm verifies correctness, hence a fully proven result is returned.

If `Bound` is set to some positive integer M, M is used instead of the Minkowski bound. The validity of the result still depends on the `"Proof"` parameter.

If `Proof := "GRH"`, everything remains as in the default case except that a bound based on the GRH is used to replace the Minkowski bound. This bound may be enlarged setting the "Bound" parameter accordingly. The result will hence be correct under the GRH.

If `Proof := "Bound"`, the computation stops if an independent set of relations between the prime ideals below the chosen bound is found. The relations may not be maximal.

If `Proof := "Subgroup"`, a maximal subset of the relations is constructed. In terms of the result, this means that the group returned will be a subgroup of the class group (i.e. the list of prime ideals considered may be to small).

If `Proof := "Full"` (the default) a guaranteed result is computed. This is equivalent to `Bound := MinkowskiBound(K)` and `Proof := "Subgroup"`.

If only `Bound` is given, the `Proof` defaults to `"Subgroup"`.

Finally, giving `Proof := "Current"` is the same as repeating the last call to `ClassGroup()`, but without the need to explicitly restate the value of `Proof` or `Bound`. If there was no prior call to `ClassGroup`, a fully proven computation will be carried out.

If `Enum := false`, then instead of enumerating short elements to get relations, Magma will use random linear combinations of a reduced basis instead. For "small" fields this will typically slow down the computations, but for large fields it is sometimes not possible to find any point using enumeration so that this is necessary for "large" fields. Unfortunately, there is no known criterion to decide beforehand if a field is "large" or "small".

If `Al` is set to `"Sieve"` (regardless of the size of the discriminant) or the discriminant of K is greater than 10^{30} then the sieving method described in [Bia] (or [Jac99] for quadratic fields) will be used. If `Al` is set to `"NoSieve"` then the sieving method will not be used regardless of the size of the discriminant.

`ConditionalClassGroup(K)`

The class group of the number field K assuming the generalized Riemann hypothesis.

`ClassNumber(K: parameters)`

`Bound`	RngIntElt	*Default :* MinkowskiBound
`Proof`	MonStgElt	*Default :* "*Full*"
`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`ClassGroup`	*Maximum :* 5

Return the class number of the ring of integers O of a number field K. The options for the parameters are the same as for `ClassGroup`.

`BachBound(K)`

An integral upper bound for norms of generators of the ideal class group for the number field K assuming the generalized Riemann hypothesis.

`MinkowskiBound(K)`

An unconditional integral upper bound for norms of the generators of the ideal class group for the number field K.

`UnitGroup(K)`

`MultiplicativeGroup(K)`

`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`UnitGroup`	*Maximum :* 6

Given a number field K, this function returns an (abstract) abelian group U, as well as a bijection m between U and the units of the maximal order. The unit group consists of the torsion subgroup, generated by the image `m(U.1)` and a free part, generated in O by the images `m(U.i)` for $2 \leq i \leq r_1 + r_2$.

The parameter `Al` can be used to specify an algorithm. It should be one of `"Automatic"`, (default, a choice will be made for the user) `"ClassGroup"`, `"Dirichlet"`, `"Mixed"` (the best known Dirichlet method), `"Relation"` or `"Short"` (which is a variation of `"Mixed"`). In the case of real quadratic fields, a continued fraction algorithm is available, `"ContFrac"`.

`TorsionUnitGroup(K)`

The torsion subgroup of the unit group of the number field K, i.e. its maximal order. The torsion subgroup is returned as an abelian group T, together with a map m from the group to the order O. The torsion subgroup will be cyclic, and is generated by `m(T.1)`.

`UnitRank(K)`

Return the unit rank of the ring of integers O of a number field K.

Example H34E14

In our field defined by $x^4 - 420 * x^2 + 40000$, we obtain the class and unit groups as follows.

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> C := ClassGroup(K);
> C;
Abelian Group of order 1
> U := UnitGroup(K);
> U;
Abelian Group isomorphic to Z/2 + Z + Z + Z
Defined on 4 generators
Relations:
      2*U.1 = 0
> T := TorsionUnitGroup(K);
> T;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*T.1 = 0
```

34.6 Galois Theory

`GaloisGroup(K)`

`Subfields(K)`

`AutomorphismGroup(K)`

See Sections 38.2, 38.3 and 38.1.

34.7 Solving Norm Equations

MAGMA can solve norm, Thue, index form and unit equations. In this section, we will only dicuss norm equations, other types of Diophantine equations are discussed in 37.8.

Norm equations in the context of number fields occur in many applications. While MAGMA contains efficient algorithms to solve norm equations it is important to understand the difference between the various types of norm equations that occur. Given some element θ in a number field k together with a finite extension K/k, there are two different types of norm equations attached to this data:

- Diophantine norm equations, that is norm equations where a solution $x \in K$ is restricted to a particular order (or any additive subgroup), and
- field theoretic norm equations where any element in $x \in K$ with $N(x) = \theta$ is a solution.

While in the first case the number of *different* (up to equivalence) solutions is finite, no such restriction holds in the field case. On the other hand, the field case often allows to prove the existence or non-existence of solutions quickly, while no efficient tests exist for the Diophantine case. So it is not surprising that different methods are applied for the different cases.

`NormEquation(F, m)`

`Primes`	ESEQ OF PRIME IDEALS	*Default :* []
`Nice`	BOOLELT	*Default :* `true`

Given a number field F and an element m of the base field of F, this function returns a boolean indicating whether an element $\alpha \in F$ exists such that $\mathrm{N}_{\mathrm{F/L}}(\alpha)$, the norm of α with respect to the base field L of F equals m, and if so, a sequence of length 1 of solutions α.

The field theoretic norm equations are all solved using S-units. Before discussing some details, we outline the method.

- Determine a set S of prime ideals. We try to obtain a solution as a S-unit for this set S.
- Compute a basis for the S-units
- Compute the action of the norm-map
- Obtain a solution as a preimage.

In general, no effective method is known for the first step. If the field is relative normal however, it is known that is S generates the class group of F and if m is a S-unit, then S is large enough (*suitable* in ([Coh00, 7.5]) [Fie97, Sim02, Gar80]. Thus to find S we have to compute the class group of F. If a (conditional) class group is already known, it is used, otherwise an *unconditional* class group is computed. The initial set S consists of all prime ideals occurring in the decomposition of $m\mathbf{Z}_F$. Note that this step includes the factorisation of m and thus can take a long time is m is large.

Next, we determine a basis for the S-unit group and the action of the norm on it. This give the norm map as a map on the S-unit group as an abstract abelian group.

Finally, the right hand side m is represented as an element of the S-unit group and a solution is then obtained as a preimage under the norm map.

If `Nice` is true, then MAGMA attempts to find a smaller solution by applying a LLL reduction to the original solution.

If `Primes` is give it must contain a list of prime ideals of L. Together with the primes dividing m it is used to form the set S bypassing the computation of an unconditional class group in this step. If L is not normal this can be used to guarantee that S is large enough. Note that the class group computation is still performed when the S-units are computed. Since the correctness of the S-unit group (we need only p-maximality for all primes dividing the (relative) degree of L) can be verified independently of the correctness of the class group, this can be used to derive provable results in cases where the class group cannot be computed unconditionally.

By default, the `MaximalOrder(L)` is used to compute the class group. If the attribute `NeqOrder` is set on L it must contain a maximal order of L. If present, this order will be used for all the subsequent computations.

`NormEquation(m, N)`

`Raw`	BOOLELT	*Default :* `false`
`Primes`	ESEQ OF PRIME IDEALS	*Default :* []

Let N be a map on the multiplicative group of some number field. Formally N may also be defined on the maximal order of the field. This intrinsic tries to find a pre-image for the element m under N.

This function works by realising N as a endomorphism of S-units for a suitable set S.

If N is a relative norm and if L is (absolutely) normal then the set S as computed for the field theoretic norm equation is guaranteed to be large enough to find a solution if it exists. Note: this condition is not checked.

If `Primes` is given it will be supplemented by the primes dividing m and then used as the set S.

If `Raw` is given, the solution is returned as an unevaluated power product. See the example for details.

The main use of this function is for Galois theoretical constructions where the subfields are defined as fields fixed by certain automorphisms. In this situation the norm function can be realised as the product over the fixed group. It is therefore not necessary to compute a (very messy) relative representation of the field.

`SimNEQ(K, e, f)`

`S`	[RNGORDIDL]	*Default :* `false`
`HasSolution`	BOOLELT	*Default :* `false`

For a number field K and subfield elements $e \in k_1$ and $f \in k_2$, try to find a solution to the simultaneous norm equations $N_{K/k_1}(x) = e$ and $N_{K/k_2}(x) = f$. The algorithm proceeds by first guessing a likely set S of prime ideals that will support

a solution - it is exists. Initially S will contain all ramified primes in K, the support of e and f and enough primes to generate the class group of K. In case K is normal over Q this set is large enough to support a solution if there is a solution at all. For arbitrary fields that is most likely not the case. However, if `S` is passed in as a parameter then the set used internally will contain at least this set. If `HasSolution` is `true`, MAGMA will add primes to S until a solution has been found. This is useful in situations where for some theoretical reason it is known that there has to be a solution.

Example H34E15

We try to solve $N(x) = 3$ in some relative extension: (Note that since the larger field is a quadratic extension, the second call tells us that there is no integral element with norm 3)

```
> x := PolynomialRing(Integers()).1;
> K := NumberField([x^2-229, x^2-2]);
> NormEquation(K, 3);
true [
  1/3*K.1 - 16/3
]
```

Next we solve the same equation but come from a different angle, we will define the norm map as an element of the group ring and, instead of explicitly computing a relative extension, work instead with the implicit fixed field.

```
> F := AbsoluteField(K);
> t := F!K.2;
> t^2;
2
> A, _, mA := AutomorphismGroup(F);
> S := sub<A | [ x : x in A | mA(x)(t) eq t]>;
> N := map<F -> F | x:-> &* [ mA(y)(x) : y in S]>;
> NormEquation(3, N);
true [
  -5/1*$.1 + 2/3*$.3
]
```

Finally, to show the effect of `Raw`:

```
> f, s, base := NormEquation(3, N:Raw);
> s;
[
  ( 0  1 -1  1 -1  0  2 -1 -1 -1 -1  2  0  0)
]
> z := PowerProduct(base, s[1]);
> z;
-5/1*$.1 + 2/3*$.3
> N(z);
3
```

34.8 Places and Divisors

A place of a number field K, an object of type `PlcNumElt`, is a class of absolute values (valuations) that induce the same topology on the field. By a famous theorem of Ostrowski, places of number fields are either finite, in which case they are in a one-to-one correspondence with the on-zero prime ideals of the maximal order, or infinite. The infinite places are identified with the embedding of K into $\mathbf{R}$ or with pairs of embeddings into $\mathbf{C}$.

The group of divisors is formally the free group generated by the finite places and the $\mathbf{R}$-vectorspace generated by the infinite ones. Divisors are of type `DivNumElt`. Places have formal parent of type `PlcNum`, while divisors belong to `DivNum`.

34.8.1 Creation of Structures

`Places(K)`

`DivisorGroup(K)`

The set of places of the number field K and the group of divisors of K respectively.

34.8.2 Operations on Structures

`d1 eq d2` `p1 eq p2`

`NumberField(P)`

`NumberField(D)`

The number field for which P is the set of places or D is the group of divisors.

34.8.3 Creation of Elements

`Place(I)`

The place corresponding to prime ideal I.

`Decomposition(K, p)`

`Decomposition(K, I)`

A sequence of tuples of places and multiplicities. When a finite prime (integer) p is given, the places and multiplicities correspond to the decomposition of p in the maximal order of K. When the infinite prime is given, a sequence of all infinite places is returned.

`Decomposition(K, p)`

For a number field K and a place p of the coefficient field of K, compute all places (and their multiplicity) that extend p. For finite places this is equivalent to the decomposition of the underlying prime ideal. The sequence returned will contain the places of K extending p and their ramification index.

For an infinite place p, this function will compute all extensions of p in K. In this case, the integer returned in the second component of the tuples will be 1 if p is complex or if p is real and extends to a real place and 2 otherwise.

`Decomposition(m, p)`

`Decomposition(m, p)`

For an extension K/k of number fields (where k can be Q as well), given by the embedding map $m : k \to K$, decompose the place p of k in the larger field. In case $k = Q$, the place is given as either a prime number or zero to indicate the infinite place. The sequence returned contains pairs where the first component is a place above p via m and the second is the ramification index.

`InfinitePlaces(K)`

A sequence containing all the infinite places of the number field K is returned.

`Divisor(pl)`

The divisor $1 * pl$ for a place pl.

`Divisor(I)`

The divisor which is the linear combination of the places corresponding to the factorization of the ideal I and the exponents of that factorization.

`Divisor(x)`

The principal divisor xO where O is the maximal order of the underlying number field of which x is an element. In particular, this computes a finite divisor.

`RealPlaces(K)`

For a number field K a sequence containing all real (infinite) places is computed. For an absolute field this are precisely the embeddings into R coming from the real roots of the defining polynomial.

34.8.4 Arithmetic with Places and Divisors

Divisors and places can be added, negated, subtracted and multiplied and divided by integers.

`d1 + d2` `- d` `d1 - d2` `d * k` `d div k`

34.8.5 Other Functions for Places and Divisors

`Valuation(a, p)`

The valuation of the element a of a number field at the place p.

`Valuation(I, p)`

The valuation of the ideal I at the finite place p.

`Support(D)`

The support of the divisor D as a sequence of places and a sequence of the corresponding exponents.

`Ideal(D)`

The ideal corresponding to the finite part of the divisor D.

`Evaluate(x, p)`

The evaluation of the number field element x in the residue class field of the place p, i.e. for a finite place p this corresponds to the image under the residue class field map for the underlying prime ideal. For infinite places, this returns the corresponding conjugate, ie. a real or complex number.

`RealEmbeddings(a)`

The sequence of real embeddings of the algebraic number a is computed, i.e. a is evaluated at all real places of the number field.

`RealSigns(a)`

A sequence containing ± 1 depending on whether the evaluation of the number field element a at the corresponding real place is positive or negative.

`IsReal(p)`

For an infinite place p, returns `true` if the corresponding embedding is real, i.e. if `Evaluate` at p will give real results.

`IsComplex(p)`

For an infinite place p, return `true` if the corresponding embedding is complex, i.e. if `Evaluate` at p will generally yield complex results.

`IsFinite(p)`

For a place p of a number field, return if the place is finite, i.e. if it corresponds to a prime ideal.

`IsInfinite(p)`

For a place p of a number field return if the place is infinite, ie. if it corresponds to an embedding of the number field into the real or complex numbers. If the place is infinite, the index of the embedding it corresponds to is returned as well.

`Extends(P, p)`

For two places P of K and p of k where K is an extension of k, check whether P extends p. For finite places, this is equivalent to checking if the prime ideal corresponding to P dives, in the maximal order of K the prime ideal of p. For infinite places `true` implies that for elements of k, evaluation at P and p will give identical results.

`InertiaDegree(P)`

`Degree(P)`

For a place P of a number field, return the inertia degree of P. That is for a finite place, return the degree of the residue class field over it's prime field, for infinite places it is always 1.

`Degree(D)`

For a divisor D of a number field, the degree is the weighted sum of the degrees of the supporting places, the weights being the multiplicities.

`NumberField(P)`

`NumberField(D)`

For a place P or divisor D of a number field, return the underlying number field.

`ResidueClassField(P)`

For a place P of a number field, compute the residue class field of P. For a finite place this will be a finite field, namely the residue class field of the underlying prime ideal. For an infinite place, the residue class field will be the field of real or complex numbers.

`UniformizingElement(P)`

For a finite place P of a number field, return an element of valuation 1. This will be the uniformizing element of the underlying prime ideal as well.

`LocalDegree(P)`

The degree of the completion at the place P, i.e. the product of the inertia degree times the ramification index.

`RamificationIndex(P)`

The ramification index of the place P. For infinite real places this is 1 and 2 for complex places.

`DecompositionGroup(P)`

For a place P of a normal number field, return the decomposition group as a subgroup of the (abstract) automorphism group.

34.9 Characters

There is an ability to use characters on number fields, similar to Dirichlet characters over the integers. These are implemented via `DirichletGroup` on number field elements, and `HeckeCharacterGroup` on ideals. The former is the dual of the `RayResidueRing` of an ideal, and the latter is the dual of the `RayClassGroup` of an ideal. Arithmetic on groups can be done multiplicatively, and the characters can be evaluated at suitable field elements and ideals. The `sub` constructor, along with `+` and `meet` for subgroups on the same modulus, should also work.

The associated types are `GrpDrchNF` and `GrpDrchNFElt`, `GrpHecke` and `GrpHeckeElt`. The number field must be an absolute extension of the rationals.

34.9.1 Creation Functions

`DirichletGroup(I)`

`DirichletGroup(I, oo)`

Given an ideal I of the integer ring of the number field K and a set of real places of K, the intrinsic `DirichletGroup` will return the dual group to the `RayResidueRing` of the specified information.

`HeckeCharacterGroup(I)`

`HeckeCharacterGroup(I, oo)`

Given an ideal I of the integer ring of the number field K and a set of real places of K, the intrinsic `HeckeCharacterGroup` will return the dual group to the `RayClassGroup` of the specified information.

`UnitTrivialSubgroup(G)`

Given a group of Dirichlet characters, return the subgroup that is trivial on the image of the field units in the residue ring.

`TotallyUnitTrivialSubgroup(G)`

Given a group of Dirichlet characters, return the subgroup that is totally trivial on the image of the field units in the residue ring. That is, it is trivial at each place individually.

34.9.2 Functions on Groups and Group Elements

`Modulus(G)`

`Modulus(G)`

`Modulus(chi)`

`Modulus(chi)`

Returns the modulus ideal and a (possibly empty) sequence of real places.

```
Order(chi)
Order(psi)
```

Returns the order of a Dirichlet or Hecke character.

```
Random(G)
Random(G)
```

Returns a random element of a Dirichlet or Hecke group.

```
Domain(G)
Domain(G)
```

Returns the number field that is the domain for the Dirichlet character.

```
Domain(G)
Domain(G)
```

Returns the set of ideals that is the domain for the Hecke character.

```
Decomposition(chi)
```

Returns a list of characters of prime power modulus (and real places) whose product (after extension to the original `DirichletGroup`) is the given Dirichlet character.

```
Conductor(chi)
Conductor(psi)
```

The product of the moduli of the all nontrivial characters in the decomposition of the given Dirichlet character, given as an ideal and a set of real places. Similarly with Hecke characters, where, in fact, one takes the `DirichletRestriction` of the Hecke character, and decomposes this.

```
AssociatedPrimitiveCharacter(chi)
AssociatedPrimitiveCharacter(psi)
```

The primitive Dirichlet character associated to the one that is given, which can be obtained by multiplying all the nontrivial characters in the decomposition. Similarly with Hecke characters, for which this decomposes the `DirichletRestriction` to finds its underlying primitive part, and then takes the `HeckeLift` of this.

```
Restrict(chi, D)
Restrict(psi, H)
Restrict(chi, I)
Restrict(psi, I)
Restrict(chi, I, oo)
Restrict(psi, I, oo)
Restrict(G, D)
Restrict(G, H)
Restrict(G, I)
Restrict(G, I)
Restrict(G, I, oo)
Restrict(G, I, oo)
```

Given a Dirichlet character modulo an ideal I and a Dirichlet character group modulo J for which $I \subseteq J$ (including behavior at real places when specified) with the character trivial on $(J/I)^\star$, this returns the restricted character on J. Similarly with Hecke characters, and with an ideal (with possible real places) at the second argument. Also with a group of characters as the first argument.

```
TargetRestriction(G, C)
TargetRestriction(H, C)
```

Given a group of Dirichlet or Hecke characters and a cyclotomic field, return the subgroup of characters whose image is contained in the cyclotomic field.

```
SetTargetRing(~chi, e)
SetTargetRing(~psi, e)
```

Given a Dirichlet or Hecke character and a suitable root of unity, modify the character to take values according to this root unity. The ring element must be a root of unity, and its order must be a multiple of the order of the character. Writing $m = \mathrm{ord}(\chi)$, if the character previously had $\chi(u) = \zeta_m^v$, it will now have $\chi(u) = (e^q)^v$ where q is $\mathrm{ord}(e)/m$.

```
Extend(chi, D)
Extend(psi, H)
Extend(chi, I)
Extend(psi, I)
Extend(chi, I, oo)
Extend(psi, I, oo)
```

```
Extend(G, D)
Extend(G, H)
Extend(G, I)
Extend(G, I)
Extend(G, I, oo)
Extend(G, I, oo)
```

Given a Dirichlet character modulo I and a Dirichlet character group modulo J for which $J \subseteq I$ (again possibly including the real places), this function returns the induced character on J, that is, the one that is trivial on $(I/J)^\star$. A second return value corresponds to a kernel. Similarly with Hecke characters, and with an ideal (with possible real places) at the second argument. Also with a group of characters as the first argument.

34.9.3 Predicates on Group Elements

```
IsTrivial(chi)
IsTrivial(psi)
```

Returns whether the given character corresponds to the trivial element in the character group.

```
IsTrivialOnUnits(chi)
```

Returns whether a Dirichlet character is trivial on the units of the number field; this determines whether the character can lift to a Hecke character on the ideals.

```
IsOdd(chi)
```

Returns whether a Dirichlet character χ has $\chi(-1) = -1$.

```
IsEven(chi)
```

Returns whether a Dirichlet character χ has $\chi(-1) = +1$.

```
IsTotallyEven(chi)
```

Returns whether a Dirichlet character χ has $\chi_p(-1) = +1$ for each χ_p in its decomposition.

```
IsPrimitive(chi)
IsPrimitive(psi)
```

Returns whether a Dirichlet character is primitive, that is, whether its conductor and modulus are equal.

34.9.4 Passing between Dirichlet and Hecke Characters

`HeckeLift(chi)`

Given a Dirichlet character that is trivial on the units of the number field, this functions returns a Hecke character that extends its domain to all the ideals of the integer ring. Also returns a kernel, so as to span the set of all possible lifts.

`DirichletRestriction(psi)`

Given a Hecke character on the ideals of the integer ring of a number field, this functions returns the Dirichlet restriction of it on the field elements.

`NormInduction(K, chi)`

Given a Dirichlet character χ over the rationals, induce it to the number field K. That is, find ψ with $\psi(a) = \chi(\mathrm{N}a)$ with ψ primitive.

Example H34E16

This example tries to codify the terminology via a standard example with Dirichlet characters over the rationals. We construct various characters modulo 5.

```
> Q := NumberField(Polynomial([-1, 1]) : DoLinearExtension);
> O := IntegerRing(Q);
> I := 5*O;
> DirichletGroup(I);
Abelian Group isomorphic to Z/4
Group of Dirichlet characters of modulus of norm 5 mapping to
 Cyclotomic Field of order 4 and degree 2
```

The above group is the Dirichlet characters modulo 5. However, the odd characters are not characters on ideals, as they are nontrivial on the units. To pass to the Hecke characters, we need to enlarge the modulus to consider embeddings at the real place. Note that this will give four more characters, corresponding to multiplying the above by the character that has χ as $+1$ on positive elements and -1 on negative elements; such characters will not be periodic in the traditional sense of Dirichlet characters, but are still completely multiplicative.

```
> D := DirichletGroup(I, [1]); D; // include first real place
Abelian Group isomorphic to Z/2 + Z/4
Group of Dirichlet characters D of modulus of norm 5 and infinite
places [ 1 ] mapping to Cyclotomic Field of order 4 and degree 2
> [ IsTrivialOnUnits(x) : x in Elements(D) ];
[ true, false, false, true, true, false, false, true ]
>  HeckeLift(D.1); // non-trivial on units
Runtime error in 'HeckeLift': Character is nontrivial on the units
> hl := HeckeLift(D.1 * D.2);
> hl(2);
zeta_4
> hl(2) eq (D.1 * D.2)(2);
```

```
true
```

So only half of the 8 completely multiplicative characters on field elements lift to characters on ideals, and these correspond exactly the standard four Dirichlet characters modulo 5, though evinced in a different guise.

`DirichletCharacter(I, B)`

`DirichletCharacter(I, oo, B)`

`DirichletCharacter(G, B)`

`HeckeCharacter(I, B)`

`HeckeCharacter(I, oo, B)`

`HeckeCharacter(G, B)`

`RequireGenerators` BoolElt *Default :* `true`

Given either an ideal (and also possibly a set of real infinite places) or a `DirichletGroup`, and a tuple of 2-tuples each containing a field element and a element of `Integers(m)` for some m, construct a Dirichlet character that sends each field element to the cyclotomic unit corresponding to the residue element. The second member of each 2-tuple can alternatively be an element of some cyclotomic field.

The parameter `RequireGenerators` demands that the given field elements should generate the `RayResidueRing` of the ideal. The second return argument is a subgroup of the ambient `DirichletGroup` by which the returned character can be translated and still retain the same values on the given elements.

Similarly for `HeckeCharacter` – there the first element in each 2-tuple should be an ideal of the field, and `RequireGenerators` demands that these generate the `RayClassGroup` of the ideal.

Example H34E17

We define a character on $5O_K$ that sends $\sqrt{-23}$ to ζ_8^2 (note that $\sqrt{-23}$ has order 8 in the `RayResidueRing` to this modulus).

```
> K := QuadraticField(-23);
> I := 5*IntegerRing(K);
> chi, SG := DirichletCharacter
>           (I, <<K.1, Integers(8)!2>> : RequireGenerators := false);
> chi(K.1);
zeta_4
> (SG.1 * chi)(K.1);
zeta_4
```

And then we define one that sends $\sqrt{-23}$ to ζ_8^6 and $(3 + 2\sqrt{-23})$, an element of order 6 in the `RayResidueRing`, to ζ_{24}^8.

```
> data := <<K.1, Integers(8)!6>, <3+2*K.1,Integers(24)!8>>;
```

```
> chi, SG := DirichletCharacter(I, data);
> chi(K.1);
-zeta_4
> chi(3+2*K.1);
zeta_3
> #SG; // this subgroup SG is trivial, as the data determine chi
1
```

Note that we can replace the Integers(8)!6 in the first tuple by ζ_8^6.

```
> C<zeta8> := CyclotomicField(8);
> data2 := <<K.1, zeta8^6>, <3+2*K.1,Integers(24)!8>>;
> chi2 := DirichletCharacter(I, data2);
> chi eq chi2;
true
```

Now we give a example with Hecke characters over a cubic field. We also note that the evaluation of a character (either Dirichlet or Hecke) can be obtained in "raw" form as an element in a residue ring via the use of the `Raw` vararg.

```
> _<x> := PolynomialRing(Integers());
> K<s> := NumberField(x^3-x^2+7*x-6); // #ClassGroup(K) is 5
> I := Factorization(11*IntegerRing(K))[2][1]; // norm 121
> HG := HeckeCharacterGroup(I,[1]); // has 20 elements
> f3 := Factorization(3*IntegerRing(K))[1][1]; // order 10
> data := < <f3, Integers(10)!7> >;
> psi := HeckeCharacter(HG, data : RequireGenerators := false);
> psi(f3);
-zeta_10^2
> psi(f3) eq CyclotomicField(10).1^7;
true
> '@'(f3,psi : Raw); // get Raw form of evaluation
14
> Parent($1);
Residue class ring of integers modulo 20
> f113 := Factorization(113*IntegerRing(K))[1][1]; // order 4
> data2 := < <f113, Integers(4)!3> >;
> psi := HeckeCharacter(HG, <data[1], data2[1]>);
> psi(f113);
-zeta_4
```

`CentralCharacter(chi)`

`CentralCharacter(psi)`

Given a Dirichlet or Hecke character, compute its central character down to the rationals. This is defined by computing a Dirichlet character that is defined over **Q** and agrees with the given character on a set of generators of the residue ring of the norm of the modulus of the given character. The `AssociatedPrimitiveCharacter` of this is then returned. It should have the same value as the original character on all unramified primes (at least). Note that the central character will always be a Dirichlet character, as the class number of the rationals is 1.

Example H34E18

```
> K := NumberField(Polynomial([4,3,-1,1])); // x^3-x^2+3*x+4
> f7 := Factorization(7*Integers(K))[1][1];
> G:= DirichletGroup(f7^2,[1]);
> chi := G.1*G.2*G.3;
> cc := CentralCharacter(chi); Conductor(cc);
Principal Ideal, Generator: [49] // conductor 49
[ 1 ] // infinite place
> cc := CentralCharacter(chi^14); Conductor(cc);
Principal Ideal, Generator: [7] // conductor 7
[ ] // no infinite places
> ////////////////
> K := NumberField(Polynomial([-10,-9,-10,1]));
> #ClassGroup(K); // C7 class group
7
> f5 := Factorization(5*Integers(K))[1][1];
> H := HeckeCharacterGroup(f5,[1]);
> cc:=CentralCharacter(H.1); Conductor(cc);
Principal Prime Ideal, Generator: [5] // conductor 5
[ 1 ] // infinite place
> Order(H.1), Order(cc);
28 4
> IsTrivial(CentralCharacter(H.1^4));
true
```

`DirichletCharacterOverNF(chi)`

`DirichletCharacterOverQ(chi)`

These are utility functions to pass between the two types of Dirichlet character over **Q** in Magma. The first takes a Dirichlet character over the `Rationals()` and returns one over the rationals as a number field, and the second reverses this.

Example H34E19

```
> SetSeed(1);
> G := DirichletGroup(16*3^2*5^2*7*11, CyclotomicField(2^6*3*5));
> #G;
57600
> chi := Random(G);
> Order(chi);
30
> psi := DirichletCharacterOverNF(chi);
> Order(psi);
30
> #Parent(psi)`supergroup;
57600
> &and[chi(p) eq psi(p) : p in PrimesUpTo(1000)];
true
> DirichletCharacterOverQ(psi) eq chi;
true
```

34.9.5 L-functions of Hecke Characters

Given a primitive Hecke character ψ, one can define the associated L-function as

$$L(\psi, s) = \prod_{\mathfrak{p}} \left(1 - \psi(\mathfrak{p})/\mathrm{N}\mathfrak{p}^s\right)^{-1},$$

and this satisfies a functional equation whose conductor is the product of the conductor of ψ and the discriminant of (the integer ring) of the number field for ψ.

Example H34E20

```
> _<x> := PolynomialRing(Integers());
> K<s> := NumberField(x^5 - 2*x^4 + 2*x + 2);
> I2 := Factorization( 2 * IntegerRing(K) ) [1][1]; // ideal above 2
> I11 := Factorization( 11 * IntegerRing(K) ) [1][1]; // above 11
> I := I2*I11; Norm(I);
22
> H := HeckeCharacterGroup(I, [1]);
> #H;
2
> psi := H.1; IsPrimitive(psi);
false
> prim := AssociatedPrimitiveCharacter(psi); Norm(Conductor(prim));
11
> L := LSeries(prim);
> LSetPrecision(L, 10); LCfRequired(L); // number of terms needed
```

```
4042
> CheckFunctionalEquation(L); // 2 or 3 seconds
-3.492459655E-10
```

34.9.6 Hecke Grössencharacters and their L-functions

A limited ability to compute Grössencharacters and their L-functions exists, but it has only really been tested for complex quadratic fields and a bit for $\mathbf{Q}(\zeta_5)$. These are "quasi-characters" in that their image is not restricted to the unit circle (and 0). The implementation in MAGMA handles the "algebraic" characters of this sort, or those of type A_0; it also requires that the field of definition be a CM-field (an imaginary quadratic extension of a totally real field).

The natural definition of Grössencharacters would be on a coset of the dual group of the `RayResidueRing` extended by the `ClassGroup`, without any modding out by units (which gives the `RayClassGroup`). However, the MAGMA implementation uses a Hecke character combined with an auxiliary Dirichlet character to simulate this. Arithmetic with Grössencharacters in also possible, even though there is no underlying group structure. However, equality with Grössencharacters is not implemented (one needs to check that various class group representatives are compatible, etc.).

`Grossencharacter(psi, chi, T)`

`RawEval(I, GR)`

Given a Hecke character ψ and a Dirichlet character χ (of the same modulus) and a compatible ∞-type T return the associated Grössencharacter.

The ∞-type is a sequence of pairs of integers which correspond to embeddings, such that

$$\psi\big((\alpha)\big) = \prod_{i=1}^{\#T} (\alpha^{\sigma_i})^{T[i][1]} (\bar{\alpha}^{\sigma_i})^{T[i][2]}$$

for all α that are congruent to 1 modulo the modulus of ψ. For a Grössencharacter to exist it follows that the ∞-type must trivialise all (totally positive) units that are congruent to 1 modulo the modulus of ψ. Each pair in T must have the same sum.

The Dirichlet character χ must correspond to the action on the image of the `UnitGroup` in the `RayResidueRing` of the modulus. In particular, for every unit u we must have that

$$\chi(u) = \prod_{i=1}^{\#T} (u^{\sigma_i})^{T[i][1]} (\bar{u}^{\sigma_i})^{T[i][2]},$$

and since the ∞-type is multiplicative, we need only check this on generators of the units.

Evaluating a Grössencharacter returns a complex number, corresponding to some choice of internal embeddings. The use of `RawEval` on an ideal will return an element in an extension of the field K (to which the ∞-type is then applied) and two elements in cyclotomic fields, corresponding to evaluations for χ and ψ respectively.

`Grossencharacter(psi, T)`

Same as above, but MAGMA will try to compute a compatible Dirichlet character χ for the given data. If there is more than one possibility, an arbitrary choice could be made.

`Conductor(psi)`

`Modulus(psi)`

`IsPrimitive(psi)`

`AssociatedPrimitiveGrossencharacter(psi)`

The conductor of the Grössencharacter is the conductor of the product of the Dirichlet part with the `DirichletRestriction` of the Hecke part. A Grössencharacter is primitive if its modulus is the same as the conductor. When taking L-functions, as before the conductor is multiplied by the discriminant of the integer ring of the field.

`Extend(psi, I)`

`Restrict(psi, I)`

Extension and restriction of a Grössencharacter. Note that the second argument is an ideal, unlike the Dirichlet/Hecke cases, where it is a group of characters.

`CentralCharacter(psi)`

Compute the central character (down to $\mathbf{Q}$) of a Grössencharacter, normalizing the result to be weight 0, and returning it as a Dirichlet character (over $\mathbf{Q}$ as a number field).

`GrossenTwist(Y, D)`

Given a Grossencharacter Y and a list D of data of pairs (a, r), find a twist Ψ of Y (by a Hilbert character) such that $\Psi(a) = r$. This intrinsic checks that the data D uniquely specifies the character, and the r should be in $\mathbf{Z}$.

Example H34E21

First, an example of $[1, 0]$-type Grössencharacters on the Gaussian field, with modulus $\mathfrak{p}_2^3$ where $\mathfrak{p}_2$ is the (ramified) prime above 2. This induces the L-function for the congruent number curve.

```
> K<i> := QuadraticField(-1);
> I := (1+i)^3*IntegerRing(K);
> HG := HeckeCharacterGroup(I, []);
> DG := DirichletGroup(I, []); #DG;
4
> GR := Grossencharacter(HG.0, DG.1^3, [[1,0]]);
> L := LSeries(GR); CheckFunctionalEquation(L);
1.57772181044202361082345713057E-30
> CentralValue(L);
```

```
0.65551438857302995261620989747 5
> CentralValue(LSeries(EllipticCurve("32a")));
0.65551438857302995261620989747 3
```

Example H34E22

An example with the canonical Grössencharacter for $K = \mathbf{Q}(\sqrt{-23})$. The ramification here is only at the prime above 23.

```
> K<s> := QuadraticField(-23);
> I := Factorization(23*IntegerRing(K))[1][1]; // ramified place
> HG := HeckeCharacterGroup(I, []);
> DG := DirichletGroup(I, []); #DG;
22
> GR := Grossencharacter(HG.0, DG.1^11, [[1,0]]); // canonical character
> CheckFunctionalEquation(LSeries(GR));
-5.17492753824983744350093938826E-28
> H := K‘extension_field; H; // defined by internal code
Number Field with defining polynomial y^3 + 1/2*(s + 3) over K
```

The values of the Grössencharacter are in the given field extension of K. We can also twist the Grössencharacter by a Hecke character on I, either via the `Grossencharacter` intrinsic, or by direct multiplication.

```
> i2 :=  Factorization(2*IntegerRing(K))[1][1]; // ideal of norm 2
> (GR*HG.1)(i2); // evaluation at i2
-0.14015763895624666594418088012 0 - 1.40725116316960648195556086783*i
> GR2 := Grossencharacter(HG.1, DG.1^11, [[1,0]]); // psi over zeta_11
> GR2(i2);
-0.14015763895624666594418088012 0 - 1.40725116316960648195556086783*i
> RawEval(i2,GR2);  // first value is in the cubic extension of K
H.1
1
zeta_33^4
> CheckFunctionalEquation(LSeries(GR2));
-7.09974814698910624870555708755E-30
```

Example H34E23

An example from Fernando Rodriguez Villegas where the Grössencharacter yields an L-function with even functional equation, but vanishing central value. This is the cube of the canonical character on $\mathbf{Q}(\sqrt{-59})$, which has class number 3. The L-function can be alternatively realised from a weight 4 modular form of level 59^2.

```
> K := QuadraticField(-59);
> I := Factorization(59*IntegerRing(K))[1][1];
> H := HeckeCharacterGroup(I);
> DG := DirichletGroup(I);
> GR := Grossencharacter(H.0, DG.1^29, [[3,0]]); // cube of canonical char
```

```
> L := LSeries(GR);
> CheckFunctionalEquation(L);
0.00000000000000000000000000000
> Sign(L);
1.00000000000000000000000000000
> CentralValue(L);
3.51858026759864075475017925650E-30
> LSetPrecision(L, 9);
> LTaylor(L, 2, 3); // first 3 terms of Taylor series about s=2
-1.09144041E-12 + 9.82515510E-11*z + 2.87637101*z^2 - 7.65817878*z^3 + ...
```

The same Grössencharacter can be obtained from cubing the canonical character (of type $[1,0]$).

```
> GR3 := Grossencharacter(H.0, DG.1^29, [[1,0]])^3;
> CentralValue(LSeries(GR3));
3.51858026759864075475017925650E-30
```

Example H34E24

An example with $\mathbf{Q}(\zeta_5)$, comparing the central value to the periods derived from Γ-values.

```
> _<x> := PolynomialRing(Rationals());
> K<z5> := NumberField(x^4+x^3+x^2+x+1);
> p5 := Factorization(5*IntegerRing(K))[1][1]; // ramified prime above 5
> H := HeckeCharacterGroup(p5^2);
> DG := DirichletGroup(p5^2);  // need p5^2 to get chi with this oo-type
> chi := DG.1^2*DG.2; // could alternatively have Magma compute this
> GR := Grossencharacter(H.0, chi, [[3,0],[1,2]]);
```

We can compute that this ∞-type sends ζ_5 to $(\zeta_5^1)^3 \cdot (\zeta_5^4)^0 \cdot (\zeta_5^2)^1 \cdot (\zeta_5^3)^2 = \zeta_5^{11}$ under the default embedding, and thus the ideal needs to afford a character of order 5 for a Grössencharacter to exist.

```
> L := LSeries(GR);
> LSeriesData(L); // Conductor is Norm(p5^2) * disc(K) = 5^2 * 5^3
<4, [ 0, -1, 1, 0 ], 3125, ... >;
> CheckFunctionalEquation(L); // functional equation works
0.00000000000000000000000000000
> CentralValue(L); // same as Evaluate(L,2)
1.25684568045898366613593980559
> Gamma(1/5)^3 * Gamma(2/5)^3 / Gamma(3/5)^2 / Gamma(4/5)^2 / 5^(7/2);
1.25684568045898366613593980558
```

The $[[3,0],[2,1]]$ ∞-type sends ζ_5 to $\zeta_5^{3+0+2\cdot 2+3} = 1$, but we still need $\mathfrak{p}_5$ in the modulus to trivialise the units of infinite order.

```
> H := HeckeCharacterGroup( 1 * IntegerRing(K)); // try conductor 1
>  GR := Grossencharacter(H.0, [[3,0],[2,1]]);
Runtime error in 'Grossencharacter':
oo-type should be trivial on all totally positive units that are 1 mod I
Fails for -zeta_5^2 - 1 which gives -1.000000000 - 3.293785801E-101*$.1
```

```
> H := HeckeCharacterGroup(p5); // conductor of norm 5
> GR := Grossencharacter(H.0, [[3,0],[2,1]]); // finds a character
> L := LSeries(GR); // CheckFunctionalEquation(L);
> PI := Pi(RealField());
> CentralValue(L); // now recognise as a product via logs and LLL
0.74985924643337212300558568330 0
> A := [ Gamma(1/5), Gamma(2/5), Gamma(3/5), Gamma(4/5), 5, PI, $1 ];
> LOGS := [ ComplexField() ! Log(x) : x in A ];
> LinearRelation(LOGS : Al := "LLL");
[ -14, 2, -2, 14, 15, -4, 4 ]
```

Example H34E25

Twisting a Grössencharacter by a Hilbert character is equivalent to changing the embedding.

```
> K := QuadraticField(-39);
> I := 39*IntegerRing(K);
> F := &*[f[1] : f in Factorization(I)]; // ideal of norm 39
> H := HeckeCharacterGroup(F); H;
Abelian Group isomorphic to Z/4 + Z/12 given as Z/4 + Z/12
> Norm(Conductor(H.1)); // H.1 is a Hilbert character of norm 1
1
> GR := Grossencharacter(H.0, [[3,0]]); // third power
```

There are four Hilbert characters here (from the class group of K), and we twist the Grössencharacter by each.

```
> L0 := LSeries(AssociatedPrimitiveGrossencharacter(GR));
> L1 := LSeries(AssociatedPrimitiveGrossencharacter(GR*H.1));
> L2 := LSeries(AssociatedPrimitiveGrossencharacter(GR*H.1^2));
> L3 := LSeries(AssociatedPrimitiveGrossencharacter(GR*H.1^3));
> Ls := [ L0, L1, L2, L3 ]; for L in Ls do LSetPrecision(L, 10); end for;
> for L in Ls do [CentralValue(L), Sign(L)]; end for;
[ 1.335826177, 1.000000000 + 2.706585223E-10*i ]
[ -1.373433032*i, -0.9999999999 - 6.351223882E-11*i ]
[ 1.335826177, 1.000000000 - 2.706585223E-10*i ]
[ 1.373433032*i, -0.9999999999 + 6.351223882E-11*i ]
```

The embedding information is stored internally in `K'Hip`, and we modify this directly to get the same L-values via a different method.

```
> K`Hip; // extension of infinite place of K
[ [ 1, 1 ] place at infinity ]
> IP := InfinitePlaces(K`extension_field); IP;
[ [ 1, 1 ] place at infinity, [ 1, 2 ] place at infinity,
  [ 1, 3 ] place at infinity, [ 1, 4 ] place at infinity ]
> for ip in IP do K`Hip := [ ip ]; // change ip, but use same GR
>        L := LSeries(AssociatedPrimitiveGrossencharacter(GR));
>        LSetPrecision(L, 10); [CentralValue(L), Sign(L)]; end for;
[ 1.335826177, 1.000000000 - 2.706585223E-10*i ]
```

```
[ 1.373433032*i, -0.9999999999 + 6.351223882E-11*i ]
[ 1.335826177, 1.000000000 - 2.706585223E-10*i ]
[ -1.373433032*i, -0.9999999999 - 6.351223882E-11*i ]
```

Finally, we can note that all the Hilbert characters have sign $+1$ in their functional equations, though two of the twists of the Grössencharacter have sign -1.

```
> Ls := [ LSeries(AssociatedPrimitiveCharacter(H.1^k)) : k in [1..4] ];
> [ Sign(L) where _:=CheckFunctionalEquation(L) : L in Ls ];
[ 0.99999999999999999999999999997, 1.0000000000000000000000000000,
  0.99999999999999999999999999997, 1.0000000000000000000000000000 ]
```

Example H34E26

A final example with characters of trivial conductor, here of type $[2,0]$ in $\mathbf{Q}(\sqrt{-23})$.

```
> K<s> := QuadraticField(-23); // class number 3
> I := 1*IntegerRing(K);
> HG := HeckeCharacterGroup(I, []);
> GR := Grossencharacter(HG.0, [[2,0]]); // of oo-type (2,0)
> Evaluate(LSeries(GR), 2); // value at edge of critical strip
1.2381910021242604040079438479 5
> Evaluate(LSeries(GR*HG.1), 2); // twist by nontrivial Hecke char
0.67033720866583940374792247 7469
> Evaluate(LSeries(GR*HG.1^2), 2);
1.06110583266449728309907405960
```

The product of these three L-values should be related to values of the Γ-function at $k/23$ for integral k. (One could alternatively relate these L-values to periods of an elliptic curve over the Hilbert class field of K having ramification only above 23.)

```
> SetDefaultRealFieldPrecision(100);
> e1 := Evaluate(LSeries(GR : Precision:=100), 2);
> e2 := Evaluate(LSeries(GR*HG.1 : Precision:=100), 2);
> e3 := Evaluate(LSeries(GR*HG.1^2 : Precision:=100), 2);
> GAMMA := [Gamma(i/23) : i in [1..22]];
> A := GAMMA cat [3,23,Pi(RealField())] cat [e1,e2,e3];
> LOGS := [ComplexField()!Log(x) : x in A];
> LinearRelation(LOGS : Al:="LLL");
[ 2,  2,  2,  2, -2,  2, -2,  2,  2, -2, -2,
  2,  2, -2, -2,  2, -2,  2, -2, -2, -2, -2,
 -2, -7,  6, -2, -2, -2 ]
> &*[ GAMMA[i]^(2*(DirichletGroup(23).1)(i)) : i in [1..22] ];
24723927.962642907904478305429424516264331853471961573093155911 28
> 3^2 * 23^7 / Pi(RealField())^6 * (e1*e2*e3)^2;
24723927.962642907904478305429424516264331853471961573093155911 28
```

Example H34E27

An example from [vGvS93, §8.8]. Here the Grössencharacter is on an ideal of norm 2^4 in the cyclotomic field $\mathbf{Q}(\zeta_8)$. The Euler factors will factor in various fields. In Table 7.6 of the cited paper, one notes that the coefficients satisfy $a_{17} = -180$ and $a_{17^2} = 15878$.

```
> Q<z8> := CyclotomicField(8);
> p2 := Factorization(2*Integers(Q))[1][1];
> G := HeckeCharacterGroup(p2^4);
> psi := G.0; // trivial
> GR := Grossencharacter(psi, [[3,0],[1,2]]);
> L:=LSeries(GR);
> CheckFunctionalEquation(L);
6.3108872417680944432938285226E-30
> Factorization(EulerFactor(L,7 : Integral)); // p is 7 mod 8
[ <343*x^2 + 1, 2> ]
> K<s2> := QuadraticField(-2);
> _<t> := PolynomialRing(K);
> Factorization(EulerFactor(L,3 : Integral),K); // 3 mod 8
[ <t^2 + 1/81*(-2*s2 - 1), 1>, <t^2 + 1/81*(2*s2 - 1), 1> ]
> K<i> := QuadraticField(-1);
> _<t> := PolynomialRing(K);
> Factorization(EulerFactor(L,5 : Integral),K); // 5 mod 8
[ <t^2 + 1/3125*(-24*i + 7), 1>, <t^2 + 1/3125*(24*i + 7), 1> ]
> EulerFactor(L,17 : Integral); // -180 and 15878 as desired
24137569*x^4 - 884340*x^3 + 15878*x^2 - 180*x + 1
```

Example H34E28

This examples exhibits the use of `GrossenTwist`, and links a Grössencharacter to a hypergeometric datum. The chosen t-value is such that the degree 3 L-function is imprimitive, and it splits as $L(\chi_{12}, s+1)L(\Psi, s)$, where Ψ is defined over $\mathbf{Q}(\sqrt{-84})$, and has trivial modulus and $[2, 0]$ ∞-type; there are still four such characters (the class number is 4), and we want the one with $\Psi(p_2) = 2$ and $\Psi(p_3) = -3$. We then check the degree 3 Euler factors on all good primes up to 100.

```
> P := PrimesInInterval(5,100);
> H := HypergeometricData([2,3],[1,6]);
> t := -27;
> ZT := Translate(LSeries(KroneckerCharacter(12)),1);
> K := QuadraticField(-84);
> DATA2 := <Factorization(2*Integers(K))[1][1],2>;
> DATA3 := <Factorization(3*Integers(K))[1][1],-3>;
> G := HeckeCharacterGroup(1*Integers(K));
> GR := Grossencharacter(G.0,[[2,0]]);
> LGR := LSeries(GrossenTwist(GR,[* DATA2, DATA3 *]));
> PROD := LGR*ZT;
> assert &and[EulerFactor(PROD,p : Integral) eq
>     EulerFactor(H,t,p) : p in P];
```

34.10 Number Field Database

An optional database of number fields may be downloaded from the MAGMA website. This section defines the interface to that database.

There are databases for number fields of degrees 2 through 9. In the case of degree 2 the enumeration is complete in the discriminant range (discriminants of absolute value less than a million); the other databases include fields with small (absolute value of) discriminant, as well as various other fields that may be of interest. The selection of fields is eclectic, and it may well be that certain "obvious" ones are missing.

For each number field in the database, the following information is stored and may be used to limit the number fields of interest via the `sub` constructor: The discriminant; the signature; the Galois group; the class number; and the class group.

34.10.1 Creation

`NumberFieldDatabase(d)`

Returns a database object for the number fields of degree d.

`sub< D | dmin, dmax : parameters >`

`sub< D | dabs : parameters >`

`sub< D | : parameters >`

`Signature`	[RNGINTELT]	*Default :*
`Signatures`	[[RNGINTELT]]	*Default :*
`GaloisGroup`	RNGINTELT *or* GRPPERM	*Default :*
`ClassNumber`	RNGINTELT	*Default :*
`ClassNumberLowerBound`	RNGINTELT	*Default :* 1
`ClassNumberUpperBound`	RNGINTELT	*Default :* ∞
`ClassNumberBounds`	[RNGINTELT]	*Default :* $[1, \infty]$
`ClassGroup`	[RNGINTELT]	*Default :*
`ClassSubGroup`	[RNGINTELT]	*Default :* [ ]
`ClassSuperGroup`	[RNGINTELT]	*Default :*
`SearchByValue`	BOOLELT	*Default :* `false`

Returns a sub-database of D, restricting (or further restricting, if D is already a sub-database of the full database for that degree) the contents to those number fields satisfying the specified conditions. Note that (with the exception of `SearchByValue`, which does not actually limit the fields in the database) it is not possible to "undo" restrictions with this constructor — the results are always at least as limited as D is.

The valid non-parameter arguments are up to two integers specifying the desired range of discriminants in the result. If two integers are provided these are taken as the lower (`dmin`) and upper (`dmax`) bounds on the discriminant. If one integer is

provided it is taken as a bound (`dabs`) on the absolute value of the discriminant. If no integers are provided then the discriminant range is the same as for D.

The parameters `Signature` or `Signatures` may be used to specify the desired signature or signatures to match. A signature is specified as a sequence of two integers $[s_1, s_2]$ where $s_1 + 2s_2 = d$.

The parameter `GaloisGroup` may be used to specify the desired Galois group of the number field. It may be given as either a permutation group, or as the explicit index of this Galois group in the transitive groups database. (i.e., the first return value of `TransitiveGroupIdentification`.)

It is possible to require certain divisibility conditions on the class number. Internally, there are lower and upper bounds on this value, such that a number field will only match if its class number is divisible by the lower bound and divides the upper bound. These bounds may be set individually using `ClassNumberLowerBound` or `ClassNumberUpperBound`; setting `ClassNumber` is equivalent to setting both bounds to the same value. Both values may be specified at once by setting `ClassNumberBounds` to the sequence of the lower and upper bounds.

When finer control is desired, it is possible to specify desired sub- and supergroups of the class group. Each group is specified by the sequence of its Abelian invariants; the desired subgroup is set using `ClassSubGroup`, and the desired supergroup is set using `ClassSuperGroup`. Both may be set at once (thus requiring the group to match exactly) using `ClassGroup`.

When iterating through the database, the default is to iterate in order of the absolute value of the discriminant. Sometimes it is desirable to iterate in order of the actual value of the discriminant; this can be accomplished by setting the parameter `SearchByValue` to true.

34.10.2 Access

`Degree(D)`

Returns the degree of the number fields stored in the database.

`DiscriminantRange(D)`

Returns the smallest and largest discriminants of the number fields stored in the database.

`#D`

`NumberOfFields(D)`

Returns how many number fields are stored in the database.

`NumberOfFields(D, d)`

Returns how many number fields of discriminant d are stored in the database.

`NumberFields(D)`

Returns the sequence of number fields stored in the database.

`NumberFields(D, d)`

Returns the sequence of number fields of discriminant d stored in the database.

Example H34E29

We illustrate with some basic examples. We start with the degree 2 number fields and get some basic information about the database.

```
> D := NumberFieldDatabase(2);
> DiscriminantRange(D);
-999995 999997
> #D;
607925
```

There are 13 fields with discriminant of absolute value less than 20:

```
> D20 := sub<D | 20>;
> #D20;
13
> [ Discriminant(F) : F in D20 ];
[ -3, -4, 5, -7, -8, 8, -11, 12, 13, -15, 17, -19, -20 ]
```

Note that these were listed in order of absolute value of the discriminant; we can also list them in value order:

```
> [ Discriminant(F) : F in sub<D20 | : SearchByValue> ];
[ -20, -19, -15, -11, -8, -7, -4, -3, 5, 8, 12, 13, 17 ]
```

Two of these number fields have class number 2:

```
> NumberFields(sub<D20 | : ClassNumber := 2>);
[
    Number Field with defining polynomial x^2 + x + 4 over the Rational Field,
    Number Field with defining polynomial x^2 + 5 over the Rational Field
]
```

Example H34E30

Now we move onto the fields of degree five, and in particular those with signature [1,2].

```
> D := NumberFieldDatabase(5);
> #D;
289040
> D12 := sub<D |: Signature := [1,2]>;
> #D12;
186906
```

We consider how many of these have class number equal to four. Note the cumulative nature of the restrictions has come into play.

```
> #sub<D12 |: ClassNumber := 4>;
1222
```

```
> #sub<D |: ClassNumber := 4>;
1255
```

The number with class group specifically isomorphic to $C_2 \times C_2$ is much less:

```
> #sub<D12 |: ClassGroup := [2,2]>;
99
```

34.11 Bibliography

[**Bai96**] Georg Baier. Zum Round 4 Algorithmus. Diplomarbeit, Technische Universität Berlin, 1996.
URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/baier.ps.gz.

[**Bia**] J.-F. Biasse. Number field sieve to compute Class groups.

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**Coh00**] Henri Cohen. *Advanced Topics in Computational Number Theory*. Springer, Berlin–Heidelberg–New York, 2000.

[**Fie97**] Claus Fieker. *Über relative Normgleichungen in algebraischen Zahlkörpern*. Dissertation, Technische Universität Berlin, 1997.
URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_CF.ps.gz.

[**Fri97**] Carsten Friedrichs. Berechnung relativer Ganzheitsbasen mit dem Round-2-Algorithmus. Diplomarbeit, Technische Universität Berlin, 1997.
URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/friedrichs.ps.gz.

[**Gar80**] Dennis A. Garbanati. An Algorithm for finding an algebraic number whose norm is a given rational number. *J. reine angew. Math.*, 316:1–13, 1980.

[**Jac99**] M. J. Jacobson, Jr. Applying sieving to the computation of quadratic class groups. *Math. Comp.*, 68(226):859–867, 1999.

[**KAN97**] KANT Group. KANT V4. *J. Symbolic Comp.*, 24(3–4):267–383, 1997.

[**KAN00**] KANT Group. The Number Theory Package KANT/KASH.
URL:http://www.math.tu-berlin.de/∼kant, 2000.

[**Pau01**] Sebastian Pauli. Factoring polynomials over local fields. *Journal of Symbolic Computation*, 32(5):533–547, 2001.

[**Poh93**] M. Pohst. *Computational Algebraic Number Theory*. DMV Seminar Band 21. Birkhäuser Verlag, Basel - Boston - Berlin, 1993.

[**PZ89**] Michael E. Pohst and Hans Zassenhaus. *Algorithmic Algebraic Number Theory*. Encyclopaedia of mathematics and its applications. Cambridge University Press, Cambridge, 1989.

[**Sim02**] Denis Simon. Solving norm equations in relative number fields using S-units. *Math. Comput.*, 71(239):1287–1305, 2002.

[**Tra76**] Barry M. Trager. Algebraic factoring and rational function integration. In R.D. Jenks, editor, *Proc. SYMSAC '76*, pages 196–208. ACM press, 1976.

[**vGvS93**] B. van Geemen and D. van Straten. The cusp forms of weight 3 on $\Gamma_2(2,4,8)$. *Math. Comp.*, 61(204):849–872, 1993.

35 QUADRATIC FIELDS

Chapter 35
QUADRATIC FIELDS

35.1 Introduction

Quadratic fields in MAGMA can be created as a subtype of the number fields `FldNum`. The advantage of the special quadratic fields is that some special (faster) algorithms have been or will be implemented to deal with them; the functions for the special quadratic fields (created with the `QuadraticField` function) are described here. Functions which work generally for number fields and their orders are described in Chapter 34.

The categories involved are `FldQuad` for fields, `RngQuad` for their orders and `FldQuadElt` and `RngQuadElt` for their elements.

35.1.1 Representation

For every squarefree integer d (not 0 or 1) there is a unique quadratic field $\mathbf{Q}(\sqrt{d})$; for any integer k we have the field $\mathbf{Q}(\sqrt{k^2 d}) \cong \mathbf{Q}(\sqrt{d})$. Given any integer m, the function `QuadraticField` will create a structure corresponding to the quadratic field $\mathbf{Q}(\sqrt{d})$, where d is the squarefree kernel of m (d will have the same sign as m and its absolute value is the largest squarefree divisor of m). In MAGMA a list of quadratic fields currently present is maintained, and if $\mathbf{Q}(\sqrt{d})$ has been created before a reference on it will be returned: two fields with the same d are the same. The discriminant D of $\mathbf{Q}(\sqrt{d})$ will be $D = d$ if $d \equiv 1 \bmod 4$ and $D = 4d$ if $d \equiv 2, 3 \bmod 4$.

Elements of $\mathbf{Q}(\sqrt{d})$ are represented by a common positive denominator b and two integer coefficients: $\alpha = \frac{1}{b}(x + y\sqrt{d})$.

The ring of integers of $F = \mathbf{Q}(\sqrt{d})$ will be $O_F = \mathbf{Z} + \epsilon_d \mathbf{Z}$, where

$$\epsilon_d = \begin{cases} \sqrt{d} & \text{if } d \equiv 2, 3 \bmod 4, \\ \frac{1+\sqrt{d}}{2} & \text{if } d \equiv 1 \bmod 4. \end{cases}$$

Elements of O_F are represented by two integer coefficients $\alpha = x + y\epsilon_d$. The pair $1, \epsilon_d$ forms an integral basis for $F = \mathbf{Q}(\sqrt{d})$, but note that elements of F are represented using the basis of the equation order $(1, \sqrt{d})$ instead.

For any positive integer f there is a suborder of conductor f in O_F, whose elements are of the form $x + yf\epsilon_d$, for any integers x, y. The discriminant of the order of conductor f is $f^2 D$, where D is the field discriminant.

The equation order of F is $E_F = \mathbf{Z} + \sqrt{d}\mathbf{Z}$. Suborders of conductor f can be formed which will contain elements of the form $x + yf\sqrt{d}$ for any integers x and y.

35.2 Creation of Structures

Squarefree integers determine quadratic fields. Associated with any quadratic field is its ring of integers (maximal order) and an equation order, and for every positive integer f there exists an order of conductor f inside the maximal order.

For information on creating elements see Section 34.2.3.

`QuadraticField(m)`

Given an integer m that is not a square, create the field $\mathbf{Q}(\sqrt{d})$, where d is the squarefree part of m. It is possible to assign a name to $\sqrt{d}$ using angle brackets: `R<s> := QuadraticField(m)`.

`EquationOrder(F)`

Creation of the order $\mathbf{Z}[\sqrt{d}]$ in the quadratic field $F = \mathbf{Q}(\sqrt{d})$, with d squarefree.

`MaximalOrder(F)`
`IntegerRing(F)`
`RingOfIntegers(F)`

Given a quadratic field $F = \mathbf{Q}(\sqrt{d})$, with d squarefree, create its maximal order. This order is $\mathbf{Z}[\sqrt{d}]$ if $d \equiv 2, 3 \bmod 4$ and $\mathbf{Z}[\frac{1+\sqrt{d}}{2}]$ if $d \equiv 1 \bmod 4$.

`NumberField(O)`

Given a quadratic order, this returns the quadratic field of which it is an order.

`sub< O | f >`

Create the sub-order of index f in the order O of a quadratic field. If O is maximal, this will be the unique order of conductor f.

`IsQuadratic(K)`
`IsQuadratic(O)`

Return `true` if the field K or order O can be created as a quadratic field or order and the quadratic field or order if so.

Example H35E1

We create the quadratic field $\mathbf{Q}(\sqrt{5})$ and an order in it, and display some elements of the order in their representation as order element and as field element.

```
> Q<z> := QuadraticField(5);
> Q eq QuadraticField(45);
true
> O<w> := sub< MaximalOrder(Q) | 7 >;
> O;
Order of conductor 7 in Q
> w;
```

```
w
> Q ! w;
1/2*(7*z + 7)
> Eltseq(w), Eltseq(Q ! w);
[ 0, 1 ]
[ 7/2, 7/2 ]
> ( (7/2)+(7/2)*z )^2;
1/2*(49*z + 147)
>  Q ! w^2;
1/2*(49*z + 147)
> w^2;
7*w + 49
```

Example H35E2

We define an injection $\phi : \mathbf{Q}(\sqrt{5}) \to \mathbf{Q}(\zeta_5)$. First a square root of 5 is identified in $\mathbf{Q}(\zeta_5)$.

```
> Q<w> := QuadraticField(5);
> F<z> := CyclotomicField(5);
> C<c> := PolynomialRing(F);
> Factorization(c^2-5);
[
   <c - 2*z^3 - 2*z^2 - 1, 1>,
   <c + 2*z^3 + 2*z^2 + 1, 1>
]
> h := hom< Q -> F | -2*z^3 - 2*z^2 - 1 >;
> h(w)^2;
5
```

35.3 Operations on Structures

The majority of functions for quadratic fields and orders apply identically to number fields and orders in general. The functions which exist only for quadratic fields and orders are listed here along with those which deserve a special mention.

`AssignNames(~F, [s])`

`AssignNames(~O, [s])`

Procedure to change the name of the generator of a quadratic field F or an order O in a quadratic field to the string s. Elements of the quadratic field $\mathbf{Q}(\sqrt{d})$ with m squarefree will be printed in the form `1/b*(x + y*s)`, where b, x, y are integers. Similarly, for an order O of conductor f in a quadratic field elements will be printed in the format x `+` y`*s`.

This procedure only changes the name used in printing the elements of F or O, it does *not* make an assignment to an identifier s. To do this, use an assignment statement, or angle brackets when creating the field or order: `F<s> := QuadraticField(-3);`.

Note that since this is a procedure that modifies F or O, it is necessary to have a reference $\sim$ in the call to this function.

`Name(F, 1)`

`Name(O, 1)`

Given a quadratic field F or one of its orders O, return the element which has the name attached to it, that is, return $\sqrt{d}$ in the field, or $f\epsilon_d$ in a suborder of the maximal order or $f\sqrt{d}$ in a suborder of the equation order.

`FundamentalUnit(K)`

`FundamentalUnit(O)`

A generator for the unit group of the order O or the maximal order of the quadratic field K.

`Discriminant(K)`

The discriminant of the field K which is only defined up to squares. The discriminant will be the discriminant of the polynomial or better.

`Conductor(K)`

The conductor of the field K which is the order of the smallest cyclotomic field containing K and a sequence containing the ramified real places of K.

`Conductor(O)`

The conductor of the order O, which equals the index of O in the maximal order.

35.3.1 Ideal Class Group

The function `ClassGroup` is available for number fields and orders in general but a different and faster algorithm is used by default for the quadratics. All of the algorithms, except for the sieving method described in [Jac99] which uses the multiple polynomial quadratic sieve (MPQS), are based on binary quadratic forms, see `ClassGroup` on page 758 in Chapter 33 for details.

`ClassGroup(K)`

`ClassGroup(O)`

`FactorBasisBound`	FldReElt	*Default :* 0.1
`ProofBound`	FldReElt	*Default :* 6
`ExtraRelations`	RngIntElt	*Default :* 1
`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`ClassGroupSieve`	*Maximum :* 5

The class group of a maximal order O or the maximal order of the quadratic field K, as an abelian group. The function also returns a map between the group and the power structure of ideals of O or the maximal order of K. The parameter `Al` can be set to `"Sieve"` or `"NoSieve"` to control whether the sieving algorithm is used or not; by default it is used when the discriminant is greater than 10^{20}. For more details on the parameters see `ClassGroup` on page 758 in Chapter 33.

`ClassNumber(K)`
`ClassNumber(O)`

`FactorBasisBound`	FldReElt	*Default :* 0.1
`ProofBound`	FldReElt	*Default :* 6
`ExtraRelations`	RngIntElt	*Default :* 1
`Al`	MonStgElt	*Default :* "*Automatic*"

The class number of the maximal order O or the maximal order of the quadratic field K.

`PicardGroup(O)`
`PicardNumber(O)`

`FactorBasisBound`	FldReElt	*Default :* 0.1
`ProofBound`	FldReElt	*Default :* 6
`ExtraRelations`	RngIntElt	*Default :* 1
`Al`	MonStgElt	*Default :* "*Automatic*"

The picard group (the group of the invertible ideals of O modulo the principal ones) of the order O or the size of this group. `PicardGroup` also returns a map from the group to the ideals of O.

Example H35E3

We give examples of class group calculations using sieving. We also show the use of the mapping between the group and the set of ideals. First a field with negative discriminant.

```
> D:=-(10^(30) + 3 );
> K := QuadraticField(D);
> time G, m := ClassGroup(K : Al := "Sieve");
Time: 10.750
> G, m;
Abelian Group isomorphic to Z/125355959329602
Defined on 1 generator
Relations:
    125355959329602*G.1 = 0
Mapping from: GrpAb: G to Set of ideals of Maximal Order of K
> m(G.1);
Ideal
Two element generators:
```

```
    385706622580333
    1487695987023274464670383908888*$.2 + 307255216496036
> $1 @@ m;
G.1
```

And now a field with positive discriminant.

```
> K := QuadraticField(NextPrime(10^24));
> time G, m := ClassGroup(K : Al := "Sieve");
Time: 3.330
> G, m;
Abelian Group isomorphic to Z/3
Defined on 1 generator
Relations:
    3*$.1 = 0
Mapping from: GrpAb: G to Set of ideals of Maximal Equation Order of K
> m(G.1);
Ideal
Two element generators:
    3847
    $.2 + 616
> $1 @@ m;
G.1
> IsPrincipal($2^3);
true
```

QuadraticClassGroupTwoPart(K)

QuadraticClassGroupTwoPart(O)

QuadraticClassGroupTwoPart(d)

Factorization RngIntEltFact *Default :* []

Use the Bosma-Stevenhagen algorithm to compute the 2-part of the class group of a quadratic order. Returned are: an array of forms that generates the 2-part and an array that gives the orders of the respective elements. The Factorization of the given discriminant can be given as additional information.

Example H35E4

```
> G, f := QuadraticClassGroupTwoPart(33923894057872); G;
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4 + Z/16 + Z/16
> Random(G);
11*G.1 + 2*G.2 + G.3 + G.4 + G.6
> f($1);
<-1212992,3947508,3780131>
> G, f := QuadraticClassGroupTwoPart(QuadraticField(33923894057872)); G;
Abelian Group isomorphic to Z/2 + Z/8 + Z/16
```

35.3.2 Norm Equations

For imaginary quadratic fields, (that is, for quadratic fields $\mathbf{Q}(\sqrt{m})$ with $m < 0$), the function `NormEquation` is provided specially for quadratics to find integral elements of a given norm. For real quadratic fields conics are used, see Section 119.5.1 for details.

`NormEquation(F, m)`

`NormEquation(F, m:` *parameters*`)`

`NormEquation(O, m)`

`NormEquation(O, m:` *parameters*`)`

`Factorization`	[<RngIntElt, RngIntElt>]	
`All`	BoolElt	*Default :* `true`
`Solutions`	RngIntElt	*Default :* All
`Exact`	BoolElt	*Default :* `false`
`Ineq`	BoolElt	*Default :* `false`
Verbose	`NormEquation`	*Maximum :* 1

Given quadratic field F and a non-negative integer m, return `true` if there exists an element α in the ring of integers O_F of F with norm m, and `false` otherwise. Instead of searching the maximal order O_F it is possible to search any suborder O of O_F for such element α by supplying O as a first argument.

For imaginary quadratic fields the method used is constructive (it uses Cornacchia's algorithm, see [Coh93] section 1.5.2), and if the value `true` is returned then a solution $[x]$ is also returned as a second return value.

Note that if the discriminant $F = \mathbf{Q}(\sqrt{d})$ with $d \equiv 1 \bmod 4$ (and squarefree) this function searches for a solution in integers to $x^2 + y^2 d = 4m$ (and the solution $\alpha = \frac{x+y\sqrt{d}}{2}$ is returned), whereas for $d \equiv 2, 3 \bmod 4$ a solution $\alpha = x + y\sqrt{d}$ with $x^2 + y^2 d = m$ in integers x, y is returned, if it exists. In an order of conductor f a search is conducted for a solution to the same equation with d replaced by $f^2 d$. Note that a version of `NormEquation` with integer arguments d and m also exists (see Section 18.12.2).

Unless m is the square of an integer, the factorization of m is used by the algorithm; if it is known, it may be supplied as the value of the optional parameter `Factorization` to speed up the calculation.

A verbose flag can be set to obtain some information on progress with the computation (see `SetVerbose` on page 102).

For real quadratic fields the same algorithm is used as for the general number fields. The last 4 parameters refer to this algorithm. See Section 34.7 for a description.

Example H35E5

```
> d := 302401481761723680;
```

```
> m := 76814814791186002463716;
> Q<z> := QuadraticField(-d);
> O<w> := sub< MaximalOrder(Q) | 6 >;
> f, s := NormEquation(O, m);
> s, Norm(s[1]);
406 + 1008*w 76814814791186002463716
```

35.4 Special Element Operations

There are a number of functions available only for elements of certain maximal orders. `Conjugate` is provided to return a quadratic element (instead of a real) as well as `ComplexConjugate`.

`a mod b`

The remainder on dividing a by b where a and b lie in the maximal order of $Q(\sqrt{d})$ for $d = -1, -2, -3, -7, -11, 2, 3, 5, 13$. `div` is provided for order elements in general, but for discriminants not in the above list `div` will fail if the division is not exact.

35.4.1 Greatest Common Divisors

`Gcd(a, b)`
`GCD(a, b)`
`GreatestCommonDivisor(a, b)`

The greatest common divisor of a and b in the maximal order of $\mathbf{Q}(\sqrt{d})$, where d must be one of the following values: $-1, -2, -3, -7, -11, 2, 3, 5, 13$.

`Lcm(a, b)`
`LCM(a, b)`
`LeastCommonMultiple(a, b)`

The least common multiple of a and b in the maximal order of $\mathbf{Q}(\sqrt{d})$, where d must be one of the following values: $-1, -2, -3, -7, -11, 2, 3, 5, 13$.

35.4.2 Modular Arithmetic

`Modexp(a, e, n)`

The computation of $a^e \bmod n$ in the maximal order of $\mathbf{Q}(\sqrt{d})$, where d is one of the following values: $-1, -2, -3, -7, -11, 2, 3, 5, 13$.

35.4.3 Factorization

MAGMA's factorization in maximal orders of quadratic number fields is based upon factoring the norm in the integers. Thus, the comments that are made about the `Factorization` command in the integers also apply here. Moreover, since the factorization may be off by a unit power, that power is also returned (the unit being -1, $\sqrt{-1}$, or $(1+\sqrt{-3})/2$).

`Factorization(n)`
`Factorisation(n)`

The factorization of n in the maximal order of the quadratic number field $Q(\sqrt{d})$, where d is one of: -1, -2, -3, -7, or -11. Returns the factorization along with the appropriate power of a unit (the unit being -1, $\sqrt{-1}$, or $(1+\sqrt{-3})/2$).

`TrialDivision(n, B)`

Trial division of n by primes of relative norm $\leq$ B in the maximal order of $Q(\sqrt{d})$, where d is one of: -1, -2, -3, -7, or -11. Returns the factored part, the unfactored part, and the power of the unit that the factorization is off by (the unit being -1, $\sqrt{-1}$, or $(1+\sqrt{-3})/2$).

35.4.4 Conjugates

`ComplexConjugate(a)`

The complex conjugate of quadratic field element a; returns a in a real quadratic field and $\bar{a} = x - y\sqrt{d}$ if $a = x + y\sqrt{d}$ in an imaginary quadratic field $\mathbf{Q}(\sqrt{d})$.

`Conjugate(a)`

The conjugate $x - y\sqrt{d}$ of $a = x + y\sqrt{d}$ in the quadratic field $\mathbf{Q}(\sqrt{d})$.

35.4.5 Other Element Functions

For the ring of integers of $\mathbf{Q}(\mathrm{i})$ the biquadratic residue symbol (generalizing the Legendre symbol) is available.

`BiquadraticResidueSymbol(a, b)`

Given a Gaussian integer a and a primary, non-unit Gaussian integer b, where a and b are coprime, return the value of the biquadratic character $\left(\frac{a}{b}\right)_4$. The value of this character is equal to i^k, for some $k \in \{0,1,2,3\}$. If a and b have a factor in common, the function returns 0, if b is not primary or b is a unit an error results.

`Primary(a)`

Return the unique associate $\bar{a}$ of the Gaussian integer a that satisfies

$$\bar{a} \equiv 1 \bmod (1+\mathrm{i})^3,$$

or 0 in case a is divisible by $1+\mathrm{i}$.

Example H35E6

The following example checks for primes p with $65 \leq p \leq 1000$ and $p \equiv 1 \bmod 4$ a result that was conjectured by Euler and proved by Gauss, namely that

$$z^4 \equiv 2 \bmod p \quad \text{has a solution} \iff p = x^2 + 64y^2 \quad \text{for some } x, y.$$

We use the function `NormEquation` to find the prime above p in the Gaussian integers, and we build the set of such primes for which 2 is a biquadratic residue (which means that $z^4 \equiv 2 \bmod p$ for some z).

```
> s := { };
> Q := QuadraticField(-1);
> M := RingOfIntegers(Q);
> for p := 65 to 1000 by 4 do
>    if IsPrime(p) then
>       _, x := NormEquation(Q, p);
>       if BiquadraticResidueSymbol(2, Primary(M!x[1])) eq 1 then
>          Include(~s, p);
>       end if;
>    end if;
> end for;
> s;
{ 73, 89, 113, 233, 257, 281, 337, 353, 577, 593, 601, 617, 881, 937 }
```

Next we create the set of all primes as above that are of the form $x^2 + 64y^2$. Note that we have to use `NormEquation` on a suborder of Q now, because we want to solve $x^2 + 64y^2 = p$, while `QuadraticField(-64)` returns just $\mathbf{Q}(\mathrm{i})$ in which we can only solve $x^2 + y^2 = p$.

```
> S := sub<MaximalOrder(Q) | 8>;
> t := { };
> for p := 65 to 1000 by 4 do
>    if IsPrime(p) then
>       if NormEquation(S, p) then
>           Include(~t, p);
>       end if;
>    end if;
> end for;
> t;
{ 73, 89, 113, 233, 257, 281, 337, 353, 577, 593, 601, 617, 881, 937 }
```

35.5 Special Functions for Ideals

Ideals of orders of quadratic fields inherit from ideals of orders of number fields (see Section 37.9 for the list of general functions and operations available). However there is also a correspondence between quadratic ideals and binary quadratic forms (see Chapter 33). The following functions make use of that correspondence.

`Content(I)`

The content of the ideal.

`Conjugate(I)`

The conjugate of the ideal.

`Discriminant(I)`

The discriminant of the quadratic form associated with I.

`QuadraticForm(I)`

The quadratic form associated with I.

`Ideal(f)`

The quadratic ideal with associated quadratic form f.

`Reduction(I)`

The quadratic ideal with associated quadratic form which is a reduction of the quadratic form associated to I.

35.6 Bibliography

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**Jac99**] M. J. Jacobson, Jr. Applying sieving to the computation of quadratic class groups. *Math. Comp.*, 68(226):859–867, 1999.

36 CYCLOTOMIC FIELDS

Chapter 36
CYCLOTOMIC FIELDS

36.1 Introduction

Cyclotomic Fields (like the Quadratic Fields) are a subtype of the Number Fields (FldNum). They have some extra functionality which is described below and use some more efficient implementations. Orders of cyclotomic fields form the category RngCyc and the fields themselves FldCyc. Functions for cyclotomic fields and orders which work generally for number fields, their orders and elements are listed in Chapter 34.

There are two different representations of cyclotomic fields available:

* The "dense" representation: the field is conceptually represented as $Q(x)/f(x)$ where f is a cyclotomic polynomial, i.e., the minimal polynomial of a primitive root of unity.
* The "sparse" representation: Let $n = \prod p_i^{r_i}$ be the factorisation of n into prime powers and $n_i := p_i^{r_i}$. Then $Q(\zeta_n) = Q(\zeta_{n_1}, \ldots, \zeta_{n_r})$ and the field is represented as $Q(x_1, \ldots, x_r)/\langle f_{n_1}(x_1), \ldots, f_{n_r}(x_r)\rangle$.

As with the number fields, the non-simple representation, the issues are the same: the "sparse" representation allows for much larger fields – as long as the elements used have only few coefficients. The "dense" representation on the other hand has the asymptotically-fastest arithmetic.

36.2 Creation Functions

Functions are provided to create fields of the special type FldCyc. Orders and elements created from a field of this type will have the special types RngCyc and FldCycElt respectively and elements created from orders RngCycElt. These functions provide an object with the correct type which will allow the extra functions and efficient implementations to be used.

36.2.1 Creation of Cyclotomic Fields

Cyclotomic fields can be created from an integer specifying which roots of unity it should contain or from a collection of elements of an existing field or order. Cyclotomic polynomials can also be retrieved independently of the fields and orders.

`CyclotomicField(m)`

`Sparse` BOOLEAN *Default :* `false`

Given a positive integer m, create the field obtained by adjoining the m-th roots of unity to $\mathbf{Q}$. It is possible to assign a name to the primitive m-th root of unity ζ_m using angle brackets: `R<s> := CyclotomicField(m)`.

If `Sparse := true`, names for all the generating elements can be assigned.

`CyclotomicPolynomial(m)`

Given a positive integer m, create the cyclotomic polynomial of order m. This function is equivalent to `DefiningPolynomial(CyclotomicField(m))`.

`MinimalCyclotomicField(a)`

Given an element a from a cyclotomic field F or ring R, this function returns the smallest cyclotomic field or order thereof (possibly the rational field or the ring of integers) $E \subset F$ containing a.

`MinimalCyclotomicField(S)`

Given a set or sequence S of cyclotomic field or ring elements, this function returns the smallest cyclotomic field or ring (possibly the rational field or integers) G containing each of the elements of S.

Example H36E1

We will demonstrate the difference between the "dense" and the "sparse" representation on the cyclotomic field of order 100.

```
> K1 := CyclotomicField(100);
> K2 := CyclotomicField(100: Sparse := true);
> K2!K1.1;
zeta(100)_4*zeta(100)_25^19
```

Where `zeta(100)_25` indicates a 25th root of unity in a field of order 100.

```
> K1!K2.1;
zeta_100^25
```

36.2.2 Creation of Elements

For elements of cyclotomic number fields the following conventions are used. Primitive roots of unity ζ_m are chosen in such a way that $\zeta_m^{m/d} = \zeta_d$, for every divisor d of m; one may think of this as choosing $\zeta_m = \mathrm{e}^{\frac{2\pi \mathrm{i}}{m}}$ (where the roots of unity are $\zeta_m^k = \mathrm{e}^{\frac{2k\pi \mathrm{i}}{m}}$) in the complex plane for every m (a convention that is followed for the explicit embedding in the complex domains).

Elements of cyclotomic fields and orders can also be created using coercion (`!`) and the elt constructor (`elt<|>`) where the left hand side is the field or order the element will lie in. For details about coercion see Section 34.2.3.

`RootOfUnity(n)`

Create the n-th root of unity ζ_n in $\mathbf{Q}(\zeta_n)$.

`RootOfUnity(n, K)`

Given a cyclotomic field $K = \mathbf{Q}(\zeta_m)$ and an integer $n > 2$, create the n-th root of unity ζ_n in K. An error results if $\zeta_n \notin K$, that is, if n does not divide m (or $2m$ in case m is odd).

`Minimise(~a)`
`Minimize(~a)`

Given an element a in a cyclotomic field F or ring R, this procedure finds the minimal cyclotomic subfield $E \subset F$ or subring $E \subset F$ containing a, and coerces a into E. Note that E may be $\mathbf{Q}$ or $\mathbf{Z}$.

`Minimise(~s)`
`Minimize(~s)`

Given a set s of cyclotomic field or ring elements, this procedure finds the minimal cyclotomic field or ring E containing all of them, and coerces each element into E. The resulting set will have universe E. Note that E may be $\mathbf{Q}$ or $\mathbf{Z}$.

`Minimise(a)`
`Minimize(a)`

Given an element a in a cyclotomic field F or ring R, this function finds the minimal cyclotomic subfield $E \subset F$ or subring $E \subset R$ containing a, and coerces a into E. Note that E may be $\mathbf{Q}$ or $\mathbf{Z}$.

`Minimise(s)`
`Minimize(s)`

Given a set s of cyclotomic field or ring elements, this function finds the minimal cyclotomic field E containing all of them, and coerces each element into E. The resulting set will have universe E. Note that E may be $\mathbf{Q}$ or $\mathbf{Z}$.

36.3 Structure Operations

In cyclotomic fields the generic ring functions are supported (see Chapter 17). The functions listed below are those functions for cyclotomic fields which are additional to those for number fields. For the list of functions applying to general number fields see Section 34.2 and Section 34.3.

36.3.1 Invariants

`Conductor(K)`

The smallest n such that the field K is contained in $\mathbf{Q}(\zeta_n)$; for a cyclotomic field that is either the 'cyclotomic order' m (see below) or half that, depending on whether $m \equiv 2 \bmod 4$. The second return value is a sequence of the ramified real places of K.

`CyclotomicOrder(K)`

The value of m for the cyclotomic field $\mathbf{Q}(\zeta_m)$. Note that this will be the m with which the cyclotomic field was created.

`CyclotomicAutomorphismGroup(K)`

Returns the automorphism group of K as an abstract abelian group G and a map from G into the set of all automorphisms. Note that similar functionality is also available through `AutomorphismGroup` however, this function returns an abelian group and uses the fact that the automorphism group is already determined by the conductor.

`CyclotomicRelativeField(k, K)`

Given two cyclotomic fields $k \subseteq K$ a number field L/k is computed that is isomorphic to K.

36.4 Element Operations

For the full range of operations for elements of a number field or order see Section 34.4.

36.4.1 Predicates on Elements

Because of the nature of cyclotomic fields and orders, some properties of elements are easier to determine than in the general case.

`IsReal(a)`

Whether the cyclotomic field or ring element a is a real number, i.e., if it is invariant under the complex conjugation.

36.4.2 Conjugates

Elements of cyclotomic fields and orders can additionally have their complex conjugate computed. Conjugates are returned as cyclotomic elements (and not reals) and which conjugate is wanted can be indicated by providing a primitive root of unity.

`ComplexConjugate(a)`

The complex conjugate of cyclotomic field or ring element a.

`Conjugate(a, n)`

The image under the map $\zeta \mapsto \zeta^n$. The second argument (n) must be coprime to the conductor.

`Conjugate(a, r)`

The conjugate of the element $a \in \mathbf{Q}(\zeta_m)$ or its order, obtained by applying the field automorphism $\zeta_m \mapsto r$ where r is a primitive root of unity.

Example H36E2

The following lines of code generate a set W of minimal polynomials for the so-called Gaussian periods

$$\eta_d = \sum_{i=0}^{(l-1)/d-1} \zeta_l^{g^{di}}$$

where ζ_l is a primitive l-th root of unity (l is prime), and where g is a primitive root modulo l. These have the property that they generate a degree d cyclic subfield of $\mathbf{Q}(\zeta_l)$. We (arbitrarily) choose $l = 13$ in this example.

```
> R<x> := PolynomialRing(RationalField());
> W := { R | };
> l := 13;
> L<z> := CyclotomicField(l);
> M := Divisors(l-1);
> g := PrimitiveRoot(l);
> for m in M do
>    d := (l-1) div m;
>    g_d := g^d;
>    w := &+[z^g_d^i : i in [0..m-1] ];
>    Include(~W, MinimalPolynomial(w));
> end for;
```

Here is the same loop in just one line, using sequence reduction:

```
> W := { R | MinimalPolynomial(&+[z^(g^((l-1) div m))^i : i in [0..m-1] ]) :
>        m in M };
> W;
{
      x^12 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 +
       x + 1,
      x^6 + x^5 - 5*x^4 - 4*x^3 + 6*x^2 + 3*x - 1,
      x^3 + x^2 - 4*x + 1,
      x + 1
      x^4 + x^3 + 2*x^2 - 4*x + 3,
      x^2 + x - 3,
}
```

37 ORDERS AND ALGEBRAIC FIELDS

Chapter 37
ORDERS AND ALGEBRAIC FIELDS

37.1 Introduction

The number field module in MAGMA is based on the Kant/Kash system (Kant-V4) [KAN97], [KAN00], developed by the group of M. Pohst in Berlin.

The three main structures which this chapter is concerned with are FldNum (number fields), RngOrd (orders in number fields) and FldOrd (fields of fractions of RngOrd's). Elements in these have types FldNumElt, RngOrdElt and FldOrdElt and ideals have types RngOrdIdl and RngOrdFracIdl. While number fields, and their sub-types cyclotomic fields and quadratic fields, are also detailed in separate chapters, (chapter 34 for number fields, 36 for cyclotomics and 35 for quadratics), almost all functionality described here applies to those objects as well.

On top of all there is a combined type that will match all "number field" types: FldAlg will match all references of type FldNum, FldOrd, FldCyc, and FldQuad (and similarly FldAlgElt will match all element types: FldNumElt, FldOrdElt, FldCycElt and FldQuadElt).

Number fields support extended types, they can be indexed by the type of the coefficient ring: FldNum[FldRat] refers to an absolute extension over $\mathbf{Q}$, while FldNum[FldNum] refers to a relative extension.

In order to use them efficiently, one has to understand the relations between the parent structures.

The basic distinction is between the number field point of view (FldNum) and the order based view (FldOrd, RngOrd).

We will start with the number fields. Formally, in MAGMA, an object K of type FldNum is an algebraic extension of finite degree over a a number field k or $\mathbf{Q}$. Thus it can be thought of as constructed as a quotient ring of a univariate polynomial ring over the base field modulo some irreducible polynomial: $K = k[t]/(f(t)k[t])$ Or, the field may be constructed as a multivariate quotient: $K = k[s_1, \ldots, s_n]/(f_1(s_1), \ldots, f_n(s_n))$ where all the polynomials are univariate. However, a slightly different representation is used internally.

It is important to remember that $\mathbf{Q}$ is **not** a number field.

One has to distinguish between number fields with primitive element α:= K.1 which is a zero of f and number fields where no primitive element is known. In this case α_i:= K.i will be a zero of f_i.

However, number fields always have a 'power' basis, i.e. a basis containing only powers of the zero(s) of the defining polynomial(s) and products of those powers.

An important consequence of this representation as a quotient of a polynomial ring is that one cannot distinguish between e.g. $\mathbf{Q}[2^{(1/3)}]$ and $\mathbf{Q}[\zeta_3 2^{(1/3)}]$ – both of them are generated using a root of $t^3 - 2$. Therefore every non trivial extension generates a **new**

object – even if the same polynomial is used repeatedly, except when the user explicitly tells MAGMA to check whether the polynomial has been used before.

An absolute extension is always an extension of $\mathbf{Q}$. An arbitrary number field K can always be converted into an isomorphic extension of $\mathbf{Q}$ using a constructive variant of the primitive element theorem.

Likewise, if a subfield k of K is known, an isomorphic field as an extension of k can be computed.

The most important facts about the various representations are the following:

* Usually, arithmetic is fastest in absolute extensions. Thus, if one wants to do lots of basic arithmetic with the elements the transformation to an absolute representation is advisable. However, typically the operations are fastest when the elements are "small" in size.
* Invariants (like `Degree`, `Discriminant`, `Norm`, `Trace` etc.) are always relative to the current representation.
* Conversions of fields tend to be time consuming thus should be avoided if possible. However, once the different field representations are computed, the conversion of elements is not too time consuming.
* Some operations and invariants can (currently) only be done for absolute representations. Essentially, these are computations involving subfields and class and unit group computations.

Number fields support only arithmetic with their elements and the computation of some invariants (`GaloisGroup`, `Subfields`, `AutomorphismGroup`). Although invariants like the class group can be computed for `FldNum`s this is only a shortcut for the corresponding computations for the maximal orders so e.g. `ClassGroup(K)` is expanded to `ClassGroup(MaximalOrder(K))`.

The other parent data-types are orders (`RngOrd`) and their fields of fractions (`FldOrd`). Orders can be constructed in basically two ways:

* as finite extensions E of a (maximal) order m (or of $\mathbf{Z}$) by a zero of a monic integral polynomial $f \in m[t]$
* via a transformation from a different order.

The main restriction for the construction of orders is that the coefficient domain (`BaseRing`) must always be a maximal order if any structural computations are desired. An order O over some maximal order m is represented using a (pseudo) m-basis (of type `PMat` similar to the way modules over Dedekind rings (`ModDed`, `PMat`) are represented in general. Every other order must be free over its base ring.

An order O is equipped with a unique field of fractions, `FieldOfFractions(O)`, which has the same basis as the order and whose base field is the field of fractions of the base ring. The fields of fractions support almost no structural computations, they merely serve as a parent structure to any elements and ideals that (may) have denominators (i.e. that are non-integral w.r.t. the current structure).

From a practical point of view, orders and their field of fractions have two different element data types, namely elements (`RngOrdElt`) and ideals (`RngOrdIdl`) (resp. `FldOrdElt`

and RngOrdFracIdl). This is technically not quite correct since ideals have formally different parents but those parents are trivial and the important information in them is the order the ideal is of. Ideals behave much more like elements than structures – they have no elements and are not rings in general. Formally, the parent structures obey the following rules:

* MaximalOrder(BaseRing(K)) eq BaseRing(MaximalOrder(K)) for all number fields and fields of fractions K.
* BaseField(FieldOfFractions(O)) eq FieldOfFractions(BaseRing(O)) for all orders O.
* BaseField(NumberField(O)) eq NumberField(BaseRing(O)) for all orders O.
* all orders within the same field share the identical number field. For example: IsIdentical(NumberField(O), NumberField(MaximalOrder(O))), while the corresponding fields of fractions will have, in general, different bases.

There are a few functions where orders behave differently to most other MAGMA objects, mostly because orders are not necessarily free modules over their base ring:

* Parent(O.i) eq FieldOfFractions(O) and furthermore O.i in O is usually false.
* Eltseq of an order element returns a sequence over the field of fractions of the base ring.
* The O.i are typically not a basis of the order – just part of a pseudo basis. However, they always form a basis of the field of fractions.

Algebraic number fields will be referred to as *number fields*. If a field may be either a number field or field of fractions it will be referred to as an *algebraic field*.

37.2 Creation Functions

The following describes how number fields, orders and their fields of fractions may be created. It also shows some ways of creating elements of these rings and homomorphisms from these rings into an arbitrary ring.

37.2.1 Creation of General Algebraic Fields

Algebraic Number Fields can be created in a various ways, all involving polynomials. The fields can be created as absolute extensions, i.e. an extension of **Q** by an irreducible polynomial, or as a relative extension which is an extension of an algebraic field by a polynomial irreducible over that field. Fields of fractions can be created as FieldOfFractions(O) where O is some order or from an existing field of fractions by using some of the constructors and functions described below. See also the corresponding sections for number fields section 34.2.1, section 36.2 for cyclotomics and section 35.2 for quadratics.

NumberField(f)

Check	BOOLELT	*Default :* true
DoLinearExtension	BOOLELT	*Default :* false

Global	BoolElt	*Default :* false

Given an irreducible polynomial f of degree $n \geq 1$ over $K = \mathbf{Q}$ or some number field K, create the number field $L = K(\alpha)$ obtained by adjoining a root α of f to K. For details see NumberField in Section 34.2.1.

The angle bracket notation may be used to assign the root α to an identifier e.g. L<y> := NumberField(f) where y will be a root of f.

RationalsAsNumberField()

QNF()

This creates a number field isomorphic to $\mathbf{Q}$. It is equivalent to NumberField(x-1) : DoLinearExtension), where x is PolynomialRing(Rationals()).1.

The result is a field isomorphic to $\mathbf{Q}$, but regarded by Magma as a number field (while $\mathbf{Q}$ itself is not, since FldRat is not a subtype of FldNum). It therefore supports all of the number field functions, while the Rationals() do not. On the other hand, arithmetic will be slower.

Coercion can be used to convert to and from the Rationals().

NumberField(s)

Check	BoolElt	*Default :* true
DoLinearExtension	BoolElt	*Default :* false
Abs	BoolElt	*Default :* false

Let K be a possibly trivial algebraic extension of $\mathbf{Q}$. Given a sequence s of nonconstant polynomials $s_1, \ldots, s_m$, that are irreducible over K, create the number field $L = K(\alpha_1, \ldots, \alpha_m)$ obtained by adjoining a root α_i of each s_i to K. For details see NumberField in Section 34.2.1.

ext< F | s1, ..., sn >

ext< F | s >

Check	BoolElt	*Default :* true
Global	BoolElt	*Default :* false
Abs	BoolElt	*Default :* false
DoLinearExtension	BoolElt	*Default :* false

Create the algebraic field defined by extending F by the polynomials s_i or the polynomials in the sequence s. Similar as for NumberField(S) described above, F may be $\mathbf{Q}$ or a field of fractions. If F is a field of fractions a field of fractions will be returned otherwise a number field will be returned. A tower of fields similar to that of NumberField is created and the same restrictions as for that function apply to the polynomials that can be used in the constructor.

`RadicalExtension(F, d, a)`

`Check`	BoolElt	*Default :* `true`

Let F be an algebraic field. Let a be an integral element of F chosen such that a is not an n-th power for any n dividing d. Returns the algebraic field obtained by adjoining the d-th root of a to F.

`SplittingField(F)`

`Abs`	BoolElt	*Default :* `true`
`Opt`	BoolElt	*Default :* `true`

Given an algebraic field F, return the splitting field of its defining polynomial. The roots of the defining polynomial in the splitting field are also returned.

If `Abs` is `true`, the resulting field will be an absolute extension, otherwise a tower is returned.

If `Opt` is `true`, an attempt of using `OptimizedRepresentation` is done. If successful, the resulting field will have a much nicer representation. On the other hand, computing the intermediate maximal orders can be extremely time consuming.

`SplittingField(f)`

Given an irreducible polynomial f over $\mathbf{Z}$, return its splitting field.

`SplittingField(L)`

`Abs`	BoolElt	*Default :* `false`
`Opt`	BoolElt	*Default :* `false`

Given a sequence L of polynomials over a number field or the rational numbers, compute a common splitting field, i.e. a field K such that every polynomial in L splits into linear factors over K. The roots of the polynomials are returned as the second return value.

If the optional parameter `Abs` is `true`, then a primitive element for the splitting field is computed and the field returned will be generated by this primitive element over $\mathbf{Q}$. If in addition `Opt` is also `true`, then an optimized representation of K is computed as well.

`sub< F | e1, ..., en >`

Given an algebraic field F with ground field G and n elements $e_i \in F$, return the algebraic field $H = G(e_1, \ldots, e_n)$ generated by the e_i (over G), as well as the embedding homomorphism from H to F.

`MergeFields(F, L)`

`CompositeFields(F, L)`

Let F and L be absolute algebraic fields. Returns a sequence of fields $[M_1, \ldots, M_r]$ such that each field M_i contains both a root of the generating polynomial of F and a root of the generating polynomial of L.

In detail: Suppose that F is the smaller field (wrt. the degree). As a first step we factorise the defining polynomial of L over F. For each factor obtained, an extension of F is constructed and then transformed into an absolute extension. The sequence of extension fields is returned to the user.

`Compositum(K, L)`

For absolute number fields K and L, at least one of which must be normal, find a smallest common over field. Note that in contrast to `CompositeFields` above the result here is essentially unique since one field was normal.

`Compositum(K, A)`

For an normal number field K and abelian extension A of some subfield of K, find a smallest common over field. Note that in contrast to `CompositeFields` above the result here is essentially unique since K is normal.

`OptimizedRepresentation(F)`

`OptimisedRepresentation(F)`

`OptimizedRepresentation(F, d)`

`OptimisedRepresentation(F, d)`

Given an algebraic field F with ground field $\mathbf{Q}$, this function will attempt to find an isomorphic field L with a better defining polynomial (in the sense defined below) than the one used to define F. If such a polynomial is found then L is returned; otherwise F will be returned.

If the argument d is not specified, a polynomial g with integer coefficients is defined to be better than f if it is monic, irreducible, defines a number field isomorphic to F and its discriminant is smaller (in absolute value) than that of f. If a second argument d is specified, then g is defined to be better if in addition to the previous requirements d is not an index divisor, that is, if d does not divide the index (defined in the Invariants sub–section) $[O_L : E_L]$ of the equation order E_L of L in the maximal order O_L, (which are defined in the next sub–section).

Note however, that as a first step this function will determine the maximal order of F which may take some time if the field is large.

Example H37E1

Some results of `OptimizedRepresentation` are shown.

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^4-420*x^2+40000);
> L := OptimizedRepresentation(K);
> L ne K;
true
> L;
Number Field with defining polynomial x^4 - 4*x^3 -
    17*x^2 + 42*x + 59 over the Rational Field
> L eq OptimizedRepresentation(L);
> f := DefiningPolynomial(L);
> Z := IntegerRing();
> Factorization(Z ! Discriminant(f));
[ <2, 18>, <5, 8>, <41, 2> ]
> g := x^4 - 4*x^3 - 17*x^2 + 42*x + 59;
> Factorization(Z ! Discriminant(g));
[ <2, 4>, <3, 4>, <5, 2>, <41, 2> ]
> OL := MaximalOrder(L);
> EL := EquationOrder(L);
> Index(OL, EL);
36
> OptimizedRepresentation(L, 2) eq L;
true
```

As we see from this computation, the prime 5 (as well as 41) divides the discriminant of g twice. This means that, potentially, 5 would still divide the index of the equation order in the maximal order O_L of L. However, in fact E_L has only index 36 in O_L.
The optimized representation of L such that 2 does not divide the index of E_L in O_L is L so 2 does divide the index seemingly contrary to the description above. However, if a more optimal representation cannot be found then the field is returned which is what happens here.

37.2.2 Creation of Orders and Fields from Orders

The maximal order $\mathcal{O}_K$ of an algebraic field and the equation order of a number field can be obtained from the field. Other orders of a field are unitary subrings of finite index in the ring of integers; they contain a subset of the integral elements in the field. The equation order $\mathcal{E}_K = \mathbf{Z}[\alpha]$ of $K = \mathbf{Q}(\alpha) \cong \mathbf{Q}[X]/f(X)$, where K is a number field defined by a monic integral polynomial, has the same basis as K, a power basis. Obviously $\mathcal{E}_K \subset \mathcal{O}_K$ since the minimal polynomial of α is integral and monic. Once an order is created in Magma further orders can be created from it.

`EquationOrder(f)`

`Check` BoolElt *Default :* `true`

Given an irreducible non-constant monic integral polynomial $f \in R[X]$, return the equation order $E = \mathbf{R}[X]/f(X)$ corresponding to f. If the optional parameter `Check` is set to `false` then the polynomial will not be checked for irreducibility.

`EquationOrder(K)`

Return the equation order corresponding to the polynomial with which the number field K was defined. K must have been defined by a monic integral polynomial. Thus this function returns the extension of the equation order of the ground field of K by the defining polynomial of K.

`SubOrder(O)`

Provided the order O is not an equation order, O is a transformation of some order O'. This function returns O'.

`EquationOrder(O)`

A suborder of the order O which is defined by a polynomial. e.g. $R[x]/f$ where R is a polynomial ring over the coefficient ring of O and f is in R. It will also be the final order of `SubOrder(SubOrder(... SubOrder(O)))`. O must have a monic defining polynomial for the equation order to exist.

`Integers(O)`
`RingOfIntegers(O)`
`IntegerRing(O)`

Returns the ring of integers in the order O, ie. O itself.

Example H37E2

Once a number field $K = \mathbf{Q}(\alpha)$ has been created, one can obtain the equation order $E = \mathbf{Z}[\alpha]$ and the ring of integers O_K simply as follows.

```
> R<x> := PolynomialRing(Integers());
> K := NumberField(x^4-420*x^2+40000);
> E := EquationOrder(K);
> O := MaximalOrder(K);
> Index(O, E);
64000
```

Note that entirely different things happen here: for the equation order nothing has to be computed, but the determination of the maximal order involves the complicated Round 2 (or 4) algorithm. In our particular example above, E is a subring of index $2^9 \cdot 5^3$ in O (see also the example for orders and ideals in the subsection 37.2.3.1 where a maximal order is created).

`sub< O | a1, ..., ar >`

Create the suborder of the order O generated (as an algebra over $\mathbf{Z}$) by the elements $a_1, \ldots, a_r \in O$, that is, create $\mathbf{Z}[a_1, \ldots, a_r]$. If the algebra does not have full rank as a sub-module of O, an error results. Note, however, that it is currently *not* required that 1 is in the sub-ring.

`ext< O | a1, ..., ar >`

Given an order O, and elements $a_1, \ldots, a_r$ lying in the maximal order of O, create the order $O[a_1, \ldots, a_r]$. Note that using this constructor O can only be extended to be as large as the maximal order. This does not cause the maximal order to get computed. See also `Order` for a different version that allows parameters to improve efficiency.

`ext< O | f >`

Given an order O and a polynomial f of degree n with coefficients in O, create the extension E of O by a root of f which forms a free module of rank n over O : $E \cong O[\alpha]$; it is necessary for f to be irreducible over O.

`FieldOfFractions(O)`

Return the field containing all fractions of elements of O. The angle bracket notation can be used to assign names to the basis elements of F and assign these elements to variables, e.g. `F<x, y> := FieldOfFractions(MaximalOrder(x^2 + 3))`.

`Order(F)`

The order of which F was created as its field of fractions. This function is an inverse to `FieldOfFractions`.

`NumberField(O)`

The number field of an order is recursively defined by:

1. the number field of $\mathbf{Z}$ is $\mathbf{Q}$
2. the number field of O is the number field of the coefficient ring of O, (i.e. the order over which O is defined), with an element α adjoined where α is a root of the defining polynomial of O.

`NumberField(F)`

The number field of `Order(F)` for a field of fractions F.

Example H37E3

The following illustrates the relationship between the bases of an order, its field of fractions and its number field.

```
> R<x> := PolynomialRing(Integers());
> f := x^5 + 5*x^4 - 75*x^3 + 250*x^2 + 65625;
> M := MaximalOrder(f);
> M;
Maximal Order of Equation Order with defining polynomial x^5 + 5*x^4 - 75*x^3 +
    250*x^2 + 65625 over its ground order
> Basis(FieldOfFractions(M));
[
    M.1,
    M.2,
    M.3,
    M.4,
    M.5
]
> Basis(NumberField(M));
[
    1,
    $.1,
    $.1^2,
    $.1^3,
    $.1^4
]
> Basis(M);
[
    M.1,
    M.2,
    M.3,
    M.4,
    M.5
]
> M.1 eq 1;
true
> M.2 eq NumberField(M).1;
false
> E := EquationOrder(M);
> NumberField(M) eq NumberField(E);
true
> Basis(FieldOfFractions(E), NumberField(M));
[
    1,
    $.1,
    $.1^2,
    $.1^3,
    $.1^4
```

```
]
> M!Basis(FieldOfFractions(E))[1];
[1, 0, 0, 0, 0]
> M!Basis(FieldOfFractions(E))[2];
[0, 5, 0, 0, 0]
> M!NumberField(M).1;
[0, 5, 0, 0, 0]
```

`OptimizedRepresentation(O)`
`OptimisedRepresentation(O)`
`OptimizedRepresentation(O, d)`
`OptimisedRepresentation(O, d)`

Given an order O with ground ring $\mathbf{Z}$, this function will attempt to find an isomorphic maximal order M with a better defining polynomial than the one used to define O. If such a polynomial is found then M is returned; otherwise O will be returned.

If the argument d is not specified, a polynomial g with integer coefficients is defined to be better than f if it is monic, irreducible, defines an order isomorphic to O and its discriminant is smaller in absolute value than that of f. If a second argument d is specified, then g is defined to be better if in addition to the previous requirements d is not an index divisor, that is, if d does not divide the index (defined in the Invariants sub–section) $[M : E_M]$ of the equation order E_M of M in M.

`O + P`

Add two orders O and P having the same equation order. Computes the smallest common over order.

`O meet P`

The intersection of two orders O and P having the same equation order.

`AsExtensionOf(O, P)`

Return the order O as an transformation of the order P where O and P have the same coefficient ring.

`Order(O, T, d)`

`Check` BoolElt *Default :* `true`

Let O be an absolute order with basis $b_1, \ldots, n_n$, $T = (T_{i,j}) \in \mathrm{GL}(n, \mathbf{Q}) \cap \mathrm{Mat}(n, \mathbf{Z})$ and $d \in \mathrm{N}$. This function creates the order with basis $(1/d \sum_{j=1}^{n} T_{i,j} b_j)_{i \le i \le n}$. `Check` can be set to `false` when the order is large to avoid checking that the result actually is an order.

`Order(O, M)`

Let O be an order with (pseudo) basis $b_1, \ldots, b_n$ and $M = \sum_{i=1}^{n} A_i \alpha_i \subseteq k^n$ be an o_k-module where o_k is the coefficient ring of O. This function creates the order $\sum_{i=1}^{n} A_i c_i$ where $c_i := \sum_{j=1}^{n} \alpha_{i,j} b_j$. If `Check` is set to `false`, then it will not be checked that the result actually is an order (potentially expensive).

`Order( [ e`$_1$`, ... e`$_n$` ] )`

`Verify`	BOOLELT	*Default :* `true`
`Order`	BOOLELT	*Default :* `false`

Given n elements $e_1, \ldots, e_n$ in an algebraic extension field F over $\mathbf{Q}$ create the minimal order O of F which contains all the e_i. If `Verify` is `true`, it is verified that the e_i are integral algebraic numbers. This can be a lengthy process if the field is of large degree.

Setting `Order` to `true` assumes that the given elements actually form a basis for the new order, thus it avoids testing for multiplicative closure. Without this parameter the order returned will have a canonical basis chosen with no direct relation to the input. By default, products of the generators will be added until the module is closed under multiplication.

37.2.3 Maximal Orders

The maximal order $\mathcal{O}_K$ is the ring of integers of an algebraic field consisting of all integral elements of the field; that is, elements which are roots of monic integer polynomials. It may also be called the number ring of a number field.

There are a number of algorithms which MAGMA uses whilst computing maximal orders. Each maximal order is a sum of p-maximal orders. The main algorithm used for p-maximal orders is a mixture of the Round–2 and Round–4 methods ([Coh93, Bai96, Poh93, PZ89]) for absolute extensions and a variant of the Round–2 for relative extensions ([Coh00, Fri97]).

However if the field is a radical (pure) extension, there is another algorithm available which is used to calculate each p-maximal order. In this case we can compute a pseudo basis for the p-maximal orders knowing only the valuation of the constant coefficient of the defining polynomial at p [Sut12].

The Round–2 and Round–4 algorithms can be selected by setting the parameter `Al` to `"Round2"` and `"Round4"` respectively. Another option for this parameter and for computation of p-maximal orders is the `"Pauli"` method. This method is only available for equation orders in simple relative extensions. It uses the factorization of the defining polynomial over the completion of the order.

Alternatively, if the discriminant of the maximal order is already known, the parameters `Discriminant` or `Ramification` can be used. If the input is an order O and the `Discriminant` or `Ramification` parameters are supplied an algorithm which can compute the maximal order given the discriminant of the maximal order will be used. `Discriminant` must be an integer if O is an absolute order and must be an ideal of the coefficient ring of O if O is a relative order. `Ramification` must contain integers if O is an absolute

order and must contain ideals of the coefficient ring of O if O is a relative order. The ramification sequence is taken to contain prime factors of the discriminant. Only one of these parameters can be specified and if one of them is then `Al` cannot be specified. This algorithm is based on [Bj94], Theorems 1.2 and 7.6.

`MaximalOrder(O)`

`Al`	MonStgElt	*Default :* "*Auto*"

`MaximalOrder(F)`
`IntegerRing(F)`
`Integers(F)`
`RingOfIntegers(F)`

`Discriminant`	Any	*Default :*
`Ramification`	SeqEnum	*Default :*
Verbose	`MaximalOrder`	*Maximum :* 5

Return the maximal order or ring of integers of the number field F. When the input is an order O or a field of fractions of an order O return the order containing O which is the largest order in the number field of O.

An integral basis for F can be found as the basis of the maximal order.

For information on the parameters, see the introduction to this section above, 37.2.3.

`MaximalOrder(f)`

`Check`	BoolElt	*Default :* `true`
`Al`	MonStgElt	*Default :* "*Auto*"
`Discriminant`	Any	*Default :*
`Ramification`	SeqEnum	*Default :*
Verbose	`MaximalOrder`	*Maximum :* 5

This is equivalent to `MaximalOrder(NumberField(f))`.

The `Check` parameter if set to `false` will prevent checking of the polynomial for irreducibility.

For information on the other parameters, see the introduction to this section above.

Example H37E4

The following shows the advantage of the `Ramification` parameter to the `MaximalOrder` function.

```
> R<t> := PolynomialRing(Integers());
> f1 := t^14 - 63*t^12 - 9555*t^11 + 118671*t^10 - 708246*t^9 - 17922660*t^8 +
> 859373823*t^7 + 2085856500*t^6 - 117366985106*t^5 - 335941176396*t^4 +
> 4638317668005*t^3 + 17926524826973*t^2 + 7429846568445*t+ 91264986397629;
> d1 := [2, 3, 5, 7, 59];
```

```
> time MaximalOrder(f1:Ramification := d1);
Maximal Order of Equation Order with defining polynomial x^14 - 63*x^12 -
    9555*x^11 + 118671*x^10 - 708246*x^9 - 17922660*x^8 + 859373823*x^7 +
    2085856500*x^6 - 117366985106*x^5 - 335941176396*x^4 + 4638317668005*x^3 +
    17926524826973*x^2 + 7429846568445*x + 91264986397629 over Z
Time: 0.230
> time MaximalOrder(f1);
Maximal Order of Equation Order with defining polynomial x^14 - 63*x^12 -
    9555*x^11 + 118671*x^10 - 708246*x^9 - 17922660*x^8 + 859373823*x^7 +
    2085856500*x^6 - 117366985106*x^5 - 335941176396*x^4 + 4638317668005*x^3 +
    17926524826973*x^2 + 7429846568445*x + 91264986397629 over Z
Time: 0.590
> f2 := t^14 - 129864*t^12 - 517832*t^11 + 6567239322*t^10 + 33352434192*t^9 -
> 166594899026864*t^8 - 752915315481312*t^7 + 2275891736459084940*t^6 +
> 7743078094604088768*t^5 - 16633213695413438344032*t^4 -
> 39871919309692447523616*t^3 + 60126791399546070679893112*t^2 +
> 778441185338527286987510 40*t - 83173498199506854751458701376;
> d2 := [2,3,7,4145023];
>
> time MaximalOrder(f2:Ramification := d2);
Maximal Order of Equation Order with defining polynomial x^14 - 129864*x^12 -
    517832*x^11 + 6567239322*x^10 + 33352434192*x^9 - 166594899026864*x^8 -
    752915315481312*x^7 + 2275891736459084940*x^6 + 7743078094604088768*x^5 -
    16633213695413438344032*x^4 - 39871919309692447523616*x^3 +
    60126791399546070679893112*x^2 + 778441185338527286987510 40*x -
    83173498199506854751458701376 over Z
Time: 0.730
> time MaximalOrder(f2);
Maximal Order of Equation Order with defining polynomial x^14 - 129864*x^12 -
    517832*x^11 + 6567239322*x^10 + 33352434192*x^9 - 166594899026864*x^8 -
    752915315481312*x^7 + 2275891736459084940*x^6 + 7743078094604088768*x^5 -
    16633213695413438344032*x^4 - 39871919309692447523616*x^3 +
    60126791399546070679893112*x^2 + 778441185338527286987510 40*x -
    83173498199506854751458701376 over Z
Time: 0.840
> f13 := t^15 - 114*t^14 + 282185319*t^13 + 1247857228852*t^12 -
> 35114805704965233*t^11 - 14152433779643338 7826*t^10 +
> 26045849804422640287440 09*t^9 + 14153948932132918272984150384*t^8 -
> 17827307724835336994132762847955 2*t^7 - 11429535068213909144192605644943047 68
> *t^6 + 15975069142211276963134599495014990639616*t^5 +
> 33516684438303088018217308253251277376159744*t^4 -
> 61758977710820371623239637245361930955447125 6064*t^3 -
> 39756141244506691954546176235488463150180617486 3360*t^2 +
> 22666571829085475706482454642151923578020471016281866 24*t
> - 13022224565327602564069162232593069605616574287778141962 24;
> d13 := [2, 3, 3377890562461, 7623585272461];
> time MaximalOrder(f13: Ramification := d13);
Maximal Order of Equation Order with defining polynomial x^15 - 114*x^14 +
```

```
    282185319*x^13 + 1247857228852*x^12 - 3511480570496523 3*x^11 -
    14152433779643338782 6*x^10 + 260458498044226402874400 9*x^9 +
    14153948932132918272984150384*x^8 - 178273077248353369941327628479552*x^7 -
    11429535068213909144192605644943047 68*x^6 +
    15975069142211276963134599495014990639616*x^5 +
    33516684438303088018217308253251277376159744*x^4 -
    61758977710820371623239637245361930955447125606 4*x^3 -
    39756141244506691954546176235488463150180617486 3360*x^2 +
    22666571829085475706482454642151923578020471016281866 24*x -
    13022224565327602564069162232593069605616574287778141962 24 over Z
Time: 0.270
>  time MaximalOrder(f13);
```

The last line did not complete running in over 7 hours.

37.2.3.1 Orders and Ideals

Orders may be created using ideals of another order. Ideals are discussed in Section 37.9. The following intrinsics form part of the computation of maximal orders as discussed above in Section 37.2.3.

`pMaximalOrder(O, p)`

`Al` MonStgElt *Default :* "*Auto*"

The p-maximal overorder of O (see also the example below). This is the largest overorder P such that the index $(P : O)$ is a power of p, a prime in the coefficient ring of O. The options for the `Al` parameter are the same as those for `MaximalOrder`.

If O is a kummer extension then specific code is used to calculate each p-maximal order, rather than the Round 2 or Round 4 methods. In this case we know 1 or 2 elements which generate the p-maximal order and can easily write the order down.

`pRadical(O, p)`

Returns the p-radical of an order O for a prime p in the coefficient ring of O, defined as the ideal consisting of elements of O for which some power lies in the ideal pO.

It is possible to call this function for p not prime (so long as p is greater than the degree of O if it is an integer). In this case the p-trace-radical will be computed, i.e.

$$\{x \in F \mid \mathrm{Tr}(xO) \subseteq p\mathbf{Z}\}.$$

If p is square free and all divisors are larger than the field degree, this is the intersection of the radicals for all l dividing p. In particular together with `MultiplicatorRing` this can sometime be used to compute maximal orders without factoring the discriminant [Bj94, Fri00] or at least "good" approximations.

`MultiplicatorRing(I)`

Returns the multiplicator ring M of the ideal I of the order O, that is, the subring of elements of the field of fractions K of O multiplying I into itself: $M = \{x \in F : xI \subset I\}$.

Example H37E5

To illustrate how the Round 2 algorithm for the determination of the ring of integers works, we present an implementation of it in the MAGMA language. The key functions are `MultiplicatorRing` and `pRadical`, called by the following function `pMaximalOverOrder`;

```
> pMaximalOverOrder := function(ord, p)
>        ovr := MultiplicatorRing(pRadical(ord, p));
>        print "index is", Index(ovr, ord);
>        return (Index(ovr, ord) eq 1) select ovr else $$(ovr, p);
> end function;
```

which finds the largest overorder in which the given order has p-power index. This function is now simply applied to the equation order, for each prime dividing the discriminant:

```
> Round2 := function(E, K)
>        // E should be some order of a number field K
>        d := Discriminant(E);
>        fact := Factorization(Abs(d));
>        print fact;
>        M := E;
>        for x in fact do
>             M := M+pMaximalOverOrder(E, x[1]);
>        end for;
>        print "index of equation order in maximal order is:", Index(M, E);
>        return M;
> end function;
```

In our running example, this produces the following output:

```
> R<x> := PolynomialRing(Integers());
> K := NumberField(x^4-420*x^2+40000);
> E := EquationOrder(K);
> Round2(E, K);
[ <2, 18>, <5, 8>, <41, 2> ]
index is 2
index is 4
index is 8
index is 4
index is 2
index is 1
index is 5
index is 25
index is 1
index is 1
index of equation order in maximal order is: 64000
Transformation of E
Transformation Matrix:
[800   0   0   0]
[  0 400   0   0]
[  0 200  20   0]
```

```
[400 180   0   1]
Denominator: 800
```

37.2.4 Creation of Elements

Elements of algebraic fields and of orders are displayed quite differently. Algebraic field elements are always printed as a linear combination with rational coefficients of the basis elements of the field. For number fields which have a power basis this is also a polynomial in the primitive element of the field with rational coefficients; that is, an element of $G[x]/f$, where f was the defining polynomial of the field over the ground field G. Since in general G will be an algebraic field itself, elements in relative extensions, i.e. G strictly bigger than $\mathbf{Q}$, will be printed as linear combinations with linear combinations as coefficients and for number fields this will look like multivariate polynomials. In fact they are recursively defined univariate polynomials.

Elements of orders are displayed as sequences of integer coefficients, referring to the basis of the order. To convert this $\mathbf{Z}$-basis representation to a polynomial expression in the primitive element of an associated number field, the element should be coerced into the number field (using `!`). To print the element as a linear combination of the basis elements, coerce the element into the field of fractions.

```
F ! a
```
```
elt< F | a >
```

Coerce a into the field F. Here a may be an integer or a rational field element, or an element from a subfield of F, or from an order in such.

```
F ! [a0, a1, ..., am−1]
```
```
elt< F | [ a0, a1, ..., am−1 ] >
```
```
elt< F | a0, a1, ..., am−1 >
```

Given the algebraic field, F of degree m over its ground field G and a sequence $[a_0, \ldots, a_{m-1}]$ of elements of G, construct the element $a_0\alpha_0 + a_1\alpha_1 + \cdots a_{m-1}\alpha_{m-1}$ of F where the α_i are the basis elements of F.

```
O ! a
```
```
elt< O | a >
```

Coerce a into the order O. Here a is allowed to be an integer, or an integral element of an associated algebraic field of O, or an element of a quotient order.

`O ! [a0, a1, ..., am−1]`

`elt< O | [ a0, a1, ..., am−1 ] >`

`elt< O | a0, a1, ..., am−1 >`

Given the order O of degree m and elements $a_0, a_1, \ldots, a_{m-1}$ in the ground order of O, construct the element $a_0\alpha_0 + a_1\alpha_1 + \cdots + a_{m-1}\alpha_{m-1}$ of O, where $\alpha_0, \ldots, \alpha_{m-1}$ is the basis for the order.

`Random(F, m)`

`Random(O, m)`

A random element of the algebraic field F or order O. The maximal size of the coefficients is determined by m.

`Random(I, m)`

A random element of the ideal I as an element of the field of fractions of the associated order. The maximal size of the coefficients with respect to the ideal basis is determined by m.

Example H37E6

Here are three ways of creating the same integral element in K as an element of the maximal order and its field of fractions.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> O := MaximalOrder(K);
> e := O ! (y^2/40 + y/4);
> f := elt< O | [0, 0, 1, 0]>;
> f eq e;
true
> F<a, b, c, d> := FieldOfFractions(O);
> g := F![0, 0, 1, 0];
> g eq e;
true
> g;
c
```

These constructions would have failed if the element was not in O.

`One(K)` `One(O)` `Identity(K)` `Identity(O)`

`Zero(K)` `Zero(O)` `Representative(K)` `Representative(O)`

37.2.5 Creation of Homomorphisms

To specify homomorphisms from algebraic fields or orders in algebraic fields, it is necessary to specify the image of the generating elements, and possible to specify a map on the ground field.

```
hom< F -> R | r >
```

```
hom< F -> R | h, r >
```

Given an algebraic field F, defined as an extension of the ground field G, as well as some ring R, build the homomorphism ϕ obtained by sending the defining primitive element α of F to the element $r \in R$.

If F is a field of fractions then r will be the image of the primitive element of the field of fractions of the equation order of `Order(F)`.

It is possible (if $G = \mathbf{Q}$) and sometimes necessary (if $G \neq \mathbf{Q}$) to specify a homomorphism ϕ on F by specifying its action on G by providing a homomorphism h with G as its domain and R its codomain together with the image of α. If R does not cover G then the homomorphism h from G into R is necessary to ensure that the ground field can be mapped into R.

```
hom< O -> R | r >
```

```
hom< O -> R | h, r >
```

Given an order O, a ring R and an element $r \in R$, construct a homomorphism ϕ by sending the primitive element of the equation order of O to r.

To be more precise: Let K be the field of fractions of O, k be the base field of K i.e. the field of fractions of the base ring of O, and $N :=$ `NumberField(O)`. As a k–algebra K is generated by $x := K!N.1$, so x is zero of `DefiningPolynomial(O)`. The element r given in the map construction will be the image of x.

When O is an equation order e.g. if O was defined using a monic integral polynomial x will be `O.2`.

As in the field case it is possible to specify a map on the coefficient ring of O (ring over which O is defined) with codomain R. This is necessary if R does not cover the coefficient ring of O.

Example H37E7

We show a way to embed the field $\mathbf{Q}(\sqrt{2})$ in $\mathbf{Q}(\sqrt{2}+\sqrt{3})$. The application of the homomorphism suggests how the image could have been chosen.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^2-2);
> KL<w> := NumberField(x^4-10*x^2+1);
> H := hom< K -> KL | (9*w-w^3)/2 >;
> H(y);
1/2*(-w^3 + 9*w)
> H(y)^2;
```

```
2
```

Homomorphisms can be created between any order or algebraic field and any ring.

```
> f := x^4 + 5*x^3 - 25*x^2 + 125*x + 625;
> M := MaximalOrder(f);
> F<a, b, c, d> := FieldOfFractions(M);
> FF := FiniteField(5, 3);
> F;
Field of Fractions of M
> FF;
Finite field of size 5^3
> h := hom< F -> FF | Coercion(Rationals(), FF), 3*FF.1>;
> h;
Mapping from: FldOrd: F to FldFin: FF
>  h(a); h(b);
1
>> h(a); h(b);
          ^
Runtime error in map application: Application of map failed
> h(5*b); h(5*5*c); h(5*5*5*d);
FF.1^94
FF.1^64
FF.1^34
```

This unexpected behaviour occurs because when the basis of F is expressed with respect to the power basis of the number field they have denominator divisible by 5. A more well–behaved example is shown below.

```
> FF := FiniteField(11, 5);
> h := hom< F -> FF | Coercion(Rationals(), FF), 7*FF.1>;
> h(a);
1
> h(b); h(c); h(d);
FF.1^48316
FF.1^96632
FF.1^144948
> 7*FF.1;
FF.1^112736
> 5*h(b);
FF.1^112736
> PrimitiveElement(F);
5/1*b
```

`hom< O -> R | `$b_1, ..., b_n$` >`

`hom< O -> R | m, `$b_1, ..., b_n$` >`

Return the map from the order O of an algebraic number field to the ring R mapping the basis elements to $b_1, .., b_n$. If given, the map m should be from the coefficient ring of O to R and will be used to map the coefficients of the basis elements. If not given the coefficient ring of O should by covered by R.

`IsRingHomomorphism(m)`

Return whether the vector space homomorphism m is a homomorphism of rings.

37.3 Special Options

It is possible to obtain output generated by the KANT package using the verbose printing feature of MAGMA, furthermore, the general style of the output can be changed. It is also possible to alter the precision used for internal real computations.

`SetVerbose(s, n)`

There are a number of verbose flags `s` applying to the functions described in this chapter. The flags and their levels `n` are mentioned in the descriptions of the functions which use them.

Note however, that setting the verbose levels may produce unexpected results since the effective scope of the flags is a little bit vague. Consider the following example:

```
> SetVerbose("MaximalOrder", 1);
> SetVerbose("Factor", 1);
> L := NumberField(PolynomialRing(Rationals()).1^2-10);
Factorize square-free polynomial over Z of degree 2
Deflation factor: 2
Number of deflated factors: 1
Factor inflated polynomial 0 of degree 2
Total factorization time: 0.000
Final irreducibility test factorization:
<x^2 - 10, 0>
> Regulator(L);
order_maximal_sub: called with algo_flag: 0
no algorithm selected
nothing about algebra-splitting selected
nothing about reduced-discriminant selected
nothing about dedekind-test selected
order_maximal_sub: calling order_maximal_sub_sub
order_maximal_sub_sub: called with algo_flag: 16
no algorithm selected
nothing about algebra-splitting selected
```

```
use reduced-discriminant selected
nothing about dedekind-test selected
red disc: f =x^2 - 10
 r_disc = 20
Reduced discriminant: 20
Factorization of reduced discriminant:
2^2 * 5^1
calculation and factorisation of reduced discriminant: 0.01
Factorization of discriminant:
2^3 * 5^1
factors with (possibly) not maximal overorder:
2^3
-----------------------
order_max_p_sub called:
prime: 2, prime_bound: 3, algo_flag: 16
-----------------------
no algorithm selected
nothing about algebra-splitting selected
nothing about dedekind-test selected
--------------------------
order_max_p_sub_sub called:
prime: 2, prime_bound: 3, algo_flag: 89
--------------------------
round2 selected
no algebra-splitting selected
use dedekind-test selected
No split performed ...
(due to user advice or impossible) ...
standard algorithm.
---------------------------
order_max_p_rnd2_sub called:
prime: 2, prime_bound: 3, algo_flag: 89
---------------------------
use dedekind-test selected
Order is already 2-maximal.
1.818446459232066823483698963560708993786253942768999999999
```

The first few lines of output are generated, because the creation of number fields involves a test of irreducibility for the defining polynomial(s).

The next group of lines come from the computation of the maximal order which is used for the regulator computation.

In general the amount of output generated increases with the value supplied. Furthermore, the output corresponding to larger values gets more and more technical.

`SetKantPrinting(f)`

Kant-style printing means that integers and rational numbers will be printed as integers and rational numbers. Especially in relative extensions this produces easier to read output - but it is no longer possible to paste the output back into the system again. Turns Kant-style printing on if f is `true` and off if f is `false`.

`SetKantPrecision(n)`

`SetKantPrecision(O, n)`

`SetKantPrecision(O, n, m)`

`SetKantPrecision(F, n)`

`SetKantPrecision(F, n, m)`

For internal real computations, some number field functions use real arithmetic to a fixed precision; therefore every algebraic field and every order comes equipped with some real-rings.

Initially, the precision (in decimal digits) will be the maximum of the set precision P, 20 and four times the degree of the algebraic field (order) in which calculations are performed. By default $P = 52$.

In addition all unit computations use a second real-ring in which the default precision is always twice the ordinary precision. This ring can be set using the third argument of `SetKantPrecision`.

Furthermore, the computation of the zeroes of the defining polynomial of the field or order uses a third, independent, precision which initially is the maximum of 32 and the (binary) logarithm of the largest coefficient of the defining polynomial.

Note however, that several functions will work automatically with a much larger precision if necessary to guarantee P digits for the user. Generally all functions take care of the necessary precision. Only under rare circumstances will any computation fail because of precision loss.

Certain calculations that fail if the precision is too small may succeed after restarting with increased precision.

If the order O or field F is given, then the precision change will apply only in the context of that order (field).

37.4 Structure Operations

In the lists below F usually refers to an algebraic field, K to a number field and O to an order.

37.4.1 General Functions

Number fields form the MAGMA category `FldNum`, orders form `RngOrd` and their fields of fractions form `FldOrd`. The notional power structures exist as parents of algebraic fields and their orders, no operations are allowed.

`Category(F)` `Parent(F)` `Category(O)` `Parent(O)`

`AssignNames(~K, s)`

Procedure to change the names of the generating elements in the number field K to the contents of the sequence of strings s.

The i-th sequence element will be the name used for the generator of the $(i-1)$-st subfield down from K as determined by the creation of K, the first element being used as the name for the generator of K. In the case where K is defined by more than one polynomial as an absolute extension, the ith sequence element will be the name used for the root of the ith polynomial used in the creation of K.

This procedure only changes the names used in printing the elements of K. It does *not* assign to any identifiers the value of a generator in K; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies K, it is necessary to have a reference `~K` to K in the call to this function.

`Name(K, i)`
`K . i`

Given a number field K, return the element which has the i-th name attached to it, that is, the generator of the $(i-1)$-st subfield down from K as determined by the creation of K. Here i must be in the range $1 \leq i \leq m$, where m is the number of polynomials used in creating K. If K was created using multiple polynomials as an absolute extension, `K.i` will be a root of the ith polynomial used in creating K.

`AssignNames(~F, s)`

Assign the strings in the sequence s to the names of the basis elements of the field of fractions F.

`F . i`
`Name(F, i)`

Return the ith basis element of the field of fractions F.

`O . i`

Return the ith basis element of the order O.

37.4.2 Related Structures

Each order and field has other orders and fields which are related to it in various ways.

`GroundField(F)`
`BaseField(F)`
`CoefficientField(F)`
`CoefficientRing(F)`

Given an algebraic field F, return the algebraic field over which F was defined. For an absolute number field F, the function returns the rational field $\mathbf{Q}$.

`BaseRing(O)`
`CoefficientRing(O)`

Given an order O, this returns the order over which O was defined. For an absolute order O this will be the integers $\mathbf{Z}$.

`AbsoluteField(F)`

Given an algebraic field F, this returns an isomorphic number field L defined as an absolute extension (i.e. over $\mathbf{Q}$).

`AbsoluteOrder(O)`

Given an order O, this returns an isomorphic order O' defined as an order in an absolute extension (over $\mathbf{Q}$).

`SimpleExtension(F)`
`SimpleExtension(O)`

Given an algebraic field F or an order O, this returns an isomorphic field L defined as an absolute simple extension or the Z-isomorphic order in it.

`RelativeField(F, L)`

Given algebraic fields L and F such that MAGMA knows that F is a subfield of L, return an isomorphic algebraic field M defined as an extension over F.

Example H37E8

It is often desirable to build up a number field by adjoining several algebraic numbers to $\mathbf{Q}$. The following function returns a number field that is the composite field of two given number fields K and L, provided that $K \cap L = \mathbf{Q}$; if K and L have a common subfield larger than $\mathbf{Q}$ the function returns a field with the property that it contains a subfield isomorphic to K as well as a subfield isomorphic to L.

```
> R<x> := PolynomialRing(Integers());
> Composite := function( K, L )
>     T<y> := PolynomialRing( K );
>     f := T!DefiningPolynomial( L );
```

```
>     ff := Factorization(f);
>     LKM := NumberField(ff[1][1]);
>     return AbsoluteField(LKM);
> end function;
```

To create, for example, the field $\mathbf{Q}(\sqrt{2}, \sqrt{3}, \sqrt{5})$, the above function should be applied twice:

```
> K := NumberField(x^2-3);
> L := NumberField(x^2-2);
> M := NumberField(x^2-5);
> KL := Composite(K, L);
> S<s> := PolynomialRing(BaseField(KL));
> KLM<w> := Composite(KL, M);
> KLM;
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

Note, that the same field may be constructed with just one call to `NumberField` followed by `AbsoluteField`:

```
> KLM2 := AbsoluteField(NumberField([x^2-3, x^2-2, x^2-5]));
> KLM2;
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

or by

```
> AbsoluteField(ext<Rationals() | [x^2-3, x^2-2, x^2-5]>);
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

In general, however, the resulting polynomials of KLM and $KLM2$ will differ. To see the difference between `SimpleExtension` and `AbsoluteField`, we will create `KLM2` again:

```
> KLM3 := NumberField([x^2-3, x^2-2, x^2-5]: Abs);
> AbsoluteField(KLM3);
Number Field with defining polynomials [ x^2 - 3, x^2 - 2,
    x^2 - 5] over the Rational Field
> SimpleExtension(KLM3);
Number Field with defining polynomial s^8 - 40*s^6 + 352*s^4 - 960*s^2 + 576
over the Rational Field
```

`Simplify(O)`

Given an order O, obtained by a chain of transformations from an equation order E, return an order that is given directly by a single transformation over E.

`LLL(O)`

Given an order O, return an order O' obtained from O by a transformation matrix T, which is returned as a second value. O' will have a LLL-reduced basis.

Example H37E9

Using LLL to reduce the basis of an order is shown.

```
> M := MaximalOrder(x^4-14*x^3+14*x^2-14*x+14);
> L, T := LLL(M);
> L;
Maximal Order, Transformation of M
Transformation Matrix:
[  1   0   0   0]
[ -3   1   0   0]
[  3 -13   1   0]
[ -7   1 -13   1]
> T;
[  1   0   0   0]
[ -3   1   0   0]
[  3 -13   1   0]
[ -7   1 -13   1]
> Basis(M);
[
    M.1,
    M.2,
    M.3,
    M.4
]
> Basis(L, M);
[
    [1, 0, 0, 0],
    [-3, 1, 0, 0],
    [3, -13, 1, 0],
    [-7, 1, -13, 1]
]
> L eq M;
true
```

Even though L and M are considered to be equal because they contain the same elements L has a basis which is LLL reduced but M does not.
Following on from the orders and ideals example (H37E5) we have

```
> R<x> := PolynomialRing(Integers());
> K := NumberField(x^4-420*x^2+40000);
> E := EquationOrder(K);
> O := Round2(E, K);
[ <2, 18>, <5, 8>, <41, 2> ]
index is 2
index is 4
index is 8
index is 4
index is 2
index is 1
```

```
index is 5
index is 25
index is 1
index is 1
index of equation order in maximal order is: 64000
> L := LLL(O);
> O;
Transformation of E
Transformation Matrix:
[800   0   0   0]
[  0 400   0   0]
[  0 200  20   0]
[400 180   0   1]
Denominator: 800
> O:Maximal;
   F[1]
    |
   F[2]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^4 - 420*x^2 + 40000
Index: 64000/1
Signature: [4, 0]
> L;
Transformation of O
Transformation Matrix:
[-1  0  0  0]
[-1 -1  0  1]
[10  1 -2  0]
[ 5  1 -1  0]
> L:Maximal;
   F[1]
    |
   F[2]
    |
   F[3]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     Given by transformation matrix
F  [ 3]     x^4 - 420*x^2 + 40000
Index: 1/1
Signature: [4, 0]
> Simplify(L):Maximal;
   F[1]
```

```
    |
   F[2]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^4 - 420*x^2 + 40000
Index: 64000/1
Signature: [4, 0]
> Simplify(L);
Transformation of E
Transformation Matrix:
[-800    0    0    0]
[-400 -220    0    1]
[8000    0  -40    0]
[4000  200  -20    0]
Denominator: 800
```

`PrimeRing(F)` `PrimeField(F)` `PrimeRing(O)`
`Centre(F)` `Centre(O)`

`Embed(F, L, a)`

Install the embedding of a simple field F in L where the image of the primitive element of F is the element a of L. This embedding will be used in coercing from F into L.

`Embed(F, L, a)`

Install the embedding of the non-simple field F in L where the image of the generating elements of F are in the sequence a of elements of L. This embedding will be used in coercing from F into L.

`EmbeddingMap(F, L)`

Returns the embedding map of F in L if an embedding is known.

Example H37E10

MAGMA does not recognize two independently created number fields as equal since more than one embedding of a field in a larger field may be possible. To coerce between them then it is convenient to be able to embed them in each other.

```
> k := NumberField(x^2-2);
> l := NumberField(x^2-2);
>  l!k.1;
>> l!k.1;
    ^
Runtime error in '!': Arguments are not compatible
LHS: FldNum
RHS: FldNumElt
> l eq k;
false
> Embed(k, l, l.1);
> l!k.1;
l.1
> Embed(l, k, k.1);
> k!l.1;
k.1
```

`Embed` is useful in specifying the embedding of a field in a larger field.

```
> l<a> := NumberField(x^3-2);
> L<b> := NumberField(x^6+108);
> Root(L!2, 3);
1/18*b^4
> Embed(l, L, $1);
> L!l.1;
1/18*b^4
```

Another embedding would be

```
> Roots(PolynomialRing(L)!DefiningPolynomial(l));
[
    <1/36*(-b^4 - 18*b), 1>,
    <1/36*(-b^4 + 18*b), 1>,
    <1/18*b^4, 1>
]
> Embed(l, L, $1[1][1]);
> L!l.1;
1/36*(-b^4 - 18*b)
```

`Lattice(O)`

`MinkowskiLattice(O)`

Given an absolute order O, returns the lattice determined by the real and complex embeddings of O.

`MinkowskiSpace(F)`

The Minkowski vector space V of the absolute field F as a real vector space, with inner product given by the T_2-norm (`Length`) on F, and by the embedding $F \to V$.

`Completion(K, P)`

`Completion(O, P)`

`comp< K|P >`

`comp< O|P >`

`Precision` RngIntElt *Default :* 50

For an absolute extension K of $\mathbf{Q}$ or O of $\mathbf{Z}$, compute the completion at a prime ideal P which must be either a prime ideal of the maximal order or unramified. The result will be a local field or ring with precision `Precision` or e`*Precision` if the ideal is ramified with ramification degree e.

The returned map is the canonical injection into the completion. It allows pointwise inverse operations.

`Completion(K, P)`

`Precision` RngIntElt *Default :* 50

For an absolute extension K over $\mathbf{Q}$ and a (finite) place P, compute the completion at P. The precision and the map are as described for `Completion`.

`LocalRing(P, prec)`

The completion of `Order(P)` at the prime ideal P up to precision `prec`.

37.4.3 Representing Fields as Vector Spaces

It is possible to express a number field or a field of fractions as a vector space of any subfield using the intrinsics below. Such a construction also allows one to find properties of elements over these subfields.

`Algebra(K, J)`

`Algebra(K, J, S)`

Returns the associative structure constant algebra which is isomorphic to the algebraic field K as an algebra over J. Also returns the isomorphism from K to the algebra mapping w^i to the $i+1$st unit vector of the algebra where w is a primitive element of K.

If a sequence S is given it is taken to be a basis of K over J and the isomorphism will map the ith element of S to the ith unit vector of the algebra.

VectorSpace(K, J)

KSpace(K, J)

VectorSpace(K, J, S)

KSpace(K, J, S)

The vector space isomorphic to the algebraic field K as a vector space over J and the isomorphism from K to the vector space. The isomorphism maps w^i to the $i+1$st unit vector of the vector space where w is a primitive element of K.

If S is given, the isomorphism will map the ith element of S to the ith unit vector of the vector space.

Example H37E11

We use the `Algebra` of a relative number field to obtain the minimal polynomial of an element over a subfield which is not in its coefficient field tower.

```
> K := NumberField([x^2 - 2, x^2 - 3, x^2 - 7]);
> J := AbsoluteField(NumberField([x^2 - 2, x^2 - 7]));
> A, m := Algebra(K, J);
> A;
Associative Algebra of dimension 2 with base ring J
> m;
Mapping from: RngOrd: K to AlgAss: A
> m(K.1);
(1/10*(J.1^3 - 13*J.1)                      0)
> m(K.1^2);
(2 0)
> m(K.2);
(1/470*(83*J.1^3 + 125*J.1^2 - 1419*J.1 - 1735) 1/940*(-24*J.1^3 - 5*J.1^2 +
    382*J.1 + 295))
> m(K.2^2);
(3 0)
> m(K.3);
(1/10*(-J.1^3 + 23*J.1)                       0)
> m(K.3^2);
(7 0)
> A.1 @@ m;
1
> A.2 @@ m;
(($.1 - 1)*$.1 - $.1 - 1)*K.1 + ($.1 + 1)*$.1 + $.1 + 1
>
> r := 5*K.1 - 8*K.2 + K.3;
> m(r);
(1/235*(-238*J.1^3 - 500*J.1^2 + 4689*J.1 + 6940) 1/235*(48*J.1^3 + 10*J.1^2 -
    764*J.1 - 590))
> MinimalPolynomial($1);
$.1^2 + 1/5*(-4*J.1^3 + 42*J.1)*$.1 + 5*J.1^2 - 180
```

```
> Evaluate($1, r);
0
> K:Maximal;
  K
  |
  |
  $1
  |
  |
  $2
  |
  |
  Q
K  : $.1^2 - 2
$1 : $.1^2 - 3
$2 : x^2 - 7
> Parent($3);
Univariate Polynomial Ring over J
> J;
Number Field with defining polynomial $.1^4 - 18*$.1^2 + 25 over the Rational
Field
```

37.4.4 Invariants

Some information describing an order can be retrieved.

`Characteristic(F)` `Characteristic(O)`

`Degree(O)`
`Degree(F)`

Given an algebraic field F, return the degree $[F : G]$ of F over its ground field G. For an order O it returns the relative degree of O over its ground order.

`AbsoluteDegree(O)`
`AbsoluteDegree(F)`

Given an order O or an algebraic field F, return the absolute degree of O over $\mathbf{Z}$ or F over $\mathbf{Q}$.

`Discriminant(O)`

`Discriminant(F)`

Given an extension F of $\mathbf{Q}$, return the discriminant of F. This discriminant is defined to be the discriminant of the order of the field of fractions or the equation order of a number field.

The discriminant of any order O defined over $\mathbf{Z}$ is by definition the discriminant of its basis, where the discriminant of any sequence of elements ω_i from K is defined to be the determinant of the trace matrix of the sequence.

The discriminant of absolute fields and orders is an integer.

The discriminant in a relative extension is the ideal generated by the discriminants of all sequences of elements ω_i from O, where the discriminant of a sequence is defined to be the determinant of its trace matrix. This can only be computed in when the coefficient ring of O is maximal or when O has a power basis.

The discriminant of relative orders is an ideal of the base ring.

`AbsoluteDiscriminant(O)`

Given an order O, return the absolute value of the discriminant of O regarded as an order over $\mathbf{Z}$.

`AbsoluteDiscriminant(K)`

Given an algebraic field K, return the absolute value of the discriminant of K regarded as an extension of $\mathbf{Q}$.

`ReducedDiscriminant(O)`

`ReducedDiscriminant(F)`

The reduced discriminant of an order O is defined as the maximal elementary divisor (elementary ideal) of the torsion module $O^{\#}/O$ where $O^{\#}$ is the dual module to O with respect to the trace form. The reduced discriminant of an algebraic field is that of the order of a field of fractions and that of the equation order of a number field.

For absolute extensions this is the largest entry of the Smith normal form of the `TraceMatrix`. For relative extensions, in addition to the `TraceMatrix` one has to consider the coefficient ideals.

For orders with a power basis, this is (a generator of) the inverse of the ideal generated by the cofactors X and Y of $Xf + Yf' = 1$ where f is the defining polynomial of the order and f' its first derivative.

`Regulator(O: parameters)`

`Current` BoolElt *Default :* `false`

`Regulator(K)`

Given a number field K or an order O, return the regulator of K or O, as a real number. Note that this will trigger the computation of the maximal order and its unit group if they are not known yet. This only works in an absolute extension.

If `Current` is `true` and a maximal system of independent units is known, then the regulator of that system is returned. In this case no effort is spent to produce a system of fundamental units.

`RegulatorLowerBound(O)`
`RegulatorLowerBound(K)`

Given an order O or number field K, return a lower bound on the regulator of O or K. This only works in an absolute extension.

`Signature(O)`
`Signature(F)`

Given an absolute algebraic field F, or an order O of F, returns two integers, one being the number of real embeddings, the other the number of pairs of complex embeddings of F.

`UnitRank(O)`
`UnitRank(K)`

The unit rank of the order O or the number field K (one less than the number of real embeddings plus number of pairs of complex embeddings).

`Index(O, S)`

The module index of order S in order O, for orders $S \subset O$. O and S must have the same equation order and S must be a suborder of O.

`DefiningPolynomial(F)`
`DefiningPolynomial(O)`

Given an algebraic field F, the polynomial defining F as an extension of its ground field G is returned. For an order O, a integral polynomial is returned that defines O over its coefficient ring.

For non simple extensions, this will return a list of polynomials.

`Zeroes(O, n)`
`Zeros(O, n)`
`Zeroes(F, n)`
`Zeros(F, n)`

Given an absolute algebraic field F or an order O in F, and an integer n, return the zeroes of the defining polynomial of F with a precision of exactly n decimal digits. The function returns a sequence of length the degree of F; all of the real zeroes appear before the complex zeroes.

Example H37E12

The information provided by `Zeros` and `DefiningPolynomial` is illustrated below.

```
> L := NumberField(x^6+108);
> DefiningPolynomial(L);
x^6 + 108
> Zeros(L, 30);
[ 1.8898815748423097471508159108999999999994 +
1.0911236359717214035600726141999999999977*i,
1.8898815748423097471508159108999999999994 -
1.0911236359717214035600726141999999999977*i,  0.E-29 +
2.1822472719434428071201452283999999999955*i,  0.E-29 -
2.1822472719434428071201452283999999999955*i,
-1.8898815748423097471508159108999999999994 +
1.0911236359717214035600726141999999999977*i,
-1.8898815748423097471508159108999999999994 -
1.0911236359717214035600726141999999999977*i ]
> l := NumberField(x^3 - 2);
> DefiningPolynomial(l);
x^3 - 2
> Zeros(l, 30);
[ 1.2599210498948731647672106072999999999994,
-0.6299605249474365823836053036391099999999 +
1.0911236359717214035600726141999999999977*i,
-0.6299605249474365823836053036391099999999 -
1.0911236359717214035600726141999999999977*i ]
```

`Different(O)`

The different of a maximal order $O \subset K$ is defined as the inverse ideal of $\{x \in K \mid \mathrm{Tr}(xO) \subset O\}$.

`Conductor(O)`

The conductor of an order O is the largest ideal of its maximal order that is still contained in O: $\{x \in M \mid xM \subseteq O\}$.

37.4.5 Basis Representation

The basis of an order or algebraic field can be expressed using elements from any compatible ring. Basis related matrices can be formed.

`Basis(O)`

`Basis(O, R)`

`Basis(F)`

`Basis(F, R)`

Return the current basis for the order O or algebraic field F over its ground ring as a sequence of elements of its field of fractions or as a sequence of elements of R.

`IntegralBasis(F)`

`IntegralBasis(F, R)`

An integral basis for the algebraic field F is returned as a sequence of elements of F or R if given. This is the same as the basis for the maximal order. Note that the maximal order will be determined (and stored) if necessary.

Example H37E13

The following illustrates how a basis can look different when expressed in a different ring.

```
> f := x^5 + 5*x^4 - 75*x^3 + 250*x^2 + 65625;
> M := MaximalOrder(f);
> M;
Maximal Order of Equation Order with defining polynomial x^5 + 5*x^4 - 75*x^3 +
    250*x^2 + 65625 over its ground order
> Basis(M);
[
    M.1,
    M.2,
    M.3,
    M.4,
    M.5
]
> Basis(NumberField(M));
[
    1,
    $.1,
    $.1^2,
    $.1^3,
    $.1^4
]
> Basis(M, NumberField(M));
[
    1,
    1/5*$.1,
```

```
    1/25*$.1^2,
    1/125*$.1^3,
    1/625*$.1^4
]
```

`AbsoluteBasis(K)`

Returns an absolute basis for the algebraic field K, i.e. a basis for K as a $\mathbf{Q}$ vector space. The basis will consist of the products of the basis elements of the intermediate fields. The expansion is done depth-first.

`BasisMatrix(O)`

Given an order O in a number field K of degree n, this returns an $n \times n$ matrix whose i-th row contains the (rational) coefficients for the i-th basis element of O with respect to the power basis of K. Thus, if b_i is the i-th basis element of O,

$$b_i = \sum_{j=1}^{n} M_{ij} \alpha^{j-1}$$

where M is the matrix and α is the generator of K.

The matrix is the same as `TransformationMatrix(O, E)`, where E is the equation order for K, except that the entries of the basis matrix are from the subfield of K, instead of the coefficient ring of the order.

`TransformationMatrix(O, P)`

Returns the transformation matrix for the transformation between the orders O and P with common equation order of degree n. The function returns an $n \times n$ matrix T with integral entries as well as a common integer denominator. The rows of the matrix express the n basis elements of O as a linear combination of the basis elements of P. Hence the effect of multiplying T on the left by a row vector v containing the basis coefficients of an element of O is the row vector $v \cdot T$ expressing the same element of the common number field on the basis for P.

`CoefficientIdeals(O)`

The coefficient ideals of the order O of a relative extension. These are the ideals $\{A_i\}$ of the coefficient ring of O such that for every element e of O, $e = \sum_i a_i * b_i$ where $\{b_i\}$ is the basis returned for O and each $a_i \in A_i$.

Example H37E14

We continue our example of a field of degree 4.
The functions `Basis` and `IntegralBasis` both return a sequence of elements, that can be accessed using the operators for enumerated sequences. Note that if, as in our example, O is the maximal order of K, both functions produce the same output:

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> O := MaximalOrder(K);
> I := IntegralBasis(K);
> B := Basis(O);
> I, B;
[
    1,
    1/2*y,
    1/40*(y^2 + 10*y),
    1/800*(y^3 + 180*y + 400)
]
[
    0.1,
    0.2,
    0.3,
    0.4
]
> Basis(O, K);
[
    1,
    1/2*y,
    1/40*(y^2 + 10*y),
    1/800*(y^3 + 180*y + 400)
]
```

The `BasisMatrix` function makes it possible to move between orders, in the following manner. We may regard orders as free **Z**-modules of rank the degree of the number field. The basis matrix then provides the transformation between the order and the equation order. The function `ElementToSequence` can be used to create module elements.

```
> BM := BasisMatrix(O);
> Mod := RSpace(RationalField(), Degree(K));
> z := O ! y;
> e := z^2-3*z;
> em := Mod ! ElementToSequence(e);
> em;
(  0 -26  40   0)
> f := em*BM;
> f;
```

```
( 0 -3  1  0)
```

So, since f is represented with respect to the basis of the equation order, which is the power basis, we indeed get the original element back. Of course it is much more useful to go in the other direction, using the inverse transformation. We check the result in the last line:

```
> E := EquationOrder(K);
> f := y^3+7;
> fm := Mod ! ElementToSequence(f);
> e := fm*BM^-1;
> e;
(-393 -360    0  800)
> &+[e[i]*B[i] : i in [1 .. Degree(K)] ];
-393/1*0.1 - 360/1*0.2 + 800/1*0.4
> K!$1;
y^3 + 7
```

`MultiplicationTable(O)`

Given an order O of some number field K of degree n, return the multiplication table with respect to the basis of O as a sequence of n matrices of size $n \times n$. The i-th matrix will have as its j-th row the basis representation of $b_i b_j$, where b_i is the i-th basis element for O.

`TraceMatrix(O)`

`TraceMatrix(F)`

Return the trace matrix of an order O or algebraic field F, which has the trace $\mathrm{Tr}(\omega_i \omega_j)$ as its i, j-th entry where the ω_i are the basis for O or F.

Example H37E15

We continue our example of a field of degree 4.
The multiplication table of the order O consists of 4 matrices, such that the i-th 4×4 matrix ($1 \leq i \leq 4$) determines the multiplication by the i-th basis element of O as a linear transformation with respect to that basis. Thus the third row of $T[2]$ gives the basis coefficients for the product of $B[2]$ and $B[3]$, and we can use the sequence reduction operator to calculate $B[2] * B[3]$ in an alternative way:

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> O := MaximalOrder(K);
> B := Basis(O);
> B[2];
0.2
> T := MultiplicationTable(O);
> T[2];
[  0   1   0   0]
```

```
[  0  -5  10   0]
[ -5  -7   5  10]
[-25  -7  15   0]
>  &+[ T[2][3][i]*B[i] : i in [1..4] ];
-5/1*0.1 - 7/1*0.2 + 5/1*0.3 + 10/1*0.4
> B[2]*B[3];
-5/1*0.1 - 7/1*0.2 + 5/1*0.3 + 10/1*0.4
```

The trace matrix may be found either by using the built-in function or by the one-line definition given below (for a field of degree 4):

```
> TraceMatrix(O);
[  4   0  21   2]
[  0 210 105 215]
[ 21 105 173 118]
[  2 215 118 226]
> MatrixRing(RationalField(), 4) ! [Trace(B[i]*B[j]): i, j in [1..4] ];
[  4   0  21   2]
[  0 210 105 215]
[ 21 105 173 118]
[  2 215 118 226]
```

37.4.6 Ring Predicates

Orders and algebraic fields can be tested for having several properties that may hold for general rings.

`N eq O`

Two orders are equal if the transformation matrix taking one to the other is integral and has determinant 1 or -1. For the transformation matrix to exist the orders must have the same number field.

`F eq L`

Returns `true` if and only if the fields F and L, are the same.

No two algebraic fields which have been created independently of each other will be considered equal since it is possible that they can be embedded into a larger field in more than one way.

`IsCommutative(R)` `IsUnitary(R)` `IsFinite(R)`

`IsOrdered(R)` `IsField(R)`

`IsNumberField(R)` `IsAlgebraicField(R)`

`IsEuclideanDomain(F)`

This is not a check for euclidean number fields. This function will always return an error.

```
IsSimple(F)
```
```
IsSimple(O)
```

Checks if the field F or the order O is defined as a simple extension over the base ring.

```
IsPID(F)
```
```
IsUFD(F)
```
```
IsPrincipalIdealRing(F)
```

Always `true` for fields.

```
IsPID(O)
```
```
IsUFD(O)
```
```
IsPrincipalIdealRing(O)
```

Always `false` for orders. Even if the class number is 1, orders are not considered to be PIDs.

```
IsDomain(R)
```
```
F ne L
```
```
O ne N
```
```
O subset P
```
```
K subset L
```
```
HasComplexConjugate(K)
```

This function returns `true` if there is an automorphism in the field K that acts like complex conjugation.

```
ComplexConjugate(x)
```

For an element x of a field K where `HasComplexConjugate` returns `true` (in particular this includes totally real fields, cyclotomic and quadratic fields and CM-extensions), the conjugate of x is returned.

37.4.7 Order Predicates

Since orders are rings with additional properties, special predicates are applicable.

```
IsEquationOrder(O)
```

This returns `true` if the basis of the order O is an integral power basis, `false` otherwise.

```
IsMaximal(O)
```

This returns `true` if the order O in the field F is the maximal order of F, `false` otherwise. The user is warned that this may trigger the computation of the maximal order.

```
IsAbsoluteOrder(O)
```

Returns `true` iff the order O is a constructed as an absolute extension of $\mathbf{Z}$.

`IsWildlyRamified(O)`

Returns `true` iff the order O is wildly ramified, i.e. if there is a prime ideal P of O such that the characteristic of its residue class field divides its ramification index.

`IsTamelyRamified(O)`

Returns `true` iff the order O is not wildly ramified, i.e. if for all prime ideals P of O the characteristic of its residue class field does not divide the ramification index.

`IsUnramified(O)`

Returns `true` iff the order O is unramified at the *finite* places.

37.4.8 Field Predicates

Here all the predicates that are specific to algebraic fields are listed.

`IsIsomorphic(F, L)`

Given two algebraic fields F and L, this returns `true` as well as an isomorphism $F \to L$, if F and L are isomorphic, and it returns `false` otherwise.

`IsSubfield(F, L)`

Given two algebraic fields F and L, this returns `true` as well as an embedding $F \hookrightarrow L$, if F is a subfield of L, and it returns `false` otherwise.

`IsNormal(F)`

Returns `true` if and only if the algebraic field F is a normal extension. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois group computation.

`IsAbelian(F)`

Returns `true` if and only if the algebraic field F is a normal extension with abelian Galois group. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois Group computation.

`IsCyclic(F)`

Returns `true` if and only if the algebraic field F is a normal extension with cyclic Galois group. At present this may only be applied if F is an absolute extension or simple relative extension. In the relative case the result is obtained via Galois and automorphism group.

`IsAbsoluteField(K)`

Returns `true` iff the algebraic field K is a constructed as an absolute extension of $\mathbf{Q}$.

`IsWildlyRamified(K)`

Returns `true` iff the algebraic field K is wildly ramified, i.e. if there is a prime ideal P of K (its maximal order) such that the characteristic of its residue class field divides the ramification index.

`IsTamelyRamified(K)`

Returns `true` iff the algebraic field K is not wildly ramified, i.e. if for all prime ideals P of the maximal order of K, the characteristic of its residue class field does not divide the ramification index.

`IsUnramified(K)`

Returns `true` iff the algebraic field K is unramified at the *finite* places.

`IsQuadratic(K)`

If the number field K is quadratic, return `true` and an isomorphic quadratic field.

`IsTotallyReal(K)`

Tests if the number field F is totally real, ie. if all infinite places are real. For absolute fields this is equivalent to the defining polynomial having only real roots.

37.4.9 Setting Properties of Orders

Several properties of orders can be set to be known and/or to have a given value.

`SetOrderMaximal(O, b)`

Set the order O to be maximal if b is `true` or known to be non maximal if b is `false`.

`SetOrderTorsionUnit(O, e, r)`

Set the torsion unit of the order O to be the element e with order r.

`SetOrderUnitsAreFundamental(O)`

Mark the currently known units of the order O to be fundamental.

37.5 Element Operations

37.5.1 Parent and Category

`Parent(a)` `Parent(w)` `Category(a)` `Category(w)`

37.5.2 Arithmetic

The table below lists the generic arithmetic functions on algebraic field and order elements. Note that automatic coercion ensures that the binary operations `+`, `-`, `*`, and `/` may be applied to an element of an algebraic field and an element of one of its orders; the result will be an algebraic field element. Since division of order elements does not generally result in an order element, the operation `/` applied to two elements of an order returns an element in the field of fractions of the order; similarly if the exponent k in `a^k` is negative.

For finding the value of an element mod an ideal or the inverse of an element mod an ideal see Section 37.9.6.

`+ a` `- a`

`a + b` `a - b` `a * b` `a / b` `a ^ k`

`w div v`

The quotient of the order element w by the order element v; v must divide w exactly, (v and w must be elements of the same order.)

`Modexp(a, n, m)`

Given a non-negative integer n and an integer m greater than 1, this function returns the modular power $a^n \ mod \ m$ of the order element a.

`Sqrt(a)`

`SquareRoot(a)`

Returns the square root of the element a if it exists in the order or field containing a.

`Root(a, n)`

Returns the n-th root of the element a if it exists in the order or field containing a.

`IsPower(a, k)`

`IsSquare(a)`

Return `true` if the element a is a kth power, (respectively square) and the root in the order or field containing a if so.

`Denominator(a)`

Returns the denominator of the element a, that is the least common multiple of the denominators of the coefficients of a.

`Numerator(a)`

Returns the numerator of the element a, that is the element multiplied by its denominator.

`Qround(E, M)`

`ContFrac` BoolElt *Default :* `true`

Finds an approximation of the field element E where the denominator is bounded by the integer M. If `ContFrac` is `true`, the approximation is computed by applying the continued fraction algorithm to the coefficients of E viewed over Q.

37.5.3 Equality and Membership

Elements may also be tested for whether they lie in an ideal of an order. See Section 37.9.5.

`a eq b` `a ne b`

`a in F`

37.5.4 Predicates on Elements

In addition to the generic predicates `IsMinusOne`, `IsZero` and `IsOne`, the predicates `IsIntegral` and `IsPrimitive` are defined on elements of algebraic fields and orders.

`IsIntegral(a)`

Returns `true` if the element a of an algebraic field F or of an order in F is contained in the ring of integers of F, `false` otherwise. This is vacuously true for order elements. We use the minimal polynomial to determine the answer, which means that the calculation of the maximal order is *not* triggered if it is not known yet. When a is a field element a denominator d such that $d * a$ is integral is also returned on request.

`IsPrimitive(a)`

Returns `true` if the element a of the algebraic field F or one of its orders O generates F.

`IsTorsionUnit(w)`

Returns `true` if and only if the order element w is a unit of finite order.

`IsPower(w, n)`

Given an element w in an order O and an integer $n > 1$, this function returns `true` if and only if there exists an element $v \in O$ such that $w = v^n$; if `true`, such an element v is returned as well.

`IsTotallyPositive(a)`

`IsTotallyPositive(a)`

Returnes `true` iff all real embeddings of the element a are positive. For elements in absolute fields this is equivalent to all real conjugates being positive.

`IsZero(a)` `IsOne(a)`

`IsMinusOne(a)` `IsUnit(a)`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsZeroDivisor(a)` `IsRegular(a)`

`IsIrreducible(a)` `IsPrime(a)`

37.5.5 Finding Special Elements

Generators of fields can be retrieved.

`K . 1`

Return the image α of x in $G[x]/f$ where f is the first defining polynomial of K and G is the base field of K.

In case of simple extensions this will be a primitive element.

`PrimitiveElement(K)`

`PrimitiveElement(F)`

Returns a primitive element for the simple algebraic field, that is an element whose minimal polynomial has the same degree as the field. For a number field K this is $K.1$ but for a field of fractions this is $F!K.1$ where K is the number field of F.

For non-simple fields, a random element is returned.

`Generators(K)`

The list of generators of K over its coefficient field, that is a sequence containing a root of each defining polynomial is returned.

`Generators(K, k)`

A list of generators of K over k is returned. That is a sequence containing a root of each defining polynomial for K and its subfield down to the level of k is returned.

`PrimitiveElement(O)`

Returns a primitive element for the field of fractions of the order O, that is an element whose minimal polynomial has the same degree as the field.

37.5.6 Real and Complex Valued Functions

The functions here return (sequences of) real or complex numbers. The precision of these numbers is governed by the appropriate order's or field's internal precision. See Section 37.3 for more information.

`AbsoluteValues(a)`

Return a sequence of length $r_1 + r_2$ of the real absolute values of the conjugates of the element a. The first r_1 values are the absolute values of the real embeddings of the element, the next r_2 are the lengths of the complex embeddings with their weight factors. That is, if the real conjugates of a are w_i, for $1 \leq i \leq r_1$, and the complex conjugates of a are $x_i \pm iy_i$ (for $1 \leq i \leq r_2$), then `AbsoluteValues` returns $[|w_1|, \ldots, |w_{r_1}|, \sqrt{\frac{x_{r_1+1}^2+y_{r_1+1}^2}{2}}, \ldots \sqrt{\frac{x_{r_1+r_2}^2+y_{r_1+r_2}^2}{2}}]$.

`AbsoluteLogarithmicHeight(a)`

Let P be the minimal polynomial of the element a over $\mathbf{Z}$, with leading coefficient a_0 and roots $\alpha_1, \ldots, \alpha_n$. Then the absolute logarithmic height is defined to be

$$h(\alpha) = \frac{1}{n} \log(a_0 \prod_{j=1}^{n} \max(1, |\alpha_j|)).$$

`Conjugates(a)`

The real and complex conjugates of the given algebraic number a, as a sequence of n complex numbers. The r_1 real conjugates appear first, and are followed by r_2 pairs of complex conjugates. The field should be an absolute extension. The ordering of the conjugates is consistent for elements of the same field (or even for elements of different fields that have the same defining polynomial).

`Conjugate(a, k)`

Equivalent to `Conjugates(a)[k]`.

`Conjugate(a, l)`

Let $l := [l_1, \ldots, l_n]$ be a sequence of positive integers and assume that the field K, the parent of a is given as a tower with n steps, $\mathbf{Q} \subseteq K_1 \subseteq \ldots \subseteq K_n = K$. This function computes the image of a in $\mathbf{C}$ or $\mathbf{R}$ under the embedding determined by l, that is under embedding obtained by extending the l_1 embedding of K_1 to K_2, then extending the l_2nd of those embeddings to K_3, ...

`Length(a)`

Return the T_2-norm of the element a, which is a real number. This equals the sum of the (complex) norms of the conjugates of a.

`Logs(a)`

Return the sequence of length $r_1 + r_2$ of logarithms of the absolute values of the conjugates of a number field or order element $a \neq 0$.

`CoefficientHeight(E)`

`CoefficientHeight(E)`

Computes the coefficient height of the element E, that is for an element of an absolute field it returns the maximum of the denominator and the largest coefficient wrt. to the basis of the parent. For elements in relative extensions, it returns the maximal coefficient height of all the coefficients wrt. the basis of the parent.

This function indicates in some way the difficulty of operations involving this element.

`CoefficientLength(E)`

`CoefficientLength(E)`

Computes the coefficient length of the element E, that is for an element of an absolute field it returns the sum of the denominator and the absolute values of all coefficients wrt. to the basis of the parent. For elements in relative extensions, it returns the sum of the coefficient length of all the coefficients wrt. the basis of the parent.

This function gives an indication on the amount of memory occupied by this element.

Example H37E16

Using the functions `Conjugates` and `Basis`, it is easy to write an alternative discriminant function.

```
> disc := func< O | Determinant( MatrixAlgebra(ComplexField(20), Degree(O) )
>        ! [ Conjugates(Basis(O)[i])[j] : i, j in [1 .. Degree(O)] ] )^2 >;
> R<x> := PolynomialRing(Integers());
> O := MaximalOrder(NumberField(x^4 - 420*x^2 + 40000));
> disc(O);
42025
> Discriminant(O);
42025
```

Thus, the new discriminant function returns a complex approximation to the built-in function `Discriminant`, giving the above result for the maximal order O of the previous example.
Here is an alternative way of getting the T_2 norm returned by `Length`, using the complex `Norm` function, together with the `Conjugates` function.

```
> norm := func< a | &+[ Norm(Conjugates(a)[i]) : \
>                       i in [1 .. Degree(Parent(a))] ] >;
```

37.5.7 Norm, Trace, and Minimal Polynomial

The norm, trace and minimal polynomial of order and algebraic field elements can be calculated both with respect to the coefficient ring and to **Z** or **Q**.

`Norm(a)`

`Norm(a, R)`

The relative norm $\mathrm{N}_{\mathrm{L/F}}(\mathrm{a})$ over F of the element a of L where F is the field or order over which L is defined as an extension. If R is given the norm is calculated over R. In this case, R must occur as a coefficient ring somewhere in the tower under L.

`AbsoluteNorm(a)`

`NormAbs(a)`

The absolute norm $\mathrm{N}_{\mathrm{L}/\mathbf{Q}}(\mathrm{a})$ over **Q** of the element a of L (or one of its orders).

`Trace(a)`

`Trace(a, R)`

The relative trace $\mathrm{Tr}_{L/F}(a)$ over F of the element a of L where F is the field or order over which L is defined as an extension. If R is given the trace is computed over R. In this case, R must occur as a coefficient ring somewhere in the tower under L.

`AbsoluteTrace(a)`

`TraceAbs(a)`

The absolute trace $\mathrm{Tr}_{L/\mathbf{Q}}(a)$ over **Q** of the element a of L (or one of its orders).

`CharacteristicPolynomial(a)`

`CharacteristicPolynomial(a, R)`

Given an element a from an algebraic field or order L, returns the characteristic polynomial of the element over R if given or the subfield or suborder F otherwise where F is the field or order over which L is defined as an extension.

`AbsoluteCharacteristicPolynomial(a)`

Given an element a from an algebraic field or one of its orders, this function returns the characteristic polynomial of the element. For field elements the polynomial will have coefficients in the rational field, for order elements the coefficients will be in the ring of integers.

`MinimalPolynomial(a)`

`MinimalPolynomial(a, R)`

Given an element a from an algebraic field or order L, returns the minimal polynomial of the element over R if given otherwise the subfield or suborder F where F is the field or order over which L is defined as an extension.

`AbsoluteMinimalPolynomial(a)`

Given an element a from an algebraic field or one of its orders, this function returns the minimal polynomial of the element. For field elements the polynomial will have coefficients in the rational field, for order elements the coefficients will be in the ring of integers.

`RepresentationMatrix(a)`

`RepresentationMatrix(a, R)`

Return the representation matrix of a, that is, the matrix which represents the linear map given by multiplication by a. If a is an order element, this matrix is with respect to the basis for the order; if a is an algebraic field element, the basis for the field is used. The ith row of the representation matrix gives the coefficients of aw_i with respect to the basis $w_1, \ldots, w_n$.

If R is given the matrix is over R and with respect to the basis of the order or field over R.

`AbsoluteRepresentationMatrix(a)`

Return the representation matrix of a relative to the $\mathbf{Q}$-basis of the field constructed using products of the basis elements, where a is an element of the relative number field L.

Let $L_i := \sum L_{i-1}\omega_{i,j}$, $L := L_n$ and $L_0 := \mathbf{Q}$. Then the representation matrix is computed with respect to the $\mathbf{Q}$-basis $(\prod_j \omega_{i_j,j})_{i\in I}$ consisting of products of basis elements of the different levels.

Example H37E17

We create the norm, trace, minimal polynomial and representation matrix of the element $\alpha/2$ in the quartic field $\mathbf{Q}(\alpha)$.

```
> R<x> := PolynomialRing(Integers());
> K<y> := NumberField(x^4-420*x^2+40000);
> z := y/2;
> Norm(z), Trace(z);
2500 0
> MinimalPolynomial(z);
ext<Q|>.1^4 - 105*ext<Q|>.1^2 + 2500
> RepresentationMatrix(z);
[     0    1/2      0      0]
[     0      0    1/2      0]
[     0      0      0    1/2]
[-20000      0    210      0]
```

The awkwardness of the printing of the minimal polynomial above can be overcome by providing a parent for the polynomial, keeping in mind that it is a univariate polynomial over the rationals:

```
> P<t> := PolynomialRing(RationalField());
> MinimalPolynomial(z);
```

```
t^4 - 105*t^2 + 2500
```

37.5.8 Other Functions

Elements can be represented by sequences and matrices. Valuation can also be calculated.

`ElementToSequence(a)`

`Eltseq(a)`

For an element a of an algebraic field F, a sequence of coefficients of length degree of F with respect to the basis is returned. For an element of an order O, the sequence of coefficients of the element with respect to the basis of O are returned.

Note however that the universe of the sequence if always a field since in general in relative extensions integral coefficients cannot be achieved.

`Eltseq(E, k)`

For an algebraic number $E \in K$ and a ring k which occurs somewhere in the defining tower for K, return the list of coefficients of E over k, that is, apply `Eltseq` to E and to its coefficients until the list is over k.

`Flat(e)`

The coefficients of the algebraic field element e wrt. to the canonical Q basis for its field. This is performed by iterating `Eltseq` until the coefficients are rational numbers. For number field elements the coefficients obtained match the coefficients wrt. to `AbsoluteBasis`.

`a[i]`

The coefficient of the ith basis element in the algebraic field or order element a.

`ProductRepresentation(a)`

Return sequences P and E such that the product of elements in P to the corresponding exponents in E is the algebraic number a.

`ProductRepresentation(P, E)`

`PowerProduct(P, E)`

Return the element a of the universe of the sequence P such that a is the product of elements of P to the corresponding exponents in the sequence E.

`Valuation(w, I)`

Given a prime ideal I and an element w of an order or algebraic field, this function returns the valuation $v_I(w)$ of w with respect to I; this valuation will be a non-negative integer. Ideals are discussed in Section 37.9.

`Decomposition(a)`

`Decomposition(a)`

The factorization of the order or algebraic field element a into prime ideals.

`Divisors(a)`

For an element a in a maximal order return a sequence containing (up to units) all the elements which divide a. The elements of the sequence will be generators for all principal ideals returned by `Divisors(Parent(a)*a)`.

`Index(a)`

The index of the module $\mathbf{Z}[a]$ in O where a lies in O, an order over $\mathbf{Z}$. If a is not a primitive element the index is infinite.

`Different(a)`

The different of the element a of an order of a number field.

37.6 Ideal Class Groups

This section describes the functions related to finding class groups and class numbers for (the maximal order O of) an absolute number field.

The method usually employed is the *relation* method ([Heß96, Coh93]), basically consisting of the following steps. In the first step a list of prime ideals of norm below a given bound is generated, the *factor basis*. In the second step a search is conducted to find in each of the prime ideals a few elements for which the principal ideals they generate factor completely over the factor basis. Using these *relations*, a generating set for the ideal class group is derived (via matrix echelonization), and in the final step it is verified that the correct orders for the generators have been found.

To determine the class group or class number correctly one has to make sure that all ideals having norm smaller than the Minkowski bound or smaller than the Bach bound, if one assumes the generalized Riemann hypothesis, are taken into consideration, and that the final stage, which may be time consuming, is properly executed. Optional arguments allow the user to override these, but correctness of such results can not be guaranteed.

It should be stressed that by default a guaranteed result is computed using the Minkowski bound. Thus even for innocent looking fields it may take considerable time. In comparison, pari (as of version 2.0) will by default use a much smaller bound giving results that are not guaranteed even under GRH. On the other hand, it will be much faster. It is possible to use the same bounds in Magma. In this case the running times will be similar.

When the discriminant of an order is very large (above 10^{30}) a sieving method developed by [Bia] is used. Using this method the relations are derived using an analogue of the number field sieve. Lattice sieving and the special-Q are also used. The verbose flag `"ClassGroupSieve"` can be set to view information about the computation.

Once a class group computation has been completed, the results are stored with the order.

All functions mentioned in this section support the verbose flag `ClassGroup` up to a maximum value of 5.

Ideals in MAGMA are discussed in Section 37.9.

`DegreeOnePrimeIdeals(O, B)`

Given an order O as well as a positive integer bound B, return a sequence consisting of all prime ideals in O whose norm is a rational prime not exceeding the bound B.

`ClassGroup(O: parameters)`

`ClassGroup(K: parameters)`

`Bound`	RNGINTELT	*Default :* MinkowskiBound
`Proof`	MONSTGELT	*Default :* "*Full*"
`Enum`	BOOLELT	*Default :* `true`
`Al`	MONSTGELT	*Default :* "*Automatic*"
Verbose	`ClassGroup`	*Maximum :* 5
Verbose	`ClassGroupSieve`	*Maximum :* 5

The group of ideal classes for the ring of integers O of the number field K is returned as an abelian group, together with a map from this abstract group to O. The map admits inverses and can therefore be used to compute "discrete logarithms" for the class group.

With the default values for the optional parameters the Minkowski bound is used and the last step of the algorithm verifies correctness (see the explanation above), hence a fully proven result is returned.

If `Bound` is set to some positive integer M, M is used instead of the Minkowski bound. The validity of the result still depends on the `"Proof"` parameter.

If `Proof := "GRH"`, everything remains as in the default case except that a bound based on the GRH is used to replace the Minkowski bound. This bound may be enlarged setting the `Bound` parameter accordingly. The result will hence be correct under the GRH.

If `Proof := "Bound"`, the computation stops if an independent set of relations between the prime ideals below the chosen bound is found. The relations may not be maximal.

If `Proof := "Subgroup"`, a maximal subset of the relations is constructed. In terms of the result, this means that the group returned will be a subgroup of the class group (i.e. the list of prime ideals considered may be to small).

If `Proof := "Full"` (the default) a guaranteed result is computed. This is equivalent to `Bound := MinkowskiBound(K)` and `Proof := "Subgroup"`.

If only `Bound` is given, the `Proof` defaults to `"Subgroup"`.

Finally, giving `Proof := "Current"` is the same as repeating the last call to `ClassGroup()`, but without the need to explicitly restate the value of `Proof` or `Bound`. If there was no prior call to `ClassGroup`, a fully proven computation will be carried out.

If `Enum := false`, then instead of enumerating short elements to get relations, MAGMA will use random linear combinations of a reduced basis instead. For "small" fields this will typically slow down the computations, but for large fields it is sometimes not possible to find any point using enumeration so that this is necessary for "large" fields. Unfortunately, there is no known criterion to decide beforehand if a field is "large" or "small".

If `Al` is set to `"Sieve"` (regardless of the size of the discriminant) or the discriminant of O is greater than 10^{30} then the sieving method developed by [Bia] is used. If `Al` is set to `"NoSieve"` then this sieving method will not be used regardless of the size of the discriminant.

`RingClassGroup(O)`

`PicardGroup(O)`

For a (possibly non-maximal) order O, compute the ring class group (Picard group) of O, ie. the group of invertible ideals in O modulo pricipal ideals. The algorithm and its implementation are due to Klüners and Pauli, [PK05].

`ConditionalClassGroup(O)`

`ConditionalClassGroup(K)`

The class group of the order O or the number field K assuming the generalized Riemann hypothesis.

`ClassGroupPrimeRepresentatives(O, I)`

For the maximal order O of some absolute number field k and an ideal I of O, compute a set of prime ideals in O that are coprime to I and represent all ideal classes. The map, mapping elements of the class group to the primes representing the ideal class is returned.

`ClassNumber(O: parameters)`

`ClassNumber(K: parameters)`

`Bound`	RNGINTELT	*Default :* MinkowskiBound
`Proof`	MONSTGELT	*Default :* "*Full*"
`Al`	MONSTGELT	*Default :* "*Automatic*"
Verbose	`ClassGroup`	*Maximum :* 5
Verbose	`ClassGroupSieve`	*Maximum :* 5

Return the class number of the ring of integers O of a number field K. The options for the parameters are the same as for `ClassGroup`.

`BachBound(K)`
`BachBound(O)`

An integral upper bound for norms of generators of the ideal class group for the number field K or the maximal order O assuming the generalized Riemann hypothesis.

`MinkowskiBound(K)`
`MinkowskiBound(O)`

An unconditional integral upper bound for norms of the generators of the ideal class group for the number field K or the maximal order O.

`FactorBasis(K, B)`
`FactorBasis(O, B)`

Given the maximal order O, or a number field K with maximal order O, this function returns a sequence of prime ideals of norm less than a given bound B.

`FactorBasis(O)`

Given the maximal order O where the class group has previously been computed, this function returns a sequence of prime ideals that have been used as factor basis for the class group computation. In addition the used upper bound for the factor basis is returned. This bound can be different from the bound passed in using the `Bound := bound` parameter.

`RelationMatrix(K, B)`
`RelationMatrix(O, B)`

Given a maximal order O, or a number field K with maximal order O, generate relations for each prime ideal in the factor basis for O with bound B on the norms of the ideals. The relations are given by rows in a matrix. If at some stage the relations generate the trivial group, no more relations are generated.

`RelationMatrix(O)`

Given a maximal order O where the class group has been computed previously, the resulting relation matrix is returned.

`Relations(O)`

Given a maximal order O where the class group has been computed previously, the vector containing the order elements used to compute the class group is returned.

`ClassGroupCyclicFactorGenerators(O)`

Let a_i be the generators for the cyclic factors of the class group of O. This function returns generators for $a_i^{c_i}$ where c_i is the order of a_i in the class group.

Example H37E18

We give an example of a class group calculation, illustrating some of the functions.

```
> R<x> := PolynomialRing(Integers());
> O := MaximalOrder(x^2-10);
> C, m := ClassGroup(O);
> C;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*C.1 = 0
> m(C.1);
Prime Ideal of O
Two element generators:
    [2, 0]
    [0, 1]
> p := Decomposition(O, 31)[1][1];
> p;
Prime Ideal of O
Two element generators:
    [31, 0]
    [45, 1]
> p @@ m;
0
> IsPrincipal(p);
true
> p := Decomposition(O, 37)[1][1];
> p @@ m;
C.1
> IsPrincipal(p);
false
> MinkowskiBound(O);
3
> F, B := FactorBasis(O);
> B;
33
```

Even though the `MinkowskiBound` is only 3, we take 33 as the bound. The reason for this behaviour is that the factor base has to have at least some elements (about 20) if it is non-empty. Otherwise the search for relations is hopeless.

```
> r := Relations(O);
> M := RelationMatrix(O);
> [ Valuation(r[1][1], x) : x in F];
[ 0, 0, 0, 0, 0, 1, 1, 0 ]
> M[1];
```

```
(0 0 0 0 0 1 1 0)
```

As one can see, the `RelationMatrix` basically stores the valuation of the elements of `Relations` at the prime ideal contained in `FactorBasis`.

```
> f,g := IsPrincipal(m(C.1)^2);
> f;
true
> g;
[2, 0]
> ClassGroupCyclicFactorGenerators(0);
[
    [2, 0]
]
```

Now we will consider some larger fields to demonstrate the effect of the `"Bound"` parameter:

```
> K := NumberField(x^5-14*x^4+14*x^3-14*x^2+14*x-14);
> MinkowskiBound(K);
21106
> BachBound(K);
7783
> time ClassGroup(K);
Abelian Group isomorphic to Z/10
Defined on 1 generator
Relations:
    10*$.1 = 0
Mapping from: Abelian Group isomorphic to Z/10
Defined on 1 generator
Relations:
    10*$.1 = 0 to Set of ideals of Maximal Equation Order with defining
polynomial x^5 - 14*x^4 + 14*x^3 - 14*x^2 + 14*x - 14 over Z
Time: 14.600
```

Note, that is an unconditional result. As one can see, the `BachBound` (proven under GRH) is much smaller than `MinkowskiBound`. The difference shows up in the running time:

```
> K := NumberField(x^5-14*x^4+14*x^3-14*x^2+14*x-14);
> time _, _ := ClassGroup(K: Bound := BachBound(K));
Time: 7.080
```

In comparison, pari (2.0.20) uses an even smaller bound as default, namely `BachBound(K)/40` for fields of degree > 2 and `BachBound(K)/20` for quadratics.

```
> K := NumberField(x^5-14*x^4+14*x^3-14*x^2+14*x-14);
> time _, _ := ClassGroup(K: Bound := Floor(BachBound(K)/40));
Time: 1.300
```

In comparison, `bnfinit` in pari (version 2.0.20) takes about 1.4 seconds for this example.
Note, that in general one cannot use arbitrarily small bounds. If they are too small, the computation time will increase again as the system will not be able to find any relations.

Example H37E19

We give some examples of class group computations using the sieving algorithm.

```
> R<x> := PolynomialRing(IntegerRing());
> f := x^3 + 3592347*x^2 - 6*x + 3989864;
> K<y> := NumberField(f);
> time G, m := ClassGroup(K : Al := "Sieve", Proof := "GRH");
Time: 3.200
> G, m;
Abelian Group isomorphic to Z/6
Defined on 1 generator
Relations:
    6*G.1 = 0
Mapping from: GrpAb: G to Set of ideals of Maximal Order of Equation Order with
defining polynomial x^3 + 3592347*x^2 - 6*x + 3989864 over Z
> m(G.1);
Ideal
Two element generators:
    [4373, 0, 0]
    [2118, 1, 0]
> $1 @@ m;
G.1
> IsPrincipal($2^6);
true
> f:= x^5 + 3330*x^3 - 50*x + 107392;
> K<y> := NumberField(f);
> time G, m := ClassGroup(K : Al := "Sieve", Proof := "GRH");
Time: 9.930
> G, m;
Abelian Group of order 1
Mapping from: GrpAb: G to Set of ideals of Maximal Order of Equation Order with
defining polynomial x^5 + 3330*x^3 - 50*x + 107392 over Z
```

It is also possible to drive the class group computation "by hand", that is one can call the individual parts one by one to for example re-create a class group computation:

`FactorBasisCreate(O,B)`

Creates a class group process by computing a factor basis containing all ideals of norm $\leq B$ in the order O and returning this factor basis.

`EulerProduct(O, B)`

Computes an approximation to the Euler product for the order O using only prime ideals over prime numbers of norm $\leq B$.

`AddRelation(E)`

Adds a relation (order element) E to the class group process of the parent of E. This function returns `true` exactly when the element factors over the factor basis and if the new relation matrix is of full rank.

`EvaluateClassGroup(O)`

Finalizes a class group process for the order O, that is, it computes (if possible) the class group structure based on the current relation matrix and a basis of the unit group generated by the nullspace of the relation matrix.

This function returns `true` when the class group is determined by the current data. In order to use this function, one has to create a factor basis and then add enough relations.

`CompleteClassGroup(O)`

This function completes an already started class group process for the order O by using the internal functions to look for relations until the class group can be determined.

`FactorBasisVerify(O, L, U)`

This function verifies the "completeness" of the current factor basis for the order O with respect to the prime ideals of norm between L and U. That is, for all prime ideals which norm is between L and U, the function tries to find a relation between the new prime ideal and the prime ideals already in the factor basis. If successful, this means that the new prime ideal does not contribute anything to the class group that is not already known and can therefore safely be ignored.

This function does not return if unsuccessful.

`ClassGroupSetUseMemory(O, f)`

For an order O where the class group is already computed, decide if results of the discrete logarithm computation for the class group are stored. For example if the order O is going to be used extensively as a coefficient ring for class field computations, then every time discrete logatrithms of ray class groups are computed, a discrete logarithm computation in the class group is triggered. In particular when investigating the cohomology of various extensions over O, this involves testing the same ideals over and over again. Setting the flag f to `true` will help to keep computation times down - at the expense of additional use of memory. This functionality is disabled by default.

`ClassGroupGetUseMemory(O)`

For an order O where the class group has been computed check if discrete logarithm values should be stored or not.

37.6.1 Setting the Class Group Bounds Globally

It is possible to preset the bounds to be used in all class group computations. Two bounds can be specified: the first controls the size of the factor base used in the first stage of the class group computation, and the second is the bound used in the checking stage (this controls the level of rigour of the computation).

The bounds may be specified using either of the intrinsics below; `SetClassGroupBounds` is a simple special case of `SetClassGroupBoundMaps`, provided for the user's convenience.

`SetClassGroupBounds(n)`

`SetClassGroupBounds(string)`

This determines the bounds that will be used in all subsequent calls to `ClassGroup`. The argument can be an integer n (then the bounds are both set to this constant). Alternatively the argument can be a string: either "GRH" (then the bounds will guarantee correctness assuming GRH) or "PARI" (this is intended to give roughly the same level of rigour as PARI).

`SetClassGroupBoundMaps(f1, f2)`

This determines the bounds that will be used in all subsequent calls to `ClassGroup`. The arguments should be maps from `PowerStructure(RngOrd)` (which is the parent object of all orders in number fields) to integers.

The bounds used when `ClassGroup` is called for a number field will be the bounds for the maximal order of that field.

Example H37E20

We select some bounds which will then be used in all calls to `ClassGroup`. (The class group computations will be rigorous, but will use a relatively small factor base for the first part of the computation).

```
> map1 := map< PowerStructure(RngOrd) -> Integers() |
>                              order :-> BachBound(order) div 10 >;
> map2 := map< PowerStructure(RngOrd) -> Integers() |
>                              order :-> MinkowskiBound(order) >;
> SetClassGroupBoundMaps( map1, map2);
```

37.7 Unit Groups

The relation method, outlined in the previous section, can also be used for unit group calculations. Therefore, unit group calculations including those triggered as a side effect may cause the creation of factor bases and relations. Other methods such as Dirichlet's method are also implemented which may be faster in certain circumstances. Descriptions of the algorithms can be found in [PZ89], (pp. 343–344), and in [Poh93]. These methods will only work for absolute extensions.

In general, the unit group related functions support a verbose flag `UnitGroup` up to a maximum of 6.

`UnitGroup(O)`

`MultiplicativeGroup(O)`

`UnitGroup(K)`

`MultiplicativeGroup(K)`

`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`UnitGroup`	*Maximum :* 6

Given an order O in a number field, this function returns an (abstract) abelian group U, as well as a bijection m between U and the units of the order. The unit group consists of the torsion subgroup, generated by the image `m(U.1)` and a free part, generated in O by the images `m(U.i)` for $2 \leq i \leq r_1 + r_2$.

If the argument to this function is a number field K, the unit group of its maximal order is returned. Note that the maximal order may have to be determined first.

The parameter `Al` can be used to specify an algorithm. It should be one of `"Automatic"`, (default, a choice will be made for the user) `"ClassGroup"`, `"Dirichlet"`, `"Mixed"` (the best known Dirichlet method), `"Relation"` or `"Short"` (which is a variation of `"Mixed"`). In the case of real quadratic fields, a continued fraction algorithm is available, `"ContFrac"`.

`UnitGroupAsSubgroup(O)`

For a (possibly non-maximal) order O in some absolute field K, return the unit group of O as a subgroup of the unit group of the maximal order of O. The algorithm and its implementatio is due to Klüners and Pauli, [PK05].

`TorsionUnitGroup(O)`

`TorsionUnitGroup(K)`

The torsion subgroup of the unit group of the order O, or, in case of a number field K, of its maximal order O. The torsion subgroup is returned as an abelian group T, together with a map m from the group to the order O. The torsion subgroup will be cyclic, and is generated by `m(T.1)`.

`IndependentUnits(O)`

`IndependentUnits(K)`

`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`UnitGroup`	*Maximum :* 6

Given an order O, this function returns a sequence of independent units; they generate a subgroup of finite index in the full unit group. Given a number field K, the function is applied to the maximal order of K. The function returns an abelian group generated by the independent units as well as a homomorphism from the group to the order.

The parameter `Al` can be used to specify an algorithm. It should be one of `"Automatic"`, `"ClassGroup"`, `"Dirichlet"`, `"Mixed"` (the best known Dirichlet-like method), `"Relation"`. or `"Short"` (which is a variation of `"Mixed"`). In the case of real quadratic fields, a continued fraction algorithm is available, `"ContFrac"`.

`pFundamentalUnits(O, p)`

`pFundamentalUnits(K, p)`

`Al`	MonStgElt	*Default :* "*Automatic*"
Verbose	`UnitGroup`	*Maximum :* 6

Given an order O in a number field, this function returns an (abstract) abelian group U, as well as a map m from U to the order. U will be a subgroup (of finite index) of the unit group G such that p does not divide the index $(G : U)$ where p is the prime number given. If a field K is given rather than an order, the above is computed for the maximal order of K.

The parameter `Al` has the same options as for `UnitGroup`.

`MergeUnits(K, a)`

`MergeUnits(O, a)`

Verbose	`UnitGroup`	*Maximum :* 6

For an order O or a number field with maximal order O and a unit $a \in O$, add the unit to the already known subgroup of U_O that is stored in O. Returns `true` if and only if the rank of the currently known unit group of O or K increases when a is merged with it.

`UnitRank(O)`

`UnitRank(K)`

Return the unit rank of the ring of integers O of a number field K.

Example H37E21

In our field defined by $x^4 - 420 * x^2 + 40000$, we obtain the class and unit groups as follows.

```
> R<x> := PolynomialRing(Integers());
> f := x^4 - 420*x^2 + 40000;
> K<y> := NumberField(f);
> C := ClassGroup(K);
> C;
Abelian Group of order 1
> U := UnitGroup(K);
> U;
Abelian Group isomorphic to Z/2 + Z + Z + Z
Defined on 4 generators
Relations:
      2*U.1 = 0
> T := TorsionUnitGroup(K);
> T;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*T.1 = 0
```

`IsExceptionalUnit(u)`

An element x of an order O is an exceptional unit if both x and $x - 1$ are units in O. This function returns `true` if and only if the order element u is an exceptional unit.

`ExceptionalUnitOrbit(u)`

If u is an exceptional unit of an order O, then all of the units $u_1 = u$, $u_2 = \frac{1}{u}$, $u_3 = 1 - u$, $u_4 = \frac{1}{1-u}$, $u_5 = \frac{u-1}{u}$, $u_6 = \frac{u}{u-1}$ are exceptional. The set $\Omega(u)$ formed by $u_1, \ldots, u_6$ is called the *orbit* of u. Usually it will have 6 elements. This function returns a sequence containing the elements of $\Omega(u)$.

`ExceptionalUnits(O)`

Verbose `UnitEq` *Maximum :* 5

This function returns a sequence S of units of the order O such that any exceptional unit u of O is either in S or is in the orbit of some element of S.

37.8 Solving Equations

MAGMA can solve norm, Thue, index form and unit equations.

37.8.1 Norm Equations

Norm equations in the context of number fields occur in many applications. While MAGMA contains efficient algorithms to solve norm equations it is important to understand the difference between the various types of norm equations that occur. Given some element θ in a number field k together with a finite extension K/k, there are two different types of norm equations attached to this data:

- Diophantine norm equations, that is norm equations where a solution $x \in K$ is restricted to a particular order (or any additive subgroup), and
- field theoretic norm equations where any element in $x \in K$ with $N(x) = \theta$ is a solution.

While in the first case the number of *different* (up to equivalence) solutions is finite, no such restriction holds in the field case. On the other hand, the field case often allows to prove the existence or non-existence of solutions quickly, while no efficient tests exist for the Diophantine case. So it is not surprising that different methods are applied for the different cases. We will discuss the differences with the individual intrinsics.

`NormEquation(O, m)`

`All`	BOOLELT	*Default :* `true`
`Solutions`	RNGINTELT	*Default :* All
`Exact`	BOOLELT	*Default :* `false`
`Ineq`	BOOLELT	*Default :* `false`

Given an order O and an element m of the ground ring of O which can be a positive integer or an element of a suborder, this intrinsic solves a Diophantine norm equation.

This function returns a boolean indicating whether an element $\alpha \in O$ exists such that $\mathrm{N}_{\mathrm{F/L}}(\alpha)$, the norm of α with respect to the subfield L of F (the field of fractions of O), equals m, and if so, a sequence of length at most `Solutions` of solutions α.

The parameter `Exact` may be used to indicate whether an exact solution is required (with `Exact := true`) or whether a solution up to a torsion unit suffices.

The maximal number of required solutions can be indicated with the `Solutions` parameter, but setting `All := true` will override this and the search will find all solutions.

If the order is absolute, then the parameter `Ineq` may be set to true. If so, all solutions x with $|N(x)| <= m$ will be found using a variation of Fincke's ellipsoid method ([Fin84, PZ89]).

Depending on whether the order is absolute maximal, absolute or (simple) relative, different algorithms are used.

If the order is an absolute maximal order, MAGMA will, in a first step, enumerate all integral ideals having the required norm (up to sign). Next, all the ideals are tested for principality using the class group based method. If `Exact := true`, then

a third step is added: we try to find a unit in O of norm -1. This unit is used to sign adjust the solution(s). If there is no such unit, we drop all solutions of the wrong sign.

If the order is absolute, but not maximal, the norm equation is first solved in the maximal order using the above outlined method. In a second step, a complete set of representatives for the unit group of the maximal order modulo the units of O is computed and MAGMA attempts to combine solutions in the maximal order with those representatives to get solutions in O.

If `Solutions` is set, the search stops after the required number of solutions is found.

In case the order is of type `RngOrd` and in some imaginary quadratic field, the norm function is a positive definite quadratic form, thus algorithms based on that property are used. In case the right hand side m equals ± 1, lattice based methods are applied.

If `Ineq` is `true`, which is only supported for absolute fields, lattice enumeration techniques ([Fin84, PZ89]) based on Fincke's ellipsoid method are used.

If the order is (simply) relative different algorithms are implemented, depending on the number of solutions sought. However, common to all of them is that they (partially) work in the `AbsoluteOrder` of O.

If O is a relative maximal order and if we only want to find 1 solution (or to prove that there is none), MAGMA first looks for (integral) solutions in the field using an S-unit based approach as outlined in `NormEquation`. This step gives an affine subspace of the S-unit group that contains all integral solutions of our equation. In a second step, a simplex based technique is used to find totally positive elements in the subspace.

In `All` is given or `Solutions` is > 1, then lattice based methods are used ([Fie97, Jur93, FJP97]).

`NormEquation(F, m)`

`Primes`	ESEQ OF PRIME IDEALS	*Default :* []
`Nice`	BOOLELT	*Default :* `true`

Given a field F and an element m of the base field of F, this function returns a boolean indicating whether an element $\alpha \in F$ exists such that $\mathrm{N}_{F/L}(\alpha)$, the norm of α with respect to the base field L of F equals m, and if so, a sequence of length 1 of solutions α.

The field theoretic norm equations are all solved using S-units. Before discussing some details, we outline the method.

- Determine a set S of prime ideals. We try to obtain a solution as a S-unit for this set S.
- Compute a basis for the S-units
- Compute the action of the norm-map
- Obtain a solution as a preimage.

In general, no effective method is known for the first step. If the field is relative normal however, it is known that is S generates the class group of F and if m is a S-unit, then S is large enough (*suitable* in ([Coh00, 7.5]) [Fie97, Sim02, Gar80]. Thus to find S we have to compute the class group of F. If a (conditional) class group is already known, it is used, otherwise an *unconditional* class group is computed. The initial set S consists of all prime ideals occurring in the decomposition of $m\mathbf{Z}_F$. Note that this step includes the factorisation of m and thus can take a long time is m is large.

Next, we determine a basis for the S-unit group and the action of the norm on it. This give the norm map as a map on the S-unit group as an abstract abelian group.

Finally, the right hand side m is represented as an element of the S-unit group and a solution is then obtained as a preimage under the norm map.

If `Nice` is true, then MAGMA attempts to find a smaller solution by applying a LLL reduction to the original solution.

If `Primes` is give it must contain a list of prime ideals of L. Together with the primes dividing m it is used to form the set S bypassing the computation of an unconditional class group in this step. If L is not normal this can be used to guarantee that S is large enough. Note that the class group computation is still performed when the S-units are computed. Since the correctness of the S-unit group (we need only p-maximality for all primes dividing the (relative) degree of L) can be verified independently of the correctness of the class group, this can be used to derive provable results in cases where the class group cannot be computed unconditionally.

By default, the `MaximalOrder(L)` is used to compute the class group. If the attribute `NeqOrder` is set on L it must contain a maximal order of L. If present, this order will be used for all the subsequent computations.

`NormEquation(m, N)`

`Raw`	BOOLELT	*Default :* `false`
`Primes`	ESEQ OF PRIME IDEALS	*Default :* []

Let N be a map on the multiplicative group of some number field. Formally N may also be defined on the maximal order of the field. This intrinsic tries to find a pre-image for m under N.

This function works by realising N as a endomorphism of S-units for a suitable set S.

If N is a relative norm and if L is (absolutely) normal then the set S as computed for the field theoretic norm equation is guaranteed to be large enough to find a solution if it exists. Note: this condition is not checked.

If `Primes` is given it will be supplemented by the primes dividing m and then used as the set S.

If `Raw` is given, the solution is returned as an unevaluated power product. See the example for details.

The main use of this function is for Galois theoretical constructions where the subfields are defined as fields fixed by certain automorphisms. In this situation the norm function can be realised as the product over the fixed group. It is therefore not necessary to compute a (very messy) relative representation of the field.

`IntegralNormEquation(a, N, O)`

`Nice`	BoolElt	*Default :* `true`

For a an integer or a unit in some number field, N being a multiplicative function on some number field k which is the field of fractions of the order O, try to find a unit in O that is the preimage of a under N. In particular, N restricted to O must be an endomorphism of O. If `Nice` is `true`, the solution will be size-reduced. In particular when the conductor of O in the maximal order of k is large, and therefore the unit index $(Z_k)^* : O^*$ is large as well, this function is much more efficient than the lattice based approach above.

`SimNEQ(K, e, f)`

`S`	[RngOrdIdl]	*Default :* `false`
`HasSolution`	BoolElt	*Default :* `false`

For a number field K and subfield elements $e \in k_1$ and $f \in k_2$, try to find a solution to the simultaneous norm equations $N_{K/k_1}(x) = e$ and $N_{K/k_2}(x) = f$. The algorithm proceeds by first guessing a likely set S of prime ideals that will support a solution - it is exists. Initially S will contain all ramified primes in K, the support of e and f and enough primes to generate the class group of K. In case K is normal over Q this set is large enough to support a solution if there is a solution at all. For arbitrary fields that is most likely not the case. However, if `S` is passed in as a parameter then the set used internally will contain at least this set. If `HasSolution` is `true`, Magma will add primes to S until a solution has been found. This is useful in situations where for some theoretical reason it is known that there has to be a solution.

Example H37E22

We try to solve $N(x) = 3$ in some relative extension: (Note that since the larger field is a quadratic extension, the second call tells us that there is no integral element with norm 3)

```
> x := PolynomialRing(Integers()).1;
> O := MaximalOrder(NumberField([x^2-229, x^2-2]));
> NormEquation(O, 3);
false
> NormEquation(FieldOfFractions(O), 3);
true [
  5/1*$.1*O.1 + 2/3*$.1*O.2
```

```
]
```

Next we solve the same equation but come from a different angle, we will define the norm map as an element of the group ring and, instead of explicitly computing a relative extension, work instead with the implicit fix-field.

```
> K := AbsoluteField(FieldOfFractions(O));
> t := K!NumberField(O).2;
> t^2;
2/1*K.1
> A, _, mA := AutomorphismGroup(K);
> F := sub<A | [ x : x in A | mA(x)(t) eq t]>;
> N := map<K -> K | x:-> &* [ mA(y)(x) : y in F]>;
> NormEquation(3, N);
true [
    5/1*K.1 + 2/3*K.3
]
```

Finally, to show the effect of `Raw`:

```
> f, s, base := NormEquation(3, N:Raw);
> s;
[
    (0 -1 0 -2 0 1 0 0 0 -1 0 1 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 0 0 0 0 1 0)
]
> z := PowerProduct(base, s[1]);
> z;
5/1*K.1 + 2/3*K.3
> N(z);
3/1*K.1;
```

37.8.2 Thue Equations

Thue equations are Diophantine equations of the form $f(x, y) = k$, where k is some integer constant and f is a homogeneous polynomial in two variables. Methods for computing all solutions to such equations are known, although the search space may be larger than is practical. To work with such equations in MAGMA a Thue object (category `Thue`) must be created to store information related to the computations. To solve Thue equations the reduction of Bilu and Hanrot ([BH96]) is used.

`Thue(f)`

Given a polynomial f of degree at least 2 over the integers, this function returns the 'Thue object' corresponding to f; such objects are used by the functions for solving Thue equations. They are printed as the homogeneous version of f.

`Thue(O)`

Given an order O with $\mathbf{Z}$ as its coefficient ring, this function returns the Thue object corresponding to the defining polynomial of O.

`Evaluate(t, a, b)`

`Evaluate(t, S)`

Given a Thue object t and integers a, b, this function returns the evaluation of the homogeneous polynomial f involved in t at (a, b), that is $f(a, b)$. The second form takes as argument the sequence $[a, b]$ instead. This can be convenient if checking the results from an inexact solution.

`Solutions(t, a)`

`Exact`	BoolElt	*Default :* `true`
Verbose	`ThueEq`	*Maximum :* 5

Given a Thue object t and an integer a, this function returns a sequence consisting of all sequences of two integers $[x, y]$ which solve the equation $f(x, y) = a$, where f is the (homogeneous form of) the Thue equation associated with t. If the optional parameter `Exact` is set to `false` then solutions to $f(x, y) = -a$ will also be found.

Example H37E23

A use of thue equations is shown.

```
> R<x> := PolynomialRing(Integers());
> f := x^3 + x + 1;
> T := Thue(f);
> T;
Thue object with form:  X^3 + X Y^2 + Y^3
> Evaluate(T, 3, 2);
47
> Solutions(T, 4);
[]
> Solutions(T, 7);
[]
> Solutions(T, 47);
[
    [ -1, 4 ],
    [ 3, 2 ]
]
> S := Solutions(T, -47 : Exact := false);
> S;
[
    [ -3, -2 ],
    [ -1, 4 ],
    [ 1, -4 ],
    [ 3, 2 ]
```

```
]
> [Evaluate(T, s) : s in S];
[ -47, 47, -47, 47 ]
```

37.8.3 Unit Equations

Unit equations are equations of the form

$$a\epsilon + b\eta = c$$

where a, b and c are some algebraic numbers and ϵ and η are unknown units in the same field.

`UnitEquation(a, b, c)`

Verbose `UnitEq` *Maximum :* 5

Return the sequence of 1×2 matrices $(e1, e2)$ such that $ae_1 + be_2 = c$ for number field elements a, b and c, where e_1 and e_2 are units in the maximal order. The algorithm uses Wildanger's method ([Wil97, Wil00]).

Example H37E24

Usage of `UnitEquation` is shown.

```
> R<x> := PolynomialRing(Integers());
> K<a> := NumberField(x^7 - x^6 + 8*x^5 + 2);
> UnitEquation(a^7, 4*a^2 + a^80, a^7 + a^80 + 4*a^2);
[
    [[1, 0, 0, 0, 0, 0, 0] [1, 0, 0, 0, 0, 0, 0]]
]
```

37.8.4 Index Form Equations

Given an absolute number field K with some order O, index form equations are equations of the form $(O : Z[\alpha]) = k$ where k is some positive integer.

In particular, if $k = 1$ this function will find "all" integral power bases.

In this context "all" means up to equivalence, where two solutions α and β are equivalent iff $\alpha = \pm\beta + r$ for some integer r.

If the field degree is larger than 4, the field must be normal and an integral power basis must already be known.

The implementation follows [Wil97, Wil00] for large degree equations, [GPP93, GPP96] for quartics and [GS89] for cubic fields.

`IndexFormEquation(O, k)`

Verbose `IndexFormEquation` *Maximum :* 5

Given an absolute order O, this function will find "all" (up to equivalence) solutions $\alpha \in OP$ to $(O : Z[\alpha]) = k$.

Example H37E25

We try to compute all integral power bases of the field defined by a zero of $x^4 - 14x^3 + 14x^2 - 14x + 14$:

```
> x := PolynomialRing(Integers()).1;
> O := MaximalOrder(x^4-14*x^3+14*x^2-14*x+14);
> IndexFormEquation(O, 1);
[
    [0, 1, 0, 0]
    [0, 1, -13, 1],
    [0, 1, -1, 1],
]
> [ MinimalPolynomial(x) :x in $1 ];
[
    x^4 - 14*x^3 + 14*x^2 - 14*x + 14,
    x^4 - 28*x^3 + 56*x^2 + 3962*x - 28014,
    x^4 - 2044*x^3 + 6608*x^2 - 7126*x + 2562
]
> [ Discriminant(x) : x in $1 ] ;
[ -80240048, -80240048, -80240048 ]
> Discriminant(O);
-80240048
```

37.9 Ideals and Quotients

Ideals of orders are of two types. All ideals are fractional and inherit from type `RngOrdFracIdl`. Integral ideals can have (the sub–)type `RngOrdIdl`. Some functions only apply to integral ideals and only ideals with the integral type can be prime. An ideal with fractional type but trivial denominator can be converted to have the integral type and any integral ideal can be converted to fractional type.

Ideals can be taken of orders over $\mathbf{Z}$ and orders defined over a maximal order. A few functions are not implemented for the latter.

Where an element or elements are returned from a function the elements are usually in the field of fractions if the ideal has the fractional type and in the order if it has integral type.

37.9.1 Creation of Ideals in Orders

The general ideal constructor can be used to create ideals in orders of algebraic fields, as described below. Since ideals in orders are allowed to be *fractional ideals*, algebraic field elements are allowed as generators. Ideals can also be created how they are written on paper: as an element multiplied by an order.

```
x * O
O * x
```

Create the ideal $x * O$ for element x and order O. If the ideal is integral it will be returned with the integral type.

```
F !! I
O !! I
```

Make the ideal I either fractional (first case where F is a field of fractions compatible with the order of I) or integral (second case where O is an order compatible with the order of I).

```
ideal< O | a1, a2, ... , am >
ideal< O | x >
ideal< O | M, d >
ideal< O | M, I1, ..., In >
```

Given an order O, as well as anything that can be used to produce a sequence of elements of the field of fractions of O return the ideal generated by those elements. If the ideal is integral then it will be returned with the integral type.

A single integer may be given in which case the principal ideal it generates will be returned. A matrix or a module over an order (`ModDed`) (or a matrix and ideals which the module could be created from) can be supplied as the basis for the resulting ideal. An optional second argument is a denominator given as an integer, (except when ideals are given).

Example H37E26

We give an example of the creation of an ideal generated by an element from an order.

```
> R<x> := PolynomialRing(Integers());
> f := x^4-420*x^2+40000;
> K<y> := NumberField(f);
> E := EquationOrder(K);
> O := MaximalOrder(K);
> elt := O ! (y^2/40+y/4);
> elt in E;
false
> I := ideal< O | elt >;
> I;
Principal Ideal of O
```

```
Generator:
    [0, 0, 1, 0]
> FieldOfFractions(O)!!I;
Principal Ideal of O
Generator:
    0.3
> O!!$1 eq I;
true
```

37.9.2 Invariants

Some information describing an ideal can be retrieved.

`Order(I)`

The order O which the ideal I is of.

`Denominator(I)`

The denominator of the fractional ideal I. This is the smallest positive integer d such that $d * I$ is an integral ideal.

`PrimitiveElement(I)`
`UniformizingElement(P)`

A primitive element of an ideal I is an element a which is in I but not in the square of I. This function returns such an element a. `UniformizingElement` returns the primitive element of a prime ideal.

`Index(O, I)`

The index of the integral ideal I, when viewed as a submodule of the order O. This is the same as the cardinality of the finite quotient ring O/I.

`Norm(I)`

The norm of the fractional ideal I. This returns the index of the ideal if the ideal is integral, and is defined on fractional ideals by multiplicativity so that the norm of I^{-1} equals the reciprocal of the norm of I.

`MinimalInteger(I)`

Given an ideal I of an order in some number field, the function returns the least positive integer contained in the ideal.

`Minimum(I)`

Given an ideal I, this function returns the least positive integer m if the ideal is integral or the least positive rational r if is fractional contained in I.

`AbsoluteNorm(I)`

The absolute norm of the fractional ideal I. This returns the index of the ideal if the ideal is integral, and is defined on fractional ideals by multiplicativity so that the norm of I^{-1} equals the reciprocal of the norm of I.

`CoefficientHeight(I)`

`CoefficientHeight(I)`

For an ideal I the coefficient height is defined to be the maximum integer occurring in the current representation of the ideal: If the ideal is given via two elements, this will be the maximal coefficient height of the generators, otherwise the maximal entry of the basis matrix.

`CoefficientLength(I)`

`CoefficientLength(I)`

For an ideal I the coefficient length is defined to be the size of the current representation: If the ideal is given via two elements, this will be the sum of the coefficient lengths of the generators, otherwise the sum of the entries of the basis matrix.

`RamificationIndex(I, p)`

`RamificationDegree(I, p)`

For a prime ideal I of an order O such that $p \in I$ returns the maximal exponent e such that I^e divides the principal ideal pO. If p is not given it is taken to be the minimal integer of I.

`RamificationDegree(I)`

`RamificationIndex(I)`

Computes the relative ramification index of the prime ideal I over the coefficient ring. To be more precise: Let I be an prime ideal of some order O with coeffcient ring o. Then $I \cap o$ is an prime ideal p in o. The ramification index $e = e(I|p)$ is the maximal exponent e such that I^e divides pO.

`ResidueClassField(O, I)`

`ResidueClassField(I)`

If I is a prime ideal of O, this function returns the finite field F isomorphic to O/I and the map $O \to F$.

`Degree(I)`

`InertiaDegree(I)`

Given a prime ideal I this function returns the relative degree f of the residue class field of the ideal I. To be more precise: Let I be a prime ideal in some order O with coefficient ring o. Then $p := o \cap I$ is a prime ideal in o and the residue class field O/I is a finite extension of degree $f = f(I|p)$ of the residue class field o/p.

`Valuation(I, p)`

Given an ideal I and a prime ideal p in an order O, returns the valuation $v_p(I)$ of I at p, that is, the number of factors p in the prime ideal decomposition of I. Note that, since the ideal I is allowed to be a fractional ideal, the returned value may be a negative integer.

`Content(I)`

The content of the ideal I, i.e. the maximal ideal of the base ring dividing I.

Example H37E27

The retrieval of some properties of an ideal is illustrated.

```
> R<x> := PolynomialRing(Integers());
> M := MaximalOrder(x^5 + 4*x^4 - x^3 + 7*x^2 - 1);
> R<x> := PolynomialRing(M);
> O := MaximalOrder(x^3 - 2);
> M;
Maximal Equation Order with defining polynomial x^5 + 4*x^4 - x^3 + 7*x^2 - 1
over Z
> O;
Maximal Order of Equation Order with defining polynomial x^3 - [2, 0, 0, 0, 0]
over M
> I := 19/43*M.4*O.3*O;
> I;
Fractional Principal Ideal of O
Generator:
    19/43*M.4*O.3
> Order(I);
Maximal Order of Equation Order with defining polynomial x^3 - [2, 0, 0, 0, 0]
over M
> Denominator(I);
86
> Denominator(O.3);
2
> PrimitiveElement(I);
19/43*M.4*O.3
> Norm(I);
Fractional Ideal of M
Basis:
[6859    0    0    0    0]
[   0 6859    0    0    0]
[   0    0 6859    0    0]
[   0    0    0 6859    0]
[   0    0    0    0 6859]
Denominator: 159014
> Minimum(I);
19/43
```

```
> p := Factorization(3*M.2*O)[1][1];
> Valuation(I, p);
0
```

37.9.3 Basis Representation

The basis of an ideal can be computed as well as related matrices.

`Basis(I)`

`Basis(I, R)`

Given an ideal I of an order O of the algebraic field F, this function returns a basis for I as a sequence of elements of F (if fractional), O (if integral) or the ring R if given.

`BasisMatrix(I)`

Returns the basis matrix for the ideal I of the order O. The basis matrix consists of the elements of a basis for the ideal written as rows of rational coefficients with respect to the basis of O. The entries of the matrix are elements of $\mathbf{Z}$ for an integral ideal of an order over $\mathbf{Z}$ only or the field of fractions of the coefficient ring of O.

`TransformationMatrix(I)`

Returns the transformation matrix for the ideal I of the order O, as well as a denominator. The transformation matrix consists of the elements of a basis for the ideal written as rows of coefficients with respect to the basis of the order O. The entries of the matrix are elements of $\mathbf{Z}$ for an integral ideal of an order over $\mathbf{Z}$ or the field of fractions of the coefficient ring of O.

`CoefficientIdeals(I)`

The coefficient ideals of the ideal I in a relative extension. These are the ideals $\{A_i\}$ of the coefficient ring of the order of I such that for every element $e \in I$, $e = \sum_i a_i * b_i$ where $\{b_i\}$ is the basis returned for I and each $a_i \in A_i$.

Example H37E28

Continuing from the last example, the use of the basis functions for ideals is shown.

```
> Basis(I);
[
    19/43*M.4*O.3,
    19/86*M.4*O.1,
    19/43*M.4*O.2
]
> Basis(I, NumberField(O));
[
    19/86*$.1^3*$.1^2,
```

```
    19/86*$.1^3,
    19/86*$.1^3*$.1
]
> BasisMatrix(I);
[0 0 19/43*M.4]
[19/86*M.4 0 0]
[0 19/43*M.4 0]
> TransformationMatrix(I);
[M.1 0 0]
[0 M.1 0]
[0 0 M.1]
1
```

For relative extensions, a different method is available:

`Module(I)`

For an ideal I in some relative extension, return a Dedekind module over the coefficient ring with the "same" basis.

37.9.4 Two–Element Presentations

All ideals of maximal orders can be generated by one or two elements of the field of fractions of the order they are an ideal of.

`Generators(I)`

Given a (fractional) ideal I of O, return a sequence containing two elements that generate I as an ideal. The elements will be in the order iff the ideal is integral, otherwise they will come from the field of fractions of O.

`TwoElement(I)`

Given a (fractional) ideal I of O, return two elements of (the field of fractions of) O that generate I as an ideal.

`TwoElementNormal(I)`

Given an integral ideal I of O (a maximal order over $\mathbf{Z}$), return two elements of O that form a two-element normal presentation for I, as well as an integer g such that I is g-normal.

Example H37E29

The generators and two element presentation of I are compared.

```
> Generators(I);
[
    19/43*M.4*O.3
]
> TwoElement(I);
19/43*M.4*O.3
19/43*M.4*O.3
```

37.9.5 Predicates on Ideals

Ideals may be tested for various properties and whether given elements lie in the ideal.

`I eq J` `I ne J`

`x in I` `x notin I` `I subset J`

`IsIntegral(I)`

Returns `true` if and only if the fractional ideal I is integral.

`IsZero(I)`

Returns `true` if the ideal I is the zero ideal, `false` otherwise.

`IsOne(I)`

Returns `true` if the ideal I is the ideal generated by 1, `false` otherwise.

`IsPrime(I)`

Returns `true` if and only if the ideal I is a prime ideal, and `false` otherwise. If I has the integral type and is not prime, the function also returns a proper (ideal) divisor.

Currently, this function becomes very slow as the norm of the ideal becomes large.

`IsPrincipal(I)`

Verbose `ClassGroup` *Maximum :* 5

Returns `true` if the fractional ideal I of the order O is a principal ideal, otherwise `false`. If I is principal, a generator (as an element of the field of fractions of O) is also returned.

If it is known or easy to decide whether I is principal the function will return quickly. If it is necessary to compute the class group of O the function will be slower. If the generator is required this may cause the function to take longer as well.

`IsRamified(P)`

Returns `true` iff the ramification index of the prime ideal P is greater than 1.

`IsRamified(P, O)`

Returns `true` iff for any prime ideal Q lying above P in the order O, the ramification index of Q is greater than 1. P must be a prime ideal of the base ring of O or a prime number (if O is an absolute order).

`IsTotallyRamified(P)`

Returns `true` iff the ramification index of the prime ideal P equals the degree of its order over the base field.

`IsTotallyRamified(P, O)`

Returns `true` iff for any prime ideal Q lying above P in the order O, the ramification index of Q equals the field degree. P must be a prime ideal of the base ring of O or a prime number (if O is an absolute order).

`IsTotallyRamified(K)`

Returns `true` if all primes dividing the discriminant the maximal order of the algebraic field K are totally ramified over the coefficient field of K.

`IsTotallyRamified(O)`

Returns `true` if all primes dividing the discriminant of the maximal order O are totally ramified over the coefficient ring of O.

`IsWildlyRamified(P)`

Returns `true` iff the ramification index of the prime ideal P is a multiple of the characteristic of the residue class field of P.

`IsWildlyRamified(P, O)`

Returns `true` iff for any prime ideal Q of the order O lying above P, the ramification index $e(Q|P)$ is a multiple of the characteristic of the residue class field of Q. P must be a prime ideal of the base ring of O or a prime number (if O is an absolute order).

`IsTamelyRamified(P)`

Returns `true` iff the prime ideal P is not wildly ramified.

`IsTamelyRamified(P, O)`

Returns `true` iff the prime ideal or integer P is not wildly ramified in the order O.

`IsUnramified(P)`

Returns `true` iff the ramification index of the prime ideal P is 1.

`IsUnramified(P, O)`

Returns `true` iff for all prime ideals Q lying above P in the order O, the ramification index of Q is 1. P must be a prime ideal of the base ring of O or a prime number (if O is an absolute order).

`IsInert(P)`

Returns `true` iff the inertia degree of the prime ideal P is the field degree.

`IsInert(P, O)`

Returns `true` iff the prime integer or ideal P in the base ring of the order O is inert, i.e. PO is an unramified prime ideal of O.

`IsSplit(P)`

Returns `true` iff the prime ideal P is not the only prime ideal which lies above its intersection with the base ring of its order.

`IsSplit(P, O)`

Returns `true` iff at least two prime ideals in the order O lie above the prime integer or ideal P of the base ring of O.

`IsTotallySplit(P)`

Returns `true` iff there are as many prime ideals lying above the intersection of the prime ideal P with the base ring of its order as the degree of the order of P.

`IsTotallySplit(P, O)`

Returns `true` iff as many prime ideals in the order O lying above the prime integer or ideal P of the base ring of O as the degree of O.

37.9.6 Ideal Arithmetic

Ideals can be multiplied in several ways, divided and added. Powers of ideals, the least common multiple and the intersection of two ideals can also be calculated.

`I * J`

The product IJ of the (fractional) ideals I and J, generated by the products of elements in I and elements in J.

`x * I`
`I * x`

Given an element x of (or coercible into) a field of fractions F, and a (fractional) ideal I in the order of F, return the product of the ideal and the principal ideal generated by x.

`&*L`

The product of all ideals in the sequence or set L.

`I / J`
`I div J`

The quotient of the (fractional) ideals I and J of an maximal order O. This is the fractional ideal K of O with the property that $JK = I$.

`I div J`

For integral ideals I and J of some maximal order such that J divides I (or, equivalently, J is contained in I), return the integral ideal I/J.

`I / x`

Given an ideal I and an element x construct the fractional ideal I/x.

`I + J`

The sum of the (fractional) ideals I and J, generated by the sums of elements in I and elements in J.

`I ^ k`

The k-th power of the (fractional) ideal I (for an integer k). If I has integral type and k is negative the result will have fractional type.

`I eq J`

Tests if the ideals I and J are equal.

`I subset J`

Tests if the two ideals I and J of the same order are contained in each other. For invertible ideals this is equivalent to checking if J divides I,

`E in I`

Tests if the element E is actually in the ideal I. The element and the ideal have to be compatible, i.e., live in the same number field.

`LCM(I, J)`
`Lcm(I, J)`
`LeastCommonMultiple(I, J)`

Return the least common multiple of ideals I and J. They must both be of the same maximal order.

`GCD(I, J)`
`Gcd(I, J)`
`GreatestCommonDivisor(I, J)`

The greatest common divisor of the ideals I and J of some maximal order of an algebraic number field.

`Content(M)`

For a matrix M with entries in some number field k, compute the gcd of all elements as principal ideals in the maximal order of k.

`I meet J`

The intersection of the (fractional) ideals I and J. For ideal in the maximal order this is the same as the lcm.

`&meetS`

The intersection of all ideals I of the sequence or set S. For ideals in some maximal order this is the same as the lcm.

`I meet R`
`R meet I`

The intersection of the ideal I with the compatible ring R. If $R = \mathbf{Q}$ an error will occur since ideals of $\mathbf{Q}$ cannot be created. If such information is required use `Minimum` instead. An ideal of R is returned.

`a mod I`

A representative of the element a of an order O in the quotient O/I.

`InverseMod(E, M)`
`Modinv(E, M)`

An element y such that $y * E = 1 \bmod M$ where M is an integral ideal or an integer and E is an element of an order.

`ColonIdeal(I, J)`
`IdealQuotient(I, J)`

The colon ideal $[I : J]$ or (I/J) is defined as $\{x \in F : xJ \subseteq I\}$ where F is the field of fractions of the order the ideals I and J belong to.

For ideals of a maximal order (or in general for *invertible* ideals) this is equivalent to I/J, otherwise only `J*ColonIdeal(I, J) subset 1*Order(I)` holds.

`IntegralSplit(I)`

Given an ideal I, return an integral ideal J and a minimal positive integer d such that $I = J/d$.

37.9.7 Roots of Ideals

It is possible to ask for the k-th root of an ideal where k is a positive integer.

`Root(I, k)`

Find the kth root of the ideal I if it exists.

`IsPower(I, k)`

Return `true` if the ideal I is a kth power of an ideal and the ideal it is a power of otherwise `false`.

`SquareRoot(I)`

`Sqrt(I)`

Return the square root of the ideal I if I is square.

`IsSquare(I)`

Return `true` if the ideal I is the square of an ideal and the ideal it is a square of otherwise `false`.

37.9.8 Factorization and Primes

The factorization of an ideal into prime ideals and the divisors of an ideal can be determined.

`Decomposition(O, p)`

Verbose `IdealDecompose` *Maximum :* 5

Given an order O and a rational prime number p or a prime ideal p of the coefficient ring of O, return a sequence of tuples consisting of prime ideals and exponents, according to the decomposition of p in O.

`DecompositionType(O, p)`

Verbose `IdealDecompose` *Maximum :* 5

Given an order O and a rational prime number p or a prime ideal p of the coefficient ring of O, return the decomposition type, ie. a sequence of tuples consisting of the degree of the prime ideals and their ramification index, according to the decomposition of p in O.

`Factorization(I)`

`Factorisation(I)`

Verbose `IdealDecompose` *Maximum :* 5

Returns the prime ideal factorization of an ideal I in an order O, as a sequence of 2-tuples (prime ideal and integer exponent).

`Divisors(I)`

Return the ideals which divide the ideal I which must be of a maximal order.

`Support(I)`

For a non-zero ideal I of some maximal order, return the set of prime ideals p dividing I.

`Support(L)`

`GaloisStable`	BoolElt	*Default :* `false`
`CoprimeOnly`	BoolElt	*Default :* `false`
`UseBernstein`	BoolElt	*Default :* `false`

For a sequence L of ideals in some maximal order (or of number field elements representing principal ideals), return the set of prime ideals dividing at least one of the ideals. If `CoprimeOnly` is given, the set returned will not in general be containing prime ideals, but will satisfy the following:

Every ideal in L can be uniquely decomposed into a power product of ideals in the set returned

The set is minimal and closed under gcd, ie. for two elements in the set, their gcd will be one.

If the number field is normal and if `GaloisStable` is given, then the set returned will be closed under the action of the galois group. Depending on the (unknown) factorisation pattern of the ideals, taking the Galois action into account will in general refine the coprime factorisation.

In general, this function will first construct a coprime basis, and the factorise the result of this step.

If `UseBernstein` is given, then Dan Berstein's asymptoically fast algorithm ([Ber05], which runs in time essentially linear in $\#L$) is used.

`CoprimeBasis(L)`

`GaloisStable`	BoolElt	*Default :* `false`
`UseBernstein`	BoolElt	*Default :* `false`

Given a sequence L of ideals in some maximal order, a coprime basis C for L is constructed. That means

- every element in L has a unique representation as a power product with elements in C
- C is closed under gcd, the ideals in C are pairwise coprime.

If the field is normal and if `GaloisStable` is given, the input sequence is supplemented by the action of the automorphism group, thus potentially refining the coprime basis.

If `UseBernstein` is given then instead of the naive algorithm with quadratic complexity in $\#L$, an asymptotically fast, almost linear algorithm by Dan Berstein is used, [Ber05].

`CoprimeBasisInsert(~L, I)`

`GaloisStable`	BoolElt	*Default :* `false`
`UseBernstein`	BoolElt	*Default :* `false`

Given a coprime basis in the sequence L, enlarge it by the ideal I, ie. enlarge L in such a way that L stays a coprime basis but allows the decomposition of I as well. If the ideals are in a normal field and if `GaloisStable` is given, then in addition of I all its Galois conjugates are inserted as well. Furthermore, a fractional ideal is always decomposed into the numerator and denominator ideal, ach of which is inserted independently.

If `UseBernstein` is given then instead of the naive algorithm with quadratic complexity in $\#L$, an asymptotically fast, almost linear algorithm by Dan Berstein is used, [Ber05].

`PowerProduct(B, E)`

Given sequences B of ideals of some maximal order and E of integers, compute the ideal $\prod B[i]^{E[i]}$.

37.9.9 Other Ideal Operations

Various other functions can be applied to ideals. In addition to those listed, the completion of an order at a prime ideal can also be taken (see Completion on page 891 and Chapter 47).

`ChineseRemainderTheorem(I1, I2, e1, e2)`

`ChineseRemainderTheorem(X, M)`

`CRT(I1, I2, e1, e2)`

`CRT(X, M)`

Returns an element e of the order O such that $(e_1 - e)$ is in the ideal I_1 of O and $(e_2 - e)$ is in the ideal I_2. If a sequence of elements X and a sequence of ideals M is given then the element e will be such that $(X[i] - e)$ is in $M[i]$ for all i.

`CRT(I1, L1, e1, L2)`

`ChineseRemainderTheorem(I1, L1, e1, L2)`

Returns an element e of the order O such that $(e_1 - e)$ is in the ideal I_1 of O and the signs of the conjugates listed in $L1$ are the same as in $L2$.

$L1$, a sorted sequence of integers $0 < l_i <= r1$, is meant to be formal product of infinite places. The signs of the l_i'th conjugate of e will be the same as the sign of $L2[i]$.

`Idempotents(I, J)`

For coprime integral ideals I and J return `true` and elements $i \in I$ and $j \in J$ such that $i + j = 1$.

`CoprimeRepresentative(I, J)`

`MakeCoprime(I, J)`

Given two integral ideals I and J in the same maximal order, find an element q in the field of fractions of this order such that qI is coprime to J.

`ClassRepresentative(I)`

Let I be an ideal in the absolute maximal order O of the number field K. Further, assume that the class group of O has been computed. The class group calculation will have chosen a set of ideal class representatives. This function returns the representative ideal for the ideal class to which I belongs.

`Lattice(I)`

`MinkowskiLattice(I)`

Given an ideal I in an absolute order, returns the lattice determined by the real and complex embeddings of I.

`Different(I)`

The different of the (possibly fractional) ideal I of an order of an algebraic number field.

`Codifferent(I)`

The codifferent of the ideal I. This will be the inverse of the different of I if I is an ideal of a maximal order.

`SUnitGroup(I)`

`SUnitGroup(S)`

`Raw`	BoolElt	*Default :* `false`
Verbose	`ClassGroup`	*Maximum :* 5

An element mu of F, the field of fractions of the order of I, is an S-unit iff $v_p(mu) = 0$ for all p not in S. This function returns the group of S-units of the prime ideals given either as a sequence S of ideals or as a product ideal I. The map from the group into the order containing the ideals is also returned.

If `Raw` is `true` then a product representation of the units is returned instead. That means a sequence L of order elements is determined such that the S-unit group is contained in their multiplicative span. The S-unit group is represented as the $\mathbf{Z}$-module of the exponent vectors for L that give rise to some S-unit. Formally we obtain the S-unit group as an abstract abelian group A, a map from A into some large `RSpace` and the sequence L.

Example H37E30

First we compute the 3-units of $\mathbf{Q}(\sqrt{10})$.

```
> K := QuadraticField(10);
> M := MaximalOrder(K);
> U, mU := SUnitGroup(3*M);
> U;
Abelian Group isomorphic to Z/2 + Z + Z + Z
Defined on 4 generators
Relations:
    2*U.1 = 0
> mU;
Mapping from: GrpAb: U to Field of Fractions of M
> u := mU(U.3); u;
7/1*M.1 - 2/1*M.2
> Decomposition(u);
[
  <Prime Ideal of M
    Two element generators:
      3
      $.2 + 1, 2>
]
```

So u is indeed a 3 unit, as the factorization contains only prime ideals over 3. Next we do the same computation but using the `Raw` option:

```
> U, mU, base := SUnitGroup(3*M:Raw);
> mU;
Mapping from: GrpAb: U to Full RSpace of degree 14 over Integer Ring
> mU(U.3);
( 0  2  1  0  0  0  0  0  0  0 -2)
> PowerProduct(base, $1);
7/1*M.1 - 2/1*M.2
> base[2]^2 * base[3]^1 * base[11]^-2;
7/1*M.1 - 2/1*M.2
```

This representation is of particular importance for large degree fields or fields with large units. To illustrate this, consider the following field from the pari-mailing list:

```
> K := NumberField(Polynomial([ 13824, -3894, -1723, 5, 1291, 1 ]));
> L := LLL(MaximalOrder(K));
> C, mC := ClassGroup(L:Bound := 500);
> U, mU, base := SUnitGroup(1*L:Raw);
> logs := Matrix([Logs(x) : x in Eltseq(base)]);
> mU(U.3)*logs;
(2815256.998090806477937318458440358713392011019115562
    -636746.052519109818325583483504897441603096166734 57
```

```
-770882.44652629342064307574571528191509290934280166)
```

As the logarithm of the absolute value of the real embeddings is of the order 10^6, we expect that a basis representation will have coefficients requiring roughly 10^6 digits. While it is feasible to compute them (using `PowerProduct`), this will take a long time.

`SUnitAction(SU, Act, S)`

`Base` SeqEnum[RngOrdElt] *Default :* []

Given a description of the S-unit group as computed by `SUnitGroup` and a (multiplicative) map of the underlying number field, this function computes the induced map on the abstract abelian group.

The argument S should be a sequence of ideals as in `SUnitGroup`. The argument SU should either be the map returned by `SUnitGroup(S)` as second return value - in which case `Base` is trivial (and not specified) or the second return value of `SUnitGroup(S:Raw)` in which case `Base` should equal the third computed value.

The argument Act must be any (multiplicative) function of the underlying number field or any order, that acts on the S-unit group.

On return, a endomorphism of the domain of SU is obtained.

`SUnitAction(SU, Act, S)`

`Base` SeqEnum[RngOrdElt] *Default :* []

Given a description of the S-unit group as computed by `SUnitGroup` and a sequence of (multiplicative) maps of the underlying number field, this function computes the induced maps on the abstract abelian group.

The argument S should be a sequence of ideals as in `SUnitGroup`. The argument SU should either be the map returned by `SUnitGroup(S)` as second return value - in which case `Base` is trivial (and not specified) or the second return value of `SUnitGroup(S:Raw)` in which case `Base` should equal the third computed value.

The argument Act must be a sequence of (multiplicative) functions of the underlying number field or any order, that acts on the S-unit group.

On return, a sequence of endomorphisms of the domain of SU is obtained.

`SUnitDiscLog(SU, x, S)`

`SUnitDiscLog(SU, L, S)`

`Base` SeqEnum[RngOrdElt] *Default :* []

Given a description of the S-unit group as computed by `SUnitGroup` and a (multiplicative) map of the underlying number field, this function computes the induced map on the abstract abelian group.

The argument S should be a sequence of ideals as in `SUnitGroup`. The argument SU should either be the map returned by `SUnitGroup(S)` as second return value - in which case `Base` is trivial (and not specified) or the second return value of `SUnitGroup(S:Raw)` in which case `Base` should equal the third computed value.

This function solves the discrete logarithm problem for the S-unit group and the algebraic number x. That is, an element in the abstract abelian group representing the S-unit group is computed which corresponds to x. If a list of algebraic numbers L is passed into this function, the discrete logarithm is computed for each of them.

Example H37E31

```
> M := MaximalOrder(Polynomial([ 25, 0, -30, 0, 1 ]));
> S := [ x[1] : x in Factorisation(30*M)];
> U, mU := SUnitGroup(S);
> L := Automorphisms(NumberField(M));
> s2 := SUnitAction(mU, L[2], S);
> s2;
Mapping from: GrpAb: U to GrpAb: U
> L[2](mU(U.2)) eq mU(s2(U.2));
```

Now the same in `Raw` representation:

```
> R, mR, Base := SUnitGroup(S:Raw);
> S2 := SUnitAction(mR, L[2], S:Base := Base);
> [S2(R.i) : i in [1..Ngens(R)]];
[
    R.1,
    R.1 - R.2,
    R.3,
    R.1 + R.3 - R.4,
    R.1 + R.5,
    R.1 + R.3 + R.7,
    R.1 - R.3 + R.6,
    R.8
]
```

If we combine `SUnitAction` with `SUnitDiscLog` we can solve norm equations:

```
> N := map<M -> M | x:-> L[1](x) * L[2](x)>;
> NR := SUnitAction(mR, N, S:Base := Base);
```

Now `NR` is the norm function with respect to the field fixed by `L[2]`.

```
> SUnitDiscLog(mR, FieldOfFractions(M)!5, S:Base := Base);
2*R.5
> $1 in Image(NR);
true
> $2 @@ NR;
R.2 + R.5
> PowerProduct(Base, mR($1));
-3/1*M.1 - 2/1*M.2 + M.4
> N($1);
[5, 0, 0, 0]
```

37.9.10 Quotient Rings

Quotients of orders defined over maximal orders and their integral ideals can be formed resulting in an object with type `RngOrdRes`. Elements of such orders can be created and elementary arithmetic and predicates may be applied to them.

37.9.10.1 Operations on Quotient Rings

The creation of quotient rings and the functions which may be applied to them are described.

`quo< O | I >`

`quo< O | M >`

`quo< O | S >`

Creates the quotient ring $Q = O/I$ of the order O. The right hand side of the constructor may contain an ideal or anything that the ideal constructor can create an ideal from.

`UnitGroup(OQ)`

`MultiplicativeGroup(OQ)`

Returns an abelian group and the map from the group into OQ. OQ must be a quotient of an absolute maximal order.

`Modulus(OQ)`

Return the denominator of the quotient ring OQ, i.e. I where $OQ = O/I$.

Example H37E32

Creation of quotient rings in shown. The orders are the same as for the ideal examples, however an integral ideal is now required.

```
> I := Denominator(I)*I;
> I;
Principal Ideal of O
Generator:
    38/1*M.4*O.3
> Basis(I);
[
    38/1*M.4*O.3,
    19/1*M.4*O.1,
    38/1*M.4*O.2
]
> Q := quo<Order(I) | I>;
> Q;
Quotient Ring of Principal Ideal of O
Generator:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 38, 0]]
> Modulus(Q);
```

```
Principal Ideal of O
Generator:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 38, 0]]
```

37.9.10.2 Elements of Quotients

Functions for elements of the quotient rings are predominantly arithmetic.

`OQ ! a`

Coerce the element a into the quotient OQ where a is anything that can be coerced into the order OQ is a quotient of.

`a mod I`

A canonical representative of the element a (belonging to an order O) in the quotient ring O/I.

`a * b` `a + b` `a - b` `a / b` `- a` `a ^ n`

`a eq b` `a ne b`

`IsZero(a)`

Returns `true` if and only if the quotient ring element a is the zero element of the quotient ring OQ.

`IsOne(a)`

Returns `true` if and only if the quotient ring element a is the one element of the quotient ring OQ.

`IsMinusOne(a)`

Returns `true` if and only if the quotient ring element a is the minus one element of the quotient ring OQ.

`IsUnit(a)`

Returns `true` if and only if the quotient ring element a has an inverse in the quotient ring OQ.

`Eltseq(a)`

`ElementToSequence(a)`

The coefficients of the quotient ring element a in the field of fractions of the coefficient ring of the order of the quotient ring containing a.

37.9.10.3 Reconstruction

Given an element e in some order O, known modulo some ideal I, a common problem in several algorithms is to recover e, ie. a unique minimal element $f \in O$ such that $e - f \in I$ and f is as "small" as possible. An equally common variation would be to ask for some field element f/d with d an integer, such that $f - de \in I$ and f as well as d are small.

In the case of O being the ring of integers and I a power of a prime (ideal), this is usually done by moving to the symmetric residue system in the first case and by rational reconstruction in the second. Here, we use techniques based on the LLL algorithm as described in [FF00].

Since the method is more complicated than in the case of the integer ring, one first has to create a "reconstruction environment" of type `RngOrdRecoEnv`, which is subsequently used to reconstruct any number of elements.

`ReconstructionEnvironment(p, k)`

`ReconstructionEnvironment(p, k)`

Given a (prime) ideal p and an exponent k, initialize the reconstruction process for the ideal $I = p^k$, that is, the object returned can be used to reconstruct elements from "approximations" modulo p^k.

`Reconstruct(x, R)`

`Reconstruct(x, R)`

`UseDenominator` BoolElt *Default :* `false`

Given an order element e, thought to be an approximation modulo p^k where p and k are stored in the reconstruction environment R, return the unique minimal f in the same order such that $e - g \in p^k$. Is `UseDenominator` is `true`, then a field element is computed, otherwise a ring element will be found.

`ChangePrecision(~ R, k)`

Change the ideal $I = p^l$ stored in R to p^k.

Example H37E33

We illustrate the use of the reconstruction environment to find roots of some polynomial over a number field. We will first compute the roots over some completion, up to some precision, then "list" the elements back from the completion into the starting order and finally use reconstruction to get the roots.

```
> f := Polynomial([1,1,1,1,1]);
> M := MaximalOrder(f);
> P := Decomposition(M, 11)[1][1]; P;
Prime Ideal of M
Two element generators:
    [11, 0, 0, 0]
    [2, 1, 0, 0]
> C, mC := Completion(M, P:Precision :=  10);
```

```
> fC := Polynomial([c@ mC : c in Eltseq(f)]);
> rt := Roots(fC); rt;
> R := ReconstructionEnvironment(P, 10);
> [Reconstruct((x[1]) @@ mC, R) : x in rt];
[
    M.4,
    M.3,
    -M.1 - M.2 - M.3 - M.4,
    M.2
]
> [ Evaluate(f, x) : x in $1];
[
    0,
    0,
    0,
    0
 ]
```

37.10 Places and Divisors

A place of a number field K is a class of absolute values (valuations) that induce the same topology on the field. By a famous theorem of Ostrowski, places of number fields are either finite, in which case they are in a one-to-one correspondence with the on-zero prime ideals of the maximal order, or infinite. The infinite places are identified with the embedding of K into $\mathbf{R}$ or with pairs of embeddings into $\mathbf{C}$.

The group of divisors is formally the free group generated by the finite places and the $\mathbf{R}$-vectorspace generated by the infinite ones.

For more information see Section 34.8.

37.10.1 Creation of Structures

`Places(K)`

`DivisorGroup(K)`

The set of places of the number field K and the group of divisors of K respectively.

37.10.2 Operations on Structures

`d1 eq d2` `p1 eq p2`

`NumberField(P)`

`NumberField(D)`

The number field for which P is the set of places or D is the group of divisors.

37.10.3 Creation of Elements

`Place(I)`

The place corresponding to prime ideal I.

`Decomposition(K, p)`
`Decomposition(K, I)`

A sequence of tuples of places and multiplicities. When a finite prime (integer) p is given, the places and multiplicities correspond to the decomposition of p in the maximal order of K. When the infinite prime is given, a sequence of all infinite places is returned.

`Decomposition(K, p)`

For a number field K and a place p of the coefficient field of K, compute all places (and their multiplicity) that extend p. For finite places this is equivalent to the decomposition of the underlying prime ideal. The sequence returned will contain the places of K extending p and their ramification index.

For an infinite place p, this function will compute all extensions of p in K. In this case, the integer returned in the second component of the tuples will be 1 if p is complex or if p is real and extends to a real place and 2 otherwise.

`Decomposition(m, p)`
`Decomposition(m, p)`

For an extension K/k of number fields (where k can be Q as well), given by the embedding map $m : k \to K$, decompose the place p of k in the larger field. In case $k = Q$, the place is given as either a prime number or zero to indicate the infinite place. The sequence returned contains pairs where the first component is a place above p via m and the second is the ramification index.

`InfinitePlaces(K)`
`InfinitePlaces(O)`

A sequence containing all the infinite places of the number field K or the order O is returned.

`Divisor(pl)`

The divisor $1 * pl$ for a place pl.

`Divisor(I)`

The divisor which is the linear combination of the places corresponding to the factorization of the ideal I and the exponents of that factorization.

`Divisor(x)`

The principal divisor xO where O is the maximal order of the underlying number field of which x is an element. In particular, this computes a finite divisor.

`RealPlaces(K)`

For a number field K all sequence containing all real (infinite) places is computed. For an absolute field this are precisely the embeddings into R coming from the real roots of the defining polynomial.

37.10.4 Arithmetic with Places and Divisors

Divisors and places can be added, negated, subtracted and multiplied and divided by integers.

`d1 + d2` `- d` `d1 - d2` `d * k` `d div k`

37.10.5 Other Functions for Places and Divisors

`Valuation(a, p)`

The valuation of the element a of a number field or order at the place p.

`Valuation(I, p)`

The valuation of the ideal I at the finite place p.

`Support(D)`

The support of the divisor D as a sequence of places and a sequence of the corresponding exponents.

`Ideal(D)`

The ideal corresponding to the finite part of the divisor D.

`Evaluate(x, p)`
`Evaluate(x, p)`
`Evaluate(x, p)`

The evaluation of the number field element x in the residue class field of the place p, ie. for a finite place p this corresponds to the image under the residue class field map for the underlying prime ideal. For infinite places, this returns the corresponding conjugate, ie. a real or complex number.

`RealEmbeddings(a)`
`RealEmbeddings(a)`

The sequence of real embeddings of the algebraic number a is computed, ie. a is evaluated at all real places of the number field.

`RealSigns(a)`

`RealSigns(a)`

A sequence containing ± 1 depending on whether the evaluation of the number field element a at the corresponding real place is positive of negative.

`IsReal(p)`

For an infinite place p, returns `true` if the corresponding embedding is real, ie. if `Evaluate` at p will give real results.

`IsComplex(p)`

For an infinite place p, return `true` if the corresponding embedding is complex, ie. if `Evaluate` at p will generally yield complex results.

`IsFinite(p)`

For a place p of a number fiel, return if the place is finite, ie. if it corresponds to a prime ideal.

`IsInfinite(p)`

For a place p of a number field return if the place is infinite, ie. if it corresponds to an embedding of the number field into the real or complex numbers. If the place is infinite, the index of the embedding it corresponds to is returned as well.

`Extends(P, p)`

For two places P of K and p of k where K is an extension of k, check whether P extends p. For finite places, this is equivalent to checking if the prime ideal corresponding to P dives, in the maximal order of K the prime ideal of p. For infinite places `true` implies that for elements of k, evaluation at P and p will give identical results.

`InertiaDegree(P)`

`Degree(P)`

For a place P of a number field, return the inertia degree of P. That is for a finite place, return the degree of the residue class field over it's prime field, for infinite places it is always 1.

`Degree(D)`

For a divisor D of a number field, the degree is the weighted sum of the degrees of the supporting places, the weights being the multiplicities.

`NumberField(P)`

For a place P or divisor D of a number field, return the underlying number field.

`ResidueClassField(P)`

For a place P of a number field, compute the residue class field of P. For a finite place this will be a finite field, namely the residue class field of the underlying prime ideal. For an infinite place, the residue class field will be the field of real or complex numbers.

`UniformizingElement(P)`

For a finite place P of a number field, return an element of valuation 1. This will be the uniformizing element of the underlying prime ideal as well.

`LocalDegree(P)`

The degree of the completion at the place P, ie. the product of the inertia degree times the ramification index.

`RamificationIndex(P)`

The ramification index of the place P. For infinite real places this is 1 and 2 for complex places.

`DecompositionGroup(P)`

For a place P of a normal number field, return the decomposition group as a subgroup of the (abstract) automorphism group.

37.11 Bibliography

[**Bai96**] Georg Baier. Zum Round 4 Algorithmus. Diplomarbeit, Technische Universität Berlin, 1996.
URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/baier.ps.gz.

[**Ber05**] Daniel J. Bernstein. Factoring into coprimes in essentially linear time. *J. of Algorithms*, 54(1):1–30, 2005.

[**BH96**] Yuri Bilu and Guillaume Hanrot. Solving Thue Equations of High Degree. *J. Number Th.*, 60:373–392, 1996.

[**Bia**] J.-F. Biasse. Number field sieve to compute Class groups.

[**Bj94**] Johannes A. Buchmann and Hendrik W. Lenstra jr. Approximating rings of integers in number fields. *J. Théor. Nombres Bordx.*, 6(2):221–260, 1994.

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**Coh00**] Henri Cohen. *Advanced Topics in Computational Number Theory*. Springer, Berlin–Heidelberg–New York, 2000.

[**FF00**] Claus Fieker and Carsten Friedrichs. On reconstruction of algebraic numbers. In Bosma [Bos00], pages 285–296.

[**Fie97**] Claus Fieker. *Über relative Normgleichungen in algebraischen Zahlkörpern.* Dissertation, Technische Universität Berlin, 1997.
URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_CF.ps.gz.

[**Fin84**] Ulrich Fincke. *Ein Ellipsoidverfahren zur Lösung von Normgleichungen in algebraischen Zahlkörpern.* Dissertation, Heinrich-Heine-Universität Düsseldorf, 1984.

[**FJP97**] C. Fieker, A. Jurk, and M. Pohst. On solving relative norm equations in algebraic number fields. *Math. Comput.*, 66(217):399–410, 1997.

[**Fri97**] Carsten Friedrichs. Berechnung relativer Ganzheitsbasen mit dem Round-2-Algorithmus. Diplomarbeit, Technische Universität Berlin, 1997.
URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/friedrichs.ps.gz.

[**Fri00**] Carsten Friedrichs. *Berechnung von Maximalordnungen über Dedekindringen.* Dissertation, Technische Universität Berlin, 2000.
URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_fried.pdf.gz.

[**Gar80**] Dennis A. Garbanati. An Algorithm for finding an algebraic number whose norm is a given rational number. *J. reine angew. Math.*, 316:1–13, 1980.

[**GPP93**] Istvan Gaál, Attila Pethő, and Michael E. Pohst. On the resolution of index form equations in quartic number fields. *J. Symbolic Comp.*, 16:563–584, 1993.

[**GPP96**] Istvan Gaál, Attila Pethő, and Michael E. Pohst. Simultaneous representation of integers by a pair of ternary quadratic forms – With an application to index form equations in quartic number fields. *J. Number Th.*, 57:90–104, 1996.

[**GS89**] Istvan Gaál and Nicole Schulte. Computing all power integral bases of cubic fields. *Math. Comp.*, 53:689–696, 1989.

[**Heß96**] Florian Heß. Zur Klassengruppenberechnung in algebraischen Zahlkörpern. Diplomarbeit, Technische Universität Berlin, 1996.
URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/hess.ps.gz.

[**Jur93**] Andreas Jurk. *Über die Berechnung von Lösungen relativer Normgleichungen in algebraischen Zahlkörpern.* Dissertation, Heinrich-Heine-Universität Düsseldorf, 1993.

[**KAN97**] KANT Group. KANT V4. *J. Symbolic Comp.*, 24(3–4):267–383, 1997.

[**KAN00**] KANT Group. The Number Theory Package KANT/KASH.
URL:http://www.math.tu-berlin.de/∼kant, 2000.

[**PK05**] Sebastian Pauli and Jüren Klüners. Computing residue class rings and Picard groups of orders. *J. of Algebra*, 292:47–64, 2005.

[**Poh93**] M. Pohst. *Computational Algebraic Number Theory.* DMV Seminar Band 21. Birkhäuser Verlag, Basel - Boston - Berlin, 1993.

[**PZ89**] Michael E. Pohst and Hans Zassenhaus. *Algorithmic Algebraic Number Theory.* Encyclopaedia of mathematics and its applications. Cambridge University Press, Cambridge, 1989.

[**Sim02**] Denis Simon. Solving norm equations in relative number fields using S-units. *Math. Comput.*, 71(239):1287–1305, 2002.

[**Sut12**] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47(5):552–567, 2012.

[**Wil97**] Klaus Wildanger. *Über das Lösen von Einheiten- und Indexformgleichungen in algebraischen Zahlkörpern mit einer Anwendung auf die Bestimmung aller ganzen Punkte einer Mordellschen Kurve.* Dissertation, Technische Universität Berlin, 1997. URL:http://www.math.tu-berlin.de/∼kant/publications/diss/KW_diss.ps.gz.

[**Wil00**] Klaus Wildanger. Über das Lösen von Einheiten- und Indexformgleichungen in algebraischen Zahlkörpern. (On the solution of units and index form equations in algebraic number fields). *J. Number Th.*, 2(82):188–224, 2000.

38 GALOIS THEORY OF NUMBER FIELDS

Chapter 38
GALOIS THEORY OF NUMBER FIELDS

The Galois theory of number fields deals with the group of automorphisms of a number field, the group of automorphisms of the normal closure of a number field and with the subfields of a given number field. While all three problems are, at least in theory, dealt with easily using the main theorems of Galois theory, they correspond to completely different and independent algorithmic problems.

The first task, that of computing automorphisms of normal extensions of $\mathbf{Q}$ (and of abelian extensions of number fields) can be thought of a special case of factorisation of polynomials over number fields: the automorphisms of a number field are in one-to-one correspondence with the roots of the defining equation in the field. However, the computation follows a different approach and is based on some combinatorial properties. It should be noted, though, that the algorithms only apply to normal fields; i.e., they cannot be used to find non-trivial automorphisms of non-normal fields!

The second task, namely that of computing the Galois group of the normal closure of a number field, is of course closely related to the problem of computing the Galois group of a polynomial. The method implemented in Magma allows the computation of Galois groups of polynomials (and number fields) of arbitrarily high degrees and is independent of the classification of transitive permutation groups. The result of the computation of a Galois group will be a permutation group acting on the roots of the (defining) polynomial, where the roots (or approximations of them) are explicitly computed in some suitable p-adic field; thus the splitting field is not (directly) part of the computation. The explicit action on the roots allows one, for example, to compute algebraic representations of arbitrary subfields of the splitting field, even the splitting field itself, provided the degree is not too large.

The last main task dealt with in this chapter is the computation of subfields of a number field. While of course this can be done using the main theorem of Galois theory (the correspondence between subgroups and subfields), the computation is completely independent; in fact, the computation of subfields is usually the first step in the computation of the Galois group. The algorithm used here is mainly combinatorical.

Finally, this chapter also deals with applications of the Galois theory:

- the computation of subfields and subfield towers of the splitting field
- solvability by radicals: if the Galois group of a polynomial is solvable, the roots of the polynomial can be represented by (iterated) radicals.
- basic Galois-cohomology; i.e., the action of the automorphisms on the ideal class group, the multiplicative group of the field and derived objects.

38.1 Automorphism Groups

Automorphisms of an algebraic field and the group they form can be calculated. Furthermore, field invariants that relate to the automorphism group can be determined.

`Automorphisms(F)`

`Abelian`	BoolElt	*Default :* `false`
Verbose	`AutomorphismGroup`	*Maximum :* 3

Given an algebraic field F, return the automorphisms of F as a sequence of maps. If the extension is known to be abelian, the parameter `Abelian` should be set to `true` in which case a much more efficient algorithm [Klü97, AK99] will be employed. If F is not a normal extension, the automorphisms are obtained by a variation of the polynomial factorisation algorithm.

`AutomorphismGroup(F)`

`Abelian`	BoolElt	*Default :* `false`
Verbose	`AutomorphismGroup`	*Maximum :* 3

Given an algebraic field F, that is either a simple normal extension of $\mathbf{Q}$ or simple abelian extension of $\mathbf{Q}$, return the automorphism group G of K as a permutation group of degree n, where n is the degree of the extension. If the extension is known to be abelian, the parameter `Abelian` should be set to `true` in which case a much more efficient algorithm [Klü97, AK99] will be employed. If F is not a normal extension of $\mathbf{Q}$ an error will occur. In addition to returning G, the function also returns the power structure Aut of all automorphisms of F, and the transfer map ϕ from G into Aut.

Example H38E1

We consider the extension obtained by adjoining a root of the irreducible polynomial $x^4 - 4x^2 + 1$ to $\mathbf{Q}$.

```
> Q := RationalField();
> R<x> := PolynomialRing(Q);
> K<w> := NumberField(x^4 - 4*x^2 + 1);
> A := Automorphisms(K);
> A;
[
    Mapping from: FldNum: K to FldNum: K,
    Mapping from: FldNum: K to FldNum: K,
    Mapping from: FldNum: K to FldNum: K,
    Mapping from: FldNum: K to FldNum: K
]
> for phi in A do phi(w); end for;
w
w^3 - 4*w
-w^3 + 4*w
```

```
-w
```

Taking the same field K we use instead the function `AutomorphismGroup`:

```
> G, Aut, tau := AutomorphismGroup(K);
> for x in G do tau(x)(w); end for;
w
w^3 - 4*w
-w^3 + 4*w
-w
```

`AutomorphismGroup(K, F)`

Computes the group of K automorphisms of F as a permutation group together with a list of all automorphisms and a map between the permutation group and explicit automorphisms of the field.

This function computes the automorphism group of F over $\mathbf{Q}$ first.

`DecompositionGroup(p)`

For an ideal p of the maximal order of some absolute normal field F with group of automorphisms G, compute the decomposition group, i.e. the subgroup U of the automorphism group such that:

$$U := \{s \in G | s(p) = p\}$$

If F is not a normal extension of $\mathbf{Q}$ an error will occur.

`RamificationGroup(p, i)`

For an ideal p of the maximal order M of some absolute normal field F with group of automorphisms G, compute the i-th ramification group, i.e. the subgroup U of the automorphism group such that:

$$U := \{s \in G | s(x) - x \in p^{i+1} \text{for all } x \text{ in } M\}$$

If F is not a normal extension of $\mathbf{Q}$ an error will occur.

`RamificationGroup(p)`

This is just an abbreviation for `RamificationGroup(p, 1)`.

`InertiaGroup(p)`

This is just an abbreviation for `RamificationGroup(p, 0)`.

`FixedField(K, U)`

Given a normal field K over $\mathbf{Q}$ and a subgroup U of the `AutomorphismGroup(K)`, compute the subfield L that is fixed by U.

This function is inverse to `FixedGroup`.

If K is not a normal extension of $\mathbf{Q}$ an error will occur.

`FixedField(K, S)`

For an algebraic field K and a list S of automorphism of K, compute the maximal subfield of K fixed by S.

`FixedGroup(K, L)`

Given a normal field K over $\mathbf{Q}$ and a subfield L, compute the subgroup U of the `AutomorphismGroup(K)` that fixes L.

This function is inverse to `FixedField`.

If K is not a normal extension of $\mathbf{Q}$ an error will occur.

`FixedGroup(K, L)`

Given a normal field K over $\mathbf{Q}$ and a sequence of number field elements L, compute the subgroup U of the `AutomorphismGroup(K)` that fixes L.

If K is not a normal extension of $\mathbf{Q}$ an error will occur.

`FixedGroup(K, a)`

Given a normal field K over $\mathbf{Q}$ and a number field element a, compute the subgroup U of the `AutomorphismGroup(K)` that fixes a.

This function is inverse to `FixedField`.

If K is not a normal extension of $\mathbf{Q}$ an error will occur.

`DecompositionField(p)`

This is an abbreviation for `FixedField(K, DecompositionGroup(p))` where K is the number field of the order of p.

`RamificationField(p, i)`

This is an abbreviation for `FixedField(K, RamificationGroup(p, i))` where K is the number field of the order of p.

`RamificationField(p)`

This is an abbreviation for `FixedField(K, RamificationGroup(p))` where K is the number field of the order of p.

`InertiaField(p)`

This is an abbreviation for `FixedField(K, InertiaField(p))` where K is the number field of the order of p.

Example H38E2

We will demonstrate the various groups and fields. In order to do so, we first construct a non-trivial normal field.

```
> o := MaximalOrder(ext<Rationals()|>.1^4-3);
> os := MaximalOrder(SplittingField(NumberField(o)));
> P := Decomposition(os, 2)[1][1];
> G, M := RayClassGroup(P^3);
> G;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*G.1 = 0
```

Since G is cyclic and the module P invariant under the automorphisms of os, the class field corresponding to G will be normal over Q. It Galois group over Q will be an extension of D_4 by C_2.

```
> A := AbelianExtension(M);
> O := MaximalOrder(EquationOrder(A));
> Oa := AbsoluteOrder(O);
> Ka := NumberField(Oa);
> Gal, _, Map := AutomorphismGroup(Ka);
> Gal;
Permutation group Gal acting on a set of cardinality 16
Order = 16 = 2^4
    (1, 2, 7, 5)(3, 8, 6, 10)(4, 12, 14, 9)(11, 16, 13, 15)
    (1, 3, 7, 6)(2, 8, 5, 10)(4, 13, 14, 11)(9, 16, 12, 15)
    (1, 4)(2, 9)(3, 11)(5, 12)(6, 13)(7, 14)(8, 15)(10, 16)
```

Now, let us pick some ideals. The only interesting primes are the primes dividing the discriminant, which in this case will be the primes over 2 and 3.

```
> P2 := Decomposition(Oa, 2)[1][1];
> P3 := Decomposition(Oa, 3)[1][1];
```

First, the valuation of the different of Oa at $P2$ should be $\sum_{i=0}^{\infty}(\#G(P2,i)-1)$ where $G(P2,i)$ is the i-th ramification group.

```
> s := 0; i := 0;
> repeat
>   G := RamificationGroup(P2, i);
>   s +:= #G-1;
>   print i, "-th ramification group is of order ", #G;
>   i +:= 1;
> until #G eq 1;
0 -th ramification group is of order  8
1 -th ramification group is of order  8
2 -th ramification group is of order  2
3 -th ramification group is of order  2
```

```
4 -th ramification group is of order  2
5 -th ramification group is of order  2
6 -th ramification group is of order  1
> s;
18
> Valuation(Different(Oa), P2);
18
```

According to the theory, $P2$ should be totally ramified over the inertia field and unramified over Q:

```
> K2 := InertiaField(P2);
> M2 := MaximalOrder(K2);
> K2r := RelativeField(K2, Ka);
> M2r := MaximalOrder(K2r);
> p2 := M2 meet (MaximalOrder(K2r)!!P2);
> IsInert(p2);
true
> IsTotallyRamified(M2r!!P2);
true
```

Now we try the same for $P3$. Since 3 is split in Ka, we may consider an additional field: the decomposition field. It should be the maximal subfield if K such that 3 is neither inert ($f = 1$) nor ramified ($e = 1$), therefore 3 has to split totally.

```
> D3 := DecompositionField(P3);
> D3M := MaximalOrder(D3);
> IsTotallySplit(3, D3M);
true
```

The inertia field is the maximal subfield such that 3 is unramified. It has to be an extension of $D3$.

```
> I3 := InertiaField(P3);
> I3;
Number Field with defining polynomial $.1^4 +
    80346384509631057182412*$.1^3 +
    22568355830378814326531151377362093966156930 22*$.\
    1^2 + 279581809285547646905698973995584573657929 1605177\
    3809455107173769804*$.1 +
    22077876856825539803853422635266440799754188013751 61428\
    41147104301325760481728833650060994 over the Rational
Field
> Discriminant($1);
10700005925626216180895747020647047166414333000723923591882\
57829873417638072117114945163507537844711544617147344227643\
21408503489566949866295669400825222748660907808235401444104\
29329493645714658394673579309893726532999745496689571082958\
82869371250900344499670337698224 64
```

This (polynomial) discriminant is huge, in fact it is so large that we should avoid the factorisation. We already know the discriminant of Ka. The discriminant of $I3$ has to be a divisor - so we can

use the `Discriminant` parameter to `MaximalOrder`: (We are going to need the `MaximalOrder` for the following embedding.)

```
> I3M := MaximalOrder(EquationOrder(I3):
> Discriminant := Discriminant(Oa));
> I3M := MaximalOrder(I3);
```

$D3$ should be a subfield of $I3$, so lets verify it:

```
> IsSubfield(D3, I3);
true Mapping from: FldNum: D3 to FldNum: I3
```

As a side-effect, MAGMA is now aware of the embedding and will use it. Without the `IsSubfield` call, the `RelativeField` function will fail.

```
> I3r := RelativeField(D3, I3);
> I3rM := MaximalOrder(I3r);
> K3r := RelativeField(D3, Ka);
> K3rM := MaximalOrder(K3r);
> IsInert(K3rM!!P3 meet D3M, I3rM);
true
```

The last step: verify that $P3$ is totally ramified over $I3$:

```
> K3r := RelativeField(I3, Ka);
> K3rM := MaximalOrder(K3r);
> IsTotallyRamified(K3rM!!P3 meet I3M, K3rM);
true
```

Using the decomposition group, we can get the splitting behaviour of any prime in any subfield of Ka.

```
> L := SubgroupLattice(Gal);
> [ IsNormal(Gal, L[x]) : x in [1..#L]];
[ true, true, true, true, false, false, false, false, true,
true, true, true, true, true, true, false, false, false,
false, true, true, true, true, true, true, true, true ]
> U := L[5];
> k := FixedField(Ka, U);
> kM := MaximalOrder(EquationOrder(k):
>   Discriminant := Discriminant(Oa));
> kM := MaximalOrder(k);
> Kr := RelativeField(k, Ka);
> KrM := MaximalOrder(Kr);
> P43 := Decomposition(Oa, 43)[1][1];
> V := DecompositionGroup(P43);
```

The splitting behaviour is determined by the double coset decomposition of Gal with respect to U and V:

```
> f, I := CosetAction(Gal, U);
> orbs := Orbits(f(V));
> reps := [];
```

```
> for o in orbs do
>   _, x := IsConjugate(I, 1, Rep(o));
>   Append(~reps, x @@ f);
> end for;
> reps;
[
    Id(G),
    (1, 2, 7, 5)(3, 8, 6, 10)(4, 12, 14, 9)(11, 16, 13, 15),
    (1, 7)(2, 5)(3, 6)(4, 14)(8, 10)(9, 12)(11, 13)(15, 16),
    (1, 8)(2, 6)(3, 5)(4, 15)(7, 10)(9, 13)(11, 12)(14, 16),
]
> #reps;
4
```

So there will be at least 4 prime ideals over 43 in k:

```
> L := [ ];
> for i in reps do
>   Append(~L, kM meet KrM !! Map(i)(P43));
> end for;
> [ IsPrime(x) : x in L];
[ true, true, true, true ]
> LL := Decomposition(kM, 43);#LL;
4
> [ Position(L, x[1]) : x in LL];
[ 4, 3, 1, 2 ]
```

`FrobeniusElement(K, p)`

Compute a Frobenius element at p in the Galois group of the Galois closure of K. This is a permutation on the roots of a polynomial defining K, which can be recovered as `DefiningPolynomial(A)` for any Artin representation A of K; the Frobenius element is well-defined up to conjugacy and modulo inertia.

Example H38E3

We take a polynomial whose Galois group is D_5 and compute Frobenius elements at $p = 2$ and $p = 5$. They in two different conjugacy classes of 5-cycles in the Galois group.

```
> load galpols;
> f:=PolynomialWithGaloisGroup(5,2);
> assert IsIsomorphic(GaloisGroup(f),DihedralGroup(5));
> K:=NumberField(f);
> FrobeniusElement(K,2);
(1, 5, 4, 3, 2)
> FrobeniusElement(K,5);
(1, 3, 5, 2, 4)
```

38.2 Galois Groups

Finding Galois groups (of normal closures) of algebraic fields is a hard problem, in general. All practical currently-used algorithms fall into two groups: The absolute resolvent method [SM85] and the method of Stauduhar [Sta73].

The MAGMA implementation is based on an extension of the method of Stauduhar [GK00, Gei03] and recent work by Klüners and Fieker [FK12].

For polynomials over $\mathbf{Z}, \mathbf{Q}$, number fields and global function fields and irreducible polynomials over function fields over $\mathbf{Q}$, MAGMA is able to compute the Galois group without any a-priori restrictions on the degree. Note, however, that the running time and memory constraints can make computations in degree > 50 impossible, although computations in degree > 200 have been successful as well. In contrast to the absolute resolvent method, it also provides the explicit action on the roots of the polynomial f which generates the algebraic field. On demand, the older version which is restricted to a maximum degree of 23, is still available.

Roughly speaking, the method of Stauduhar traverses the subgroup lattice of transitive permutation groups of degree n from the symmetric group to the actual Galois group. This is done by using so-called relative resolvents. Resolvents are polynomials whose splitting fields are subfields of the splitting field of the given polynomial which are computed using approximations of the roots of the polynomial f.

If the field (or the field defined by a polynomial) has subfields (i.e. the Galois group is imprimitive) the current implementation changes the starting point of the algorithm in the subgroup lattice, to get as close as possible to the actual Galois group. This is done via computation of subfields of a stem field of f, that is the field extension of $\mathbf{Q}$ which we get by adjoining a root of f to $\mathbf{Q}$. Using this knowledge of the subfields, the Galois group is found as a subgroup of the intersection of suitable wreath products which may be easily computed. This intersection is a good starting point for the algorithm.

If the field (or the field defined by a polynomial) does not have subfields (i.e. the Galois group is primitive) we use a combination of the method of Stauduhar and the absolute resolvent method. The Frobenius automorphism of the underlying p-adic field or the complex conjugation, when using complex approximations of the roots of the polynomial f, already determines a subgroup of the Galois group, which is used to speed up computations in the primitive case.

`GaloisGroup(f)`

`Prime`	RngElt	*Default :*
`ShortOK`	BoolElt	*Default :* `false`
`Ring`	GaloisData	*Default :*
`NextPrime`	UserProgram	*Default :*
Verbose	`GaloisGroup`	*Maximum :* 5
Verbose	`Invariant`	*Maximum :* 3

Given a polynomial f over the integers, rationals, a number field or an order thereof, compute the Galois group of a splitting field for f, ie. determine the subgroup

of the permutations of the roots of f in a splitting field that correspond to field automorphisms. The method applied here is a variant of Stauduhar's algorithm, but with no dependency on the explicit classification of transitive groups and thus no a-priori degree limitation. It must be stated though that this function does not return proven results, if such results are necessary, one needs to call `GaloisProof` afterwards.

The prime to use for splitting field computations can be given via the parameter `Prime`. The method of choosing of primes for splitting field computations can be given by the parameter `NextPrime`.

`GaloisGroup(K)`

`Prime`	RngElt	*Default :*
`ShortOK`	BoolElt	*Default :* `false`
`Ring`	GaloisData	*Default :*
`NextPrime`	UserProgram	*Default :*
`Current`	BoolElt	*Default :* `false`
`Subfields`	BoolElt	*Default :* `true`
`Old`	BoolElt	*Default :* `false`
`Type`	MonStgElt	*Default :* "p-Adic"
`Prec`	RngIntElt	*Default :* 20
`Time`	Rec	*Default :*
Verbose	`GaloisGroup`	*Maximum :* 5
Verbose	`Invariant`	*Maximum :* 3

Given a number field K, compute the Galois group of a normal closure of K. The group is returned as an abstract permutation group acting on the roots of the defining polynomial of k in a suitable splitting field. The method applied here is a variant of Stauduhar's algorithm, but with no dependency on the explicit classification of transitive groups and thus no a-priori degree limitation. It must be stated though that this function does not return proven results, if such results are necessary, one needs to call `GaloisProof` afterwards.

The prime to use for splitting field computations can be given via the parameter `Prime`. The method of choosing of primes for splitting field computations can be given by the parameter `NextPrime`.

`GaloisProof(f, S)`

`GaloisProof(K, S)`

Given the result of a (conditional) computation by `GaloisGroup` for either an irreducible polynomial over the integers or an absolute number field, try to find proofs for the conditional steps of the algorithm. The method employed here is to show that a suitable absolute resolvent polynomial has a factor of a specific degree that can be used to differentiate between the possible groups.

`GaloisRoot(f, i, S)`

`Prec`	RngIntElt	*Default :* 20
`Bound`	RngIntElt	*Default :* 0

Given a polynomial f and the result S of a computation of its Galois group, return the ith root in the ordering obtained form the Galois process to the given precision. The precision can be specified either directly by setting `Prec` to the desired p-adic precision or by giving a bound B in `Bound`. In the latter case the p-adic precision k will be calculated such that $p^k > B$.

`GaloisRoot(i, S)`

Given the result S of a computation of a Galois group and an integer i, compute the ith conjugate of the primitive element used during the computation. The precision can be specified either directly by setting `Prec` to the desired p-adic precision or by giving a bound B in `Bound`. In the latter case the p-adic precision k will be calculated such that $p^k > B$.

`Stauduhar(G, H, S, B)`

Verbose	`GaloisGroup`	*Maximum :* 5
Verbose	`Invariant`	*Maximum :* 3
`AlwaysTransform`	BoolElt	*Default :* `false`
`Coset`	SeqEnum	*Default :*
`PreCompInvar`	UserProgram	*Default :*

This function gives access to a single step of the Stauduhar method: Let G be a permutation group known to contain the Galois group of the object under investigation with the numbering of the "roots" determined by S. Furthermore, let B be a bound on the absolute value of the complex roots of the object and H be a (maximal) subgroup of G. Under these circumstances, the intrinsic will decide if there is some $g \in G$ such that the Galois group is contained in H^g. The primary return value can be:

1 if the Galois group is proven to be a subgroup of H^g up to precision problems, indicated by the 3rd value

-1 if there is a proof that the Galois group is contained in a proper subgroup of G and maybe in H^g

-2 if the Galois group may be in H^g, but we could not prove that it is in a proper subgroup of G

0 the Galois group is not contained in a conjugate of H.

In case of a non-zero result, the second return value will be the element g conjugating H, the third value will be `true` or `false`, depending on whether the p-adic bound used were proven or heuristic and the fourth value is the invariant used to separate the groups.

The optional parameter `Coset` can be used to pass a transversal of G/H in, while `PreCompInvar` should contain a suitable invariant separating G and H if set.

`IsInt(x, B, S)`

Given an element x in the splitting field determined by S and a bound B on the complex absolute value, determine if there exists an element $y \in \mathbf{Z}$ or an extension of $\mathbf{Z}$ defined by the polynomial the Galois group is being computed for, such that $y = x$ up the precision of x and such that $|y| < B$. In case such a y exists, it is returned as a second return value.

Example H38E4

A Galois group computation is shown below.

```
> Z:= Integers();
> P<x>:= PolynomialRing(Z);
> G, R, S := GaloisGroup(x^6-108);
> G;
Permutation group G acting on a set of cardinality 6
Order = 6 = 2 * 3
    (1, 5, 3)(2, 6, 4)
    (1, 2)(3, 6)(4, 5)
> R;
[ -58648*$.1 + 53139 + O(11^5), 58648*$.1 - 19478 +
O(11^5), -43755*$.1 - 72617 + O(11^5), 58648*$.1 -
    53139 + O(11^5), -58648*$.1 + 19478 + O(11^5),
    43755*$.1 + 72617 + O(11^5) ]
> S;
GaloisData over Z_11
> time G, _, S := GaloisGroup(x^32-x^16+2);
Time: 65.760
> #G;
2048
```

Some examples for the relative case

```
> load galpols;
> f := PolynomialWithGaloisGroup(9, 14);
> G := GaloisGroup(f);
> TransitiveGroupIdentification(G);
14 9
> M := MaximalOrder(f);
> kM := FieldOfFractions(M);
> f:= Factorisation(Polynomial(kM, f))[2][1];
> f;
$.1^8 + (-2/1*kM.1 + kM.2)*$.1^7 + (-60/1*kM.1 -
    2/1*kM.2 + kM.3)*$.1^6 + (120/1*kM.1 - 60/1*kM.2 -
    2/1*kM.3 + kM.4)*$.1^5 + (980/1*kM.1 + 120/1*kM.2 -
    60/1*kM.3 - 2/1*kM.4 + kM.5)*$.1^4 + (-1808/1*kM.1
    + 980/1*kM.2 + 120/1*kM.3 - 60/1*kM.4 - 2/1*kM.5 +
    kM.6)*$.1^3 + (-4012/1*kM.1 - 1808/1*kM.2 +
    980/1*kM.3 + 120/1*kM.4 - 60/1*kM.5 - 2/1*kM.6 +
```

```
    kM.7)*$.1^2 + (4936/1*kM.1 - 4013/1*kM.2 -
    1809/1*kM.3 + 979/1*kM.4 + 118/1*kM.5 - 60/1*kM.6 -
    4/1*kM.7 + 3/1*kM.8)*$.1 - 208769062021/1*kM.1 +
    51146604497/1*kM.2 - 30878218588/1*kM.3 +
    50063809507/1*kM.4 - 52067647419/1*kM.5 -
    94281823910/1*kM.6 + 69906801827/1*kM.7 -
    182364865509/1*kM.8 + 214706745867/1*kM.9
> g, r, p:= GaloisGroup(f);
> TransitiveGroupIdentification(g);
5 8
```

Since g is derived from a factor of the original f, the Galois group should be isomorphic to a subgroup of G:

```
> Subgroups(G:OrderEqual := #g);
Conjugacy classes of subgroups
------------------------------
[1]     Order 8            Length 9
        Permutation group acting on a set of
        cardinality 9
        Order = 8 = 2^3
            (2, 4, 5, 3)(6, 8, 7, 9)
            (2, 7, 5, 6)(3, 9, 4, 8)
            (2, 5)(3, 4)(6, 7)(8, 9)
> IsIsomorphic(g, $1[1]`subgroup);
true Homomorphism of GrpPerm: g, Degree 8, Order 2^3
into GrpPerm: $, Degree 9, Order 2^3 induced by
    (1, 2, 3, 8)(4, 5, 6, 7) |--> (2, 4, 5, 3)(6, 8, 7, 9)
    (1, 7, 3, 5)(2, 6, 8, 4) |--> (2, 7, 5, 6)(3, 9, 4, 8)
```

38.2.1 Straight-line Polynomials

One of the most important tools in the computational Galois theory are invariants, that is multivariate polynomials that are invariant under some permutation group. While invariant theory in general is a rich and classical branch of mathematics, and is supported by a powerful magma module, Chapter 110, the more specific needs in the Galois theory are best met with a different set of functions. Invariants, in this chapter are multivariate polynomials in straight-line representation, the polynomials are represented as programs without branches. The category of this polynomials is of type `RngSLPol` and its elements are of type `RngSLPolElt`. A consequence of this representation is that certain operations are very fast, while others are impossible - or at least very difficult. For example, representing $(a-b)^{1000}(a+b)^{1000} - (a^2-b^2)^{1000}$ is trivial, this is a short program with just a few steps:

1 subtract b from a

2 raise to the 1000th power

3 add a and b

4 raise to the 1000th power

5 multiply the results of steps 2 and 4

6 subtract b^2 from a^2 and raise to the 1000th power

7 subtract the result of step 7 from 5

Now, while it is trivial to evaluate this polynomial at, for example, any pair of elements in any finite ring, it is very difficult to see that, in fact, the polynomial is identical to zero – when expanded as a polynomial.

`SLPolynomialRing(R, n)`

`Global` BoolElt *Default :* `false`

Creates the ring of multivariate straight-line polynomials over the ring R with n indeterminates.

`Name(R, i)`

`R . i`

Return the ith indeterminate of the SL-polynomial ring R.

`BaseRing(R)`

`CoefficientRing(R)`

Return the coefficient ring of R.

`Rank(R)`

Return the rank of the SL-polynomial ring, ie the number of independent indeterminates over the coefficient ring.

`SetEvaluationComparison(R, F, n)`

For a SL-polynomial ring R, prepare "probabilistic" comparison of straight-line polynomials, using evaluation at n tuples drawn at random from the finite field F.

In order to allow a probabilistic test for "equality" of SL-polynomials in places where a strict, deterministic test is not necessary, this allows to compare SL-polynomials through their values at random evaluation points.

`GetEvaluationComparison(R)`

Return the finite field and the number of random samples used to compare polynomials. If `SetEvaluationComparison` has not been called, the 1st return value will be `false` while the second is undefined.

`x * y` `x + y` `x - y` `- x`

`Derivative(x, i)`

The ith partial derivative of the SL-polynomial x.

38.2.2 Invariants

At the core of the computation of Galois groups is the single Stauduhar step where, for a group G and a (maximal) subgroup U the programme decides if the Galois group is a subgroup of U - provided it was contained in G. This is achieved by evaluating a G-relative U-invariant polynomial $f \in \mathbf{Z}[x_1, \ldots, x_n]$ (or $f \in \mathbf{F}_q[t][x_1, \ldots, x_n]$ when the characteristic of the coefficient ring of the input polynomial is prime). In this subsection several functions are collected that allow a user to access MAGMA's internally used invariants. In what follows, an invariant is always a multivariate polynomial f in n indeterminates where n is the degree of G i.e. $G < S_n$. Invariants are represented as straight-line polynomials that allow the very compact representation and fast evaluation of polynomials.

`GaloisGroupInvariant(G, H)`

`DoCost`	BOOLELT	*Default :* `false`
`Worklevel`	RNGINTELT	*Default :* -1
Verbose	`GaloisGroup`	*Maximum :* 3

For subgroups $H < G$ of the symmetric group on n elements, where H is maximal in G and G is transitive, compute a G-relative H-invariant. This is done by carefully comparing certain group theoretical properties of the group pair in question to find invariant polynomials of special types that are easy to evaluate. If this fails, generic invariants will be used.

If `DoCost` is `true`, two values are returned: the first return value in this case is an estimate for the number of multiplications necessary to evaluate the invariant, while the second value is a function that can be evaluated without arguments to compute the invariant. This is done to allow to compare invariants by their computational complexity before actually committing and computing them explicitly as this can be very time consuming.

If `Worklevel` is set to an integer different from -1 only certain types of invariants are tested for suitability for this particular pair of groups. In this case a special return value of `false` indicates that MAGMA was unable to find an invariant at this level. Roughly speaking, the higher the `Worklevel`, the more time-consuming the invariant will be, both in terms of the time spend in finding as well as the time necessary to evaluate the invariant.

`RelativeInvariant(G, H)`

`IsMaximal`	BOOLELT	*Default :* `false`
`Risk`	BOOLELT	*Default :* `false`
Verbose	`Invariant`	*Maximum :* 3

For a pair of subgroups $H < G$ of the symmetric group where H is not necessarily maximal in G, find a G-relative H-invariant polynomial. The computation splits into three phases:

- First, a subgroup chain between H and G is computed such that each step in the chain is a maximal subgroup.

- Second, for each pair $U_i < U_{i+1}$ of maximal subgroups one fixed invariant is computed
- Third, in the last step, the invariants are combined to produce a G-relative H-invariant.

If `IsMaximal` is set to `true`, MAGMA will not compute a subgroup chain but instead assume that H is a maximal subgroup of G. If `Risk` is `true`, then MAGMA will use `GaloisGroupInvariant` to compute invariants on each level. By default, MAGMA will use generic invariants (ie. orbit sums of monomials) on each level. The problem with using special invariants as produced by `GaloisGroupInvariant` is that in the third step we can no longer guarantee that the invariant returned will be a true G-relative H-invariant as the G-stabilizer might be too large. On the other hand, the special invariants are much faster to evaluate and will give the desired result almost always.

`CombineInvariants(G, H1, H2, H3)`

Given a subgroup $G < S_n$ and three maximal subgroups H_1, H_2 and H_3 of G two of which have already known invariants, try to derive an invariant for H_3 from the known ones. The input for H_1 and H_2 consists of a tuple with two (or three) entries, the first specifying the actual subgroupm the second the G-relative H_i-invariant and the optional third a Tschirnhaus transformation that should be done before the invariant is evaluated.

They typical situation in which this function is used is the case of H_1 and H_2 being index 2 subgroups of G. In this case elementary theory immediately guarantees a third subgroup H_3 of index 2. For this function to work, the core of $H_1 \cap H_2$ must be contained in H_3. This is only useful if the index of the core is not too large.

`IsInvariant(F, p)`

`Sign` BOOLELT *Default :* `false`

For a multivariate polynomial F in straight-line representation and a permutation p this functions test if $F^p eq F$ with a high probability. In particular, this function will evaluate F at random elements in some large finite field, then permute the evaluation points by p and evaluate again. If the values agree, the polynomial is most likely invariant under p, if they disagree than the polynomial is definitely not invariant. The probability of failure is related to the probability of guessing a zero of a multivariate polynomial at random.

In order to get a proof for the invariants, one can convert F into a standard multivariate polynomial and check directly that this is invariant. However, for the invariants typically constructed in the Galois package, the conversion into a multivariate polynomial will not be possible due to the large degree of the polynomial and the resulting large number of terms.

If `Sign` is set to `true`, the function checks instead for $F^p = -F$.

`Bound(I, B)`

Given a multivariate polynomial I in straight-line representation and an integer B, compute an integer M such that

$$|I(x_1, \ldots, x_n)| \leq M$$

for all complex numbers $|x_i| \leq B$, it it returns a bound for the size of an evaluation of I.

`Bound(I, B)`

Given a multivariate polynomial I in straight-line representation and B, a power series over the integers, compute a power series M such that for all choices of power series x_i such that the coefficients of x_i are bounded in absolute value by those of B we have that the power series

$$I(x_1, \ldots, x_n)$$

has coefficients bounded by those of M.

38.2.3 Subfields and Subfield Towers

The result of a Galois group computation contains, in addition to the Galois group as an abstract group, the explicit action of the group on the roots of the underlying polynomial in some splitting field. This explicit action, together with the availability of invariants for group pairs, can be used to compute arbitrary subfields of the splitting field.

`GaloisSubgroup(K, U)`

`GaloisSubgroup(S, U)`

`GaloisSubgroup(f, U)`

Verbose `Invariant` *Maximum :* 2

Given either a polynomial f or number field K or a successful computation of a Galois group in S and a subgroup $U < G$ where G is the Galois group, find a defining polynomial for the subfield of the splitting field that is fixed by U.

`GaloisQuotient(K, Q)`

`GaloisQuotient(f, Q)`

`GaloisQuotient(S, Q)`

Verbose `Invariant` *Maximum :* 2

Given either a polynomial f or number field K or a successful computation of a Galois group in S and a permutation group Q, find all subfields of the splitting field that have a Galois group isomorphic to Q. This is done by finding all subgroups U of the Galois group G such that the permutation action of G on the cosets G/U is isomorphic to Q.

`GaloisSubfieldTower(S, L)`

`Risk`	BoolElt	*Default :* `false`
`MinBound`	RngIntElt	*Default :* 1
`MaxBound`	RngIntElt	*Default :* ∞
`Inv`	[RngSLPolElt]	*Default :* `false`
Verbose	`GaloisTower`	*Maximum :* 2

For data computed as the third return value of `GaloisGroup` and a subgroup chain $U_1 > U_2 > \ldots > U_s$, compute the corresponding tower of fixed fields K_i that is fixed by the operation of U_i on the roots of f as ordered in S.

Currently, this function only works for polynomials defined over $\mathbf{Q}$ or absolute extensions of $\mathbf{Q}$.

The first return value is the largest number field in the tower, that is the field fixed by the smallest group in the chain as an extension of the fixed field of the second group ..... The second return value is a sequence of tuples each containing the data used to generate one step:

- The first item is the invariant used in this step. This corresponds directly to the choice of the primitive element.
- The second item is the Tschirnhaus transformation on this level
- The third item is a transversal of U_i over U_{i+1}, the fixed ordering of which gives the ordering of the "relative conjugates"

The third and fourth return values can be used to algebraically identify arbitrary elements of the splitting field that are defined by multivariate polynomials. The third is a function that takes a vector of p-adic conjugates and returns an algebraic representation of the element, the fourth takes an invariant and computes precision bounds for the precision necessary so that the algebraic recognition will work.

If `Risk` is set to `true`, then for non-maximal subgroup pairs $U_i > U_{i+1}$ the "risky" version of `RelativeInvariant` is used.

The parameter `MinBound` can be used to specify a minimal p-adic precision that should be used internally. This can be used to avoid the calculation of an increase in precision which can be costly. On the other hand, to work in larger precision than necessary also incurs a time penalty.

The parameter `MaxBound` can be used to limit the p-adic precision used internally. Especially when the chain get longer, the internally used precision estimates become more and more pessimistic thus forcing higher and higher precision. In certain cases when it is possible to verify the correctness of the result independently, a smaller precision can speed the computation up considerably.

If the parameter `Inv` is given it should contain a sequence of invariants, the i-th entry need be an U_i relative U_{i+1} invariant. The invariants used correspond almost directly to the relative primitive elements computed at each step in the tower. This is useful in situation where either certain primitive elements are necessary or where certain invariants are known.

`GaloisSplittingField(f)`

`Galois`	TUP<GRPPERM, [RNGELT], GALOISDATA>	
`Roots`	BOOLELT	*Default :* `true`
`AllAuto`	BOOLELT	*Default :* `false`
`Stab`	BOOLELT	*Default :* `true`
`Chain`	[GRPPERM]	*Default :* `false`
`Inv`	[RNGSLPOLELT]	*Default :* `false`
`Name`	MONSTGELT	*Default :* `false`

For a polynomial f in $\mathbf{Z}[t]$, $\mathbf{Q}[t]$ or over an absolute number field this function computes the splitting field of f as a tower of fields. The various parameter can be used to force certain subfield towers and/ or compute additional data. By default `Stab` is set to `true`, which means that the splitting field will be the tower corresponding to the chain of stabilizers of $\{1\}$, $\{1,2\}$, ..., $\{1,\ldots,n\}$. Also by default `Roots` is set to `true`, which means that the roots of f are expressed as elements of the splitting field. If `Roots` is set to `false`, only the field is computed and returned.

The third return value will be the Galois group, the optional fourth value the automorphisms.

If the parameter `Galois` is used, it should contain a list or triplet containing the output of `GaloisGroup(f);`.

If `Chain` is set to a sequence of subgroups, this chain is used to compute a subfield tower. In this case the first elements must be G, the full Galois group. If `Chain` is used, `Inv` can be used to provide the invariants as well.

If `AllAuto` is set to `true`, the full automorphism group of the splitting field is computed as a sequence of sequences giving the all the roots of the relative polynomials.

If `Name` is given, it should be set to a string. In this case the primitive element of the i-subfield in the tower will be called `Name.i`.

Example H38E5

We start with a small example, to illustrate some of the parameters and their influence:

```
> P<x> := PolynomialRing(IntegerRing());
> f := x^3-2;
> GaloisSplittingField(f);
Number Field with defining polynomial $.1^2 +
    $.1*$.1 + $.1^2 over its ground field
[
    $.1,
    $.1,
    -$.1 - $.1
]
Symmetric group acting on a set of cardinality 3
Order = 6 = 2 * 3
```

```
    (1, 2, 3)
    (1, 2)
> K, R, G := $1;
> K:Maximal;
  K
  |
  |
  $1
  |
  |
  Q
K  : $.1^2 + $.1*$.1 + $.1^2
$1 : x^3 - 2
> [x^3 : x in R];
[
    2,
    2,
    2
]
```

The fact that by default all generators are called `$.1` makes this hard to read, so let us assign other names:

```
> GaloisSplittingField(f:Name := "K");
Number Field with defining polynomial K1^2 + K2*K1
    + K2^2 over its ground field
[
    K2,
    K1,
    -K1 - K2
]
Symmetric group G acting on a set of cardinality 3
Order = 6 = 2 * 3
    (1, 2, 3)
    (1, 2)
> (K where K := $1):Maximal;
  $1<K1>
    |
    |
  $2<K2>
    |
    |
    Q
$1 : K1^2 + K2*K1 + K2^2
$2 : x^3 - 2
```

So now we can easily see that the splitting field is a relative quadratic extension of the degree 3 stem field. Now we try a different subgroup chain:

```
> G, r, S := GaloisGroup(f);
```

```
> GaloisSplittingField(f:Galois := <G, r, S>,
>     Chain := CompositionSeries(G), Name := "K", AllAuto);
Number Field with defining polynomial K1^3 - 2
over its ground field
[
    K1,
    1/6*K2*K1,
    1/6*(-K2 - 6)*K1
]
Symmetric group G acting on a set of cardinality 3
Order = 6 = 2 * 3
    (1, 2, 3)
    (1, 2)
[
    [
        K2,
        -K2 - 6
    ],
    [
        K1,
        1/6*K2*K1,
        1/6*(-K2 - 6)*K1
    ]
]
> (K where K := $1):Maximal;
  $1<K1>
    |
    |
  $2<K2>
    |
    |
    Q
$1 : K1^3 - 2
$2 : x^2 + 6*x + 36

> f := x^10 - 20*x^8 + 149*x^6 - 519*x^4 + 851*x^2 - 529;
> G, r, S := GaloisGroup(f);G,r,S;
> TransitiveGroupIdentification(G);
8 10
> TransitiveGroupDescription(G);
[2^4]5
```

Thus the Galois group of f is isomorphic to ${}_{10}T_8$ of type `[2^4]5` and order 80.
We first compute the splitting field directly:

```
>  time _ := SplittingField(f);
```

This takes a long time, mainly because of the type of the Galois group which will require a field tower involving 5 steps and factorisation of a polynomial in such a tower. Now, we try the same

by using the Galois information:

```
> time K, R := GaloisSplittingField(f:Name := "K");
Time: 4.740
> K:Maximal;
  K<K1>
    |
    |
  $1<K2>
    |
    |
  $2<K3>
    |
    |
  $3<K4>
    |
    |
    Q
K  : K1^2 + 1/23*(-12*K4^8 + 217*K4^6 - 1374*K4^4
    + 3606*K4^2 - 3381)
$1 : K2^2 + 1/23*(18*K4^8 - 314*K4^6 + 1877*K4^4 -
    4512*K4^2 + 3588)
$2 : K3^2 + 1/23*(-5*K4^8 + 77*K4^6 - 377*K4^4 +
    663*K4^2 - 437)
$3 : x^10 - 20*x^8 + 149*x^6 - 519*x^4 + 851*x^2 -
    529
> [ Evaluate(f, x) eq 0 : x in R];
[ true, true, true, true, true, true, true, true,
true, true ]
```

From the type of the Galois group, `[2^4]5` we expect G to have a normal subgroup A of type C_2^4 such that the quotient G/A is a cyclic group of order 5. To find that subfield we can for example use the Galois computations again:

```
> A := NormalSubgroups(G:OrderEqual := 16)[1]`subgroup;
> GaloisSubgroup(S, A);
x^5 + 1682*x^4 + 715964*x^3 + 99797360*x^2 +
    5206504944*x + 88019915488
(x5 + x10)
```

The second return value $x_5 + x_{10}$ also tells us that the primitive element of the subfield is the sum of two roots of f, namly the 5-th and 10-th in our fixed ordering.
Suppose we want to work in the degree 16 extension over this field, that is we want to work in the fixed field of the trivial subgroup over the field fixed by A:

```
> K, D, Reco, Bnd := GaloisSubfieldTower(S, [A, sub<G|>]);
>  GK := GaloisGroup(K);
>  #GK;
16
>  AbelianInvariant(GK);
```

```
[ 2, 2, 2, 2 ]
```

As an abstract field, K as the splitting field can be described as a quotient of $\mathbf{Q}[x_1, \ldots, x_{10}]/I$ for some suitable ideal I also known as the Galois ideal. On the other hand, by tensoring with some p-adic complection $\mathbf{Q}_p$ we get an embedding of K into $K_p^{\#G} =: \Gamma$. The sequence D that is returned as the 2nd value contains the information necessary to map elements in $\mathbf{Z}[x_1, \ldots, x_{10}]$ via Γ to K. Suppose we want to find x_1, ie. a root of f in K. We first have to get the p-adic image in Γ, with appropriate precision. Step one is to define a suitable multivariate polynomial i that will represent x_1.
The second step is to compute an integer B such that all complex conjugates if i are bounded by B in absolute value. For this we can use information about the size of the roots of f stored in S. The next step now is to get a bound for the p-adic precision.

```
> R := SLPolynomialRing(Integers(), 10);
> i := R.1;
> B := S‘max_comp;
> bound := Bnd(B);
```

Now, we need to get the p-adic conjugates of x_1, ie. the image in Γ. The third entry in each of the elements of D contains coset representatives that give the relative conjugates:

```
> rt := [GaloisRoot(i, S:Bound := bound) : i in [1..10]];
> con := CartesianProduct(Reverse([x[3]: x in D]));
> gamma := [Evaluate(i, PermuteSequence(rt, &*p)) : p in con];
> im := Reco(gamma: Bound := B);
> time Evaluate(f, im) eq 0;
```

Before we try to find an automorphism of the base field using this method we want to find the primitive element of the base field of K. The primitive element is essentially given by the 1st part of D. Note that here a Tschirnhaus-tranformation was necessary.

```
> i := D[1][1]; t := D[1][2];
> B := Bound(i, Evaluate(t, S‘max_comp));
> bound := Bnd(B);
> rt := [GaloisRoot(i, S:Bound := bound) : i in [1..10]];
> rt := [Evaluate(t, x) : x in rt];
> gamma := [Evaluate(i, PermuteSequence(rt, &*p)) : p in con];
> im := Reco(gamma : Bound := B);
> im;
$.1
> im eq K.2;
true;
```

Now for the automorphism - all we have to change is to permute the roots as we already have the permutation group.

```
> rt := PermuteSequence(rt, Random(G));
> gamma := [Evaluate(i, PermuteSequence(rt, &*p)) : p in con];
> au := Reco(gamma : Bound := B);
> au;
```

```
1/92*(-9*$.1^4 + 386*$.1^3 - 5854*$.1^2 + 37120*$.1 - 82288)
```

In order to "find" arbitrary (integral) elements this way one has to

- define the element as a multivariate polynomial in the roots, i
- with the aid of `Bound` and the knowledge of the complex roots of f, find a bound B of the complex embeddings of i and use `Bnd` as above to find a bound M on the p-adic precision
- use the information in D to compute i in Γ, ie all p-adic conjugates in the "correct" ordering
- use `Reco` to find the algebraic representation.

38.2.4 Solvability by Radicals

For a polynomial $f \in \mathbf{Z}[t]$ with solvable Galois group it is well known that the roots of f can be expressed as nested radicals. On the other hand no good algorithm is known to achieve this. Here we use the explicit action of the Galois group of f as a permutation group on the p-adic roots to compute such an representation.

`SolveByRadicals(f)`

`Prime`	RngIntElt	*Default :* `false`
`Name`	MonStgElt	*Default :* `false`
`Galois`	Tup<GrpPerm, [RngElt], GaloisData>	
`UseZeta_p`	BoolElt	*Default :* `false`
`MaxBound`	RngIntElt	*Default :* ∞
Verbose	`GaloisTower`	*Maximum :* 3

For a polynomial $f \in \mathbf{Z}[t]$ with solvable Galois group, a splitting field as a tower of radical extensions is computed together with algebraic representations of the roots of f as elements in the splitting field. The third return value contains the non-trivial roots of unity which are used.

If the parameter `Galois` is used, it should contain a list or triplet containing the output of `GaloisGroup(f);`.

If `Prime` is used, and `Galois` is unspecified, the value of `Prime` is passed onto the Galois group computation and can therefore be used to choose the p-adic field.

If `UseZeta_p` is set to `true`, then the expression for the roots of p will contain pure radicals and roots of unity. By default, if `UseZeta_p` is `false`, radical expressions for the roots of unit necessary will also be computed.

If `MaxBound` is given, it will be used as an upper bound for the p-adic precision used internally. Expecially when the radical tower contains many steps, the internally used precision estimates become more and more pessimistic, thus resulting in larger and larger precision.

If `Name` is set to some string, the i-th level primitive element in the tower will be called `Name.i`.

CyclicToRadical(K, a, z)

Let K/k be a number field with cyclic automorphism group of order n generated by $K.1 \to a$ and z be a n-th root of unity in k. This function will return a field L isomorphic to K such that L is a Kummer extension, ie. the defining polynomial for L will be of the form $t^n - b$ for some b in the coefficient field k of K. The second returned value contains the roots of f in L while the third return value contains the roots of unity used.

Example H38E6

```
> P<x> := PolynomialRing(IntegerRing());
> f := x^6 - x^5 - 6*x^4 + 7*x^3 + 4*x^2 - 5*x + 1;
> K, R := SolveByRadicals(f:Name := "K.");
> K:Maximal;
   K<K.1>
      |
      |
  $1<K.2>
      |
      |
  $2<K.3>
      |
      |
  $3<K.4>
      |
      |
      Q
K  : K.1^3 + 1/2*(3*K.4 - 11)*K.2 + 1/2*(-27*K.4 + 23)
$1 : K.2^2 - 5
$2 : K.3^3 - 228*K.4 + 532
$3 : K.4^2 + 3
> [ Evaluate(f, x) eq 0 : x in R];
[ true, true, true, true, true, true ]
```

Note that every step in the tower defining K is radical, ie. given by an equation of type $x^n - a$.

38.2.5 Linear Relations

An important question for various problems is that of finding all linear (additive) relations between the roots of some integral polynomial. While there is a obvious algorithm if the splitting field can be constructed explicitly, there is no obvious way of doing it in general. In this section we provide two algorithms to find those and more general relations and a third that can verify arbitrary relations.

`LinearRelations(f)`

`Proof`	BoolElt	*Default :* `true`
`Galois`	Tup<GrpPerm, [RngElt], GaloisData>	
`UseAction`	BoolElt	*Default :* `false`
`UseLLL`	BoolElt	*Default :* `true`
`Power`	RngIntElt	*Default :* 1
`kMax`	RngIntElt	*Default :* ∞
`LogLambdaMax`	RngIntElt	*Default :* ∞

Given an integral monic polynomial f, this function finds a basis for the module of additive relations between the roots of f in some algebraic closure. The ordering of the roots is the same as chosen by the computation of the Galois group of f, in fact, the roots used are precisely the ones returned by `GaloisRoot`. The output consists of a basis for the relation module encoded in a matrix and the Galois data encoding the ordering of the roots. The algorithm is described in [dGF07].

If `Power` is set to an integer larger than one, the module of relations between the powers of the roots is computed.

`LinearRelations(f, I)`

`Proof`	BoolElt	*Default :* `true`
`Galois`	Tup<GrpPerm, [RngElt], GaloisData>	
`UseAction`	BoolElt	*Default :* `false`
`UseLLL`	BoolElt	*Default :* `true`
`Power`	RngIntElt	*Default :* 1
`kMax`	RngIntElt	*Default :* ∞
`LogLambdaMax`	RngIntElt	*Default :* ∞

Let f be an integral monic polynomial and α_1, ..., α_n be the roots of f in some splitting field in a fixed ordering. The field and the ordering used here are the ones chosen by the computation of the Galois group of f. The splitting field K of f be represented as a quotient $\mathbf{Q}[x_1, \ldots, x_n]/J$ for some suitable ideal J, thus elements in K can be represented as multivariate polynomials in the roots α_i. The sequence I that is passed into this function is interpreted to contain elements in K given via the polynomials in I. This function computes a basis for the module of relations between the elements represented by I. The algorithm is described in [dGF07].

`VerifyRelation(f, F)`

`Galois`	Tup<GrpPerm, [RngElt], GaloisData>	
`kMax`	RngIntElt	*Default :* ∞

Let f be an integral monic polynomial and α_1, ..., α_n be the roots of f in some splitting field in a fixed ordering. The field and the ordering used here are the ones chosen by the computation of the Galois group of f. The splitting field K of f be

represented as a quotient $\mathbf{Q}[x_1, \ldots, x_n]/J$ for some suitable ideal J, thus elements in K can be represented as multivariate polynomials in the roots α_i. For a polynomial F in the roots of f, this function verifies if F evaluated at the roots of f equals zero, ie. if F describes a relation between the roots. The algorithm is described in [dGF07].

Example H38E7

The following example originates in a paper [BDE$^+$] where, among other things, polynomials are constructed whose roots have a maximal number of linear dependencies. Note, that the polynomial constructed here is not extremal.

```
> G := ShephardTodd(8);
> R := InvariantRing(G);
> p := PrimaryInvariants(R);
> p;
[
    x1^8 + (-4*i - 4)*x1^7*x2 + 14*i*x1^6*x2^2 + (-14*i +
        14)*x1^5*x2^3 - 21*x1^4*x2^4 + (14*i + 14)*x1^3*x2^5
        - 14*i*x1^2*x2^6 + (4*i - 4)*x1*x2^7 + x2^8,
    x1^12 + (-6*i - 6)*x1^11*x2 + 33*i*x1^10*x2^2 + (-55*i +
        55)*x1^9*x2^3 - 231/2*x1^8*x2^4 + (66*i +
        66)*x1^7*x2^5 + (-66*i + 66)*x1^5*x2^7 -
        231/2*x1^4*x2^8 + (55*i + 55)*x1^3*x2^9 -
        33*i*x1^2*x2^10 + (6*i - 6)*x1*x2^11 + x2^12
]
> f := Resultant(p[1]-2, p[2]-3, 2);
> f := Polynomial(Rationals(), UnivariatePolynomial(f));
> IsPower(f, 4);
true 27/4*x^24 - 135*x^16 + 405*x^12 - 405*x^8 + 162*x^4 - 1
> _, f := $1;
> GG, R, S := GaloisGroup(f);
> #GG;
192
> rel := LinearRelations(f:Galois := <GG, R, S>);
> #rel;
20
> rel[3];
[ 0, 0, 0, -1, 0, -1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
> IR := SLPolynomialRing(Integers(), 24);
> r := &+ [rel[3][i]*IR.i : i in [1..24]];
> r;
(x7 + ((-1 * x4) + (-1 * x6)))
> VerifyRelation(f, r:Galois := <GG, R, S>);
true
```

Now we use this to construct a field of degree 192 over Q where the conjugates of a primitive element span only a 4-dimensional vectorspace. We first define an invariant (a multivariate poly-

nomial) that corresponds to a primitive element. This can be verified by computing all p-adic conjugates and checking that they are pairwise different.

```
> i := &+ [ i*IR.i : i in [1..23]];
> I := [Apply(g, i) : g in GG];
> #{Evaluate(i, R) : i in I};
192
```

So, since $i = \sum i\alpha_i$ is a linear combination of the roots α_i of f, the dimension of the vectorspace spanned by the conjugates of i is 4 again. Note that this is a property of the polynomial not of the field.

```
> i := RelativeInvariant(GG, sub<GG|>);
> I := [Apply(g, i) : g in GG];
> #{Evaluate(i, R) : i in I};
192
>  #LinearRelations(f, I:Galois := <GG, R, S>, Proof := false);
```

38.2.6 Other

`ConjugatesToPowerSums(I)`

For elements in a sequence I, compute the sequence containing the power sums $\sum I_i^j$ for $j = 1, \ldots, \#I$. If I is interpreted to contain the Galois conjugates of some algebraic number (or the roots of some polynomial) then this computed the power sums.

`PowerSumToElementarySymmetric(I)`

Given a sequence I of elements, interpreted as power sums of some algebraic number x, use Newton's relations to compute the elementary symmetric functions in the conjugates of x. In general for this to succeed, the characteristic of the underlying ring needs to be larger than the length of the sequence.

38.3 Subfields

This section contains functions for the computation of all subfields of any number field or all subfields of a given degree of a simple absolute algebraic field or a simple relative extension.

These computations are independent of the computation of the Galois groups, but similiarly there is no limit on the degrees of the field.

The algorithms used are Klüner's method as presented in [Klü95, Klü97, KP97, Klü98] and the newer method of Klüners, van Hoeij and Novocin [vHKN11].

`Subfields(K, n)`

Verbose	`Subfields`	*Maximum :* 4

Given a simple absolute algebraic field or a simple relative extension K and an integer n greater than 1, this function returns a sequence of pairs (2-tuples) containing the subfields of K of degree n together with the embedding homomorphisms of each subfield into K. It is possible that the sequence contains isomorphic fields, but the embeddings will be distinct in such a situation.

`Subfields(K)`

`Al`	MonStgElt	*Default :* "*Default*"
`Current`	BoolElt	*Default :* `false`
`Proof`	RngIntElt	*Default :* 1
Verbose	`Subfields`	*Maximum :* 4

Given an algebraic field K, this function returns a sequence of pairs (2-tuples) containing the subfields of K (except $\mathbf{Q}$) together with the embedding homomorphisms of each subfield into K. It is possible that the sequence contains isomorphic fields.

For fields K which are extensions of an algebraic number field the more recent algorithm developed by Klüners and van Hoeij [vHKN11] is used. For fields K which are extensions of $\mathbf{Q}$, this algorithm can be selected by setting the parameter `Al` to `"KluenersvanHoeij"` or may be chosen by `"Default"` as the optimal algorithm. By default, the `"Klueners"` algorithm is chosen if the defining polynomial of K factors into large degree factors (with respect to the degree of K) over the residue field of some prime or the coefficients of the defining polynomial are large, otherwise `"KluenersvanHoeij"` is used which is optimal when the defining polynomial only has small degree factors (with respect to the degree of K) over the residue fields of some number of primes, that is, K has a large number of subfields.

38.3.1 The Subfield Lattice

Subfields of number fields can also be retrieved in the form of a lattice from which additional information can be discovered.

`SubfieldLattice(K)`

Verbose	`Subfields`	*Maximum :* 4

The lattice of subfields of an absolute number field K.

`#L`

The number of fields in the lattice L.

`Representative(L)` `Rep(L)`

`Bottom(L)`

The bottom element of the subfield lattice L (this corresponds to $\mathbf{Q}$).

`Top(L)`

The top element of the subfield lattice L (this corresponds to the original number field).

`Random(L)`

A random element of the subfield lattice L.

`L ! n`
`L[n]`

The n-th element of the subfield lattice L.

`NumberField(e)`

The number field corresponding to the given subfield lattice element e.

`EmbeddingMap(e)`

The mapping from `NumberField(e)` into the top number field of the subfield lattice.

`Degree(e)`

The (absolute) degree of the number field corresponding to the subfield lattice element e.

`e eq f`

Returns `true` if and only if the subfield lattice elements e and f are equal.

`e subset f`

Returns `true` f and only if e is a subfield of f.

`e * f`

The smallest field containing both e and f.

`e meet f`

The intersection of e and f. This is the largest field common to both of them.

`&meetS`

The intersection of the subfields in the sequence S.

`MaximalSubfields(e)`

The sequence of maximal subfield lattice elements contained in e.

`MinimalOverfields(e)`

The sequence of minimal subfield lattice elements containing e.

Example H38E8

A subfield lattice is shown.

```
> Zx<x> := PolynomialRing(Integers());
> K<a> := NumberField(x^8 - x^4 + 1);
> L := SubfieldLattice(K);
> L;
Subfield Lattice of K
[1] Rational Field
[2] Subfield generated by a root of x^2 - 4*x + 1
[3] Subfield generated by a root of x^2 - 26*x + 241
[4] Subfield generated by a root of x^2 - 10*x + 241
[5] Subfield generated by a root of x^2 + 1
[6] Subfield generated by a root of x^2 - 10*x + 1
[7] Subfield generated by a root of x^2 - 6*x + 1
[8] Subfield generated by a root of x^2 + x + 1
[9] Subfield generated by a root of x^4 + 4*x^2 + 1
[10] Subfield generated by a root of x^4 - x^2 + 1
[11] Subfield generated by a root of x^4 - 8*x^3 + 20*x^2 - 16*x + 1
[12] Subfield generated by a root of x^4 - 4*x^3 + 8*x^2 - 4*x + 1
[13] Subfield generated by a root of x^4 - 6*x^3 + 13*x^2 - 6*x + 1
[14] Subfield generated by a root of x^4 - 4*x^3 + 8*x^2 + 4*x + 1
[15] Subfield generated by a root of x^4 - 2*x^3 + 5*x^2 + 2*x + 1
[16] Subfield generated by a root of x^8 - x^4 + 1
```

Observe that subfields 2, 5 and 8 give the square roots of 2, -1 and 3, respectively. Since these are incommensurate radicals the field generated by them has degree 8 and so must be isomorphic to K.

```
> K2 := AbsoluteField( NumberField([ x^2 + 1, x^2 - 2, x^2 - 3 ]));
> K2;
Number Field with defining polynomial x^8 - 16*x^6 + 88*x^4 + 192*x^2 + 144
over the Rational Field
> IsIsomorphic(K2, K);
true
```

In fact, K is just a "better" version of K2.

```
> OptimizedRepresentation(K2);
Number Field with defining polynomial x^8 - x^4 + 1 over the Rational Field
Mapping from: FldNum: K2 to Number Field with defining polynomial x^8 - x^4 + 1
over the Rational Field
```

38.4 Galois Cohomology

MAGMA has some rudimentary functions to aid computations in Galois cohomology of number fields.

`Hilbert90(a, M)`

S	[RngOrdIdl]	*Default :* `false`

Let K be a number field and $M : K \to K$ be an automorphism of K furthermore, denote by k the fixed field of M, thus M generates the automorphism group of the relative cyclic extension K/k. For some element a in K, such that $N_{K/k}(a) = 1$, this function will find some element b such that $a = b/M(b)$. If `S` is given it should contain a sequence of prime ideals such that there exists some b in the S-unit group over S.

`SUnitCohomologyProcess(S, U)`

ClassGroup	BoolElt	*Default :* `false`
Ramification	BoolElt	*Default :* `false`

Let k be a normal number field with (abstract) automorphism group G. For a set of prime ideals S of k, which is closed under the action of the subgroup U of G, a process is created that allows working with the cohomology of the multiplicative group of k - partially represented by a group of S-units. If `ClassGroup` is given, the set S is enlarged to support the current generators of the class group. If `Ramification` is present, then all ramified primes are also included in S.

During the computations with this object the set S can be increased to allow the representation of a larger number of elements.

`IsGloballySplit(C, l)`

Sub	GrpPerm	*Default :* `false`
Verbose	Cohomology	*Maximum :* 2

For a cohomology process C as created by `SUnitCohomologyProcess` and a 2-cocyle $l : U \times U \to k$ given as a MAGMA-function, decide if l is split, ie. if there exists a 1-cochain $m : U \to k$ such that $\delta m = l$ for the cohomological coboundary map δ. If `Sub` is given it has to be a subgroup of the automorphism group of the number field underlying the cohomology process, otherwise the full automorphism group is used. This allows to restrict a cocyle easily.

As a fixed cocyle l assumes only finitely many values, we can consider it as a cocyle with values in some suitable S-unit group. Similarly, it is exists, m also has values in some S'-unit group for a potentially larger set S'. This function first tries to "remove" ideals from the support of l, to make the set S as small as possible. Then the set is enlarged to make sure that m, if exists, can be found with values in the $S' = S$-unit group. Since the final problem now involves only finitely generated abelian groups, it can be solved by MAGMA's general cohomology machinery.

`IsSplitAsIdealAt(I, l)`

`Sub` GrpPerm *Default :* `false`

Let U be a subgroup of the automorphism group G of some number field k, $l : U \times U \to k^*$ a 2-cocyle and I some ideal in k. If `Sub` is given, U is taken to be `Sub`, otherwise $U := G$. Assuming that each element $l(u, v)$ has a valuation at all ideals in the U-orbit of I, ie. we have a unique decomposition of ideals $l(u, v) = J^{x(u,v)}A(u, v)$ for integers $x(u, v)$ and ideals $A(u, v)$ coprime to J for all J in I^U. Then we can use l to define a cocyle with values in I^U which is a finitely generated group. This function determines if this cocycle splits, and if so, computes a 1-cochain with values in I^U for some fixed ordering of I^U. The cochain and I^U are returned on success.

38.5 Bibliography

[**AK99**] Vincenzo Acciaro and Jürgen Klüners. Computing Automorphisms of Abelian Number Fields. *Math. Comp.*, 68(227):1179–1186, 1999.

[**BDE**$^+$] Neil Berry, Arturas Dubickas, Noam Elkies, Bjorn Poonen, and Chris Smyth. The conjugate dimension of algebraic numbers. *Quart. J. Math.*, 55:237–252.

[**dGF07**] Willem de Graaf and Claus Fieker. Finding integral linear dependencies of algebraic numbers and algebraic Lie algebras. *LMS Journal of Computation and Mathematics*, 11, 2007.

[**FK12**] C. Fieker and J. Klüners. Computational Galois Theory I: Invariants and Computations over **Q**. submitted, http://arxiv.org/abs/1211.3588, 2012.

[**Gei03**] Katharina Geißler. *Berechnung von Galoisgruppen über Zahl- und Funktionenkörpern*. PhD Thesis, TU-Berlin, 2003. available at URL:http://www.math.tu-berlin.de/~kant/publications/diss/geissler.pdf.

[**GK00**] Katharina Geißler and Jürgen Klüners. The determination of Galois Groups. *J. Symbolic Comp.*, 30(6):653–674, 2000.

[**Klü95**] Jürgen Klüners. Über die Berechnung von Teilkörpern algebraischer Zahlkörper. Diplomarbeit, Technische Universität Berlin, 1995. URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/klueners.ps.gz.

[**Klü97**] Jürgen Klüners. *Über die Berechnung von Automorphismen und Teilkörpern algebraischer Zahlkörper*. Dissertation, Technische Universität Berlin, 1997. URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_jk.ps.gz.

[**Klü98**] Jürgen Klüners. On computing subfields. A detailed description of the algorithm. *J. Theor. Nombres Bordx.*, 2(10):243–271, 1998.

[**KP97**] Jürgen Klüners and Michael E. Pohst. On Computing Subfields. *J. Symbolic Comp.*, 24(3):385–397, 1997.

[**SM85**] Leonhard H. Soicher and John McKay. Computing Galois Groups over the rationals. *J. Number Th.*, 20:273–281, 1985.

[**Sta73**] Richard P. Stauduhar. The determination of Galois Groups. *Math. Comp.*, 27:981–996, 1973.

[**vHKN11**] M. van Hoeij, J. Klüners, and A. Novocin. Generating Subfields. In Anton Leykin, editor, *Proceedings ISSAC 2011*, 2011.

39 CLASS FIELD THEORY

Chapter 39
CLASS FIELD THEORY

39.1 Introduction

This chapter presents the facilities provided in MAGMA for class field theory. The main objects of interest are abelian extensions of number fields (`FldAb`) and maps between abelian groups and ideal groups.

Class field theory is concerned with the classification of all abelian extensions of a given field. In particular, this covers abelian extensions of number fields, local fields and global function fields. While this chapter deals with the number field case only, MAGMA can also perform computations in the other cases. For the case of global function fields, see Chapter 43 and for the case of p-adic local fields, Section 47.14.

Abstractly, class field theory parametrizes abelian extensions in terms of abelian groups defined with respect to the base field. In MAGMA, ray class groups and their quotients are used to define the extensions. Ray class groups and their quotients are always represented as maps between a finite abelian group (`GrpAb`) and the power structure of ideals (`PowIdeal`). The maps are usually obtained as products of the map returned by ray class groups and quotient maps.

In theory, a class field is completely determined by the corresponding class group (map). Currently there is only a small number of invariants that can be computed directly from the map; for most other properties the class field has to be converted into a number field by computing a set of defining equations.

39.1.1 Overview

Class field theory classifies all abelian extensions of a given number field k in terms of quotients of ray class groups. Ray class groups should be thought of as generalized class groups in that they can be defined similarly to class groups:

$$1 \to U \to k^* \to I \to \mathrm{Cl} \to 1$$

where k^* is the multiplicative group of k, U is the group of units, and I the group of fractional ideals. (Recall, the class group is defined as the quotient of the ideals modulo the principal ideals). To define ray class groups we need to refine all of the above terms. Let $o = \mathbf{Z}_k$ be the ring of integers in k and fix an integral ideal $\mathfrak{m}_0$ in o. Furthermore, let $\mathfrak{m}_\infty$ be a subset of the real embeddings of k into $\mathbf{C}$ and, formally, $\mathfrak{m} := (\mathfrak{m}_0, \mathfrak{m}_\infty)$. Then for $x \in k$ define

$$x \bmod^* \mathfrak{m} = 1 \quad \text{iff} \begin{cases} v_{\mathfrak{p}}(x-1) \geq v_{\mathfrak{p}}(\mathfrak{m}_0) & \text{for all prime ideals } \mathfrak{p} \\ s(x) > 0 & \text{for } s \in \mathfrak{m}_\infty \end{cases}$$

Let $I^{\mathfrak{m}}$ denote the group of fractional ideals that are coprime to $\mathfrak{m}_0$, $U_{\mathfrak{m}}$ the units u such that $u \bmod^* \mathfrak{m} = 1$ and $P_{\mathfrak{m}}$ the elements $x \in k$ such that $x \bmod^* \mathfrak{m} = 1$. Then

$$1 \to U_{\mathfrak{m}} \to P_{\mathfrak{m}} \to I^{\mathfrak{m}} \to \mathrm{Cl}_{\mathfrak{m}} \to 1$$

defines the ray class groups modulo $\mathfrak{m}$. For $\mathfrak{m} = (1o, \emptyset)$, this corresponds to the usual class group. Given two moduli $\mathfrak{m}$ and $\mathfrak{n}$ such that $\mathfrak{m}_0 | \mathfrak{n}_0$ and $\mathfrak{m}_\infty \subseteq \mathfrak{n}_\infty$ we have canonical surjections

$$\mathrm{Cl}_{\mathfrak{n}} \to \mathrm{Cl}_{\mathfrak{m}}\,.$$

Now, let H be a subgroup such that $P_{\mathfrak{m}} < H < I^{\mathfrak{m}}$. There exists a minimal $\mathfrak{n}$ such that the canonical embedding $\mathrm{Cl}_{\mathfrak{m}}/H \to \mathrm{Cl}_{\mathfrak{n}}$ is injective. This $\mathfrak{n}$ is called the conductor of H.

The main theorem of class field theory asserts the following correspondence between abelian extensions K/k of k and quotients of ray class groups: Let K/k be abelian and $G := \mathrm{Gal}(K/k)$ the group of k-automorphisms of K. For each prime ideal $\mathfrak{p}$ of k that is unramified in K, let $\mathrm{Frob}(\mathfrak{p}, K/k)$ be the Frobenius automorphism, i.e. the unique $s \in G$ such that $s(x) = x^N \bmod \mathfrak{p}$ for all $x \in k$. Extend this map multiplicatively to all ideals coprime to $d_{K/k}$, the discriminant of K/k. This is known as the Artin-map and is denoted by $(b, K/k)$. Class field theory asserts the existence of some modulus $\mathfrak{m}$ and a subgroup H such that $G \cong \mathrm{Cl}_{\mathfrak{m}}/H$ by the Artin-map.

Conversely, for each ray class group $\mathrm{Cl}_{\mathfrak{m}}$ and each subgroup $P_{\mathfrak{m}} < H$ there is an abelian extension K/k with the above property.

The correspondence is one-to-one if one restricts to pairs $\mathrm{Cl}_{\mathfrak{m}}$ and H such that $\mathfrak{m}$ is the conductor of H.

For $\mathfrak{m} = (1o, \emptyset)$, the corresponding field K is called the Hilbert class field of k (in the wide sense).

Perhaps best understood is the Hilbert class field. As conjectured by Hilbert and proved by Furtwängler, all ideals of k become principal in the Hilbert class field H_k. Futhermore, H_k is the maximal unramified abelian extension of k.

A different interpretation of H is provided by norm groups. We have

$$H = \langle N_{K/k}(b) | b \in I_K^{\mathfrak{m}} \rangle$$

This result may be used to convert an abelian number field into an abelian extension. In addition, class field theory asserts that, for an arbitrary normal extension K/k, the norm group defined above corresponds to the maximal abelian subfield.

39.1.2 Magma

In Magma maps are used to represent the ideal groups. If necessary, the map can be augmented by supplying $\mathfrak{m}$ i.e. an integral ideal $\mathfrak{m}_0$ and a sequence of indices of the real places in $\mathfrak{m}_\infty$. The map returned as a second return value from `ClassGroup` or `RayClassGroup` carries the necessary information to recover $\mathfrak{m}$, even if this is hidden from the user. As an example, we consider the Hilbert class field of $k := \mathbf{Q}(\alpha)$ with $\alpha^3 + \alpha^2 + 3\alpha - 6 = 0$ with class group C_4.

Example H39E1

```
> k := NumberField(Polynomial([ -6, 3, 1, 1 ]));
```

Now, the easiest way to get the Hilbert class field is to call `HilbertClassField` directly on k:

```
> K := HilbertClassField(k);
> K;
Number Field with defining polynomial $.1^4 + (76*k.1^2 - 420*k.1 - 488)*$.1^2\
 +    32080*k.1^2 + 41984*k.1 + 95168 over k
```

Let us now verify some of the properties, starting with the discriminant. Since K/k is totally unramified, the discriminant of the maximal order should be 1:

```
> O := MaximalOrder(K);
> O;
Maximal Order of Equation Order with defining polynomial x^4 + [-488, -420,
    76]*x^2 + [95168, 41984, 32080] over its ground order
> Discriminant(O);
Ideal
Basis:
[1 0 0]
[0 1 0]
[0 0 1]
```

Now let us check that all ideals of k are principal in K. In order to do so, it is sufficient to demonstrate that a generator of the class group becomes principal:

```
> g, m := ClassGroup(k);
> g;m;
Abelian Group isomorphic to Z/4
Defined on 1 generator
Relations:
    4*g.1 = 0
Mapping from: GrpAb: g to Set of ideals of Maximal Equation Order with
defining polynomial x^3 + x^2 + 3*x - 6 over its ground order
> i := m(g.1);
> I := O!!i;
> IsPrincipal(I);
true
> f,g := IsPrincipal(I);
> g;
[[2193497788678474035456, -1238066307883451022336,
-2319265120953032748288], [394272965034846395136,
-222653406238254306432, -417025104282566694144],
[167087257584, 71708840496, 32825469008], [495632919,
-279332235, -523526213]] / 10917386545536
```

However, to use the more sophisticated functions, the construction of the class fields needs to be done step–by–step. We first create an abelian extension as an object of type `FldAb`. Since we wish

to see how much of the class group becomes trivial in the quadratic subfield K_1 of K (capitulates in K_1), we also define this.

```
> aK := AbelianExtension(m);
> g, m := ClassGroup(k);
> q, mq := quo<g | 2*g.1>;
> m2 := Inverse(mq)*m;m2;
Mapping from: GrpAb: q to Set of ideals of Maximal
Equation Order with defining polynomial x^3 + x^2 + 3*x
    - 6 over Z
Composition of Mapping from: GrpAb: q to GrpAb: g and
Mapping from: GrpAb: g to Set of ideals of Maximal
Equation Order with defining polynomial x^3 + x^2 + 3*x
    - 6 over Z
> aK2 := AbelianExtension(m2);
```

This demonstrates a very important technique: the creation of a quotient group of the class group together with the corresponding map.
A few invariants such as degree and discriminant may be calculated directly from `aK` without the (costly) computation of a defining equation.

```
> Discriminant(aK);
Principal Ideal
Generator:
    [1, 0, 0]
[ 4, 4 ]
```

The second return value denotes the signature of the field as an extension of $\mathbf{Q}$.
Now we compute a defining equation for `aK2` and see what happens to the class group of k.

```
> O := MaximalOrder(aK2);O;
Maximal Order of Equation Order with defining
polynomial x^2 + [-5, -2, -1] over its ground order
> O!!m(g.1);
> IsPrincipal($1);
false
> IsPrincipal($2^2);
true
```

So only "half" of the class group capitulates in `aK2`, the reminding part collapses in `aK`.

39.2 Creation

The most powerful way to create class fields or abelian extensions in MAGMA is to use the AbelianExtension function that enables the user to create the extension corresponding to some ideal group.

So, before we can describe the creation functions for the class fields, we have to deal with the ideal groups.

39.2.1 Ray Class Groups

The classical approach to class field theory, which is well suited for computation, is based on *ideal groups* which are generalisations of the ideal class group.

In this section we describe in detail how to create *full* ideal groups, mainly ray class groups. As ray class groups are closely related to the unit groups of residue class rings of maximal order, these too are presented here.

In addition to the functions listed here, the CRT on page 946 is relevant in this context.

```
RayClassGroup(I)
```

```
RayClassGroup(I, T)
```

Given an integral ideal I belonging to the maximal order of a number field, the *ray class group modulo* I is the quotient of the subgroup generated by the ideals coprime to I by the subgroup generated by the principal ideals generated by elements congruent to 1 modulo I and T if present.

The sequence T contains the numbers $[i_1, \ldots, i_r]$ of certain real infinite places. When the sequence is supplied, the generators of the principal ideals must take positive values at the places indicated by T. The sequence T must be strictly ascending containing only positive integers, each not exceeding the number of real embeddings.

This function requires the class group to be known. If it is not already stored, it will be computed in such a way that its correctness is not guaranteed. However, it will almost always be correct. If the user requires a guaranteed result, then the class group must be verified by the user or computed up to the proof level required beforehand.

The ray class group is returned as an abelian group A, together with a mapping between A and a set of representatives for the ray classes.

The algorithm used is a mixture of Pauli's approach following Hasse ([Pau96, HPP97]) and Cohen's method ([CDO96, CDO97, Coh00]).

```
RayClassGroup(D)
```

Given a divisor (or place) of an absolute number field, compute the Ray class group defined modulo the divisor.

This function requires the class group to be known. If it is not already stored, it will be computed in such a way that its correctness is not guaranteed. However, it will almost always be correct. If the user requires a guaranteed result, then the

class group must be verified by the user or computed up to the proof level required beforehand.

The ray class group is returned as an abelian group A, together with a mapping between A and a set of representatives for the ray classes.

The algorithm used is a mixture of Pauli's approach following Hasse ([Pau96, HPP97]) and Cohen's method ([CDO96, CDO97, Coh00]).

Example H39E2

Some ray class groups are computed below. The example merely illustrates the fact that ray class groups tend to grow if their defining modul grows and can be arbitrarily large. This should be compared to class groups where it is rather difficult to give examples of fields having "large" class groups — unless one takes imaginary quadratic fields.

```
> R<x> := PolynomialRing(Integers());
> o := MaximalOrder(x^2-10);
> RayClassGroup(2*o, [1,2]);
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
Mapping from: Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0 to Set of ideals of o
> RayClassGroup(2*o);
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*$.1 = 0
Mapping from: Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*$.1 = 0 to Set of ideals of o
```

As one can see, the inclusion of the infinite places only added a C_2 factor to the group. In general, a set of infinite places containing n elements can at most add n C_2 factors to the group without infinite places.
Now we enlarge the modulus by small primes. As one can see, the ray class group gets bigger.

```
> RayClassGroup(8*3*5*7*11*13*101*o);
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4 + Z/4 + Z/24 + Z/24 + Z/120 +
Z/600
Defined on 9 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
```

```
    2*$.3 = 0
    8*$.4 = 0
    24*$.5 = 0
    4*$.6 = 0
    120*$.7 = 0
    12*$.8 = 0
    600*$.9 = 0
Mapping from: Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4 + Z/4 + Z/24 +
Z/24 + Z/120 + Z/600
Defined on 9 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
    2*$.3 = 0
    8*$.4 = 0
    24*$.5 = 0
    4*$.6 = 0
    120*$.7 = 0
    12*$.8 = 0
    600*$.9 = 0 to Set of ideals of o
```

`RayResidueRing(I)`

`RayResidueRing(I, T)`

Given an integral ideal I belonging to the maximal order of a number field, the *ray residue ring modulo* I is the unit group of the maximal order modulo I extended by one C_2 factor for each element of T. The sequence T should be viewed as a condition on the signs of the numbers factored out.

Let I be an integral ideal of an absolute maximal order and let T be a set of real places given by an increasing sequence containing integers i, $1 \leq i \leq r_1$ where r_1 is the number of real zeros of the defining polynomial of the field. This function computes the group of units $\mathrm{mod}^*(I, T)$. The result is a finite abelian group and a map from the group to the order to which the ideal belongs.

When T is not given the unit group of the residue ring mod m is returned. This is equivalent to formally setting $T := [\,]$ to be the empty sequence.

`RayResidueRing(D)`

Given an effective divisor D of a number field, compute the unit group of the residue class ring defined modulo the divisor, ie. compute the group of elements that for the finite places in the support of D approximate 1 and have positive sign at the real infinite places of the support of D.

39.2.2 Selmer groups

Let S be a finite set of prime ideals in a number field K. For an integer p, the p-Selmer group of S is defined as

$$K_p(S) := \{x \in K^\times/(K^\times)^p \mid v_Q(x) = 0 \bmod p \ \forall Q \notin S\}$$

$K_p(S)$ is a finite abelian group of exponent p.

`pSelmerGroup(p, S)`

`Integral`	BoolElt	*Default :* `true`
`Nice`	BoolElt	*Default :* `true`
`Raw`	BoolElt	*Default :* `false`

For a prime integer p and a set of prime ideals S in a number field K, the function returns the p-Selmer group of S as an abstract group G, together with a map m from K to G. The map comes with an inverse.

In principle, the domain of m is the set of x in K satisfying the condition in the definition of $K_p(S)$ above. When $m(x)$ is invoked for x in K, it is assumed without checking that x satisfies the condition. If x does not, either a runtime error occurs, or the map returns a random element of G. (Checking the condition would require a far more expensive computation. The algorithm identifies the class of x in $K_p(S)$ by computing the multiplicative orders of residues of x, and of the group generators, modulo some unrelated primes.)

The role of the optional parameters is as follows. The p-Selmer group is realized as a subgroup of a quotient of a suitable group of $\tilde{S}$-units of the number field, the images of the map returned by `pSelmerGroup` are $\tilde{S}$-units. Initially, $\tilde{S}$ is chosen as S and then enlarged until the p-part of the ideal class group of K is generated by the ideals in S. The parameter `Raw` is related to the same parameter in `SUnitGroup`, see SUnitGroup for more information. If the parameter is set to `true`, the objects returned as images of the `pSelmerGroup` map are exponent vectors that are applied to a fixed sequence of elements to get actual S-units, see the following example for a demonstration.

In addition to changing the return type, `Raw` also implies a reduction of the results, elements returned under `Raw` are reduced by removing the projection of the lattice generated by pth powers of $\tilde{S}$-units. As a side effect of this reduction, elements are no longer guaranteed to be integral. To offset this, the parameter `Integral` can be set to `true`, in which case the sequence of multiplicative generators will be extended to contain uniformizing elements for all ideals in $\tilde{S}$ and the exponent vectors will be supplemented accordingly to achieve integrality.

Example H39E3

We compute the 3-Selmer group of $\mathbf{Q}(\sqrt{10})$ with respect to the primes above 2, 3, 11:

```
> k := NumberField(Polynomial([-10, 0,1]));
```

```
> m := MaximalOrder(k);
> lp := Factorization(2*3*11*m);
> S := [ i[1] : i in lp];
> KpS, mKpS := pSelmerGroup(3, Set(S));
> KpS;
Abelian Group isomorphic to Z/3 + Z/3 + Z/3 + Z/3 + Z/3
Defined on 5 generators
Relations:
    3*KpS.1 = 0
    3*KpS.2 = 0
    3*KpS.3 = 0
    3*KpS.4 = 0
    3*KpS.5 = 0
> mKpS;
Mapping from: RngOrd: m to GrpAb: KpS given by a rule
> mKpS(m!11);
KpS.2
> mKpS(m!11*2);
KpS.2 + 2*KpS.3 + 2*KpS.5
> mKpS(m!11*2*17^3);
KpS.2 + 2*KpS.3 + 2*KpS.5
```

So as long as the argument to `mKpS` is only multiplied by cubes, the image will be stable. Next, we do the same again, but this time using `Raw`:

```
> KpS, mKpS, mB, B := pSelmerGroup(3, Set(S):Raw);
> B;
(-m.1 -11/1*m.1 3/1*m.1 2/1*m.1 13/1*m.1 5/1*m.1 31/1*m.1
    3/1*m.1 - 2/1*m.2 3/1*m.1 + 2/1*m.2 m.1 - 2/1*m.2 m.1
    + 2/1*m.2 m.2 m.1 + m.2 m.1 - m.2 -2/1*m.1 - m.2 m.2
    3/1*m.1 3/1*m.1 11/1*m.1)
> #Eltseq(B);
19
> mB;
Mapping from: GrpAb: KpS to Full RSpace of degree 19 over
Integer Ring given by a rule [no inverse]
> r := KpS.1 + KpS.2 + 2*KpS.4 + KpS.5;
> r @@ mKpS;
396/1*m.1 + 99/1*m.2
> r @ mB;
( 0  1 -2  0  0  0  0  0  0  0  0  1  0  1  0  3  0  0)
> PowerProduct(B, $1);
396/1*m.1 + 99/1*m.2
```

39.2.3 Maps

`InducedMap(m1, m2, h, c)`

Given maps $m_1 : G_1 \to I_1$, $m_2 : G_2 \to I_2$ from some finite abelian groups into the ideals of some maximal order, and a map $h : I_1 \to I_2$ on the ideals, compute the map induced by h on the abelian groups.

For this to work, m_1, and m_2 need to be maps that can be used to define abelian extensions. This implies that m_i has to be a composition of maps where the last component is either the map returned by `ClassGroup` or by `RayClassGroup`. The argument c should be a multiple of the minima of the defining moduli.

The result is the map as defined by

```
hom<G_1 -> G_2 | [ h(r_1(G_1.x)) @@ r_2 : x in [1..Ngens(G_1)]]>
```

For larger modules however, this function is much faster than the straightforward approach. This function tries to find a set of "small" generators for both groups. Experience shows that ray class groups (and their quotients) can usually defined by a "small" set of "small" prime ideals. Since solving the discrete logarithm in ray class groups depends upon solving the discrete logarithm for class groups which is quite slow for "large" ideals, it is much faster in general to use this rather roundabout approach. "Small" ideal in this context means "small" norm.

`InducedAutomorphism(r, h, c)`

An abbreviation for `InducedMap(r, r, h, c)`.

Example H39E4

Consider a "large" ray class group over $k := \mathbf{Q}[\sqrt{10}, \zeta_{16}]$

```
> k := NumberField([Polynomial([-10, 0, 1]), CyclotomicPolynomial(16)]);
> k := OptimizedRepresentation(AbsoluteField(k));
> o := MaximalOrder(k);
> ClassGroup(o:Bound := 500);
Abelian Group of order 1
Mapping from: Abelian Group of order 1 to Set of ideals of o
> IndependentUnits(o);
> SetOrderUnitsAreFundamental(o);
> p := &* [113, 193, 241];
> r, mr := RayClassGroup(p*o);
> #r; Ngens(r);
170613117637701990523621885614354740012596396318100496286167859 2\
000000000
26
```

The automorphisms of k act on this group. One way of obtaining the action is:

```
> autk := Automorphisms(k);
> time m1 := hom<r -> r | [ autk[2](mr(r.i))@@ mr : i in [1..Ngens(r)]]>;
```

```
Time: 26.300
```

In contrast, using `InducedAutomorphism`:

```
> time InducedAutomorphism(mr, autk[2], p);
Time: 19.080
```

Now we increase p:

```
> p *:= 257*337;
> r, mr := RayClassGroup(p*o);
> #r; Ngens(r);
383174842075502327821254012562863503503880824795585976926638885 1\
825941987170813422862370672745342599097489263347018928299704320 0\
0000000
42
> time m1 := hom<r -> r | [ autk[2](mr(r.i))@@ mr : i in [1..Ngens(r)]]>;
Time: 115.120
> time InducedAutomorphism(mr, autk[2], p);
Time: 82.390
```

For "small" examples the direct approach is much faster, but for large ones, especially if one is only interested in certain quotients, the other approach is faster.

39.2.4 Abelian Extensions

The ultimate goal of class field theory is the classification of all abelian extensions of a given number field. Although the theoretical question was settled in the 1930's, it is still difficult to explicitly compute defining equations for class fields. For extensions of imaginary quadratic fields, there are well known analytic methods available. This section explains the basic operations implemented to create class fields in MAGMA.

`RayClassField(m)`
`AbelianExtension(m)`
`RayClassField(m, I, T)`
`AbelianExtension(m, I, T)`
`RayClassField(m, I)`
`AbelianExtension(m, I)`

Given a map $m : G \to I_k$ where G is a finite abelian group and I_k is the set of ideals of some absolute maximal order construct the class field defined by m.

More formally, m^{-1} must be a homomorphism from some ray class group R onto an finite abelian group G. If either I or I and T are given, they must define R. This implies that I has to be an integral ideal and that T has to be a sequence containing the relevant infinite places. Otherwise, MAGMA will try to extract this information from m. The class field defined by m has Galois group isomorphic to $R/\ker(m^{-1})$ under the Artin map.

Note that MAGMA cannot check whether the map passed in is valid. If an invalid map is supplied, the output will most likely be garbage.

`AbelianExtension(I)`

Creates the full ray class field modulo the ideal I.

`RayClassField(D)`

Create the full Ray class field defined modulo the divisor D, ie. an abelian extension that is unramified outside the support of D and such that the (abelian) automorphism group is canonically isomorphic to the ray class group modulo D.

`AbelianpExtension(m, p)`

For a map m as in `AbelianExtension` and a prime number p, create the maximal p-field, i.e. the maximal subfield having degree a p–power.

Example H39E5

The abelian extensions of $\mathbf{Q}$ are known to lie in some cyclotomic field.
We demonstrate this by computing the 12-th cyclotomic field using class fields:
Unfortunately, as ray class groups are not defined for $\mathbf{Z}$ in MAGMA, we must work in a degree 1 extension of $\mathbf{Q}$:

```
> x := ext<Rationals()|>.1;
> Q := ext<Rationals()| x-1 :DoLinearExtension>;
> M := MaximalOrder(Q);
```

The ray class group that defines $\mathbf{Q}(\zeta_{12})$ is defined mod$(12, \infty)$ where ∞ is the unique infinite place of $\mathbf{Q}$. There are at least two ways of looking at this: First, since $Q(\zeta_{12})$ is a totally complex field, the infinite place of $\mathbf{Q}$ must ramify (by convention: $\mathbf{C}$ is ramified over $\mathbf{R}$), so we must include the infinite place in the definition of the ray class group.
Secondly, the ray class group mod(12) without the infinite place is too small. We know $\phi(12) = 4$, $(\mathbf{Z}/12\mathbf{Z})^* = \{1, 5, 7, 11\} = \{\pm 1, \pm 5\}$. As ideals we have $(1) = o = (-1) = (11)$ and $(5) = (-5) = (7)$ so that $\mathrm{Cl}_{12} \cong C_2$. By introducing $T = [1]$, we distinguish ± 1 and ± 5.

```
> G, m := RayClassGroup(12*M, [1]);
> G;
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*G.1 = 0
    2*G.2 = 0
> A := AbelianExtension(m);
> E := EquationOrder(A);
> Ea := SimpleExtension(E);
> Ma := MaximalOrder(Ea);
> Discriminant(Ma);
144
> Factorization(Polynomial(Ma, CyclotomicPolynomial(12)));
[
```

```
    <ext<Ma|>.1 + [0, 1, 0, -1], 1>,
    <ext<Ma|>.1 + [0, -1, 0, 0], 1>,
    <ext<Ma|>.1 + [0, 1, 0, 0], 1>,
    <ext<Ma|>.1 + [0, -1, 0, 1], 1>
]
```

The main advantage of this method over the use of the cyclotomic polynomials is the fact that we can directly construct certain subfields:

```
> x := ext<Integers()|>.1;
> M := MaximalOrder(x^2-10);
> G, m := RayClassGroup(3615*M, [1,2]);
> G; m;
Abelian Group isomorphic to Z/2 + Z/2 + Z/4 + Z/80 + Z/240
Defined on 5 generators
Relations:
    4*G.1 = 0
    80*G.2 = 0
    240*G.3 = 0
    2*G.4 = 0
    2*G.5 = 0
Mapping from: GrpAb: G to Set of ideals of M
```

We will only compute the 5-part of this field:

```
> h := hom<G -> G | [5*G.i : i in [1..#Generators(G)]]>;
> Q, mq := quo<G|Image(h)>;
> mm := Inverse(mq) * m;
> mm;
Mapping from: GrpAb: Q to Set of ideals of M
> A := AbelianExtension(mm);
> E := EquationOrder(A);
> E;
Non-simple Equation Order defined by x^5 - [580810, 0]*x^3 +
    [24394020, -40656700]*x^2 + [15187310285, 2799504200]*x
    + [1381891263204, 530506045900], x^5 - [580810, 0]*x^3 +
    [-109192280, 34848600]*x^2 + [30584583385,
    16797025200]*x + [-341203571896, 109180663800] over its
ground order
> C := Components(A);
```

The function `Components` gives a list of cyclic extensions of M that correspond to the cyclic factors of G.

```
> GaloisGroup(NumberField(C[1]));
Permutation group acting on a set of cardinality 5
Order = 5
    (1, 4, 2, 5, 3)
[ -9 + O(41), 15 + O(41), 6 + O(41), 18 + O(41), 11 + O(41) ]
GaloisData over Z_Prime Ideal
Two element generators:
```

```
[41, 0]
[25, 1] - relative case
> GaloisGroup(NumberField(C[2]));
Order = 5
    (1, 4, 2, 5, 3)
[ 38 + O(79), -28 + O(79), -31 + O(79), 27 + O(79), -6 + O(79) ]
GaloisData over Z_Prime Ideal
Two element generators:
[79, 0]
[57, 1] - relative case
```

Thus the Galois group is indeed proven to be $C_5 \times C_5$.

`AbelianExtension(I, P)`

Creates the full ray class field modulo the ideal I and the infinite places in P.

`HilbertClassField(K)`

Creates the Hilbert class field of K, i.e. the maximal unramified abelian extension of K. This is equivalent to `AbelianExtension(1*MaximalOrder(K))`.

`MaximalAbelianSubfield(M)`
`MaximalAbelianSubfield(F)`
`MaximalAbelianSubfield(K)`

`Conductor` [RngOrdIdl, [RngIntElt]] *Default :* [ ]

Let k be the coefficient field of the given number field K. This function creates the maximal abelian extension A of k inside K. If `Conductor` is given, it must contain a multiple of the true conductor. If no value is specified, the discriminant of K is used.

The correctness of this function is based on some heuristics. The algorithm is similar to [Coh00, Algorithm 4.4.3]

`AbelianExtension(K)`
`AbelianExtension(M)`

`Conductor` [RngOrdIdl, [RngIntElt]] *Default :* [ ]

Creates an abelian extension A of k the coefficient field of the input K that is isomorphic to K. If a value for `Conductor` is given, it must contain a multiple of the true conductor, otherwise the discriminant of K is used to be more specific, in case K is a number field, the discriminant of it's maximal order is used as an initial guess, while for field of fractions of orders (K of type `FldOrd`), the defining order gives the initial guess.

In contrast to `MaximalAbelianSubfield`, provided the field is abelian, this function always computes a correct answer.

Example H39E6

We will compute the Hilbert class field of the sextic field defined by a zero of the polynomial $x^6 - 3x^5 + 6x^4 + 93x^3 - 144x^2 - 153x + 2601$ which has a class group isomorphic to $C_3 \times C_3$.

```
> m := LLL(MaximalOrder(Polynomial([2601, -153, -144, 93, 6, -3, 1 ])));
```

The call to LLL is not necessary, but quite frequently class group computations are faster if the order basis is LLL-reduced first.

```
> a,b := ClassGroup(m:Bound := 300);a;
Abelian Group isomorphic to Z/3 + Z/3
Defined on 2 generators
Relations:
    3*a.1 = 0
    3*a.2 = 0
```

`HilbertClassField` on `m` computes a Non-simple number field defining the Hilbert class field of `m`.

```
> H := HilbertClassField(NumberField(m)); H;
Number Field with defining polynomial [ $.1^3 + 1/595*(-460*$.1^5
    + 2026*$.1^4 - 4052*$.1^3 - 52572*$.1^2 + 229338*$.1 -
    529159), $.1^3 + 1/37485*(82*$.1^5 + 4344*$.1^4 + 8805*$.1^3
    + 15990*$.1^2 + 410931*$.1 + 1098693)] over its ground field
> time _ := MaximalOrder(H);
Time: 56.150
```

However, in order to access more of the structural information of `H` we have to create it as an abelian extension, using `b`.

```
> A := AbelianExtension(b);
> HH := NumberField(A);
```

Now we are able to compute the maximal order of `HH` using `A` and verify the discriminant:

```
> time M := MaximalOrder(A:Al := "Discriminant");
Time: 3.260
> Discriminant(M);
Ideal of m
Basis:
[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
```

Note, that as a side effect, the maximal order of `HH` is now known:

```
> time MaximalOrder(HH);
Maximal Order of Equation Order with defining polynomials x^3 + [841, 841, 312,
    -534, 222, 0], x^3 + [25, -2, -9, 7, -11, -8] over m
```

```
Time: 0.000
```

We will continue this example following the next section.

39.2.5 Binary Operations

The model underlying the class field theory as implemented in MAGMA is based on the (generalized) ideal class groups. Based on this group-theoretical description, certain binary operations are easily possible:

`A eq B`

Gives two abelian extensions with the same base field, decide if they are the same.

`A subset B`

Gives two abelian extensions with the same base field, decide if they are contained in each other.

`A * B`

Given two abelian extensions with the same base field, find the smallest abelian extension containing both.

`A meet B`

Given two abelian extensions with the same base field, find the largest common subfield.

39.3 Galois Module Structure

If the base field k for class field constructions is normal with respect to some subfield k_0, i.e. k/k_0 is normal with Galois group G and if the defining modulus of the ideal group is G–invariant, then G acts on the ideal group. The following functions view ideal groups as Galois modules. Given an abelian extension A and parameters `All` and `Over`, we will consider this setup:

Let k be the `BaseField` of A and k_1 the coefficient field of k. If `All` is `true`, let $g := \mathrm{Aut}(k/k_1)$, otherwise, $g := \langle$`Over`$\rangle$. In both cases we define $k_0 := \mathrm{Fix}(k, g)$. In particular, if k is normal over the coefficient field k_1 then $k_0 = k_1$ and g is the full Galois group.

In general g is not required to contain k_1 automorphisms, so that any subset of the **Q** automorphism group is valid as input. By construction, k is normal over k_0, and g acts on the ideals of k. In general however, g does not act on the ideal groups used to define A.

39.3.1 Predicates

`IsAbelian(A)`

`All`	BoolElt	*Default :* `false`
`Over`	[Map]	*Default :* []

Returns `true` if and only if the abelian extension A is abelian over k_0.

`IsNormal(A)`

`All`	BoolElt	*Default :* `false`
`Over`	[Map]	*Default :* []

Returns `true` if and only if the abelian extension A is normal over k_0. This tests whether the defining ideal group is a g-module.

`IsCentral(A)`

`All`	BoolElt	*Default :* `false`
`Over`	[Map]	*Default :* []

Returns `true` if and only if the abelian extension A is central over k_0. If k is cyclic over k_0 then this is equivalent to checking if A is abelian over k_0. This tests whether the defining ideal group is a g–module with trivial action: If N is the norm group of A, the group extension

$$1 \to N \to G \to g \to 1$$

is central.

39.3.2 Constructions

`GenusField(A)`

`All`	BoolElt	*Default :* `false`
`Over`	[Map]	*Default :* []

The genus field is the maximal abelian extension of k_0 that is contained in the abelian extension A. The result of this function is an abelian extension of k_0.

`H2_G_A(A)`

For A such that A is normal over $\mathbf{Q}$ with base field k that is normal too, compute the 2nd cohomology group of the Galois group of k acting on the ideal group defining A.

`NormalSubfields(A)`

`Quot`	SeqEnum[RngIntElt]	*Default :* []

For an abelian extension, normal over Q and defined over a normal number field k as base field, return a list of all normal intermediate fields. If `Quot` is given, restrict to fields where the norm group has the abelian invariants as specified in `Quot`.

`AbelianSubfield(A, U)`

`FixedField(A, U)`

`IsNormal` BoolElt *Default :* `false`

For an abelian extension A with norm group map $G \to I$ for some finite abelian group G and a subgroup $U < G$, define the field corresponding to G/U, ie. the field fixed by U. If `IsNormal` is given then any cohomology information that is present is transferred to the new field - if possible.

`CohomologyModule(A)`

For an abelian extension A defined over some normal field k/Q, compute the cohomology module (see Chapter 68). The maps returned give the transition between the Z-modules used in the cohomology package and the ideal groups used to define A.

The first map returned maps between the automorphism group of k (as an permutation group) and the actual automorphisms of the field. It is obtained as the third return value of `AutomorphismGroup`.

The second map maps between the ideal group used to create A and a standart representation of the same group.

The third map maps between the standart representation of the norm group and the Z-module.

39.4 Conversion to Number Fields

Although in theory an abelian extension "uniquely" defines a number field and therefore all its properties, not all of them are directly accessible (in Magma at least). The functions listed here perform the conversion to a number field, the most important being of course the function that computes defining equations.

`EquationOrder(A)`

Verbose `ClassField` *Maximum :* 5

Given an abelian extension A of a number field, using the algorithm of Fieker ([Fie00, Coh00]) defining equations for A are computed. For each cyclic factor of prime power degree, one polynomial will be constructed. Depending on the size of the cyclic factors encountered, this may be a very lengthy process.

`NumberField(A)`

Converts the abelian extension A into a number field. This is equivalent to `NumberField(EquationOrder(A))`.

`MaximalOrder(A)`

`Al`	MonStgElt	*Default :* "*Kummer*"
`Partial`	BoolElt	*Default :* `false`

Computes the maximal order of the abelian extension A. The result is the same as that given by `MaximalOrder(EquationOrder(A))` but this functions uses the special structure of A and should be much faster in general.

The first step involves computing the maximal orders of each component.

If `Al eq "Kummer"` the maximal order computation uses Kummer theory to compute maximal orders of kummer extensions known to each component then intersects these with the component to gain the maximal order of that component [Sut12].

If `Al eq "Round2"`, the ordinary round 2 maximal order function is used on the components.

If `Al eq "Discriminant"`, the discriminant of the components is passed into the maximal order computation.

In the second step, the components are combined into an approximation of the full maximal order of A.

If `Partial` is `true`, the computations stop at this point, otherwise `MaximalOrder` is again called and the discriminant of A is passed in.

`Components(A)`

Verbose	`ClassField`	*Maximum :* 5

A list of relative extensions is determined. One extension per cyclic factor is computed.

`Generators(A)`

The first return value is a sequence of generating elements for `NumberField(A)`, the second contains the same elements but viewed as elements of the Kummer extension used in the construction. The third list contains the images of the second list under the action of a generator of the automorphism group corresponding to this cyclic factor.

39.5 Invariants

Several invariants of an abelian extension can easily be obtained from the ideal groups without first computing defining equations for the field.

`Discriminant(A)`

Let A be an abelian extension. Based on the conductor-discriminant relation made explicit by [Coh00, Section 3.5.2], the discriminant of the class field A is computed. This does not involve the computation of defining equations. The second return value is the signature of the resulting field.

`AbsoluteDiscriminant(A)`

The absolute discriminant of A as a number field over $\mathbf{Q}$.

`Conductor(A)`

Computes the conductor of the abelian extension A, i.e. the smallest ideal and the smallest set of infinite places that are necessary to define A. The algorithm used is based on [Pau96, HPP97].

`Degree(A)`

The degree of the abelian extension A.

`AbsoluteDegree(A)`

The degree of the abelian extension A over $\mathbf{Q}$.

`CoefficientRing(A)`
`CoefficientField(A)`
`BaseField(A)`

The base field of the abelian extension A, that is `FieldOfFractions(BaseRing(A))`.

`BaseRing(A)`
`CoefficientRing(A)`

The base ring of the abelian extension A, that is the maximal order used to define the underlying ray class group.

`NormGroup(A)`

The norm group (see the definition of norm group on page 1000) used to define the abelian extension A.

`DecompositionField(p, A)`

The decomposition field of the finite prime p in the abelian extension A as an abelian (sub)extension.

`DecompositionField(p, A)`

The decomposition field of the place p in the abelian extension A as an abelian extension.

`DecompositionGroup(p, A)`
`DecompositionGroup(p, A)`

The decomposition group of the finite prime p in the abelian extension A. The abelian group returned is a subgroup of the norm group.

`DecompositionGroup(p, A)`

The decomposition group of the place p in the abelian extension A. The abelian group returned is a subgroup of the `NormGroup`.

`DecompositionType(A, p)`

The "type" of the decomposition of the finite prime ideal p in the abelian extension A as a sequence of pairs $\langle f, e \rangle$ giving the degrees and the ramification indices.

`DecompositionType(A, p)`

The "type" of the decomposition of the place p in the abelian extension A as a sequence of pairs $\langle f, e \rangle$ giving the degrees and the ramification indices.

`DecompositionType(A, p)`

`Normal` BoolElt *Default :* `false`

The "type" of the decomposition over $\mathbf{Q}$ of the prime number p in the abelian extension A as a sequence of pairs $\langle f, e \rangle$ giving the degrees and the ramification indices. If `Normal` is set to `true` then the algorithm assumes that the base field of A is normal. This is used to speed up the computations.

`DecompositionTypeFrequency(A, l)`

`Normal` BoolElt *Default :* `false`

Computes the decomposition type of all elements in l and returns them as a multi-set. The list l must only contain objects for which `DecompositionType` is defined. If `Normal eq true` then the underlying `DecompositionType` function must be able to deal with it too. If `Normal` is set to `true` then the algorithm assumes that the base field of the abelian extension A is normal. This is used to speed up the computations.

`DecompositionTypeFrequency(A, a, b)`

`Normal` BoolElt *Default :* `false`

Computes the decomposition type over $\mathbf{Q}$ in the abelian extension A of all prime numbers $a \leq p \leq b$ and returns them as a multi set.

If `Normal` is set to `true` then the algorithm assumes that the base field of A is normal. This is used to speed up the computations.

39.6 Automorphisms

The group of relative automorphisms of the abelian extension is isomorphic via the Artin map to the ideal group used to define the field. After defining equations are computed, the user can explicitly map ideals that are coprime to the defining modulus to automorphisms of the field.

`ArtinMap(A)`

Returns a map from the defining group (considered as a "subgroup" of the ideals of the base ring) into the automorphisms of the abelian extension A over the base field.

By the defining property of class fields, this map induces an isomorphism on the defining group (as an abelian group) onto the relative automorphisms of A.

Since this function constructs the number field defined by A, this may involve a lengthy calculation.

`FrobeniusAutomorphism(A, p)`

Computes the relative automorphism of the abelian extension A that is the Frobenius automorphism of p. Since this function constructs the number field defined by A, this may involve a lengthy calculation.

`AutomorphismGroup(A)`

`All`	BoolElt	*Default :* `false`
`Over`	[Map]	*Default :* []

If `IsNormal` is true for the abelian extension A with the given parameters, then the automorphism group of A over k_0 is computed. Since this function constructs the number field defined by A, this may involve a lengthy calculation.

`ProbableAutomorphismGroup(A)`

`Factor`	RingIntElt	*Default :* 1

In case of A and it's base field k both begin normal over $\mathbf{Q}$, the automorphism group G of $A/\mathbf{Q}$ is a group extension of the abelian group coming from the definition of A and the automorphism group of $k/\mathbf{Q}$. This functions sets up the corresponding group extension problem and uses `DistinctExtensions` to compute all group theoretical possibilities for G. In case of several possible groups, a further selection based on cycle types and their frequencies is attempted. The optional parameter `Factor` is passed on to `ImproveAutomorphismGroup` to control the amount of time spent on improving the guess.

While this function can be much faster than the direct computation of the automorphism group, the result of this computation is in general not guaranteed. Furthermore, as there are groups that cannot be distinguished by cycle types and their frequencies alone, correctness cannot be achieved by increasing the value of `Factor`. The intended use of this function is to have a (reasonable fast) method of checking is the field under consideration has an interesting group before an unnecessary long call to `AutomorphismGroup` is attempted.

`ImproveAutomorphismGroup(F, E)`

`Factor` RngIntElt *Default :* 1

Given the output of `ProbableAutomorphismGroup` or `ImproveAutomorphismGroup` try to improve the quality of the guess by splitting more primes to get more data for a cycle-type frequency analysis.

Example H39E7

We will demonstrate the use of `ProbableAutomorphismGroup` by investigating extensions of $\mathbf{Q}(\sqrt{10})$:

```
> k := NumberField(Polynomial([-10, 0, 1]));
> R, mR := RayClassGroup(4*3*5*MaximalOrder(k));
> s := [x`subgroup : x in Subgroups(R:Quot := [2,2])];
> a := [ AbelianExtension(Inverse(mq)*mR) where
>                              _, mq := quo<R|x> : x in s ];
> n := [ x : x in a | IsNormal(x:All)];
> ProbableAutomorphismGroup(n[2]);
Finitely presented group on 3 generators
Relations
    $.2^2 = Id($)
    $.3^2 = Id($)
    ($.2, $.3) = Id($)
    ($.1, $.2^-1) = Id($)
    $.1^-1 * $.3 * $.1 * $.3^-1 * $.2^-1 = Id($)
    $.1^2 = Id($)
```

This shows that since there is only one group extension of a V_4 by C_2 with the action induced from the action of the Galois group of k on R, the Automorphism group is already determined. On the other hand, for the first subgroup there are more possibilities:

```
> g, c := ProbableAutomorphismGroup(n[1]);
> #c;
2
```

We will try to find the "correct" guess by looking at the orders of elements which correspond to decomposition types and their frequencies:

```
> {* Order(x) : x in CosetImage(c[1], sub<c[1]|>) *};
{* 1, 2^^7 *}
> {* Order(x) : x in CosetImage(c[2], sub<c[2]|>) *};
{* 1, 2^^3, 4^^4 *}
```

So, if we find a prime of degree 4 we know it's the second group. Looking at the frequencies, we can be pretty confident that we should be able to find a suitable prime - if it exists. Since among the first 100 primes there is not a single prime with a factor of degree 4 we are pretty confident that the first group is the correct one. By setting the verbose level, we can see how the decision is made:

```
> SetVerbose("ClassField", 2);
```

```
> _ := ImproveAutomorphismGroup(n[1], c:Factor := 2);
Orders and multiplicities are  [
    {* 1, 2^^7 *},
    {* 1, 2^^3, 4^^4 *}
]
Probable orders and multiplicities are  {* 1^^3, 2^^34 *}
Error terms are  [ 0.006993296366796325415873541339 07,
0.993006703633203674584126458661 ]
```

This indicates that out of 37 primes considered, non had a degree 4 factor, thus we are confident that the first group is the correct one.

`AbsoluteGaloisGroup(A)`

Given an abelian extension, compute it's Galois group over **Q**, ie. the abstract automorphism group of a **Q**-normal closure of A. This function requires the defining equations for A as a number field to be known, but is considerably faster than calling `GaloisGroup` for the number field directly. The group is returned as an permutation group. All roots of the defining polynomial of the coefficient field of A as well as of the defining polynomials of A itself are returned in some local field. The zeros as well as data required for further computations are contained in the 3rd return value.

`TwoCocycle(A)`

For an abelian extension that is normal over Q and defined over a normal base field k/Q, the automorphism group of A/Q is a group extension of the Galois group of k by A. As a group extension it corresponds to an element in the second cohomology group and can be represented by an explicit 2-cocyle with values in the norm group. This function computes such a cocycle. It can be used as an element of the second cohomology group of the cohomology module of A, see `CohomologyModule`.

39.7 Norm Equations

For cyclic fields Hasse's famous norm theorem states that when considering the solvability of norm equations, local solvability everywhere is equivalent to global solvability. Unfortunately, this local-global principle fails in general even for fields with Galois group isomorphic to Klein's group V_4. The extent of this failure is measured by the number knot which is the quotient of the numbers that are everywhere local norms by the global norms. As it turns out, the structure and size of this quotient can be easily computed, so that it is possible to test if Hasse's theorem is sufficient.

As a second consequence, solvability can be decided by looking at the maximal p-subfields for all primes that divide the degree of the field. Even better, a global solution can be obtained by combining solutions from the maximal p-subfields.

It is important to note that local solvability can be decided by analyzing the ideal groups only. Thus, all the "local" functions will avoid computing defining equations and are therefore reasonably fast.

`IsLocalNorm(A, x, p)`

Returns `true` if and only if x is a local norm in the abelian extension A at the finite prime p, i.e. if x is a norm in the extension of the local field obtained by taking the completion at p.

`IsLocalNorm(A, x, i)`

Returns `true` if and only if x is a local norm in the abelian extension A at the infinite prime i.

`IsLocalNorm(A, x, p)`

Returns `true` if and only if x is a local norm in the abelian extension A at the place p, i.e. if x is a norm in the extension of the local fields obtained by taking the completion at p.

`IsLocalNorm(A, x)`

Returns `true` if and only if x is a local norm everywhere in the abelian extension A.

`Knot(A)`

The (number) knot is defined as the quotient group of the group consisting of those elements of the base field of the abelian extension A that are local norms everywhere modulo the elements that are norms. Therefore, if the knot is trivial, an element is a local norm if and only if it is a norm.

Hasse's norm theorem states that for cyclic fields A the knot is always trivial. In general, this is not true for non-cyclic fields.

`NormEquation(A, x)`

Checks if x is a norm, and if so returns an element of the pre-image. As a first step this function verifies if x is a local norm. If x passes this test, the number field of A is computed and by combining solutions of the norm equation in certain subfields a solution in A is constructed. If the knot is non-trivial, the last step may fail.

This function can be extremely time consuming as not only defining equations for A are computed but class groups in some of them. For large A this is much more efficient than just solving the norm equation in the number field.

`IsNorm(A, x)`

As a first step, this function verifies if x is a local norm. If x passes this test, MAGMA verifies whether the knot is trivial. If it is, `true` is returned. However, if the knot is non-trivial, then the function `NormEquation` is invoked.

Example H39E8

We illustrate the power of the class field theoretic approach with the following example from group theory. We want to solve for elements of norm 2 or 5 in the field $\mathbf{Q}(\zeta_5)(\eta)$ where $\eta^{20} - \zeta_5\eta^{10} + \zeta_5^2 = 0$ over the cyclotomic field. This field has degree 80 over $\mathbf{Q}$ and is therefore far too large for a direct method.

```
> k := CyclotomicField(5);
> kt := ext<k|>;
> K := NumberField(kt.1^20 - k.1*kt.1^10 +k.1^2);
```

Now we convert K into an abelian extension of k:

```
> A := AbelianExtension(K);
> A;
FldAb, defined by (<[590490000000000000000000, 0, 0, 0]>, [])
of structure: Z/2 + Z/10
> Conductor(A);
Ideal
Two element generators:
    [37500, 0, 0, 0]
    [12060, 15120, 7440, 1680]
[]
```

We now recreate A using the smaller conductor. This will significantly speed up the following computations.

```
> m_0, m_inf := $1;
> A := AbelianExtension(K : Conductor := [* m_0, m_inf *]);
```

We first check the local solvability:

```
> IsLocalNorm(A, BaseRing(A)!2);
false
> IsLocalNorm(A, BaseRing(A)!5);
true
> Knot(A);
Abelian Group isomorphic to Z/2
Defined on 1 generator in supergroup:
    $.1 = $.1
Relations:
    2*$.1 = 0
```

Since the knot is isomorphic to a C_2, the local solvability is not sufficient, but we can attempt to solve the equation:

```
> NormEquation(A, BaseRing(A)!5);
true [
    1/10*(-3*zeta_5^3 - 6*zeta_5^2 + zeta_5 -
        7)*$.1*$.2*$.3^4 + 1/10*(9*zeta_5^3 + 3*zeta_5^2 +
        2*zeta_5 + 6)*$.1*$.2*$.3^3 + 1/10*(-6*zeta_5^3 +
        3*zeta_5^2 - 3*zeta_5 + 1)*$.1*$.2*$.3^2 +
```

```
        1/10*(-3*zeta_5^3 - 6*zeta_5^2 + zeta_5 -
        7)*$.1*$.2*$.3 + 1/2*(2*zeta_5^3 + zeta_5^2 + zeta_5
        + 2)*$.1*$.2 + 1/10*(-12*zeta_5^3 - 4*zeta_5^2 -
        6*zeta_5 - 13)*$.1*$.3^4 + 1/10*(11*zeta_5^3 +
        2*zeta_5^2 + 13*zeta_5 + 4)*$.1*$.3^3 +
        1/10*(zeta_5^3 + 2*zeta_5^2 - 7*zeta_5 +
        9)*$.1*$.3^2 + 1/10*(-12*zeta_5^3 - 4*zeta_5^2 -
        6*zeta_5 - 13)*$.1*$.3 + 1/2*(2*zeta_5^3 - zeta_5^2
        + 2*zeta_5 + 1)*$.1 + 1/10*(-zeta_5^3 + 3*zeta_5^2 -
        3*zeta_5 + 1)*$.2*$.3^4 + 1/10*(-2*zeta_5^3 +
        zeta_5^2 + 4*zeta_5 - 3)*$.2*$.3^3 +
        1/10*(3*zeta_5^3 - 4*zeta_5^2 - zeta_5 +
        2)*$.2*$.3^2 + 1/10*(-zeta_5^3 + 3*zeta_5^2 -
        3*zeta_5 + 1)*$.2*$.3 + 1/2*(-zeta_5^3 - zeta_5^2 -
        1)*$.2 + 1/10*(-14*zeta_5^3 - 13*zeta_5^2 - 2*zeta_5
        - 21)*$.3^4 + 1/10*(22*zeta_5^3 + 4*zeta_5^2 +
        11*zeta_5 + 13)*$.3^3 + 1/10*(-8*zeta_5^3 +
        9*zeta_5^2 - 9*zeta_5 + 8)*$.3^2 +
        1/10*(-14*zeta_5^3 - 13*zeta_5^2 - 2*zeta_5 -
        21)*$.3 + 1/2*(5*zeta_5^3 + zeta_5^2 + 3*zeta_5 + 4)
]
> _, s := $1;
> Norm(s[1]);
5
```

Thus, the largest field we had to work in was of degree 16.

39.8 Attributes

In this rather technical section, we describe the programming interface to the abelian extension module. Most of the internal representation is available as attributes and may be used to extend the package.

39.8.1 Orders

The only attribute of orders `RngOrd` that is of interest here concerns cyclotomic extensions. The first and usually most time consuming step while computing defining extensions is to adjoin certain roots of unity and to compute class groups of these rather large fields.

The cyclotomic extensions are stored and are available via an attribute. This attribute is a read-only attribute.

`o‘CyclotomicExtensions`

If defined, `CyclotomicExtensions` is a list of records, each containing data for one cyclotomic extension, i.e. for an order o this is an extension of the form $O := o[\zeta_l]$ for a prime power $l = p^n$. The components are

`Abs` : a maximal order Oa of O given as an absolute extension

`Rel` : O as an extension of o

`p2n` : the "l", i.e. the cyclotomic order

`Zeta` : a primitive lth root of unity as an element of the absolute extension `Abs`

`Aut` : a list of records describing generators for the automorphism group of O over o. If p is an odd prime then the list will always be of length 1. This list consists of the following components:

`Aut‘Abs` : the automorphism as an automorphism of Oa

`Aut‘Rel` : the automorphism as an automorphism of O. Note that in general this is not an o-automorphism, as o itself may contain roots of unity.

`Aut‘Order` : the order of the automorphism

`Aut‘r` : the image of ζ_l, this automorphism sends ζ_l to ζ_l^r.

Due to possible common subfields of o and $\mathbf{Q}(\zeta_k)$, the degree of O over o may be smaller than expected. Furthermore, Oa will be in optimized representation.

Example H39E9

We will demonstrate this with the Hilbert class field of $\mathbf{Q}(\sqrt{-1001})$:

```
> Zx<x> := PolynomialRing(Integers());
> k := NumberField(x^2+1001);
> g, m :=ClassGroup(k); g;
Abelian Group isomorphic to Z/2 + Z/2 + Z/10
Defined on 3 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
    10*$.3 = 0
> K := HilbertClassField(k);
Number Field with defining polynomial [ $.1^2 - 2*k.1 + 136,
    $.1^2 - 4*k.1 - 47, $.1^2 + 2*k.1 - 136, $.1^5 + 37210*$.1^3
    + (-2104500*k.1 + 61148840)*$.1^2 + (-292068000*k.1 +
    14636593760)*$.1 + 305212632000*k.1 - 1779009360128] over k
> o := MaximalOrder(k);
> c := o‘CyclotomicExtensions;
> #c;
2
```

Since there are two non-isomorphic direct cyclic factors of prime power order in the class group of k, we have two cyclotomic extensions stored in o. One has order 2 and the other has order 5:

```
> c[1]‘p2n;
```

```
2
> c[2]‘p2n;
5
```

Since the order 2 extension is essentially trivial, we will discuss the other extension in detail.

```
> c[2]‘Abs;
Maximal Order of Equation Order with defining polynomial x^8 +
    4*x^7 + 4016*x^6 + 12034*x^5 + 6032056*x^4 + 12044060*x^3 +
    4016040045*x^2 + 4010020020*x + 1000000005005 over Z
> c[2]‘Rel;
Maximal Equation Order with defining polynomial x^4 + x^3 + x^2 +
    x + [1, 0] over o
```

The relative extension is generated by a root of the 5th-cyclotomic polynomial of degree 4, so there are no common subfields. The corresponding absolute extension has degree 8. As one can see, the class group (at least a conditional class group) is known:

```
> c[2]‘Abs:Maximal;
   F[1]
    |
   F[2]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^8 + 4*x^7 + 4016*x^6 + 12034*x^5 + 6032056*x^4 +
    12044060*x^3 + 4016040045*x^2 + 4010020020*x + 1000000005005
Discriminant: 4016024016004000000
Index: 4100090871676479535981/1
Class Number 12800
Class Group Structure C2 * C4 * C20 * C80
Signature: [0, 4]
```

We will illustrate the `Aut` entries by showing their effect on ζ_5. Since the maps operate between the number fields rather than the orders, we will coerce the results back into the orders for clarity. The only difference between `Rel` and `Abs` is that `Rel` implements the automorphism in the relative extension.

```
> Oa := c[2]‘Abs;
> z := c[2]‘Zeta; z;
[0, 1, 0, 0, 0, 0, 0, 0]
> Oa!c[2]‘Aut[1]‘Abs(z);
[0, 0, 1, 0, 0, 0, 0, 0]
> z^c[2]‘Aut[1]‘r;
[0, 0, 1, 0, 0, 0, 0, 0]
```

The `Order` entry gives the order of the automorphism:

```
> c[2]‘Aut[1]‘Order;
4
```

```
> Oa!c[2]‘Aut[1]‘Abs(z);
[0, 0, 1, 0, 0, 0, 0, 0]
> Oa!(c[2]‘Aut[1]‘Abs($1));
[-1, -1, -1, -1, 0, 0, 0, 0]
> Oa!(c[2]‘Aut[1]‘Abs($1));
[0, 0, 0, 1, 0, 0, 0, 0]
> Oa!(c[2]‘Aut[1]‘Abs($1));
[0, 1, 0, 0, 0, 0, 0, 0]
```

As a by–product one can see the optimized representation: the first four basis elements are clearly 1, ζ_5, ζ_5^2, ζ_5^3 which are the T_2 shortest integral elements in m.

39.8.2 Abelian Extensions

Abelian extensions have several attributes. Most of them are only useful in programming.

`A‘Components`

This read-only atribute contains a record for each cyclic factor of prime power degree that occurs in the abelian extension. In order to describe the contents we have to fix some definitions. Let o be the base ring of the abelian extension A. Then O will be the maximal order of the cyclotomic extension $o[\zeta_l]$ as an extensions of $\mathbf{Z}$. The algorithm for the computation of defining extensions will firstly compute a generator $a \in O$ such that $O(a^{1/l})$ equals $A(\zeta_l)$. This element will be a certain S-unit of O.

The components are

`Basis` : a matrix B containing order elements. This represents a “multiplicative basis” for the generator a and all the S-units that are used.

`GenRaw` : the exponent vector G defining a, i.e.

$$a = \prod_i {B_{1,i}}^{G_{i,1}}.$$

`UnitsRaw` : a matrix U defining a basis for the group of S-units, i.e.

$$u_j = \prod_i {B_{1,i}}^{U_{j,i}}.$$

`S` : a list containing the prime ideals of S

`Gen` : a as an element of Oa

`GenAut` : the image of a generator for the class field (an image of `Class Field.2`) in the field of fractions of O under a generator for the cyclic group corresponding to this component.

`GenInv` : $1/a$ as an element of the field of fractions of Oa

`O` : the big Kummer extension $Oa(a^{1/l})$

`ClassField` : an equation order for the cyclic extension corresponding to this component. This will be an extension of o.

`Artin` : the Artin map on the big Kummer extension $o[\zeta_l](U_S^{1/l})$ where the automorphisms are represented as elements in some abelian group of type $C_l^{\#S}$.

`A‘DefiningGroup`

`A‘NormGroup`

These two attributes give access to the ideal group used to define A, and they have the same structure. `DefiningGroup` is the group used to create A in the first place whereas `NormGroup` contains the group defined modulo the conductor. The user needs to call the function `Conductor` in order to define this component. These attributes are read-only.

The records contain the following components:

`Map` : the map as passed into the `AbelianExtension` constructor, respectively, an equivalent map.

`m0` : the "finite" part of the modulus underlying `Map`

`m_inf` : the "infinite" part of the modulus underlying `Map`

`RcgMap` : if present, contains the map returned from the call to `RayClassGroup` with `m0` and `m_inf`.

`GrpMap` : if present, the "rest" of `Map`, i.e. `Map` = `RcgMap` ∘ `GrpMap`.

`A‘IsAbelian`

Stores the result of a call to `IsAbelian` with parameter `All := true`.

`A‘IsNormal`

Stores the result of a call to `IsNormal` with parameter `All := true`.

`A‘IsCentral`

Stores the result of a call to `IsCentral` with parameter `All := true`.

Example H39E10

To illustrate the preceeding examples we will investigate the 3-part of the ray class field modulo 36 over $\mathbf{Z}[\sqrt{10}]$:

```
> Zx<x> := PolynomialRing(Integers());
> o := MaximalOrder(x^2+10);
> r, mr := RayClassGroup(36*o);
> q, mq := quo<r| [r.i*3 : i in [1..Ngens(r)]]>;
> A := AbelianExtension(Inverse(mq)*mr); A;
FldAb, defined by (<[36, 0]>, [])
```

```
of structure: Z/3 + Z/3
```

At this stage, the only attribute that is assigned is `DefiningGroup`.

```
> la := GetAttributes(FldAb);
> [ <assigned A``i, i> : i in la];
[ <false, Components>, <true, DefiningGroup>, <false, IsAbelian>,
<false, IsCentral>, <false, IsNormal>, <false, NormGroup> ]
```

So let us have a closer look at `DefiningGroup`:

```
> d := A`DefiningGroup;
> d;
rec<recformat<Map, m0, m_inf, RcgMap, GrpMap> | Map := Mapping
from: GrpAb: q to Set of ideals of o
Composition of Mapping from: GrpAb: q to GrpAb: r and
Mapping from: GrpAb: r to Set of ideals of o, m0 := Principal
Ideal of o
Generator:
    [36, 0], m_inf := [], RcgMap := Mapping from: GrpAb: r to Set
of ideals of o, GrpMap := Mapping from: GrpAb: q to GrpAb: r>
```

As one can see, `Map` is the original map used to define A. Note that this is the composition of `RcgMap` and `GrpMap`, the first being equivalent to `mr`, the second to `mq`. Furthermore, the defining modulus is now available in `m0` and `m_inf`.
After having set up A, we now need to transform it into a number field in order for the `Components` to be assigned:

```
> K := NumberField(A);
> K;
Number Field with defining polynomial [ $.1^3 - 3*$.1 - 6*$.1 -
    11, $.1^3 - 3*$.1 - 6*$.1 + 11] over its ground field
> c := A`Components;
> #c;
2
```

We will focus on the first component only.

```
> B := c[1]`Basis;
> a := &* [ B[1][i] ^ c[1]`GenRaw[i][1] : i in [1..Ncols(B)]];
-11/1*$.2 + 4/1*$.3 + 2/1*$.4
> c[1]`Gen;
[0, -11, 4, 2]
> $1 eq $2;
true
> c[2]`GenInv * a eq 1;
true
```

In S we need all primes dividing the degree 9 of our extension, all primes dividing the modulus 36 and enough primes to generate the 3-part of the class group. Since the class group can be generated by the prime ideal over 2, this leaves us with only two prime ideals in S. Since k is imaginary quadratic, the base field $k[\zeta_3]$ is totally complex of degree 4 which implies unit rank 1.

So the free rank of our S-unit group will be 3, and since we have to take care of the torsion unit, `GenRaw` will have 4 columns.

```
> #c[1]‘S;
2
> UnitRank(o);
1
> Ncols(c[1]‘UnitsRaw);
4
> U := c[1]‘UnitsRaw;
> u := [ &* [B[1][i] ^ U[i][j] : i in [1..Ncols(B)]] : j in [1..4]];
```

Now u_1 should be a fundamental unit for Oa, u_2 and u_3 are S-units and u_4 is the torsion unit, since we adjoin the 3rd-roots of unity, this will be a 6th root of unity.

```
> Oa := Parent(B[1][1]);
> Oa!u[1];
[0, -11, 4, 2]
> IsUnit($1);
true
> Decomposition(u[2]);
[
    <Prime Ideal of Oa
    Two element generators:
        [3, 0, 0, 0]
        [-9, 0, 1, 2], 1>
]
> u[4]^6 eq 1;
```

Now O should be an extension of Oa:

```
> c[1]‘O:Maximal;
         F[1]
           /
          /
      E2[1]
        /
       /
   E1[1]
  /
 /
Q
F  [ 1]     x^3 + [[0, -4], [11, -2]]
E 2[ 1]     x^2 + x + [1, 0]
E 1[ 1]     x^2 + 10
Index : <[[1, 0], [0, 0]]>
```

The class field K we are seeking is a subfield of O:

```
> c[1]‘ClassField;
Equation Order with defining polynomial x^3 + [-3, 0]*x + [-11,
```

```
    -6] over o
> g := c[1]‘ClassField.2;
> c[1]‘O!g;
[[[0, 0], [0, 0]], [[1, 0], [0, 0]], [[11, 2], [11, -2]]]
```

Note that the absolute term of the defining polynomial is a. Since `GenAut` is an image of g, its minimal polynomial should be the same as that of g:

```
> MinimalPolynomial(g);
$.1^3 - 3/1*o.1*$.1 - 11/1*o.1 - 6/1*o.2
> Evaluate($1, c[1]‘GenAut);
0
```

39.9 Group Theoretic Functions

39.9.1 Generic Groups

Quite frequently in computational algebra one constructs a set of objects that generate a group under some operation. Generic groups are finite groups that are defined by generators that have implicit relations. In order to use them, one has to provide a function for the multiplication of two elements and one to check equality. If known, the identity object can also be passed in.

Generic groups are used in the class field package for the automorphism groups. A frequent situation is that one knows certain automorphisms (as maps) and would like to get the group generated by them. If the group is reasonably small, this can be done using the functions in this section.

All functions here rely on the group being small enough to allow complete enumeration of all elements.

The main application are situations where multiplication of the actual objects is time consuming so one would like to transfer as much as possible to some abstract finite group.

`GenericGroup(X)`

`Mult`	INTRINSIC	*Default :* $'*'$
`Eq`	INTRINSIC	*Default :* $'eq'$
`Id`	ANY	*Default :*
Verbose	`GrpGen`	*Maximum :* 3

Creates the group G generated by the elements of `X`. The function assumes that the group is finite. The second return value is a map from G onto a list of elements of G which are of the same type as the elements of `X`.

Since this function will enumerate all group elements, the group cannot be too large.

`AddGenerator(G, x)`

Verbose `GrpGen` *Maximum :* 3

Adds a new generator `x` to G. If `x` was already in G, the value `false` is returned and the other return values are unassigned. Otherwise, the new group and the corresponding map is returned.

G has to be a generic group as returned by `GenericGroup`.

The function applies a version of Dimino's algorithm [But91a] to find all elements of G with as few operations as possible.

`FindGenerators(G)`

Given a generic group G as returned by `GenericGroup`, find a small set of generators.

39.10 Bibliography

[**But91a**] Gregory Butler. *Dimino's Algorithm*, pages 13 – 23. Volume 559 of *LNCS* [But91b], 1991.

[**But91b**] Gregory Butler. *Fundamental Algorithms for Permutation Groups*, volume 559 of *LNCS*. Springer-Verlag, 1991.

[**CDO96**] Henri Cohen, Francisco Diaz y Diaz, and Michel Olivier. Computing Ray Class Groups, Conductors and Discriminants. In Cohen [Coh96], pages 52–59.

[**CDO97**] Henri Cohen, Francisco Diaz y Diaz, and Michel Olivier. Computing Ray Class Groups, Conductors and Discriminants. *Submitted to Math. Comp.*, 1997.

[**Coh96**] Henri Cohen, editor. *ANTS II*, volume 1122 of *LNCS*. Springer-Verlag, 1996.

[**Coh00**] Henri Cohen. *Advanced Topics in Computational Number Theory*. Springer, Berlin–Heidelberg–New York, 2000.

[**Fie00**] Claus Fieker. Computing Class Fields via the Artin Map. *Math. Comput.*, 70(235):1293–1303, 2000.

[**HPP97**] Florian Heß, Sebastian Pauli, and Michael E. Pohst. On the computation of the multiplicative group of residue class rings. *Math. Comp.*, 1997.

[**Pau96**] Sebastian Pauli. Zur Berechnung von Strahlklassengruppen. Diplomarbeit, Technische Universität Berlin, 1996.
URL:http://www.math.tu-berlin.de/~kant/publications/diplom/pauli.ps.gz.

[**Sut12**] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47(5):552–567, 2012.

40 ALGEBRAICALLY CLOSED FIELDS

Chapter 40
ALGEBRAICALLY CLOSED FIELDS

40.1 Introduction

MAGMA contains a system [Ste02, Ste10] for computing with algebraically closed fields, which have the property that they always contain all the roots of any polynomial defined over them. It is of course not possible to construct explicitly the closure of a field, but the system works by automatically constructing larger and larger algebraic extensions of an original base field as needed during a computation, thus giving the illusion of computing in the algebraic closure of the base field.

Such a system was already proposed before (the D5 system [DDD85]), but this has difficulty with the parallelism which occurs when one must compute with several conjugates of a root of a reducible polynomial, leading to situations where a certain expression evaluated at a root is invertible but evaluated at a conjugate of that root is not invertible.

MAGMA's system has no such problem and one can compute with the field just like any other field in MAGMA; all standard algorithms which work over generic fields or which use factorization work automatically without having to be adapted to handle the many conjugates of a root.

Especially significant is also the fact that all the Gröbner basis algorithms work well over such fields. One can compute the variety of any zero-dimensional multivariate polynomial ideal over the algebraic closure of its base field. Puiseux expansions of polynomials are also successfully computed using an algebraically closed field.

40.2 Representation

An algebraically closed field is based on an affine algebra (or quotient ring of a multivariate polynomial ring by an ideal of "relation" polynomials). The defining polynomials of this affine algebra are not necessarily irreducible – the system avoids factorization over an algebraic number field when possible, and automatically splits the defining polynomials of the affine algebra when factors are found during computations with the field. These factors often arise automatically because of the structure of the algorithm which is computing over the field.

Many technical optimizations have been designed to make the system practical. For example, the most expensive arithmetic operation by far in the whole system is, quite surprisingly, the testing of whether an element of the field is zero or not (which is utterly trivial for most other rings, of course)! To allow for the parallelism amongst conjugates to work, MAGMA performs a recursive GCD computation with the element, considered as a polynomial in its highest variable, and the appropriate defining polynomial, to determine the result. If the GCD is non-trivial, then this forces a splitting of the defining polynomial,

all elements of the field are reduced, and the original element may now be zero. More details concerning the internal design of the system can be found in [Ste10].

Care must be taken with the interpretation of the roots of a polynomial in this system. The roots of polynomials are only defined algebraically, and the user may wish to identify them with some particular elements of the complex field, for example, but one cannot assume that the system will follow the embedding one wishes. An example will demonstrate. Suppose that α is a root of $x^2 - 2$, β is a root of $x^2 - 3$, and γ is a root of $x^2 - 6$. Does $\gamma = \alpha \cdot \beta$ or does $\gamma = -\alpha \cdot \beta$? The system will have to make a choice between the two possibilities if the situation arises, but the choice which it will make cannot be predicted beforehand. That is, even if we might like to interpret things as $\alpha = \sqrt{2}$, $\beta = \sqrt{3}$ and $\gamma = \sqrt{6}$ (referring to the positive real roots in each case), it cannot be assumed that this will hold in the particular algebraically closed field, since the roots are only defined algebraically.

In the following descriptions, the adjective *invariable*, applied to a value, means that the mathematical value of a function will not change, despite simplifications, etc. which may occur after the function is called. That is, the value returned by the function may be considered constant with respect to the `eq` operator. A value may print differently later (because of simplifications of the field), but the mathematical value will never change.

In contrast, the adjective *variable* means that the mathematical value of a function may change (as a result of simplifications). This is usually only a property of access-like functions like `Degree(A, v)` (see below). But the fact that most functions below have invariable return values enables the illusion of a true field to be sustained.

Separate algebraic fields should be created for separate problems which involve root taking (if the roots of the different problems are unrelated). Otherwise it may be unnecessarily expensive to compute in the field with roots of unrelated polynomials.

Currently, the base field of an algebraic closure may be a finite field, the rational field **Q**, or a rational function field over a finite field or **Q**.

40.3 Creation of Structures

`AlgebraicClosure(K)`

Create the algebraic closure **A** of the field K. Currently K may be **Q**, a finite field or a rational function field over a finite field or the rational field.

`AlgebraicClosure()`

Create the algebraic closure **A** of the rational field **Q**.

`AssignNamePrefix(A, S)`

(Procedure.) Given an algebraically closed field A and a string S, reassign the string prefix of the names of A to be S. By default, this prefix is `"r"`. When new variables are introduced by root taking, they are named by this prefix with the number appended (see `A.i` below).

40.4 Creation of Elements

The usual way of creating elements within an algebraically closed field A is by coercion from the base field into A, or by construction of roots of polynomials over A (and this may be done indirectly via other functions).

40.4.1 Coercion

`A ! a`

Given a finite field A create the element specified by a; here a is allowed to be an element coercible into A, which means that a may be

(i) an element of A;

(ii) an integer or rational.

`One(A)` `Identity(A)`

`Zero(A)` `Representative(A)`

These generic functions create `A!1`, `A!1`, `A!0` and `A!0`, respectively.

40.4.2 Roots

`Roots(f)`

`Roots(f, A)`

`Max` RngIntElt *Default :*

Given a polynomial f over an algebraically closed field A, or given a polynomial f over some subring of A together with A itself, this function computes all roots of f in A, and returns a sorted sequence of tuples (pairs), each consisting of a root of f in A and its multiplicity. Since A is algebraically closed, f always splits completely.

If the parameter `Max` is set to a non-negative number m, at most m roots are returned. This feature can be quite useful when one wishes, say, only one root of a polynomial and not all the conjugates of the root, as they will cause the field to have more variables than necessary and this can make the full simplification of the field much more difficult later (if that is requested).

Note that the function `Factorization(f)` is also supported, and simply returns the linear factors and multiplicities corresponding to the roots returned by `Roots(f)`.

`RootOfUnity(n, A)`

Return a primitive n-th root of unity in A, i.e., an element $\omega \in A$ such that $\omega^n = 1$ and $\omega^i \neq 1$ for $1 \leq i < n$. This always exists since the field is algebraically closed, and the return value is invariable. This function is equivalent to `Roots(CyclotomicPolynomial(n), A: Max := 1)[1, 1]`.

`SquareRoot(a)`

`Sqrt(a)`

A square root of the element a from the field A, i.e., an element y of A such that $y^2 = a$. A square root always exists since the field is algebraically closed, and the return value is invariable.

`IsSquare(a)`

Return `true` and a square root of the element a from the field A, i.e., `true` and an element y of A such that $y^2 = a$. A square root always exists since the field is algebraically closed, and the return value is invariable.

`Root(a, n)`

Return an n-th root of the element a from the field A, i.e., an element y of A such that $y^n = a$. A root always exists since the field is algebraically closed, and the return value is invariable.

`IsPower(a, n)`

Return `true` and an n-th root of the element a from the field A, i.e., `true` and an element y of A such that $y^n = a$. A root always exists since the field is algebraically closed, and the return value is invariable.

40.4.3 Variables

`A . i`

Return the i-th variable of A. i must be between 1 and the rank of A (the current number of variables in A). Initially A has no variables, and new variables are only created by calling `Roots` above (or similar functions such as `Sqrt`). As long as `Prune` or `Absolutize` are not called (which shift the variable numbers – see below), the return value of this function is invariable, so $A.i$ for fixed i will always return the same mathematical object despite any simplifications or constructions of new roots. New roots are always assigned higher generator numbers.

Example H40E1

We first show the most common way of creating roots: by the `Roots` function.

```
> A := AlgebraicClosure();
> P<x> := PolynomialRing(IntegerRing());
> r := Roots(x^3 + x + 1, A);
> r;
[
    <r1, 1>,
    <r2, 1>,
    <r3, 1>
]
> A;
```

```
Algebraically closed field with 3 variables
Defining relations:
[
    r3^3 + r3 + 1,
    r2^3 + r2 + 1,
    r1^3 + r1 + 1
]
> a := r[1,1];
> a^3 + a;
-1
> A.1;
r1
> A.2;
r2
> A.3;
r3
> A.1 eq a;
true
```

It is often useful to use the `Max` parameter with the `Roots` function. Note that in this case A does not have the extra variables found in the previous example.

```
> A := AlgebraicClosure();
> r := Roots(x^3 + x + 1, A: Max := 1);
> A;
Algebraically closed field with 1 variable
Defining relations:
[
    r1^3 + r1 + 1
]
```

One can also create elements by `Sqrt`, etc.

```
> A := AlgebraicClosure();
> sqrt2 := Sqrt(A ! 2);
> cube3 := Root(A!3, 3);
> A;
Algebraically closed field with 2 variables
Defining relations:
[
    r2^3 - 3,
    r1^2 - 2
]
> sqrt2^2;
2
> cube3^3;
3
```

Example H40E2

The n-th Swinnerton-Dyer polynomial is defined to be

$$\prod(x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \cdots \pm \sqrt{p_n}),$$

where p_i is the i-th prime and the product runs over all 2^n possible combinations of $+$ and $-$ signs. Such polynomials lie in $\mathbf{Z}[x]$ and are irreducible over $\mathbf{Z}$. It is very easy to compute them using algebraically closed fields. We simply construct the square roots we need and multiply out the expression, coercing the resulting polynomial to $\mathbf{Z}[x]$.

```
> Z := IntegerRing();
> function SwinnertonDyer(n)
>     P := [2];
>     for i := 2 to n do
>         Append(~P, NextPrime(P[#P]));
>     end for;
>     A := AlgebraicClosure();
>     S := [Sqrt(A ! p): p in P];
>     P<z> := PolynomialRing(A);
>     f := &*[z + &+[t[i]*S[i]: i in [1..n]]: t in CartesianPower({-1, 1}, n)];
>     return PolynomialRing(Z) ! f;
> end function;
> P<x> := PolynomialRing(Z);
> [SwinnertonDyer(i): i in [1..5]];
[
    x^2 - 2,
    x^4 - 10*x^2 + 1,
    x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576,
    x^16 - 136*x^14 + 6476*x^12 - 141912*x^10 + 1513334*x^8 -
        7453176*x^6 + 13950764*x^4 - 5596840*x^2 + 46225,
    x^32 - 448*x^30 + 84864*x^28 - 9028096*x^26 + 602397952*x^24 -
        26625650688*x^22 + 801918722048*x^20 - 16665641517056*x^18 +
        239210760462336*x^16 - 2349014746136576*x^14 +
        15459151516270592*x^12 - 65892492886671360*x^10 +
        172580952324702208*x^8 - 255690851718529024*x^6 +
        183876928237731840*x^4 - 44660812492570624*x^2 +
        2000989041197056
]
```

The Swinnerton-Dyer polynomials yield worse-case inputs for the Berlekamp-Zassenhaus factorization algorithm for polynomials over $\mathbf{Z}$, but they are no longer difficult to factor using van Hoeij's new algorithm (see Example H23E6).

We can even define a simple extension of the Swinnerton-Dyer polynomials. Let $Q = \{q_1, \ldots, q_n\}$ be a set of n distinct primes or negatives of primes. Define:

$$\mathrm{GSD}_Q := \prod(x \pm \sqrt{q_1} \pm \sqrt{q_2} \pm \cdots \pm \sqrt{q_n}),$$

where the product runs over all 2^n possible combinations of + and − signs. Then $\mathrm{GSD}_Q \in \mathbf{Z}[x]$, is irreducible over $\mathbf{Z}$, has degree 2^n, and has at least 2^{n-1} factors mod any prime. A function to compute these polynomials is only a slight variation on the previous function.

```
> function GSD(Q)
>     n := #Q;
>     A := AlgebraicClosure();
>     S := [Sqrt(A ! x): x in Q];
>     z := PolynomialRing(A).1;
>     f := &*[z + &+[t[i]*S[i]: i in [1..n]]: t in CartesianPower({-1, 1}, n)];
>     return PolynomialRing(Z) ! f;
> end function;
```

One can multiply GSD_Q for various Q to construct more reducible polynomials with many modular factors. We first note the effects of changing the sign of the input primes.

```
> GSD([2, 3, 5]);
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
> GSD([-2, -3, -5]);
x^8 + 40*x^6 + 352*x^4 + 960*x^2 + 576
> GSD([-2, 3, 5]);
x^8 - 24*x^6 + 224*x^4 + 960*x^2 + 1600
> GSD([2, -3, 5]);
x^8 - 16*x^6 + 184*x^4 + 960*x^2 + 3600
> GSD([2, 3, -5]);
x^8 + 152*x^4 + 1920*x^2 + 5776
```

We now form a polynomial f which is the product of two degree-64 irreducible polynomials. f has at least 64 factors modulo any prime, but is not difficult to factor using van Hoeij's algorithm.

```
> f := GSD([2, 3, 5, 7, 11, 13])*GSD([-2, -3, -5, -7, -11, -13]);
> Degree(f);
128
> Max([Abs(x): x in Coefficients(f)]);
74356932844713201802276382813294219572861455394629943303351262856515487990904 17
> time L:=Factorization(f);
Time: 9.850
> [Degree(t[1]): t in L];
[ 64, 64 ]
> Max([Abs(x): x in Coefficients(L[1,1])]);
17710807204306291616851589788921525994 56 11
```

Example H40E3

This example shows how one can compute Puiseux expansions over an algebraic closure. The `PuiseuxExpansion` function calls the `Roots` function internally as it needs to.

```
> A := AlgebraicClosure();
> S<y> := PuiseuxSeriesRing(A);
> P<x> := PolynomialRing(S);
```

```
> f := (x^2 - y^2 - 1)^5 + x*y + 1;
> time S := PuiseuxExpansion(f, 3);
Time: 0.210
> S;
[
    r1*y + (102/2525*r1 - 2/505)*y^3 + O(y^4),
    r2*y + (102/2525*r2 - 2/505)*y^3 + O(y^4),
    r3 + (1/10*r3^2 - 1/10)*y + (-101/1000*r3^7 + 101/200*r3^5 -
        209/200*r3^3 + 26/25*r3)*y^2 + O(y^3),
    r4 + (1/10*r4^2 - 1/10)*y + (-101/1000*r4^7 + 101/200*r4^5 -
        209/200*r4^3 + 26/25*r4)*y^2 + O(y^3),
    r5 + (1/10*r5^2 - 1/10)*y + (-101/1000*r5^7 + 101/200*r5^5 -
        209/200*r5^3 + 26/25*r5)*y^2 + O(y^3),
    r6 + (1/10*r6^2 - 1/10)*y + (-101/1000*r6^7 + 101/200*r6^5 -
        209/200*r6^3 + 26/25*r6)*y^2 + O(y^3),
    r7 + (1/10*r7^2 - 1/10)*y + (-101/1000*r7^7 + 101/200*r7^5 -
        209/200*r7^3 + 26/25*r7)*y^2 + O(y^3),
    r8 + (1/10*r8^2 - 1/10)*y + (-101/1000*r8^7 + 101/200*r8^5 -
        209/200*r8^3 + 26/25*r8)*y^2 + O(y^3),
    r9 + (1/10*r9^2 - 1/10)*y + (-101/1000*r9^7 + 101/200*r9^5 -
        209/200*r9^3 + 26/25*r9)*y^2 + O(y^3),
    r10 + (1/10*r10^2 - 1/10)*y + (-101/1000*r10^7 + 101/200*r10^5 -
        209/200*r10^3 + 26/25*r10)*y^2 + O(y^3)
]
> A;
Algebraically closed field with 10 variables
Defining relations:
[
    r10^8 - 5*r10^6 + 10*r10^4 - 10*r10^2 + 5,
    r9^8 - 5*r9^6 + 10*r9^4 - 10*r9^2 + 5,
    r8^8 - 5*r8^6 + 10*r8^4 - 10*r8^2 + 5,
    r7^8 - 5*r7^6 + 10*r7^4 - 10*r7^2 + 5,
    r6^8 - 5*r6^6 + 10*r6^4 - 10*r6^2 + 5,
    r5^8 - 5*r5^6 + 10*r5^4 - 10*r5^2 + 5,
    r4^8 - 5*r4^6 + 10*r4^4 - 10*r4^2 + 5,
    r3^8 - 5*r3^6 + 10*r3^4 - 10*r3^2 + 5,
    r2^2 + 1/5*r2 - 1,
    r1^2 + 1/5*r1 - 1
]
```

We check that f evaluated at each expansion in S is zero up to the precision.

```
> [Evaluate(f, p): p in S];
[
    O(y^5),
    O(y^5),
    O(y^3),
    O(y^3),
    O(y^3),
```

```
    O(y^3),
    O(y^3),
    O(y^3),
    O(y^3),
    O(y^3)
]
```

40.5 Related Structures

`Category(A)` `Parent(A)` `Centre(A)`
`PrimeRing(A)` `PrimeField(A)`
`FieldOfFractions(A)`

40.6 Properties

`BaseField(A)`

Return the base field over which A is defined.

`Rank(A)`

Return the current rank of A, that is, the number of variables which currently define A. This can increase by the construction of new roots, or decrease by pruning (see `Roots` and `Prune` respectively below), so the return value of this function is *variable*.

`Degree(A, v)`

Given an algebraically closed field A of rank r and an integer v in the range $1 \leq v \leq r$, return the current degree of the defining polynomial for variable v. The return value of this function is *variable*, as A may be simplified between invocations, making the defining polynomial for v have smaller degree.

`Degree(A)`

Return the current absolute degree of A, that is, the degree over its base field. This necessitates the simplification of A (see `Simplify` below), so may be very time consuming. The return value varies only when new roots of polynomials over the field are computed, but until then, the return value is invariable (as the field will remain simplified, even if `Prune` or `Absolutize` is called – see below).

AffineAlgebra(A)

QuotientRing(A)

Return the affine algebra (or multivariate quotient ring) R which currently represents A. The quotient relations of R consist of the defining polynomials A, and one may coerce between A and R, but note that the variable numbers are inverted. The reason for this is that for the system to work, the first root $A.1$ must be the smallest variable with respect to the lexicographical order in the corresponding affine algebra R, so that reductions modulo the Gröbner basis of relations are in the correct form.

Note also that if A changes in any way (whether from simplification or by pruning), then the affine algebra R of course stays the same and will not be comparable with the new form of A if A has a different number of variables than before.

Ideal(A)

Return the ideal of defining polynomials currently defining A. This is simply equivalent to `DivisorIdeal(AffineAlgebra(A))`. See the relevant comments for the function AffineAlgebra.

40.7 Ring Predicates and Properties

IsCommutative(A) IsUnitary(A)

IsFinite(A) IsOrdered(A)

IsField(A) IsEuclideanDomain(A)

IsPID(A) IsUFD(A)

IsDivisionRing(A) IsEuclideanRing(A)

IsPrincipalIdealRing(A) IsDomain(A)

A eq B A ne B

Characteristic(A)

40.8 Element Operations

40.8.1 Arithmetic Operators

This section lists the basic arithmetic operations available for elements of an algebraically closed field. Elements are always kept in normal form with respect to the defining relations of the field. Computing the inverse of an element may cause a simplification of the field to be performed.

```
+ a
- a
a + b
a - b
a * b
a / b
a ^ k
a +:= b
a -:= b
a *:= b
```

40.8.2 Equality and Membership

40.8.3 Parent and Category

```
Parent(a)
Category(a)
```

40.8.4 Predicates on Ring Elements

```
IsZero(a)
```

Return whether a is the zero element of its field. This is the most difficult of all arithmetic functions for algebraically closed fields! To determine whether a is zero, MAGMA computes the recursive GCD of a, considered as a polynomial in its highest variable, and the appropriate defining polynomial, to determine the result. If the GCD is non-trivial, then this forces a splitting of the defining polynomial, all elements of the field are reduced, and the original element may now be deemed to be zero (it may not be zero because the cofactor of the GCD may be used to perform the simplification). Despite the fact that simplifications may occur, the return value of this function is *invariable*, and this fact is the most important feature of the whole system, enabling the illusion of a true field to be achieved!

```
IsOne(a)
```

Return whether a is one in its field, which is determined by testing whether $(a-1)$ is zero. Consequently, a simplification of the field may occur, but the return value is invariable.

```
IsMinusOne(a)
```

Return whether a is minus one in its field, which is determined by testing whether $(a+1)$ is zero. Consequently, a simplification of the field may occur, but the return value is invariable.

`a eq b`

Return whether $a = b$, which is determined by testing whether $(a - b)$ is zero. Consequently, a simplification of the field may occur, but the return value is invariable.

`a ne b`

`a in A` `a notin A`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsUnit(a)` `IsZeroDivisor(a)` `IsRegular(a)`

`IsIrreducible(a)` `IsPrime(a)`

40.8.5 Minimal Polynomial, Norm and Trace

`MinimalPolynomial(a)`

Return the minimal polynomial of the element a of the field A, relative to the base field of A. This is the unique minimal-degree *irreducible* monic polynomial with coefficients in the base field, having a as a root.

This function works as follows. First the minimal polynomial M of a in the affine algebra corresponding to A is computed. M may be reducible in general, so M is factored, and for each irreducible factor F of M, $F(a)$ is evaluated and it is tested whether this evaluation is zero (using the `IsZero` algorithm). If M is not irreducible, some of these evaluations will cause simplifications of the field, but exactly one of the evaluations will be zero and the corresponding irreducible F is the minimal polynomial of a. Consequently, after F is returned, $F(a)$ will be identically zero so the return value of this function *is invariable*.

Thus the illusion of a true field is sustained by forcing the minimal polynomial of a to be irreducible, by first performing whatever simplifications of A are necessary for this. (In fact, computing minimal polynomials in this way is one method of achieving simplifications.)

`Norm(a)`

Given an element a from an algebraically closed field A, return the absolute norm of a to the base field of A. This is simply computed as $(-1)^{\text{Degree}(M)}$ times the constant coefficient of M, where M is the irreducible minimal polynomial of a returned by `MinimalPolynomial`(a). Consequently, a simplification of the field may occur, but the return value is invariable.

`Trace(a)`

Given an element a from an algebraically closed field A, return the absolute trace of a to the base field of A. This is simply computed as the negation of the coefficient of x^{n-2} in M, where M is the irreducible minimal polynomial of a returned by `MinimalPolynomial`(a). Consequently, a simplification of the field may occur, but the return value is invariable.

`Conjugates(a)`

Given an element a from an algebraically closed field A, return the conjugates of a as a sequence of elements. The conjugates of a are defined to be the roots in A of the minimal polynomial of a, and a is always included. This function is thus simply equivalent to:

```
[t[1] :  t in Roots(MinimalPolynomial(a), A)].
```

(No multiplicities are returned as in the `Roots` function since the minimal polynomial is always squarefree, of course.) As this function first computes the minimal polynomials of a, a simplification of the field may occur, but the return value is invariable.

Example H40E4

We create two elements of an algebraically closed field and note that they are conjugate.

```
> A := AlgebraicClosure();
> x := Sqrt(A!2) + Sqrt(A!-3);
> y := Sqrt(A ! (-1 + 2*Sqrt(A!-6)));
> A;
Algebraically closed field with 4 variables
Defining relations:
[
    r4^2 - 2*r3 + 1,
    r3^2 + 6,
    r2^2 + 3,
    r1^2 - 2
]
> x;
r2 + r1
> y;
r4
> x eq y;
false
> Conjugates(x);
[
    r4,
    -r4,
    r5,
    r6
]
> y in Conjugates(x);
true
```

Of course, x and y are conjugate if and only if they have the same minimal polynomial, which is the case here:

```
> P<z> := PolynomialRing(RationalField());
```

```
> MinimalPolynomial(x);
z^4 + 2*z^2 + 25
> MinimalPolynomial(y);
z^4 + 2*z^2 + 25
```

40.9 Simplification

The following procedures allow one to simplify an algebraically closed field so that it is a true field.

`Simplify(A)`

`Partial` BoolElt *Default :* `false`

(Procedure.) Simplify the algebraically closed field A so that the affine algebra which represents it is a true field, modifying A in place. Equivalently, simplify A so that the multivariate polynomial ideal corresponding to the defining polynomials A is maximal.

The procedure first partially simplifies A by calling `MinimalPolynomial` on all variables and sums of two variables of A. This will usually cause many simplifications, since this forces the corresponding minimal polynomials to be irreducible (see `MinimalPolynomial`). The procedure then performs all other necessary simplifications by successively computing absolute representations and factorizing the absolute polynomials which arise. This may be very expensive (in particular, if the final absolute degree is greater than 20), so is only practical for fairly small degrees.

If the parameter `Partial` is set to `true`, then only the partial simplification is performed, which is usually rather fast, and may be sufficient.

`Prune(A)`

Prune the algebraically closed field A by removing useless variables, modifying A in place. That is, for each variable v of A such that its defining polynomial is a linear polynomial, remove v and the corresponding defining polynomial from A, and shift variables higher than v appropriately.

Note that elements of A are kept reduced to normal form with respect to the defining polynomials of A (at least according the user's perception), so for each such v having a linear relation, v cannot occur in any element of A, so the removal of v from A is possible.

40.10 Absolute Field

One may construct an absolute field isomorphic to the current subfield represented by an algebraically closed field. The construction of the absolute field may be very expensive, as it involves factoring polynomials over successive subfields. In fact, it is often the case that the degree of the absolute field is an extremely large integer, so that an absolute field is not practically representable, yet the system may allow one to compute effectively with the original non-absolute presentation.

`AbsoluteAffineAlgebra(A)`

`AbsoluteQuotientRing(A)`

Simplify the algebraically closed field A fully (see `Simplify(A)` above) and then return an absolute field as a univariate affine algebra R which is isomorphic to the current (true) algebraic field represented by A, and also return the isomorphism $f : A \to R$.

`AbsolutePolynomial(A)`

Simplify the algebraically closed field A fully (see `Simplify(A)` above) and then compute an absolute field isomorphic to the current (true) algebraic field represented by A and return the defining polynomial of the absolute field. That is, return a polynomial f such that $K[x]/ < f >$ is isomorphic to A in its current state.

`Absolutize(A)`

Modify the algebraically closed field A in place so that has an absolute presentation. That is, compute an isomorphic absolute field and absolute polynomial f as in `AbsolutePolynomial` and modify A and its elements in place so that A now only has one variable v and corresponding defining polynomial $f(v)$ and the elements of A correspond via the isomorphism to their old representation.

Example H40E5

We show how one can easily compute the number field over which the complete variety of the Cyclic-6 ideal can be defined.
We first create the ideal I over $\mathbf{Q}$ and compute its variety over A, the algebraic closure of $\mathbf{Q}$.

```
> P<a,b,c,d,e,f> := PolynomialRing(RationalField(), 6);
> B := [
>     a + b + c + d + e + f,
>     a*b + b*c + c*d + d*e + e*f + f*a,
>     a*b*c + b*c*d + c*d*e + d*e*f + e*f*a + f*a*b,
>     a*b*c*d + b*c*d*e + c*d*e*f + d*e*f*a + e*f*a*b + f*a*b*c,
>     a*b*c*d*e + b*c*d*e*f + c*d*e*f*a + d*e*f*a*b +
>         e*f*a*b*c + f*a*b*c*d,
>     a*b*c*d*e*f - 1];
> I := ideal<P | B>;
> time Groebner(I);
Time: 1.459
```

```
> A := AlgebraicClosure();
> time V := Variety(I, A);
Time: 4.219
> #V;
156
```

We now notice that there are 28 variables in A and we check that all elements of V satisfy the original polynomials.

```
> Rank(A);
28
> V[1];
<-1, -1, -1, -1, r1 + 4, -r1>
> V[156];
<r28^3 + 2*r28^2*r9 - 2*r9, -r28^3 - 2*r28^2*r9 + 2*r9, r9, -r28, r28, -r9>
> {Evaluate(f, v): v in V, f in B};
{
    0
}
```

We now simplify A to ensure that it represents a true field, and prune away useless variables now having linear defining polynomials.

```
> time Simplify(A);
Time: 3.330
> Prune(A);
> A;
Algebraically closed field with 3 variables
Defining relations:
[
    r3^2 - 1/3*r3*r2*r1 - 5/3*r3*r2 + 2/3*r3*r1 - 2/3*r3 + r2*r1 + 4*r2 + 1,
    r2^2 - r2*r1 - 4*r1 - 1,
    r1^2 + 4*r1 + 1
]
> V[1];
<-1, -1, -1, -1, r1 + 4, -r1>
> V[156];
<2/3*r3*r2*r1 + 7/3*r3*r2 - 1/3*r3*r1 - 2/3*r3 + 5/3*r2*r1 +
    19/3*r2 + 2/3*r1 + 4/3, -2/3*r3*r2*r1 - 7/3*r3*r2 + 1/3*r3*r1
    + 2/3*r3 - 5/3*r2*r1 - 19/3*r2 - 2/3*r1 - 4/3, -4/3*r2*r1 -
    14/3*r2 - 1/3*r1 - 2/3, 2/3*r3*r2*r1 + 7/3*r3*r2 - 1/3*r3*r1
    - 2/3*r3 - 5/3*r2*r1 - 19/3*r2 + 1/3*r1 - 1/3, -2/3*r3*r2*r1
    - 7/3*r3*r2 + 1/3*r3*r1 + 2/3*r3 + 5/3*r2*r1 + 19/3*r2 -
    1/3*r1 + 1/3, 4/3*r2*r1 + 14/3*r2 + 1/3*r1 + 2/3>
```

Finally we compute an absolute polynomial for A, and then modify A in place using `Absolutize` to make A be defined by one polynomial of degree 8.

```
> time AbsolutePolynomial(A);
x^8 + 4*x^6 - 6*x^4 + 4*x^2 + 1
Time: 0.080
```

```
> time Absolutize(A);
Time: 0.259
> A;
Algebraically closed field with 1 variables
Defining relations:
[
    r1^8 + 4*r1^6 - 6*r1^4 + 4*r1^2 + 1
]
> V[1];
<-1, -1, -1, -1, 1/2*r1^6 + 2*r1^4 - 7/2*r1^2 + 3, -1/2*r1^6 -
    2*r1^4 + 7/2*r1^2 + 1>
> V[156];
<r1^7 + 4*r1^5 - 6*r1^3 + 4*r1, -r1^7 - 4*r1^5 + 6*r1^3 - 4*r1,
    -1/4*r1^7 - 3/4*r1^5 + 11/4*r1^3 - 7/4*r1, -r1, r1, 1/4*r1^7
    + 3/4*r1^5 - 11/4*r1^3 + 7/4*r1>
> {Evaluate(f, v): v in V, f in B};
{
    0
}
```

Example H40E6

In this example we compute the splitting field of a certain polynomial of degree 8.
We first set f to a degree-8 polynomial using the database of polynomials with given Galois group. The Galois group has order 16, so we know that the splitting field will have absolute degree 16.

```
> P<x> := PolynomialRing(IntegerRing());
> load galpols;
Loading "/home/magma/libs/galpols/galpols"
> PolynomialWithGaloisGroup(8, 6);
x^8 - 2*x^7 - 9*x^6 + 10*x^5 + 22*x^4 - 14*x^3 - 15*x^2 + 2*x + 1
> f := $1;
> #GaloisGroup(f);
16
```

We next create an algebraic closure A and compute the roots of f over A.

```
> A := AlgebraicClosure();
> r := Roots(f, A);
> #r;
8
> A;
Algebraically closed field with 8 variables
Defining relations:
[
    r8^8 - 2*r8^7 - 9*r8^6 + 10*r8^5 + 22*r8^4 - 14*r8^3 - 15*r8^2 + 2*r8 + 1,
    r7^8 - 2*r7^7 - 9*r7^6 + 10*r7^5 + 22*r7^4 - 14*r7^3 - 15*r7^2 + 2*r7 + 1,
    r6^8 - 2*r6^7 - 9*r6^6 + 10*r6^5 + 22*r6^4 - 14*r6^3 - 15*r6^2 + 2*r6 + 1,
    r5^8 - 2*r5^7 - 9*r5^6 + 10*r5^5 + 22*r5^4 - 14*r5^3 - 15*r5^2 + 2*r5 + 1,
```

```
    r4^8 - 2*r4^7 - 9*r4^6 + 10*r4^5 + 22*r4^4 - 14*r4^3 - 15*r4^2 + 2*r4 + 1,
    r3^8 - 2*r3^7 - 9*r3^6 + 10*r3^5 + 22*r3^4 - 14*r3^3 - 15*r3^2 + 2*r3 + 1,
    r2^8 - 2*r2^7 - 9*r2^6 + 10*r2^5 + 22*r2^4 - 14*r2^3 - 15*r2^2 + 2*r2 + 1,
    r1^8 - 2*r1^7 - 9*r1^6 + 10*r1^5 + 22*r1^4 - 14*r1^3 - 15*r1^2 + 2*r1 + 1
]
```

Finally we simplify A. There are defining polynomials of degrees 2 and 8 in the simplified field. The absolute polynomial of degree 16 defines the splitting field of f.

```
> time Simplify(A);
Time: 2.870
> A;
Algebraically closed field with 8 variables
Defining relations:
[
    r8 + 1/2*r3*r1^6 - 2*r3*r1^5 - r3*r1^4 + 8*r3*r1^3 - 2*r3*r1^2 -
        5*r3*r1 - 1/2*r3 + r1^7 - 3/2*r1^6 - 9*r1^5 + 4*r1^4 + 19*r1^3
        - r1^2 - 9*r1 - 5/2,
    r7 - 1/2*r3*r1^6 + 2*r3*r1^5 + r3*r1^4 - 8*r3*r1^3 + 2*r3*r1^2 +
        5*r3*r1 + 1/2*r3,
    r6 + r3 - 3/2*r1^7 + 2*r1^6 + 14*r1^5 - 5*r1^4 - 28*r1^3 + 3*r1^2
        + 19/2*r1,
    r5 - 1/2*r3*r1^6 + 2*r3*r1^5 + r3*r1^4 - 8*r3*r1^3 + 2*r3*r1^2 +
        4*r3*r1 + 1/2*r3 + r1^6 - r1^5 - 9*r1^4 - r1^3 + 14*r1^2 +
        6*r1,
    r4 + 1/2*r3*r1^6 - 2*r3*r1^5 - r3*r1^4 + 8*r3*r1^3 - 2*r3*r1^2 -
        4*r3*r1 - 1/2*r3 - 1,
    r3^2 - 3/2*r3*r1^7 + 2*r3*r1^6 + 14*r3*r1^5 - 5*r3*r1^4 -
        28*r3*r1^3 + 3*r3*r1^2 + 19/2*r3*r1 + 3/2*r1^6 - r1^5 -
        15*r1^4 - 4*r1^3 + 27*r1^2 + 11*r1 - 9/2,
    r2 + 1/2*r1^7 - 3/2*r1^6 - 4*r1^5 + 10*r1^4 + 10*r1^3 - 16*r1^2 -
        11/2*r1 + 3/2,
    r1^8 - 2*r1^7 - 9*r1^6 + 10*r1^5 + 22*r1^4 - 14*r1^3 - 15*r1^2 +
        2*r1 + 1
]
> AbsolutePolynomial(A);
x^16 - 36*x^14 + 488*x^12 - 3186*x^10 + 10920*x^8 - 19804*x^6 + 17801*x^4 -
    6264*x^2 + 64
```

40.11 Bibliography

[**DDD85**] J. Della Dora, C. Dicrescenzo, and D. Duval. About a new method for computing in algebraic number fields. In B.F. Caviness, editor, *Proc. EUROCAL '85*, volume 204 of *LNCS*, pages 289–290, Linz, 1985. Springer.

[**FK02**] Claus Fieker and David R. Kohel, editors. *ANTS V*, volume 2369 of *LNCS*. Springer-Verlag, 2002.

[**Ste02**] Allan Steel. A new scheme for computing with algebraically closed fields. In Fieker and Kohel [FK02], pages 491–505.

[**Ste10**] Allan K. Steel. Computing with algebraically closed fields. *J. Symb. Comput.*, 45(3):342–372, March 2010.

41 RATIONAL FUNCTION FIELDS

Chapter 41
RATIONAL FUNCTION FIELDS

41.1 Introduction

Given a ring R such that there is a greatest-common-divisor algorithm for polynomials over R, MAGMA allows the construction of a rational function field K in any number of indeterminates over R. Such function fields are objects of type `FldFunRat` with elements of type `FldFunRatElt`. The elements of K are fractions whose numerators and denominators lie in the corresponding polynomial ring over R. As for polynomial rings, the different univariate and multivariate cases are distinguished, since the fractions just use the different representations given by the different cases of polynomial rings.

A fraction f lying in a function field K is always *reduced*; this means that the numerator and denominator of f are coprime and the denominator of f is *normalized* (monic over fields and positive over $\mathbf{Z}$). Note that R itself need not be a field. Thus it is possible, for example, to create the rational function field $K = \mathbf{Z}(t)$ which is mathematically equal to $\mathbf{Q}(t)$ of course, but will be represented slightly differently. A fraction in $\mathbf{Q}(t)$ will have a monic denominator (and the coefficients of both the numerator and denominator may be non-integral), while a fraction in $\mathbf{Z}(t)$ will have a positive denominator (and the coefficients of both the numerator and denominator will be integral). Thus the fractions $(3t+2)/(4t-2) \in \mathbf{Z}(t)$ and $((3/4)t+1/2)/(t-1/2) \in \mathbf{Q}(t)$ are equal and are both reduced in their respective fields. It is generally much better to use the domain of integers instead of the field of fractions for the coefficient ring R (so it is better to use $\mathbf{Z}(t)$ instead of $\mathbf{Q}(t)$) since arithmetic is much faster, but the use of a field of fractions for the coefficient ring may be more desirable for output purposes.

41.2 Creation Functions

41.2.1 Creation of Structures

`FunctionField(R)`

`RationalFunctionField(R)`

`Global` BOOLELT *Default :* `true`

Create the field $\mathbf{F}$ of rational functions in 1 indeterminate (consisting of quotients of univariate polynomials) over the integral domain R. The angle bracket notation may be used to assign names to the indeterminates, just as in the case of polynomial rings, e.g.: `K<t> := FunctionField(IntegerRing());`.

By default, the unique *global* univariate function field over R will be returned; if the parameter `Global` is set to `false`, then a non-global univariate function field over R will be returned (to which a separate name for the indeterminate can be assigned).

`FunctionField(R, r)`

`RationalFunctionField(R, r)`

`Global` BoolElt *Default :* `false`

Create the field **F** of rational functions in r indeterminates over the integral domain R. may be used to assign names to the indeterminates, just as in the case of polynomial rings, e.g.: `K<a,b,c> := FunctionField(IntegerRing(), 3);`.

By default, a *non-global* function field will be returned; if the parameter `Global` is set to `true`, then the unique global function field over R with n variables will be returned. This may be useful in some contexts, but a non-global result is returned by default since one often wishes to have several function fields with the same numbers of variables but with different variable names (and create mappings between them, for example). Explicit coercion is always allowed between function fields having the same number of variables (and suitable base rings), whether they are global or not, and the coercion maps the i-variable of one function field to the i-th variable of the other function field.

`FieldOfFractions(P)`

Given a polynomial ring P, return its field of fractions F, consisting of quotients f/g, with $f, g \in P$. The angle bracket notation may be used to assign names to the indeterminates, just as in the case of polynomial rings: `K<t> := FieldOfFractions(P);`. If this function is called more than once for a fixed P, then the identical function field will be returned each time.

41.2.2 Names

`AssignNames(~F, s)`

Procedure to change the name of the indeterminates of a function field F. The i-th indeterminate will be given the name of the i-th element of the sequence of strings s (for $1 \leq i \leq \#s$); the sequence may have length less than the number of indeterminates of F, in which case the remaining indeterminate names remain unchanged.

This procedure only changes the name used in printing the elements of F. It does *not* assign to identifiers corresponding to the strings the indeterminates in F; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies F, it is necessary to have a reference `~F` to F in the call to this function.

`Name(F, i)`

Given a function field F, return the i-th indeterminate of F (as an element of F).

41.2.3 Creation of Elements

`F ! [a, b]`

`elt< F | a, b >`

Given the rational function field F (which is the field of fractions of the polynomial ring R), and polynomials a, b in R (with $b \neq 0$), construct the rational function a/b.

`F ! a`

Given the rational function field F as a field of fractions of R, and a polynomial $a \in R$, create the rational function $a = a/1$ in F.

`K . i`

The i-th generator for the field of fractions K of R over the coefficient ring of R.

`One(F)` `Identity(F)`

`Zero(F)` `Representative(F)`

Example H41E1

We create the field of rational functions over the integers in a single variable w.

```
> R<x> := PolynomialRing(Integers());
> F<w> := FieldOfFractions(R);
> F ! x+3;
w + 3
> F ! [ x, x-1 ];
w/(w - 1)
```

41.3 Structure Operations

41.3.1 Related Structures

`IntegerRing(F)`

`RingOfIntegers(F)`

Given the rational function field F this returns the polynomial ring from which F was constructed as its field of fractions.

`BaseRing(F)`

`CoefficientRing(F)`

The coefficient ring of the (ring of integers of) the rational function field F.

`Rank(F)`

The rank (number of indeterminates) of the rational function field F.

`ValuationRing(F)`

Given the rational function field F for which the coefficients come from a field, this returns the valuation ring of F with respect to the valuation given by the degree. This valuation ring consists of those rational functions g/h for which the degree of h is greater than or equal to that of g.

`ValuationRing(F, f)`

Given the rational function field F for which the coefficients come from a field, and an irreducible polynomial f in the ring of integers of F, this returns the valuation ring of F with respect to the valuation associated with f. This valuation ring consists of those rational functions g/h for which f divides g but not h.

`Category(R)` `Parent(R)` `PrimeRing(R)`

41.3.2 Invariants

`Characteristic(F)`

41.3.3 Ring Predicates and Booleans

`IsCommutative(F)` `IsUnitary(F)`
`IsFinite(F)` `IsOrdered(F)`
`IsField(F)` `IsEuclideanDomain(F)`
`IsPID(F)` `IsUFD(F)`
`IsDivisionRing(F)` `IsEuclideanRing(F)`
`IsPrincipalIdealRing(F)` `IsDomain(F)`
`F eq G` `F ne G`

41.3.4 Homomorphisms

In its general form a ring homomorphism taking a function field $R(x_1, \ldots, x_n)$ as domain requires $n + 1$ pieces of information, namely, a map (homomorphism) telling how to map the coefficient ring R together with the images of the n indeterminates.

`hom< P -> S | f, y1, ..., yn >`

`hom< P -> S | y1, ..., yn >`

Given a function field $F = R(x_1, \ldots, x_n)$, a ring S, a map $f : F \to S$ and n elements $y_1, \ldots, y_n \in S$, create the homomorphism $g : F \to S$ by applying the rules of $g(rx_1^{a_1} \cdots x_n^{a_n}) = f(r)y_1^{a_1} \cdots y_n^{a_n}$ for monomials, linearity for polynomials, i.e., $g(M + N) = g(M) + g(N)$, and division for fractions, i.e., $g(n/d) = g(n)/g(d)$.

The coefficient ring map may be omitted, in which case the coefficients are mapped into S by the unitary homomorphism sending 1_R to 1_S. Also, the images y_i are allowed to be from a structure that allows automatic coercion into S.

Example H41E2

In this example we map $\mathbf{Q}(x, y)$ into the number field $\mathbf{Q}(\sqrt[3]{2}, \sqrt{5})$ by sending x to $\sqrt[3]{2}$ and y to $\sqrt{5}$ and the identity map on the coefficients (which we omit).

```
> Q := RationalField();
> F<x, y> := FunctionField(Q, 2);
> A<a> := PolynomialRing(IntegerRing());
> N<z, w> := NumberField([a^3-2, a^2+5]);
> h := hom< F -> N | z, w >;
> h(x^11*y^3-x+4/5*y-13/4);
-40*w*z^2 - z + 4/5*w - 13/4
> h(x/3);
1/3*z
> h(1/x);
1/2*z^2
> 1/z;
1/2*z^2
```

41.4 Element Operations

41.4.1 Arithmetic

`+ a` `- a`

`a + b` `a - b` `a * b` `a / b` `a ^ k`

41.4.2 Equality and Membership

`a eq b` `a ne b`

`a in F` `a notin F`

41.4.3 Numerator, Denominator and Degree

`Numerator(f)`

Given a rational function $f \in K$, the field of fractions of R, return the numerator P of $f = P/Q$ as an element of the polynomial ring R.

`Denominator(f)`

Given a rational function $f \in K$, the field of fractions of R, return the denominator Q of $f = P/Q$ as an element of the polynomial ring R.

`Degree(f)`

Given a rational function f in a univariate function field, return the degree of f as an integer (the maximum of the degree of the numerator of f and the degree of the denominator of f).

`TotalDegree(f)`

Given a rational function f in a multivariate function field, return the total degree of f as an integer (the total degree of the numerator of f minus the total degree of the denominator of f).

`WeightedDegree(f)`

Given a rational function f in a multivariate function field, return the weighted degree of f as an integer (the weighted degree of the numerator of f minus the weighted degree of the denominator of f).

41.4.4 Predicates on Ring Elements

`IsZero(a)` `IsOne(a)` `IsMinusOne(a)`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsUnit(a)` `IsZeroDivisor(a)` `IsRegular(a)`

41.4.5 Evaluation

`Evaluate(f, r)`

Given a univariate rational function f in F, return the rational function in F obtained by evaluating the indeterminate in r, which must be from (or coercible into) the coefficient ring of the integers of F.

`Evaluate(f, v, r)`

Given a multivariate rational function f in F, return the rational function in F obtained by evaluating the v-th variable in r, which must be from (or coercible into) the coefficient ring of the integers of F.

41.4.6 Derivative

`Derivative(f)`

Given a univariate rational function f, return the first derivative of f with respect to its variable.

`Derivative(f, k)`

Given a univariate rational function f, return the k-th derivative of f with respect to its variable. k must be non-negative.

`Derivative(f, v)`

Given a multivariate rational function f, return the first derivative of f with respect to variable number v.

`Derivative(f, v, k)`

Given a multivariate rational function f, return the k-th derivative of f with respect to variable number v. k must be non-negative.

41.4.7 Partial Fraction Decomposition

`PartialFractionDecomposition(f)`

Given a univariate rational function f in $F = K(x)$, return the (unique) complete partial fraction decomposition of f. The decomposition is returned as a (sorted) sequence Q consisting of triples, each of which is of the form $< d, k, n >$ where d is the denominator, k is the multiplicity of the denominator, and n is the corresponding numerator, and also d is irreducible and the degree of n is strictly less than the degree of d. Thus f equals the sum of the ${n_t}/{d_t}^{k_t}$, where t ranges over the triples contained in Q. If f is improper (the degree of its numerator is greater than or equal to the degree of its denominator), then the first triple of Q will be of the form $< 1, 1, q >$ where q is the quotient of the numerator of f by the denominator of f.

`SquarefreePartialFractionDecomposition(f)`

Given a univariate rational function f in $F = K(x)$, return the (unique) complete squarefree partial fraction decomposition of f. The decomposition is returned as a (sorted) sequence Q consisting of triples, each of which is of the form $< d, k, n >$ where d is the denominator, k is the multiplicity of the denominator, and n is the corresponding numerator, and also d is squarefree and the degree of n is strictly less than the degree of d. Thus f equals the sum of the ${n_t}/{d_t}^{k_t}$, where t ranges over the triples contained in Q. If f is improper (the degree of its numerator is greater than or equal to the degree of its denominator), then the first triple of Q will be of the form $< 1, 1, q >$ where q is the quotient of the numerator of f by the denominator of f.

Example H41E3

We compute the squarefree and complete (irreducible) partial fraction decompositions of a fraction in $Q(t)$.

```
> F<t> := FunctionField(RationalField());
> P<x> := IntegerRing(F);
> f := ((t + 1)^8 - 1) / ((t^3 - 1)*(t + 1)^2*(t^2 - 4)^2);
> SD := SquarefreePartialFractionDecomposition(f);
> SD;
[
    <x^4 + 2*x^3 - x - 2, 1, 467/196*x^3 + 1371/196*x^2 +
        1391/196*x + 234/49>,
    <x^2 - x - 2, 1, -271/196*x + 505/98>,
    <x^2 - x - 2, 2, 271/14*x + 139/7>
]
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in SD] eq f;
> D := PartialFractionDecomposition(f);
> D;
[
    <x - 2, 1, -3683/2646>,
    <x - 2, 2, 410/63>,
    <x - 1, 1, 85/36>,
    <x + 1, 1, 1/108>,
    <x + 1, 2, 1/18>,
    <x + 2, 1, 1/18>,
    <x^2 + x + 1, 1, -5/147*x - 8/147>
]
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in D] eq f;
true
```

Note that doing the same operation in the function field $Z(t)$ must modify the numerators and denominators to be integral but the result is otherwise the same.

```
> F<t> := FunctionField(IntegerRing());
> P<x> := IntegerRing(F);
> f := ((t + 1)^8 - 1) / ((t^3 - 1)*(t + 1)^2*(t^2 - 4)^2);
> D := PartialFractionDecomposition(f);
> D;
[
    <2646*x - 5292, 1, -3683>,
    <63*x - 126, 2, 25830>,
    <36*x - 36, 1, 85>,
    <108*x + 108, 1, 1>,
    <18*x + 18, 2, 18>,
    <18*x + 36, 1, 1>,
    <147*x^2 + 147*x + 147, 1, -5*x - 8>
]
```

```
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in D] eq f;
true
```

Finally, we compute the partial fraction decomposition of a fraction in a function field whose coefficient ring is a multivariate function field.

```
> R<a, b> := FunctionField(IntegerRing(), 2);
> F<t> := FunctionField(R);
> P<x> := IntegerRing(F);
> f := 1 / ((t^2 - a)^2*(t + b)^2*t^3);
> SD := SquarefreePartialFractionDecomposition(f);
> SD;
[
    <x^3 + b*x^2 - a*x - a*b, 1, (-3*a - 2*b^2)/(a^3*b^4)*x^2 +
        (-a - 2*b^2)/(a^3*b^3)*x + (3*a + 3*b^2)/(a^2*b^4)>,
    <x^3 + b*x^2 - a*x - a*b, 2, (a + b^2)/(a^2*b^3)*x^2 +
        1/a^2*x + (-a - b^2)/(a*b^3)>,
    <x, 1, (3*a + 2*b^2)/(a^3*b^4)>,
    <x, 2, -2/(a^2*b^3)>,
    <x, 3, 1/(a^2*b^2)>
]
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in SD] eq f;
true
> D := PartialFractionDecomposition(f);
> D;
[
    <x, 1, (3*a + 2*b^2)/(a^3*b^4)>,
    <x, 2, -2/(a^2*b^3)>,
    <x, 3, 1/(a^2*b^2)>,
    <x + b, 1, (-3*a + 7*b^2)/(a^3*b^4 - 3*a^2*b^6 + 3*a*b^8 -
        b^10)>,
    <x + b, 2, -1/(a^2*b^3 - 2*a*b^5 + b^7)>,
    <x^2 - a, 1, (-3*a^2 - 3*a*b^2 + 2*b^4)/(a^6 - 3*a^5*b^2 +
        3*a^4*b^4 - a^3*b^6)*x + (6*a*b - 2*b^3)/(a^5 - 3*a^4*b^2
        + 3*a^3*b^4 - a^2*b^6)>,
    <x^2 - a, 2, (a + b^2)/(a^4 - 2*a^3*b^2 + a^2*b^4)*x -
        2*b/(a^3 - 2*a^2*b^2 + a*b^4)>
]
> // Check appropriate sum equals f:
> &+[F!t[3] / F!t[1]^t[2]: t in D] eq f;
true
```

41.5 Padé-Hermite Approximants

41.5.1 Introduction

A given rational function $F(z) \in k(z)$ over a field k can be written as a power series $f(z)$ in the completion $k((z))$ of $k(z)$ at the place (z). It thus is a good approximation of $f(z)$ in the sense that $F(z)$ is equal to $f(z)$ up to infinite order.

Padé-Hermite approximants deal with the converse. Given a (formal) power series $f(z) \in k[[z]]$ and some non- negative integers n_P and n_Q, find polynomials $P(z), Q(z) \in k[z]$ of degrees at most n_P and n_Q, respectively, such that $P(z)*f(z)-Q(z) = O(z^{n_P+n_Q+1})$ holds. In other words, $P/Q(z)$ is an approximation for $f(z)$ for polynomials P and Q of limited degree in z. In this notation the pair $[P, Q]$ is known as the Padé-Hermite approximant for the power series tuple $(f, -1)$.

The idea of approximating one power series by two polynomials can be extended to approximating several power series at the same time as follows. Consider the vector $\underline{f}^T := (f_1, f_2, \ldots, f_m)^T$ in the m-dimensional vector space $k((z))^m$ over a power series ring. Let $\underline{n} = (n_1, n_2, \ldots, n_m)$ be an m-tuple of non-negative integers. A *Padé-Hermite approximant* of $\underline{f}^T(z)$ of type $\underline{n}$ is a non-zero vector of polynomials $\underline{P} = (P_1, P_2, \ldots, P_m)$ in $k[z]^m$ such that $\underline{P} \cdot \underline{f}^T = O(z^N)$, $N = n_1 + n_2 + \cdots + n_m + m - 1$, is satisfied. A non-trivial Padé-Hermite approximant always exists. The approximant is contained in the sub-space $V_{\underline{f},N} = \{\underline{Q} \in k[z]^m : \underline{Q} \cdot \underline{f}^T = O(z^N)\}$.

The implementation of Padé-Hermite approximants is based on [Der94] and [BL94]. They are implemented in MAGMA as sequences rather than vectors.

41.5.2 Ordering of Sequences

A Padé-Hermite approximant to some sequence of power series does not have to be unique. They can be ordered according to their maximum degree and their type of sequence. A sequence $\underline{P} = [P_1, P_2, ..., P_m]$ is of degree i if the maximum of the weak degrees of $P_1, P_2, \ldots, P_m$ is i. An extension of the definition maximum degree is when a distortion of non-negative integers $\underline{d} = [d_1, d_2, \ldots, d_m]$ on the degrees is allowed. In this case the maximum degree is defined as the maximum of $\deg P_i - d_i$. The type of $\underline{P}$ identifies the last P_i in $\underline{P}$ whose weak degree equals the maximum degree of $\underline{P}$. Again a distortion on the degrees is allowed.

`MaximumDegree(f)`

`Distortion` SEQENUM *Default :* []

MaximumDegree returns the degree of a sequence of polynomials or power series, defined as the maximum of the degrees of $f[i] - d[i]$, where d is the distortion. The value -infinity is returned in the case that f is weakly equal to the zero-sequence.

Example H41E4

```
> S<u> := PowerSeriesRing(Rationals());
> f := [u+u^2, 2+u^2+u^3,0];
```

```
> MaximumDegree(f);
3
> MaximumDegree(f:Distortion:=[]);
3
> MaximumDegree(f:Distortion:=[0,2,1]);
2
> MaximumDegree([S|0,0]);
-Infinity
> MaximumDegree([O(u)]);
-Infinity
```

`TypeOfSequence(f)`

`Distortion` SEQENUM *Default :* []

Returns the highest index i for those $f[i]$ whose (distorted) degree is weakly equal to the maximum of the degrees of all entries. The second integer returned is the maximum degree of the sequence.

Example H41E5

```
> S<u> := PowerSeriesRing(Rationals());
> f := [u+u^2, 2+u^2+u^3,0];
> TypeOfSequence(f);
2 3
> TypeOfSequence(f:Distortion:=[]);
2 3
> TypeOfSequence(f:Distortion:=[0,2,1]);
1 2
> TypeOfSequence([S|0,0]);
2 -Infinity
```

The Padé-Hermite approximant of $\underline{f}$ can be seen as an element of the free module $k[z]^m$ of rank m, or as an element of the sub-space $V_{\underline{f},N}$.

A free sub-model $V \subset k[z]^m$ is generated by m polynomial vectors $\underline{Q}_i$, $1 = 1, 2, \ldots, m$, such that $\underline{Q}_1(z), \underline{Q}_2(z), \ldots, \underline{Q}_m(z)$ form a minimal vector sequence of V. Such a sequence is defined as a sequence S of m vectors in V, such that $S[i]$ is a non-trivial polynomial vector in V of minimal degree of type i, for $i = 1, 2, \ldots, m$. A minimal vector sequence is not unique.

Two variations on a minimal vector sequence are implemented. The first allows a distortion as an attribute. The sequence is the based on the distorted maximal degree. The second attribute sets the positive power p of z in each of the $\underline{Q}_i$ as follows. Instead of considering $\underline{Q}_i(z)$ for each i, one considers $\underline{Q}_i(z^p)$.

`MinimalVectorSequence(f,n)`

`Distortion`	SeqEnum	*Default :* []
`Power`	RngIntElt	*Default :* 1

A minimal sequence of vectors $\underline{Q}_1, \underline{Q}_2, \ldots, \underline{Q}_m$ with respect to the sequence f of length m whose entries are polynomials or power series. The order of $\underline{Q}_i \cdot f$ is at least m.

Example H41E6

```
> S<u> := PowerSeriesRing(Rationals());
> f := [u+u^2, 2+u^2+u^3];
> seq := MinimalVectorSequence(f, 2);
> seq;
[
    (u 0),
    (   -1 1/2*u)
]
> sums := [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
> sums;
[
    u^2 + u^3,
    -u^2 + 1/2*u^3 + 1/2*u^4
]
> seq := MinimalVectorSequence(f,3);
> #seq eq 2,  seq[1], seq[2];
true (u^2   0)
(-1 + u  1/2*u)
```

Example H41E7

```
> L<x> := PolynomialRing(Rationals());
> f := [1+x, 3-x^2, 5+x+x^3-x^5];
> seq := MinimalVectorSequence(f, 10);
> seq[1];
(-2*x^2 + 6   -2*x - 2          0)
> sums := [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
> sums;
[
    0,
    0,
    x^10
]
```

Example H41E8

```
> S<u> := PowerSeriesRing(Rationals());
> f := [2*u^4,2+u^3+u^6];
> seq := MinimalVectorSequence(f, 10);
> seq;
[
    (1/2*u^6       0),
    (-1/2 - 1/4*u^3        1/2*u^4)
]
> sums := [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
> sums;
[
    u^10,
    1/2*u^10
]
```

Example H41E9

```
> S<u> := PowerSeriesRing(Rationals());
> f :=  [1+u-7*u^2, 6-3*u+1/2*u^2-u^3, 5-u+u^2];
> seq := MinimalVectorSequence(f, 5);
>  [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
[
    0,
    u^5,
    0
]
> seq := MinimalVectorSequence(f, 5:Distortion:=[2,0,1]);
> [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
[
    -7/2*u^5,
    u^5,
    0
]
> p:=2;
> seq := MinimalVectorSequence(f, 5:Distortion:=[2,0,1], Power:=p);
> sums := [&+([Q[i]*f[i]: i in [1..#f]]) : Q in seq];
> sums;
> mp:= map<S->S|  x :-> (IsWeaklyZero(x) select 0
>     else  &+([Coefficient(x,i)*(S.1)^(p*i) :  i in Exponents(x)]))
>       + (ISA(Type(v),RngIntElt) select O((S.1)^(p*v))
>   else S!0 where v := AbsolutePrecision(x))>;
> sums := [&+([mp(Q[i])*f[i]: i in [1..#f]]) : Q in seq];
> sums;
[
    u^6 + u^7 - 7*u^8,
```

```
    u^5 - 23/3*u^6,
    -5/3*u^5 + 35/3*u^6
]
```

41.5.3 Approximants

The Padé-Hermite approximant of type $\underline{d} = [d_1, d_2, \ldots, d_m]$ with respect to the tuple $\underline{f}^T \in k((z))^m$ is an element of the space $V_{\underline{f},N} = \{\underline{Q} \in k[z]^m : \underline{Q} \cdot \underline{f}^T = O(z^N)\}$ for N equal to $d_1 + d_2 + \cdots + d_m + m - 1$. This space is generated by the vectors in the minimal vector sequence with respect to $\underline{f}$ with distortion $\underline{d}$. The routine PadeHermiteApproximant returns one that is smallest with respect to the degree on sequences. The input sequence $\underline{f}$ must be a sequence of polynomial ring elements, or be a power series sequence. While the Padé Hermite approximants theoretically are polynomials, MAGMA returns them as elements of the same ring the entries of $\underline{f}$ are contained in.

`PadeHermiteApproximant(f,d)`

`Power` RNGINTELT *Default :* 1

Returns a Padé-Hermite form $\underline{P}$ of f with distortion d, smallest with respect to the degree on sequences, and the corresponding minimal vector sequence. The third argument returned is the order in the order term of $\underline{P} \cdot f$.

Example H41E10

This example can be found on page 813 in [BL94].

```
> S<u> := PowerSeriesRing(Rationals());
> f := [1,u,u/(1-u^4)+u^10+O(u^16),u/(1+u^4)+u^12+O(u^16)];
> pade, padebasis, ord := PadeHermiteApproximant(f,[2,2,2,2]);
> pade, ord;
( u -1  0  0)
11
> BaseRing(Parent(pade)) eq S;
true
> MinimalVectorSequence(f,10);
[
    ( u -1  0  0),
    (   0  u^4 -1/2  1/2),
    (         0    1 - u^4 -1/2 + u^4       -1/2),
    (    0    -u 1/2*u 1/2*u)
]
>
> p := 2;
> seq := MinimalVectorSequence(f,10: Distortion :=[2,2,2,2],Power := p);
> seq;
[
    (u^5   0   0   0),
```

```
    (   0  u^2 -1/2  1/2),
    (         0    1 - u^2 -1/2 + u^2       -1/2),
    (    0    -u 1/2*u 1/2*u)
]
> mp:= map<S->S|  x :-> (IsWeaklyZero(x) select 0
>     else  &+([Coefficient(x,i)*(S.1)^(p*i) :  i in Exponents(x)]))
>       + (ISA(Type(v),RngIntElt) select O((S.1)^(p*v))
>   else S!0 where v := AbsolutePrecision(x))>;
> sums := [&+([mp(Q[i])*f[i]: i in [1..#f]]) : Q in seq];
> [Valuation(v) : v in sums];
[ 10, 10, 10, 11 ]
> sums;
[
    u^10,
    -1/2*u^10 + 1/2*u^12 - u^13 + O(u^16),
    -1/2*u^10 - 1/2*u^12 + u^13 + u^14 + O(u^16),
    u^11 + 1/2*u^12 + 1/2*u^14 + O(u^18)
]
```

Example H41E11

This example covers the example on page 815 in [BL94].

```
> S<u> := PowerSeriesRing(Rationals());
> f := [1,u, -1-u^4-2*u^8+u^10+u^11-u^12+O(u^16),-u-u^5-u^9-u^14-u^15+O(u^16)];
> dist:=[2,2,3,3];
> seq := MinimalVectorSequence(f,13:Distortion:=dist);
> pade, padebasis, ord :=  PadeHermiteApproximant(f,dist);
> pade, ord;
(-u  1  0  0)
13
> padebasis;
[
    (-u  1  0  0),
    (1/2*u -1/2*u^4 1/2*u + 1/2*u^3 + 1/2*u^4 -1/2*u^2 - 1/2*u^3 - u^4),
    (-1/2 1/2*u^3 -1/2 - 1/2*u^2 - 1/2*u^3 - u^4 1/2*u + 1/2*u^2 + 2*u^3),
    (      -u        0        0 -1 + u^4)
]
> padebasis eq seq;
true
> [[Valuation(w[i]): i in [1..Degree(w)]] : w in seq];
[
    [ 1, 0, Infinity, Infinity ],
    [ 1, 4, 1, 2 ],
    [ 0, 3, 0, 1 ],
    [ 1, Infinity, Infinity, 0 ]
]
> [[MaximumDegree([w[i]])-dist[i]: i in [1..Degree(w)] ] : w in seq];
```

```
[
    [ -1, -2, -Infinity, -Infinity ],
    [ -1, 2, 1, 1 ],
    [ -2, 1, 1, 0 ],
    [ -1, -Infinity, -Infinity, 1 ]
]
> p:=2;
> seq := MinimalVectorSequence(f,12:Distortion:=dist,Power:=p);
> seq;
[
    (  u - u^3         0 u - 2*u^3         0),
    (-1 - u + u^2 -u^3 -1 - u + 2*u^2 + u^3 -u^3),
    (u + u^2 - u^3 0 u + u^2 - 2*u^3 - u^4 0),
    (      0       1       0 1 - u^2)
]
> seq[1]-seq[3];
(      -u^2          0 -u^2 + u^4          0)
> mp:= map<S->S|  x :-> (IsWeaklyZero(x) select 0
>     else  &+([Coefficient(x,i)*(S.1)^(p*i) :  i in Exponents(x)]))
>       + (ISA(Type(v),RngIntElt) select O((S.1)^(p*v))
>   else S!0 where v := AbsolutePrecision(x))>;
>  [Valuation(&+([mp(Q[i])*f[i]: i in [1..#f]])) : Q in seq];
[ 12, 12, 13, 13 ]
```

Example H41E12

This example considers the example on page 816 in [BL94].

```
> S<u> := PowerSeriesRing(Rationals());
> f := [1,u,-1-u^4-2*u^8+u^10+O(u^12),-u-u^5-u^9+u^10+O(u^12)];
> dist := [2,2,3,3];
> seq := MinimalVectorSequence(f,12: Distortion := dist);
> [[MaximumDegree([w[i]])-dist[i]: i in [1..Degree(w)] ] : w in seq];
[
    [ -1, -2, -Infinity, -Infinity ],
    [ -2, 1, 0, 0 ],
    [ -Infinity, -Infinity, 1, 0 ],
    [ -1, -Infinity, 0, 1 ]
]
> [Valuation(&+([(Q[i])*f[i]: i in [1..#f]]) ) : Q in seq];
[ Infinity, 12, 12, 12 ]
> [MaximumDegree([ &+([(Q[i])*f[i]: i in [1..#f]]) ]) : Q in seq];
[ -Infinity, -Infinity, 14, -Infinity ]
> PadeHermiteApproximant(f,[2,2,3,3]);
> p := 2;
> seq := MinimalVectorSequence(f,12:Distortion:=[2,2,3,3],Power:=p);
> seq;
[
```

```
    (  1 - u^2        -1 1 - 2*u^2  -1 + u^2),
    (   0 -u^4    0 -u^4),
    (      -u       -1 -u + u^3 -1 + u^2),
    (      0       u       0 u - u^3)
]
> [[MaximumDegree([w[i]])-dist[i]: i in [1..Degree(w)] ] : w in seq];
[
    [ 0, -2, -1, -1 ],
    [ -Infinity, 2, -Infinity, 1 ],
    [ -1, -2, 0, -1 ],
    [ -Infinity, -1, -Infinity, 0 ]
]
> mp:= map<S->S|  x :-> (IsWeaklyZero(x) select 0
>     else  &+([Coefficient(x,i)*(S.1)^(p*i) :  i in Exponents(x)]))
>       + (ISA(Type(v),RngIntElt) select O((S.1)^(p*v))
>   else S!0 where v := AbsolutePrecision(x))>;
>
> [Valuation(&+([mp(Q[i])*f[i]: i in [1..#f]])) : Q in seq];
[ 12, 13, 12, 12 ]
```

A variant of the Padé-Hermite approximant is when the exponent in the order term is set rather that the type of the sequence. It is also possible to let $\underline{f}$ be a sequence such that its entries themselves are vectors of polynomials or power series.

`PadeHermiteApproximant(f,m)`

`Power` RngIntElt *Default :* 1

Returns a Padé-Hermite form of minimal degree in the corresponding minimal vector sequence, such that its inproduct with f has order at least m. The second argument returned is the corresponding minimal vector sequence.

Example H41E13

This example can be found on page 813 in [BL94].

```
> S<u> := PowerSeriesRing(Rationals());
> f := [Vector([1]), Vector([u])];
> pade, seq := PadeHermiteApproximant(f,3);
Calculating the Pade'-Hermite approximant for the sequence [
    1,
    u
]
with order term 3 and power 1 .
> pade;
( u -1)
> seq;
[
```

```
    ( u -1),
    (  0 u^2)
]
> mat := Matrix([Eltseq(v): v in f]);
> pade*mat;
(0)
> PadeHermiteApproximant([1,u],5);
( u -1)
[
    ( u -1),
    (  0 u^4)
]
> PadeHermiteApproximant(f,3:Power:=2);
> g:= [Vector([1,0,0]), Vector([0,1,0]), Vector([1+u,2+u^2,u^3])];
> pade := PadeHermiteApproximant(g,5);
Calculating the Pade'-Hermite approximant for the sequence [
    1,
    u,
    1 + 2*u + u^3 + u^7 + u^11
]
with order term 15 and power 3 .
> pade;
(-u^3 - u^4     -2*u^3        u^3)
> pade*Matrix([Eltseq(v): v in g]);
(  0 u^5 u^6)
```

Example H41E14

This example considers Padé-Hermite approximants for some series that have non-trivial power series expansions.

```
> S<u> := PowerSeriesRing(Rationals());
> f := [Sin(u), Cos(u), Exp(u)];
> [Valuation(f[i]) : i in [1..#f]], [Degree(f[i]) : i in [1..#f]];
[ 1, 0, 0 ]
[ 19, 20, 20 ]
> [AbsolutePrecision(f[i]) : i in [1..#f]];
[ 21, 22, 21 ]
> dist := [3,2,5];
> pade, seq, ord := PadeHermiteApproximant(f,dist);
> 1/420*pade;
(-1275 - 255*u + 45*u^2 + 5*u^3 120 + 495*u + 75*u^2 -120 + 900*u - 600*u^2 +
    160*u^3 - 20*u^4 + u^5)
> ord eq &+(dist)+#f-1, ord;
true 12
> [Degree(pade[i]) : i in [1..Degree(pade)]];
[ 3, 2, 5 ]
> g:= [Cos(2*u)*(u+1)+3,Cos(u)^2+u*Cos(u)+1,Cos(2*u)+1,Cos(u)];
```

```
> pade, basis := PadeHermiteApproximant(g,20);
> 131/75880*pade;
(          2    -4 + 2*u        -3*u 4*u - 2*u^2)
> h := [ 1+u^2-u^7+u^12, Sin(u), Exp(u) ];
> dist:=[3,1,2];
> seq := MinimalVectorSequence(h,8:Distortion := dist);
> sums := [&+([Q[i]*h[i]: i in [1..#f]]) : Q in seq];
> [Valuation(s) : s in sums];
[ 8, 8, 8 ]
> [[MaximumDegree([w[i]]): i in [1..Degree(w)] ] : w in seq];
[
    [ 4, 1, 2 ],
    [ 4, 2, 2 ],
    [ 3, 1, 2 ]
]
>  [[MaximumDegree([w[i]])-dist[i]: i in [1..Degree(w)] ] : w in seq];
[
    [ 1, 0, 0 ],
    [ 1, 1, 0 ],
    [ 0, 0, 0 ]
]
```

41.6 Bibliography

[**BL94**] Bernhard Beckermann and George Labahn. A uniform approach for the fast computation of matrix-type Padé approximants. *SIAM J. Matrix Anal. Appl.*, 15(3): 804–823, 1994.

[**Der94**] Harm Derksen. An algorithm to compute generalized Padé-Hermite Forms. Technical Report 9403, Department of Mathemtaics, Catholic University Nijmegen, jan 1994.

42 ALGEBRAIC FUNCTION FIELDS

Chapter 42
ALGEBRAIC FUNCTION FIELDS

42.1 Introduction

This version of MAGMA is based on the KANT Version 4 system for algebraic function field computations. Special functions for rational function fields are described in Chapter FldFunRat.

An algebraic function field F/k (in one variable) over a field k is a field extension F of k such that F is a finite field extension of $k(x)$ for an element $x \in F$ which is transcendental over k. As a MAGMA object it is of type `FldFun` with elements of type `FldFunElt`. For perfect k it is always possible to choose $x \in F$ so that $F/k(x)$ is also separable. For such x there exists a primitive element $\alpha \in F$ with $F = k(x, \alpha)$ where α is a root of an irreducible, separable polynomial in $k(x)[y]$.

42.1.1 Representations of Fields

All function fields in MAGMA can be represented as described above, i.e. as $k(x, \alpha)$ where x is transcendental and α is algebraic. This is the representation which has the most functionality.

Alternatively, one may wish to consider a function field as a combined transcendental and algebraic extension of its constant field, more like a curve. Such a field would be a quotient of $k[x, y]$. Fields in this representation do not have orders.

Algebraic function fields may be extended to create relative finite extensions of $k(x)$ like $F = k(x, \alpha_1, \ldots, \alpha_n)$. The functionality described in this chapter which is not available for these relative extensions is that involving series rings, galois groups and subfields.

It is also possible to make non–simple extensions where more than one root of a polynomial is added at each step by extending by several polynomials. These extensions have the same functionality as the relative finite extensions, except that primitive elements and some functions involving differentials are not available.

Function fields represented as finite extensions may have orders. Some orders will have a basis different to that of their function fields. Orders may have a field of fractions. A field of fractions of an order is isomorphic to the function field of the order, however its elements are represented with respect to the basis of the order. So there are 3 different types of rings covered in this chapter, function fields (`FldFun`), orders of function fields (`RngFunOrd`) and fields of fractions of orders of function fields (`FldFunOrd`). Each ring type has its own corresponding element type.

42.2 Creation of Algebraic Function Fields and their Orders

42.2.1 Creation of Algebraic Function Fields

`ext< K | f >`

`FunctionField(f :`*parameters*`)`

`Check`	BoolElt	*Default :* `true`
`Global`	BoolElt	*Default :* `true`

Let k be a field and $K = k(x)$ or $K = k(x, \alpha_1, \ldots, \alpha_r)$ some finite extension of $k(x)$. Given an irreducible and separable polynomial $f \in K[y]$ of degree greater than zero with coefficients within K, create the algebraic function field $F = K[y]/\langle f \rangle = k(x, \alpha_1, \ldots, \alpha_r, \alpha)$ obtained by adjoining a root α of f to K. F will be viewed as a (finite) extension of K. The polynomial f is also allowed to be $\in k[x][y]$.

The optional parameter `Check` may be used to prevent some conditions from being tested. The default is `Check := true`, so that f is verified to be irreducible and separable. The optional parameter `Global` may be used to allow another copy of the field to be returned if it is set to `false`, otherwise if a field has already been constructed using f over K and has not been deleted then the existing field will be returned.

The angle bracket notation may be used to assign the root α to an identifier: `F<a> := FunctionField(f)`.

`FunctionField(f :`*parameters*`)`

`Check`	BoolElt	*Default :* `true`
`Global`	BoolElt	*Default :* `true`

Let k be a field. Given an irreducible polynomial $f \in k[x, y]$ of degree greater than zero, create the algebraic function field F which is the field of fractions of $k[x, y]/\langle f \rangle$. The polynomial f must be separable in at least one variable. F will be viewed as (infinite) extension of k.

The optional parameter `Check` may be used to prevent some conditions from being tested. The default is `Check := true`, so that f is verified to be irreducible and separable in at least one variable. The optional parameter `Global` may be used to allow another copy of the field to be returned if it is set to `false`, otherwise if a field has already been constructed using f over K and has not been deleted then the existing field will be returned.

The angle bracket notation may be used to assign the images of x, y in F to identifiers: `F<a, b> := FunctionField(f)`.

`FunctionField(S)`

`Check`	BoolElt	*Default :* `true`

Return the function field F whose defining polynomials are the polynomials in the sequence S. If `Check` is set to `false` then it will not be checked the polynomials actually define a field.

`FunctionField(S)`

Return the function field F whose defining polynomials are the polynomials in the sequence S. This field F will be represented as an infinite degree extension of the coefficient field of the polynomials in S.

`HermitianFunctionField(p, d)`

`HermitianFunctionField(q)`

Create the Hermitian function field $F = \mathbf{F}_{q^2}(x, \alpha)$ defined by $\alpha^q + \alpha = x^{q+1}$, where q is the d-th power of the prime number p.

`sub< F | S >`

`sub< F |` $s_1, \ldots, s_r$ `>`

The subfield of the function field F containing the elements in the sequence S or the elements s_i.

`AssignNames(~F, s)`

`AssignNames(~a, s)`

Procedure to change the name of the generating element(s) in the function field F (a in F) to the contents of the sequence of strings s, which must have length 1 or 2 in this case.

This procedure only changes the name(s) used in printing the elements of F. It does *not* assign to any identifier(s) the value(s) of the generator(s) in F; to do this, use an assignment statement, or use angle brackets when creating the field.

Note that since this is a procedure that modifies F, it is necessary to have a reference `~F` to F (or a) in the call to this function.

`FunctionField(R)`

`Global`	BoolElt	*Default :* `true`
`Type`	Cat	*Default :* `FldFunRat`

Return the rational function field over R in one variable. If `Global` is `false` then create a new copy of the field, otherwise reuse any globally created field which already exists. If `Type` is `FldFun` create the field as an algebraic function field, otherwise create as a rational function field.

Example H42E1

Let $\mathbf{F}_5$ be the finite field of five elements. To create the function field extension $\mathbf{F}_5(x, \alpha)/\mathbf{F}_5(x)$, where α satisfies

$$\alpha^2 = 1/x,$$

one may proceed in the following, equivalent ways:

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> F<alpha> := FunctionField(y^2 - 1/x);
> F;
Algebraic function field defined over Univariate rational function field over
GF(5)
Variables: x by
y^2 + 4/x
```

or

```
> R<x> := PolynomialRing(GF(5));
> P<y> := PolynomialRing(R);
> F<alpha> := FunctionField(x*y^2 - 1);
> F;
Algebraic function field defined over Univariate rational function field over
GF(5)
Variables: x by
x*y^2 + 4
```

or

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> F<alpha> := ext< R | y^2 - 1/x >;
> F;
Algebraic function field defined over Univariate rational function field over
GF(5)
Variables: x by
y^2 + 4/x
```

Example H42E2

An extension of F may be created as follows.

```
> R<y> := PolynomialRing(F);
> FF<beta> := FunctionField(y^3 - x/alpha : Check := false);
> FF;
Algebraic function field defined over F by
y^3 + 4*x^2*alpha
```

Example H42E3

To create a non–simple extension:

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> FF<alpha, beta> := FunctionField([y^2 - 1/x, y^3 + x]);
> FF;
Algebraic function field defined over Univariate rational function field over
GF(5) by
y^2 + 4/x
y^3 + x
```

or

```
> P<y> := PolynomialRing(F);
> FF<beta, gamma> := FunctionField([y^2 - x/alpha, y^3 + x]);
> FF;
Algebraic function field defined over F by
y^2 + 4*x^2*alpha
y^3 + x
```

Example H42E4

The creation of an Hermitian function field:

```
> F := HermitianFunctionField(9);
> F;
Algebraic function field defined over GF(3^4) by
y^9 + y + 2*x^10
```

42.2.2 Creation of Orders of Algebraic Function Fields

Equation orders, maximal orders and other orders of algebraic function fields can be created.

`EquationOrderFinite(F)`

Create the 'finite' equation order of the function field $F/k(x, \alpha_1, \ldots, \alpha_r)$, i.e. $k[x, d_1\alpha_1, \ldots, d_r\alpha_r, d\alpha]$ where $d_j, d \in k[x]$ is chosen such that $d_j\alpha_j, d\alpha$ are integral over $k[x]$.

`MaximalOrderFinite(F)`

Create the 'finite' maximal order of the function field $F/k(x, \alpha_1, \ldots, \alpha_r)$. This is the integral closure of $k[x, d_1\alpha_1, \ldots, d_r\alpha_r]$ in F.

`EquationOrderInfinite(F)`

Create the 'infinite' equation order of the function field $F/k(x, \alpha_1, \ldots, \alpha_r)$, i.e. $o_\infty[\alpha_1, \ldots, \alpha_r, \beta]$ where o_∞ denotes the valuation ring of the degree valuation in $k(x)$ and β is a primitive element of $F/k(x, \alpha_1, \ldots, \alpha_r)$ which is integral over o_∞.

`MaximalOrderInfinite(F)`

Create the 'infinite' maximal order of the function field $F/k(x, \alpha_1, \ldots, \alpha_r)$. This is the integral closure of o_∞ in F.

`IntegralClosure(R, F)`

The integral closure of the subring R of the function field F in itself.

`EquationOrder(O)`

The equation order of the order O. An order whose basis is a transformation of that of O and is a power basis.

`MaximalOrder(O)`

`Discriminant`	ANY	*Default :*
`Ramification`	SEQENUM	*Default :*
`Al`	MONSTGELT	*Default :* "*Auto*"
Verbose	`MaximalOrder`	*Maximum :* 5

The maximal order of the order O of an algebraic function field.

If O is a radical (pure) extension then specific code is used to calculate each p-maximal order, rather than the Round 2 method. In this case we can compute a pseudo basis for the p-maximal orders knowing only the valuation of the constant coefficient of the defining polynomial at p [Sut12].

If O is an Artin–Schreier extension then the maximal order can be computed directly without computing the p-maximal orders. The proof of Proposition III.7.8 of [Sti93] gives us a start on some elements which are a basis for the maximal order [Sut13].

If the `Discriminant` or `Ramification` parameters are supplied an algorithm ([Bj94], Theorems 1.2 and 7.6) which can compute the maximal order given the discriminant of the maximal order will be used. `Discriminant` must be an element of the coefficient ring of O if O is a non relative order and must be an ideal of O if O is a relative order. `Ramification` must contain elements of the coefficient ring if O is a non relative order and must contain ideals of O if O is a relative order. The ramification sequence is taken to contain prime factors of the discriminant. Only one of these parameters can be specified and if one of them is then `Al` cannot be specified. Otherwise `Al` may be set to `"Round2"` to avoid using the algorithms for the special cases above.

`SetOrderMaximal(O, b)`

Set the order O of a function field to be maximal if b is `true` and to be non–maximal if b is `false`.

`ext< O | f >`

`Check` BOOL *Default :* `true`

The order O with a root of f adjoined.

Example H42E5

Creation of orders is shown below.

```
> PR<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(PR);
> FR1<a> := FunctionField(y^3 - x*y^2 + y + x^4);
> P<y> := PolynomialRing(FR1);
> FR2<c> := FunctionField(y^2 + y - a/x^5);
> EFR1F := EquationOrderFinite(FR1);
> MFR1F := MaximalOrderFinite(FR1);
> EFR1I := EquationOrderInfinite(FR1);
> MFR1I := MaximalOrderInfinite(FR1);
> EFR2F := EquationOrderFinite(FR2);
> MFR2F := MaximalOrderFinite(FR2);
> EFR2I := EquationOrderInfinite(FR2);
> MFR2I := MaximalOrderInfinite(FR2);
>  MaximalOrder(EFR2I);
>> MaximalOrder(EFR2I);
               ^
Runtime error in 'MaximalOrder': Order must be defined over a maximal order
> MFR2I;
Maximal Order of FR2 over MFR1I
> P<y> := PolynomialRing(FR1);
> MaximalOrder(ext<MFR1F | y^2 + y - a*x^5>); MFR2F;
Maximal Equation Order of Algebraic function field defined over FR1 by
y^2 + y - x^5*a over EFR1F
Maximal Order of FR2 over EFR1F
> MaximalOrder(ext<MFR1I | y^2 - 1/a>);
Maximal Order of Algebraic function field defined over FR1 by
y^2 + 1/x^4*a^2 - 1/x^3*a + 1/x^4 over MFR1I
```

Example H42E6

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> f := y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2;
> F<alpha> := FunctionField(f);
> IntegralClosure(R, F);
```

```
Algebraic function field defined over GF(5) by
y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2
> IntegralClosure(PolynomialRing(GF(5)), F);
Maximal Order of F over Univariate Polynomial Ring in x over GF(5)
> IntegralClosure(ValuationRing(R), F);
Maximal Order of F over Valuation ring of Rational function field of
rank 1 over GF(5)
Variables: x with generator 1/x
```

`Order(O, T, d)`

`Check` BoolElt *Default :* `true`

Create the order whose basis is that of the order O multiplied by the matrix T over the coefficient ring of O divided by the scalar d. If the parameter `Check` is set to `false` then it will not be checked that the result is actually an order (potentially expensive).

`Order(O, M)`

`Check` BoolElt *Default :* `true`

Create the order whose basis is that of the order O multiplied by the dedekind module M. If the parameter `Check` is set to `false` then it will not be checked that the result is actually an order (potentially expensive).

`Order(O, S)`

`Verify` BoolElt *Default :* `true`

`Order` BoolElt *Default :* `false`

Given a sequence S of elements in an algebraic function field F create the minimal order R of F which contains all elements of S.

The order O may be an order of F which will be used as the suborder of R, in which case its coefficient ring should be maximal, or O may be a maximal order of the coefficient field of F.

If `Verify` is `true`, it is verified that the elements of S are integral algebraic numbers. This can be a lengthy process if the field is of large degree.

Setting `Order` to `true`, when the order is large will avoid verifying that the module generated by the elements of S is closed under multiplication. By default, products of the generators will be added until the module is closed under multiplication.

`Simplify(O)`

Return the order O as a direct transformation of its equation order, instead of a composition of transformations.

`O1 + O2`

The smallest common over order of $O1$ and $O2$ where $O1$ and $O2$ have the same equation order.

`O1 meet O2`

The intersection of orders $O1$ and $O2$ which must have the same equation order.

`AsExtensionOf(O1, O2)`

Return the order $O1$ as an transformation of the order $O2$ where $O1$ and $O2$ have the same coefficient ring.

Example H42E7

Some of the above order creations are shown below.

```
> P<x> := PolynomialRing(GF(5));
> P<y> := PolynomialRing(P);
> F<a> := FunctionField(y^3 - x^4);
> O := Order(EquationOrderFinite(F), MatrixAlgebra(Parent(x), 3)!1, Parent(x)!3);
> O;
Order of F over Univariate Polynomial Ring in x over GF(5)
> Basis(O);
[
    2,
    2*a,
    2*a^2
]
> P<y> := PolynomialRing(O);
> EO := ext<MaximalOrder(O) | y^2 + O!(2*a)>;
> V := KModule(F, 2);
> M := Module([V | [1, 0], [4, 3], [9, 2]]);
> M;
Module over Maximal Order of F over Univariate Polynomial Ring in x over GF(5)
Ideal of Maximal Order of F over Univariate Polynomial Ring in x over GF(5)
Generator:
1 car Ideal of Maximal Order of F over Univariate Polynomial Ring in x over
GF(5)
Generator:
2
> O2 := Order(EO, M);
> O2;
Order of Algebraic function field defined over F by
$.1^2 + 2*a over Maximal Order of F over Univariate Polynomial Ring in x over
GF(5)
Transformation of EO
Transformation Matrix:
[[ 1, 0, 0 ] [ 0, 0, 0 ]]
[[ 0, 0, 0 ] [ 1, 0, 0 ]]
```

```
> Basis(O2);
[ 1, $.1 ]
```

42.2.3 Orders and Ideals

Orders may be created using ideals of other orders. Ideals are discussed in Section 42.12.

`MultiplicatorRing(I)`

Returns the multiplicator ring of the ideal I of the order O, that is, the subring of elements of the field of fractions of O that multiply I into itself.

`pMaximalOrder(O, p)`

The p-maximal over order of O where p is a prime polynomial or ideal of the coefficient ring of O or an element of valuation 1 of the valuation ring.

If O is a Kummer extension then specific code is used to calculate each p-maximal order, rather than the Round 2 method. In this case we know 1 or 2 elements which generate the p-maximal order and can write the order down.

If O is an Artin–Schreier extension then we can also write down a basis for the p-maximal order and avoid the Round 2 algorithm. We use [Sti93] Proposition III.7.8 to get a start on computing these elements.

`pRadical(O, p)`

Returns the p-radical of an order O for a prime p (polynomial or ideal of the coefficient ring or element of valuation 1 of the valuation ring), defined as the ideal consisting of elements of O for which some power lies in the ideal pO.

It is possible to call this function even if p is not prime. In this case the p-trace-radical will be computed, i.e.

$$\{x \in F \mid \mathrm{Tr}(xO) \subseteq C\}$$

for F the field of fractions of O and C the order of p (if p is an ideal) or the parent of p otherwise. If p is square free and all divisors are larger than the field degree, this is the intersection of the radicals for all l dividing p.

42.3 Related Structures

42.3.1 Parent and Category

Function fields form the MAGMA category `FldFun` and function field orders form the MAGMA category `RngFunOrd`. The notional power structures exist as parents of function fields and their orders but allow no operations.

`Category(F)` `Category(O)`

`Parent(F)` `Parent(O)`

42.3.2 Other Related Structures

More interesting related structures (than above) are listed below.

`PrimeRing(F)`
`PrimeField(F)`
`PrimeRing(O)`

The prime field of the function field F or the order O (prime ring of the constant field).

`ConstantField(F)`
`DefiningConstantField(F)`

The constant field k, where $F = k(x, \alpha)$.

`ExactConstantField(F)`

The exact constant field of the algebraic function field F/k, i.e. the algebraic closure in F of the constant field k of F, together with the inclusion map.

`BaseRing(F)`
`BaseField(F)`
`CoefficientRing(F)`
`CoefficientField(F)`

The rational function field $k(x)$ if the function field F is an extension of $k(x)$ and k if F is an extension of k. If F is an extension of another algebraic function field then this field will be returned.

`ISABaseField(F,G)`

Applies to more general fields within MAGMA than function fields. Returns whether G is amongst the recursively defined base fields of F.

`BaseRing(O)`

`CoefficientRing(O)`

The polynomial algebra $k[x]$ if the order O is finite or the degree valuation ring if O is infinite. If O is an extension of another order of an algebraic function field this order will be returned.

`BaseRing(FF)`

`BaseField(FF)`

`CoefficientRing(FF)`

`CoefficientField(FF)`

Given a field of fractions FF of an order O return the field of fractions of the coefficient ring of O.

`SubOrder(O)`

For a non equation order O returns the order which O was created as a transformation of. This order is one transformation closer to the equation order.

`FunctionField(O)`

The function field which O is an order of.

`FieldOfFractions(O)`

`FieldOfFractions(FF)`

`FieldOfFractions(F)`

Given an order O, this function returns the field of fractions, a field with the same basis as O. On a function field or a field of fractions this function is trivial.

`Order(FF)`

Given a field of fractions FF return the order O which is the ring of integers of FF.

`RationalExtensionRepresentation(F)`

The function field F represented as an extension of a rational function field. This function gives the representation of function fields F/k as finite extensions.

`AbsoluteOrder(O)`

The order O as an extension of its bottom coefficient ring, (i.e. the order of the `RationalExtensionRepresentation` of the field of fractions of O corresponding to O).

`AbsoluteFunctionField(F)`

The function field F expressed as an extension of its constant field.

`UnderlyingRing(F)`

`UnderlyingField(F)`

`UnderlyingRing(F, R)`

`UnderlyingField(F, R)`

Return the underlying ring of the function field F over R. This is F expressed as an extension of R. If R is not given then it is taken to be the coefficient field of the coefficient field of F. The field R must appear in the tower of coefficient fields under F.

`Embed(F, L, a)`

`Embed(F, L, s)`

Install the embedding of F into L with the image(s) of the primitive element(s) of F being the element a in L or the images in s in L.

`Places(F)`

The set of places of the algebraic function field F/k.

`DivisorGroup(F)`

The group of divisors of the algebraic function field F/k.

`DifferentialSpace(F)`

The space of differentials of the algebraic function field F/k.

Example H42E8

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> f := y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2;
> F<alpha> := FunctionField(f);
> ConstantField(F);
Finite field of size 5
> CoefficientField(F);
Univariate rational function field over GF(5)
Variables: x
> CoefficientRing(MaximalOrderFinite(F));
Univariate Polynomial Ring in x over GF(5)
> FieldOfFractions(IntegralClosure(ValuationRing(R), F));
Algebraic function field defined over Univariate rational function field over
GF(5)
Variables: x by
y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2
> Order(IntegralClosure(ValuationRing(R), F),
>     MatrixAlgebra(CoefficientRing(MaximalOrderInfinite(F)), 3)!4,
>     CoefficientRing(MaximalOrderInfinite(F))!1);
```

```
Maximal Order of F over Valuation ring of Univariate rational function field
over GF(5) with generator 1/x
> SubOrder($1);
Maximal Order of F over Valuation ring of Univariate rational function field
over GF(5) with generator 1/x
> $1 eq $2;
false
> Places(F);
Set of places of F
> DivisorGroup(F);
Divisor group of F
```

Example H42E9

Output from `UnderlyingRing` is shown.

```
> PF<x> := PolynomialRing(GF(31, 3));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x^3 + 1>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b*x*y - 1>;
> RationalExtensionRepresentation(FF2);
Algebraic function field defined over Univariate rational function field over
GF(31^3) by
y^6 + 29*y^3 + (30*x^5 + x^2)*y^2 + 1
> UnderlyingRing(FF2);
Algebraic function field defined over Univariate rational function field over
GF(31^3) by
y^6 + 29*y^3 + (30*x^5 + x^2)*y^2 + 1
> UnderlyingRing(FF2, FieldOfFractions(PF));
Algebraic function field defined over GF(31^3) by
$.1^6 + 29*$.1^3 + 30*$.1^2*$.2^5 + $.1^2*$.2^2 + 1
```

`WeilRestriction(E, n)`

`Reduction`	BoolElt	*Default :* `true`
Verbose	`WeilRes`	*Maximum :* 1

A hyperelliptic function field in the Weil restriction over $\mathbf{F}_q$ of the elliptic function field E: $y^2+xy+x^3+ax^2+b$ defined over $\mathbf{F}_{q^n}$ where q is a power of 2. Also returns a function which can be used to map a place (not a pole or zero of x) of F into a divisor of the result. See [Gau00]. `Reduction` indicates whether a (possibly quite expensive) reduction step is performed at the end of the computation. It defaults to `true`.

`ConstantFieldExtension(F, E)`

Return the function field with constant field E which contains the function field F. The ring E must cover the constant field of F. If E is contained in the exact constant field of F then F and the new field will be isomorphic.

Example H42E10

Changing the constant field to the exact constant field is shown below.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(y^6 + y + 2);
> E<a> := ExactConstantField(F);
> C, r := ConstantFieldExtension(F, E);
> r(c);
1/16*(a^5 + 4*a^4 + 6*a^3 + 4*a^2 + a)
> $1 @@ r;
c
> e := Random(C, 2);
> e @@ r;
1/2*x*c^5 - 3*x*c^4 + (-12*x + 8)*c^3 + (-16*x + 24)*c^2 + (-8*x + 16)*c - 1
> r($1);
1/2*(-a^5 - a^2 + a)*$.1 + a^5 + a^4 - a^3 - a^2 - 1
```

`Reduce(O)`

Given an order O belonging to a function field F, this function returns the order obtained by applying size-reduction to the basis of O.

42.4 General Structure Invariants

`Characteristic(F)`

`Characteristic(O)`

The characteristic of the function field F/k or one of its orders O.

`IsPerfect(F)`

Applies to any field in MAGMA. Returns whether F is perfect.

`Degree(F)`

`Degree(F, G)`

`Degree(O)`

The degree $[F : G]$ of the field extension F/G where G is the base field of F unless specified. For an order O, this function returns the rank of O as a module over its coefficient ring. Note that this rank is equal to the degree $[F : G]$ where F and G are the field of fractions of O and the coefficient ring of O respectively.

`AbsoluteDegree(F)`

`AbsoluteDegree(O)`

The degree of the function field F or the order O as a finite extension of $k(x)$ or $k[x]$ or as an infinite extension of k.

`DefiningPolynomial(F)`

`DefiningPolynomial(O)`

The defining polynomial of the function field F over its coefficient ring. For an order O belonging to a function field F, this function returns the defining polynomial of O, which may be different from that of $F/k(x, \alpha_1, \ldots, \alpha_r)$.

`DefiningPolynomials(F)`

`DefiningPolynomials(O)`

Return the defining polynomials of the function field F or the order O as a sequence of polynomials over the coefficient ring.

`Basis(F)`

`Basis(O)`

`Basis(O, R)`

The basis $1, \alpha, \ldots, \alpha^{n-1}$ of the function field $F[\alpha]$ over the coefficient field.

Given an order O belonging to a function field F, this function returns the basis of O in the form of function field elements.

Given an additional ring R, return the basis of O as elements of R.

`TransformationMatrix(O1, O2)`

Return the matrix M and a denominator d which transforms elements of the order $O1$ into elements of the order $O2$.

`CoefficientIdeals(O)`

The coefficient ideals of the order O of a relative extension. These are the ideals $\{A_i\}$ of the coefficient ring of O such that for every element e of O, $e = \sum_i a_i * b_i$ where $\{b_i\}$ is the basis returned for O and each $a_i \in A_i$.

`BasisMatrix(O)`

Given an order O in a function field F of degree n, this returns an $n \times n$ matrix whose i-th row contains the coefficients for the i-th basis element of O with respect to the power basis of F. Thus, if b_i is the i-th basis element of O,

$$b_i = \sum_{j=1}^{n} M_{ij} \alpha^{j-1}$$

where M is the matrix and α is the generator of F.

`PrimitiveElement(O)`

A root of the defining polynomial of the order O.

`Discriminant(O)`

The discriminant of the order O, up to a unit in its coefficient ring.

`AbsoluteDiscriminant(O)`

The discriminant of the order O of an algebraic function field F over the bottom coefficient ring of O, (the subring of the rational function field F extends).

`DimensionOfExactConstantField(F)`

`DegreeOfExactConstantField(F)`

The dimension of the exact constant field of the function field F/k over k. The exact constant field is the algebraic closure of k in F.

`Genus(F)`

The genus of the function field F/k.

Example H42E11

```
> PF<x> := PolynomialRing(GF(31, 3));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x^3 + 1>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b*x*y - 1>;
> Characteristic(FF2);
31
> EFF2I := EquationOrderInfinite(FF2);
> MFF2I := MaximalOrderInfinite(FF2);
> Degree(MFF2I) eq 3;
true
> AbsoluteDegree(EFF2I);
6
> Genus(FF2);
9
> DefiningPolynomial(EFF2I);
$.1^3 + [ 0, 30/x^3 ]*$.1 + [ 30/x^9, 0 ]
> Basis(MFF2I);
[ 1, 1/x*d, 1/x^2*d^2 ]
> Discriminant(EFF2I);
Ideal of Maximal Equation Order of FF1 over Valuation ring of Univariate
rational function field over GF(31^3)
Variables: x with generator 1/x
Generator:
(4*x^3 + 27)/x^15*b + 4/x^18
> AbsoluteOrder(EFF2I);
```

```
Order of Algebraic function field defined over Univariate rational function
field over GF(31^3) by
y^6 + 29*y^3 + (30*x^5 + x^2)*y^2 + 1 over Valuation ring of Univariate rational
function field over GF(31^3) with generator 1/x
> AbsoluteDiscriminant(EFF2I);
(2*x^9 + 25*x^6 + 6*x^3 + 29)/x^33
> Discriminant($2);
(30*x^24 + 6*x^21 + 16*x^18 + 20*x^15 + 16*x^12 + 7*x^9 + 27*x^6 + 3*x^3 +
    30)/x^48
```

Example H42E12

Invariants are slightly different for non–simple fields.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<a, b> := FunctionField([3*y^3 - x^2, x*y^2 + 1]);
> DefiningPolynomials(F);
[
    3*y^3 - x^2,
    x*y^2 + 1
]
> DefiningPolynomials(EquationOrderFinite(F));
[
    y^3 - 1/3*x^2,
    y^2 + x
]
> DefiningPolynomials(EquationOrderInfinite(F));
[
    $.1^3 - 1/3/$.1^4,
    $.1^2 + 1/$.1
]
> Basis(F);
[
    1,
    a,
    a^2,
    $.1*b,
    $.1*a*b,
    $.1*a^2*b
]
> TransformationMatrix(EquationOrderFinite(F), MaximalOrderFinite(F));
[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 x 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 x 0]
[0 0 0 0 0 x]
```

```
1
> TransformationMatrix(MaximalOrderFinite(F), EquationOrderFinite(F));
[x 0 0 0 0 0]
[0 x 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 x 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
x
```

`GapNumbers(F)`

`SeparatingElement` FldFunGElt *Default :*

The sequence of global gap numbers of the function field F/k (in characteristic zero this is always $[1, \ldots, g]$). A separating element used internally for the computation can be specified, it defaults to `SeparatingElement(F)`. See the description of `GapNumbers` on page 1167.

`GapNumbers(F, P)`

The sequence of gap numbers of the function field F/k at P where P must be a place of degree one. See the description of `GapNumbers` on page 1167.

`SeparatingElement(F)`

Returns a separating element of the function field F/k.

`RamificationDivisor(F)`

`SeparatingElement` FldFunGElt *Default :*

The ramification divisor of the function field F/k. The semantics of calling `RamificationDivisor()` with F or the zero divisor of F are identical. For further details see the description of `RamificationDivisor` on page 1169.

`WeierstrassPlaces(F)`

`SeparatingElement` FldFunGElt *Default :*

The Weierstrass places of the function field F/k. The semantics of calling `WeierstrassPlaces` with F or the zero divisor of F are identical. See the description of `WeierstrassPlaces` on page 1170.

`WronskianOrders(F)`

`SeparatingElement` FldFunGElt *Default :*

The Wronskian orders of the function field F/k. The semantics of calling `WronskianOrders` with F or the zero divisor of F are identical. See the description of `WronskianOrders` on page 1170.

`Different(O)`

The different of the maximal order O.

`Index(O, S)`

The index of S in O where S is a suborder of O and O and S have the same equation order.

42.5 Galois Groups

Finding Galois groups (of normal closures) of polynomials over rational function fields over $k \in \{\mathbf{Q}, \mathbf{F}_q\}$, where $\mathbf{F}_q$ denotes the finite field of characteristic p with $q = p^r$, $r \in \mathbf{Z}_{>0}$ is a hard problem, in general. All practical algorithms used the classification of transitive groups, which is known up to degree 31 [CHM98]. These algorithms fall into two groups: The absolute resolvent method [SM85] and the method of Stauduhar [Sta73].

The MAGMA implementation is based on an extension of the method of Stauduhar by Klüners, Geißler [Gei03, GK00] and, more recently, Fieker [FK12] and Sutherland [Sut]. There is no longer any limit on the degree of the polynomials or fields as this algorithm does not use the classification of transitive groups. In contrast to the absolute resolvent method, it also provides the explicit action on the roots of the polynomial f which generates the function field. The algorithm strongly depends on the fact that the corresponding problem is implemented for the residue class field.

Roughly speaking, the method of Stauduhar traverses the subgroup lattice of transitive permutation groups of degree n from the symmetric group to the actual Galois group. This is done by using so-called relative resolvents. Resolvents are polynomials whose splitting fields are subfields of the splitting field of the given polynomial which are computed using approximations of the roots of the polynomial f.

If the field (or the field defined by a polynomial) has subfields (i.e. the Galois group is imprimitive) the current implementation changes the starting point of the algorithm in the subgroup lattice, to get as close as possible to the actual Galois group. This is done via computation of subfields of a stem field of f, that is the field extension of $k(t)$ which we get by adjoining a root of f to $k(t)$. The Galois group is found as a subgroup of the intersection of suitable wreath products (using the knowledge of the subfields) which may be easily computed.

If the field (or the field defined by a polynomial) does not have subfields (i.e. the Galois group is primitive) we use a combination of the method of Stauduhar and the absolute resolvent method. The Frobenius automorphism of the underlying field already determines a subgroup of the Galois group, which is used to speed up computations in the primitive case.

The algorithms used here are similar to those use for number fields. See also Chapter 38. In addition to the intrinsics described here, some of the intrinsics described in Section 38.2 apply to polynomials over function fields also.

`GaloisGroup(f)`

`Prime`	RngElt	*Default :*
`NextPrime`	UserProgram	*Default :*
`ProofEffort`	RngIntElt	*Default :* 10
`Ring`	GaloisData	*Default :*
`ShortOK`	BoolElt	*Default :* `false`
`Old`	BoolElt	*Default :* `false`
Verbose	`GaloisGroup`	*Maximum :* 5

Given a separable, (irreducible if k is $\mathbf{Q}$) polynomial $f(t, x)$ of degree n over the rational function field $k(t)$, $k \in \{\mathbf{Q}, \mathbf{F}_q\}$, or an extension K thereof (if $k = \mathbf{F}_q$) this function returns a permutation group that forms the Galois group of a splitting field for f in some algebraic closure of K. The permutation group acts on the points $1, 2, \ldots, n$. The roots of f are calculated in the process, expressed as power series and returned as the second argument: For a prime polynomial $p(t) \in k(t)$ denote by $\bar{N}$ the splitting field of the polynomial $f(t, x)$ mod $p(t)$. It is well known that the roots of the polynomial $f(t, x)$ can be expressed as power series in $\bar{N}[[t]]$. We embed $\bar{N}$ in an unramified p-adic extension. The third return value is a structure containing information about the computation that can be used to compute the roots of f to arbitrary precision. This can be used for example in `GaloisSubgroup` on page 979 to compute arbitrary subfields of the splitting field.

The required precision increases linearly with the index of the subgroups, which are passed traversing the subgroup lattice. Therefore computations may slow down considerably for higher degrees and large indices.

The default version employs series computations over either unramified p-adic fields ($k = \mathbf{Q}$) or finite fields ($k = \mathbf{F}_q$). The prime polynomial is determined during a Galois group computation in such a way that f is squarefree modulo p.

The prime to use for splitting field computations can be given via the parameter `Prime`. The method of choosing primes for splitting field computations can be given by the parameter `NextPrime`. An indication of how much effort the computation should make to prove the results can be provided by altering the parameter `ProofEffort`, it should be increased if more effort should be made.

If `Old` is set to `true`, then the old version is called if available. Since the return values of the new version differ substantially from the old one, this may be used in old applications.

`GaloisGroup(F)`

`Prime`	RngElt	*Default :*
`NextPrime`	UserProgram	*Default :*
`ProofEffort`	RngIntElt	*Default :* 10
`Ring`	GaloisData	*Default :*
`ShortOK`	BoolElt	*Default :* `false`

Given a function field F defined as an extension of either a rational function field or a global algebraic function field by one polynomial compute the Galois group of a normal closure of F.

The prime to use for splitting field computations can be given via the parameter `Prime`. The method of choosing primes for splitting field computations can be given by the parameter `NextPrime`. An indication of how much effort the computation should make to prove the results can be provided by altering the parameter `ProofEffort`, it should be increased if more effort should be made.

Example H42E13

A Galois group computation is shown below.

```
> k<t>:= FunctionField(Rationals());
> R<x>:= PolynomialRing(k);
> f:= x^15 + (-1875*t^2 - 125)*x^3 + (4500*t^2 + 300)*x^2 +
>     (-3600*t^2 - 240)*x + 960*t^2+ 64;
> G, r, S:= GaloisGroup(f);
> TransitiveGroupDescription(G);
1/2[S(5)^3]S(3)
> A := Universe(r);
> AssignNames(~A,  ["t"]);
> A;
Power series ring in t over Unramified extension
defined by the polynomial (1 + O(191^20))*x^4 +
    O(191^20)*x^3 + (7 + O(191^20))*x^2 + (100 +
    O(191^20))*x + 19 + O(191^20)
 over Unramified extension defined by the
polynomial (1 + O(191^20))*x + 190 + O(191^20)
 over pAdicField(191)
> r[1];
> r[1];
-54*$.1^3 + 68*$.1^2 + 31*$.1 - 12 + O(191) +
    (-15*$.1^3 - 66*$.1^2 - 2*$.1 - 39 + O(191))*t
    + O(t^2)
> S;
GaloisData of type p-Adic (FldFun over Q)
> TransitiveGroupIdentification(G);
99 15
```

Example H42E14

Some examples for polynomials over rational function fields over finite fields

```
> k<x>:= FunctionField(GF(1009));
> R<y>:= PolynomialRing(k);
> f:= y^10 + (989*x^4 + 20*x^3 + 989*x^2 + 20*x + 989)*y^8 + (70*x^8 +
> 869*x^7 + 310*x^6 + 529*x^5 + 600*x^4 + 479*x^3 + 460*x^2 + 719*x +
```

```
> 120)*y^6 + (909*x^12 + 300*x^11 + 409*x^10 + 1000*x^9 + 393*x^8 +
> 657*x^7 + 895*x^6 + 764*x^5 + 420*x^4 + 973*x^3 + 177*x^2 + 166*x +
> 784)*y^4 + (65*x^16 + 749*x^15 + 350*x^14 + 909*x^13 + 484*x^12 +
> 452*x^11 + 115*x^10 + 923*x^9 + 541*x^8 + 272*x^7 + 637*x^6 + 314*x^5 +
> 724*x^4 + 490*x^3 + 948*x^2 + 99*x + 90)*y^2 + 993*x^20 + 80*x^19 +
> 969*x^18 + 569*x^17 + 895*x^16 + 101*x^15 + 742*x^14 + 587*x^13 +
> 55*x^12+ 437*x^11 + 97*x^10 + 976*x^9 + 62*x^8 + 171*x^7 + 930*x^6 +
> 604*x^5 + 698*x^4 + 60*x^3 + 60*x^2 + 1004*x + 1008;
> G, r, p:= GaloisGroup(f);
> t1, t2:= TransitiveGroupIdentification(G);
> t1;
1
> t2;
10
```

And a second one.

```
> k<t>:= FunctionField(GF(7));
> R<x>:= PolynomialRing(k);
> f:= x^12 + x^10 + x^8 + (6*t^2 + 3)*x^6 + (4*t^4 + 6*t^2 + 1)*x^4 +
> (5*t^4 + t^2)*x^2 + 2*t^4;
> G, r, p:= GaloisGroup(f);
> G;
Permutation group G acting on a set of cardinality 12
    (2, 8)(3, 9)(4, 10)(5, 11)
    (1, 5, 9)(2, 6, 10)(3, 7, 11)(4, 8, 12)
    (1, 12)(2, 3)(4, 5)(6, 7)(8, 9)(10, 11)
> A := Universe(r);
> AssignNames(~A,  ["t"]);
> r;
[
    w^950*t^13 + w^1350*t^12 + w^1900*t^11 + w^500*t^10 + w^2050*t^9 + 2*t^8 +
        w^1350*t^7 + w^300*t^6 + w^350*t^5 + w^1450*t^4 + w^950*t^3 + w^1000*t^2
        + w^1100*t + w^550,
    w^1175*t^13 + w^1825*t^12 + w^1675*t^11 + w^725*t^10 + w^1025*t^9 +
        w^1825*t^8 + w^1325*t^7 + w^775*t^6 + w^1775*t^5 + w^1325*t^4 +
        w^1575*t^3 + w^1175*t^2 + w^2225*t + w^2275,
    w^25*t^13 + w^1075*t^12 + w^425*t^11 + w^925*t^10 + w^225*t^9 + w^2375*t^8 +
        w^2125*t^7 + w^625*t^6 + w^1175*t^5 + w^425*t^4 + w^575*t^3 + w^825*t^2
        + w^1175*t + w^2375,
    w^175*t^13 + w^1525*t^12 + w^575*t^11 + w^475*t^10 + w^1575*t^9 + w^1025*t^8
        + w^475*t^7 + w^775*t^6 + w^1025*t^5 + w^1775*t^4 + w^1625*t^3 +
        w^2175*t^2 + w^1025*t + w^1025,
    w^1025*t^13 + w^1975*t^12 + w^2125*t^11 + w^1475*t^10 + w^2375*t^9 +
        w^1975*t^8 + w^2075*t^7 + w^1825*t^6 + w^425*t^5 + w^875*t^4 +
        w^1425*t^3 + w^2225*t^2 + w^1175*t + w^325,
    w^650*t^13 + w^2250*t^12 + w^100*t^11 + w^1100*t^10 + w^1150*t^9 + 2*t^8 +
        w^1050*t^7 + w^2100*t^6 + w^1250*t^5 + w^550*t^4 + w^650*t^3 +
        w^2200*t^2 + w^1700*t + w^1450,
```

```
    w^2150*t^13 + w^150*t^12 + w^700*t^11 + w^1700*t^10 + w^850*t^9 + 5*t^8 +
        w^150*t^7 + w^1500*t^6 + w^1550*t^5 + w^250*t^4 + w^2150*t^3 +
        w^2200*t^2 + w^2300*t + w^1750,
    w^2375*t^13 + w^625*t^12 + w^475*t^11 + w^1925*t^10 + w^2225*t^9 + w^625*t^8
        + w^125*t^7 + w^1975*t^6 + w^575*t^5 + w^125*t^4 + w^375*t^3 +
        w^2375*t^2 + w^1025*t + w^1075,
    w^1225*t^13 + w^2275*t^12 + w^1625*t^11 + w^2125*t^10 + w^1425*t^9 +
        w^1175*t^8 + w^925*t^7 + w^1825*t^6 + w^2375*t^5 + w^1625*t^4 +
        w^1775*t^3 + w^2025*t^2 + w^2375*t + w^1175,
    w^1375*t^13 + w^325*t^12 + w^1775*t^11 + w^1675*t^10 + w^375*t^9 +
        w^2225*t^8 + w^1675*t^7 + w^1975*t^6 + w^2225*t^5 + w^575*t^4 +
        w^425*t^3 + w^975*t^2 + w^2225*t + w^2225,
    w^2225*t^13 + w^775*t^12 + w^925*t^11 + w^275*t^10 + w^1175*t^9 + w^775*t^8
        + w^875*t^7 + w^625*t^6 + w^1625*t^5 + w^2075*t^4 + w^225*t^3 +
        w^1025*t^2 + w^2375*t + w^1525,
    w^1850*t^13 + w^1050*t^12 + w^1300*t^11 + w^2300*t^10 + w^2350*t^9 + 5*t^8 +
        w^2250*t^7 + w^900*t^6 + w^50*t^5 + w^1750*t^4 + w^1850*t^3 + w^1000*t^2
        + w^500*t + w^250
]
> p;
t^2 + 4
```

42.6 Subfields

For finite extensions L of $\mathbf{Q}(t)$, $\mathbf{F}_q(t)$ or extensions K of $\mathbf{F}_q(t)$ MAGMA can compute all fields between L and $\mathbf{Q}(t)$, $\mathbf{F}_q(t)$ or K. It should be noted that the computation of subfields does not depend on the Galois groups. The implementation over $\mathbf{Q}(t)$ uses the algorithm of Klüners [Klü02]. The implementation over $\mathbf{F}_q(t)$ and extensions thereof follows the newer ideas of Klüners and van Hoeij [vHKN11].

`Subfields(F)`

All algebraic function fields G with $k(x) \subset G \subseteq F$ or $K \subset G \subseteq F$ where K is an extension of $\mathbf{F}_q(x)$ and the coefficient field of F.

Example H42E15

A subfield computation is shown below.

```
> k<x>:= FunctionField(Rationals());
> R<y>:= PolynomialRing(k);
> f:= y^14 - 3234*y^12 + (8*x + 123480)*y^11 + (-696*x - 1152480)*y^10 +
> (27672*x - 43563744)*y^9 + (-663544*x + 1795525424)*y^8 + (10660416*x -
>  33905500608)*y^7 + (-120467088*x + 409661347536)*y^6 + (976911040*x -
>  3428257977088)*y^5 + (-5684130144*x + 20264929189344)*y^4 + (23251514496*x -
>  83582683562112)*y^3 + (-63672983360*x + 229899367865216)*y^2 +
>  (105037027200*x - 380160309247488)*y - 79060128000*x + 286518963720192;
```

```
> F:= FunctionField(f);
> Subfields(F);
[
    <Algebraic function field defined over Univariate rational function field
    over Rational Field
    Variables: x by
    y^14 - 3234*y^12 + (8*x + 123480)*y^11 + (-696*x - 1152480)*y^10 + (27672*x
        - 43563744)*y^9 + (-663544*x + 1795525424)*y^8 + (10660416*x -
        33905500608)*y^7 + (-120467088*x + 409661347536)*y^6 + (976911040*x -
        3428257977088)*y^5 + (-5684130144*x + 20264929189344)*y^4 +
        (23251514496*x - 83582683562112)*y^3 + (-63672983360*x +
        229899367865216)*y^2 + (105037027200*x - 380160309247488)*y -
        79060128000*x + 286518963720192, Mapping from: FldFun: F to FldFun: F>,
    <Algebraic function field defined over Univariate rational function field
    over Rational Field
    Variables: x by
    y^7 + 294*y^6 - 107016*y^5 + (2744*x + 576240)*y^4 + (-806736*x +
        2469418896)*y^3 + (88740960*x - 312072913824)*y^2 + (-4329483200*x +
        15606890921216)*y + 79060128000*x - 286518963720192, Mapping from:
    Algebraic function field defined over Univariate rational function field
    over Rational Field
    Variables: x by
    y^7 + 294*y^6 - 107016*y^5 + (2744*x + 576240)*y^4 + (-806736*x +
        2469418896)*y^3 + (88740960*x - 312072913824)*y^2 + (-4329483200*x +
        15606890921216)*y + 79060128000*x - 286518963720192 to FldFun: F>
]
```

42.7 Automorphism Group

Let K be a finite extension of the rational function field over a finite field, the rationals or a number field. In contrast to the number field situation, there are two different natural notions of automorphisms here: we distinguish between automorphisms that fix the base field and arbitrary automorphisms that can also induce non-trivial maps of the constant field.

The first case, automorphisms fixing the base field of K, is analogous to the number field case and was implemented by Jürgen Klüners.

The second case of more general automorphisms has been implemented by Florian Heß along the lines of his paper [Heß04]. Here the constant field of K can, in fact, be any exact perfect field in Magma with a few provisos.

42.7.1 Automorphisms over the Base Field

`Automorphisms(K, k)`

Computes all $\mathbf{Q}(t)$ automorphisms of the absolute finite extension K that fix k. The field k has to be $\mathbf{Q}(t)$ for this function.

`AutomorphismGroup(K, k)`

Return the group of k-automorphisms of the algebraic function field K together with the map from the group to the sequence of automorphisms of K. The field k has to be $\mathbf{Q}(t)$.

Example H42E16

We define an extension of degree 7 over $\mathbf{Q}(t)$ and compute the automorphisms.

```
> Q:=Rationals();
> Qt<t>:=PolynomialRing(Q);
> Qtx<x>:=PolynomialRing(Qt);
> f := x^7 + (t^3 + 2*t^2 - t + 13)*x^6 + (3*t^5 - 3*t^4
>     + 9*t^3 + 24*t^2 - 21*t + 54)*x^5 + (3*t^7 -
>     9*t^6 + 27*t^5 - 22*t^4 + 6*t^3 + 84*t^2 -
>     121*t + 75)*x^4 + (t^9 - 6*t^8 + 22*t^7 -
>     57*t^6 + 82*t^5 - 70*t^4 - 87*t^3 + 140*t^2 -
>     225*t - 2)*x^3 + (-t^10 + 5*t^9 - 25*t^8 +
>     61*t^7 - 126*t^6 + 117*t^5 - 58*t^4 - 155*t^3
>     + 168*t^2 - 80*t - 44)*x^2 + (-t^10 + 8*t^9 -
>     30*t^8 + 75*t^7 - 102*t^6 + 89*t^5 + 34*t^4 -
>     56*t^3 + 113*t^2 + 42*t - 17)*x + t^9 - 7*t^8
>     + 23*t^7 - 42*t^6 + 28*t^5 + 19*t^4 - 60*t^3 -
>     2*t^2 + 16*t - 1;
> K:=FunctionField(f);
> A:=Automorphisms(K, BaseField(K));
> #A;
7
```

Now we transform this list into a group to see that it is really cyclic. We pass in special functions for equality testing and multiplication to speed the algorithm up.

```
> G := GenericGroup(A: Eq := func<a,b | a`Images eq b`Images>,
>                      Mult := func<a,b | hom<K -> K | a`Images @ b>>);
> G;
Finitely presented group G on 2 generators
Relations
    G.1 = Id(G)
    G.1 * G.2 = G.2 * G.1
    G.1 * G.2^2 = G.2^2 * G.1
    G.1 * G.2^3 = G.2^3 * G.1
    G.1 * G.2^4 = G.2^4 * G.1
    G.1 * G.2^5 = G.2^5 * G.1
```

```
    G.1 * G.2^6 = G.2^6 * G.1
    G.1 = G.2^7
```

Finally, we verify that this gives the same result as `AutomorphismGroup`.

```
> AutomorphismGroup(K, BaseField(K));
Finitely presented group on 2 generators
Relations
    $.1 = Id($)
    $.1 * $.2 = $.2 * $.1
    $.1 * $.2^2 = $.2^2 * $.1
    $.1 * $.2^3 = $.2^3 * $.1
    $.1 * $.2^4 = $.2^4 * $.1
    $.1 * $.2^5 = $.2^5 * $.1
    $.1 * $.2^6 = $.2^6 * $.1
    $.1 = $.2^7
Mapping from: GrpFP to [
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K,
    Mapping from: FldFun: K to FldFun: K
] given by a rule
```

`IsSubfield(K, L)`

Given two absolute finite extensions K and L of $\mathbf{Q}(t)$, decide if L is an extension of K. If this is the case, return an embedding map from K into L.

`IsIsomorphicOverQt(K, L)`

Given two absolute finite extensions K and L of $\mathbf{Q}(t)$, decide if L is $\mathbf{Q}(t)$-isomorphic to K. If this is the case, return a map from K onto L.

Example H42E17

`Subfields` and `IsIsomorphic` are illustrated below.

```
> Q:=Rationals();
> Qt<t>:=PolynomialRing(Q);
> Qtx<x>:=PolynomialRing(Qt);
> K:=FunctionField(x^4-t^3);
> L:=Subfields(K);
> #L;
2
> L:=L[2][1]; L;
Algebraic function field defined over Univariate
```

```
rational function field over Rational Field
Variables: t by
x^2 - t^3
```

Now we will check if L is indeed a subfield of K:

```
> IsSubfield(L,K);
true Mapping from: FldFun: L to FldFun: K
```

Obviously, L can be defined using a simpler polynomial:

```
> LL:=FunctionField(x^2-t);
> IsIsomorphicOverQt(LL,L);
true Mapping from: FldFun: LL to FldFun: L
```

42.7.2 General Automorphisms

`Isomorphisms(K, E)`

`BaseMorphism`	MAP	*Default :* `false`
`Bound`	RNGINTELT	*Default :* ∞
`Strategy`	MONSTGELT	*Default :* "*None*"

Given two function fields K and E, this function computes a list of at most `Bound` field isomorphisms from K to E.

If `BaseMorphism` is given it should be an isomorphism f between the constant fields of K and E. In this case only isomorphisms extending f are considered.

The default behaviour is for all isomorphisms from K to E which extend SOME isomorphism of the constant field of K to that of E considered. In this case (no base morphism is specified), the constant fields must be finite, the rationals or a number field. If the base morphism f is specified then the constant fields can be any exact perfect fields (finite or characteristic 0).

If the base morphism f is specified, it can be defined in the natural way for most constant field types. For example, for finite fields and number fields, the usual `hom<k->l|x>`, where `x` gives the image of $k.1$, can be used. A common situation is where the constant fields of K and E are equal to k and f is the identity. This can be defined very simply for any k by `IdentityFieldMorphism(k)` . Several more intrinsics related to field morphisms are described in the following subsection.

The possible choices of `Strategy` are `"None"`, `"Weierstrass"` or `"DegOne"`. If `Strategy` is different to `"None"`, this determines the places that are used as the basis of the construction of the maps. In all cases, a finite set of places of E and K which must correspond under any isomorphism are used. All isomorphisms are found between the canonical affine models (as defined by Heß) obtained by omitting one of these places from each of E and K.

`DegOne` can only be used with finite constant fields. In this case, isomorphisms are determined which map a fixed degree one place of K to any one of the finite

number of degree one places of E. This function can fail in rare situations if the constant field of K is too small and no degree one place exists. In this case an appropriate error message is displayed.

`Weierstrass` uses the Weierstrass places of the fields. Isomorphisms are determined which map a fixed Weierstrass place of K to any of those of E with the same degree and Riemann-Roch data. This strategy can be very fast if the residue field and Riemann-Roch data of a particular place of K match those of only a few (or no!) Weierstrass places of E.

In case of fields of genus < 2, the constant field must be finite.

`IsIsomorphic(K, E)`

`BaseMorphism`	MAP	*Default :* `false`
`Strategy`	MONSTGELT	*Default :* "*None*"

As above, except the function only computes a single isomorphism if one exists.

`Automorphisms(K)`

`BaseMorphism`	MAP	*Default :* `false`
`Bound`	RNGINTELT	*Default :* ∞
`Strategy`	MONSTGELT	*Default :* "*None*"

This function computes a list of at most `Bound` automorphisms of the function field K. This is an abbreviation for `Isomorphisms(K, K)` and the parameters are as described above.

An important difference is that the BaseMorphism, if specified, must be of field morphism type. `IdentityFieldMorphism` may be used, but basic constrictors for non-trivial constant field maps f will usually cause an error if used directly. The way around this is to use the conversion `f := FieldMorphism(f)` (see the following subsection).

`Isomorphisms(K,E,p1,p2)`

`Automorphisms(K,p1,p2)`

`Bound`	RNGINTELT	*Default :* ∞

As above except that the constant field morphism is taken as the identity and only iso/automorphisms which take function field place $p1$ to $p2$ are computed.

`AutomorphismGroup(K)`

`BaseMorphism`	MAP	*Default :* `false`
`Strategy`	MONSTGELT	*Default :* "*None*"

Given a function field K, this function computes that group of automorphisms satisfying the conditions specified by the parameters and returns it as a finitely-presented group. The map also returned is invertible and takes a group element to the function field isomorphism that it represents.

`AutomorphismGroup(K,f)`

`Strategy` MonStgElt *Default : "None"*

In this variation, the automorphism group of the function field K is computed in its permutation representation on a set of places or divisors or in its linear representation on a space of differentials or subspace of K.

The return values consist of the representing group G, a map (with inverse) from G to the maps of K giving the actual isomorphisms, and a sequence of isomorphisms of K which consist of the kernel of the representation.

Only automorphisms fixing the constant field are considered here. If the set/space on which the representation is to be defined is not invariant by the automorphism group, a run-time error will result.

The argument f should be a map defining the representation.

Its domain must be an enumerated sequence for a permutation representation or a vector space for a linear representation.

Its codomain should be K or a space or enumerated sequence of elements of K, places of K, divisors of K or differentials of K. The examples below show some common ways of producing f by using functions like `DifferentialSpace` and `RiemannRochSpace`.

42.7.3 Field Morphisms

The isomorphisms returned by the functions in the last subsection are of general `Map` type but contain some extra internal structure. The same is true of the maps used to specify `BaseMorphism`. These objects come in two flavours: *field morphisms*, that represent maps between general fields, and the more specialised *function field morphisms*, representing maps between algebraic function fields. This subsection contains several related functions that are very useful when working with (function) field morphisms.

`IsMorphism(f)`

Returns `true`, if the map is a field or function field morphism; `false` otherwise.

`FieldMorphism(f)`

Converts a homomorphism between fields into a field morphism.

`IdentityFieldMorphism(F)`

Returns the identity automorphism of field F as a field morphism.

`IsIdentity(f)`

Returns `true` if f is the identity morphism; `false` otherwise.

`Equality(f, g)`

Returns `true`, if the two maps are both field morphisms or function field morphisms and are equal; `false` otherwise.

`HasInverse(f)`

Either returns "true" and the inverse morphism for (function) field morphism f, or "false" if inverse does not exist, or "unknown" if it cannot be computed.

`Composition(f, g)`

The composition of the field morphisms f and g.

Example H42E18

We illustrate the use of the general isomorphism functions with some examples. In the first, we have a rational function field of characteristic 5:

```
> k<w> := GF(5);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> F<a> := FunctionField(x^2+y^2-1);
> L := Isomorphisms(kxf, F);
> #L eq #PGL(2, k);
```

In the next example we consider the function field of a hyperelliptic curve defined over $\mathbf{Q}(i)$ $[i^2 = -1]$ and a Galois twist of it. The fields are not isomorphic over $\mathbf{Q}(i)$ but they are over $\mathbf{Q}$:

```
> k<i> := QuadraticField(-1);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> F1<a> := FunctionField(y^2-x^5-x^2-i);
> F2<b> := FunctionField(i*y^2-x^5-i*x^2+1);
> c := IdentityFieldMorphism(k);
> IsIsomorphic(F1,F2 : BaseMorphism := c);
false
> IsIsomorphic(F1,F2);
true Mapping from: FldFun: F1 to FldFun: F2 given by a rule
> L := Isomorphisms(F1, F2);
> [<f(a), f(x), f(i)> : f in L];
[
    <-i*b, i*x, -i>,
    <i*b, i*x, -i>
]
```

In the next example we consider the function field of the genus 3 plane curve $x^3*y+y^3*z+z^3*x = 0$, which has full automorphism group $PGL_2(\mathbf{F}_7)$. We compute automorphisms over different finite fields and also compute the automorphisms group as an FP group.

```
> k := GF(11);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> K<y> := FunctionField(x^3*y+y^3+x);
> L := Automorphisms(K);
> #L;
3
```

```
> // Extend base field to get all autos
> k := GF(11^3);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> K<y> := FunctionField(x^3*y+y^3+x);
> L := Automorphisms(K);
> #L;
504
> // restrict to just "geometric" autos, which fix the base
> c := IdentityFieldMorphism(k);
> L := Automorphisms(K : BaseMorphism := c);
> #L;
168
> // get the automorphism group instead as an FP group
> G,mp := AutomorphismGroup(K : BaseMorphism := c);
> G;
Finitely presented group G on 2 generators
Relations
    G.2^3 = Id(G)
    (G.1^-1 * G.2)^3 = Id(G)
    G.1^7 = Id(G)
    (G.2^-1 * G.1^-3)^2 = Id(G)
    (G.2^-1 * G.1^-1)^4 = Id(G)
> #G;
168
> IdentifyGroup(G); // find in small group database
<168, 42>
```

Finally, we give an example of a genus 1 function field over $\mathbf{F}_5$ where the group of automorphisms is computed acting on various spaces of functions and differentials.

```
> k<w> := GF(5);
> kxf<x> := RationalFunctionField(k);
> kxfy<y> := PolynomialRing(kxf);
> f := x^3 + y^3 + 1;
> F<a> := FunctionField(f);
> f := Numeration(Set(Places(F, 1)));
> G, h, K := AutomorphismGroup(F, f);
> #G; Type(G);
12
GrpPerm
> V, f := SpaceOfDifferentialsFirstKind(F);
> G, h, K := AutomorphismGroup(F, f);
> #G; Type(G);
2
GrpMat
> D := &+ Places(F, 1);
> V, f := DifferentialSpace( -D );
> G, h := AutomorphismGroup(F, f);
```

```
> #G;
12
> V, f := RiemannRochSpace( D );
> G, h, ker := AutomorphismGroup(F, f);
> #G; #ker;
12
1
```

42.8 Global Function Fields

F/k denotes a global function field in this section.

42.8.1 Functions relative to the Exact Constant Field

`NumberOfPlacesOfDegreeOverExactConstantField(F, m)`

`NumberOfPlacesDegECF(F, m)`

The number of places of degree m of the global function field F/k. Contrary to the `Degree` function the degree is here taken over the respective exact constant fields.

`NumberOfPlacesOfDegreeOneOverExactConstantField(F)`

`NumberOfPlacesOfDegreeOneECF(F)`

The number of places of degree one in the global function field F/k. Contrary to the `Degree()` function the degree is here taken over the exact constant field.

`NumberOfPlacesOfDegreeOneOverExactConstantField(F, m)`

`NumberOfPlacesOfDegreeOneECF(F, m)`

The number of places of degree one in the constant field extension of degree m of the global function field F/k. Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

`SerreBound(F)`

`SerreBound(F, m)`

`SerreBound(q, g)`

The Serre bound on the number of places of degree one in a global function field of genus g over the exact constant field of q elements (of the global function field F, of the constant field extension of degree m of F). Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

`IharaBound(F)`

`IharaBound(F, m)`

`IharaBound(q, g)`

The Ihara bound on the number of places of degree one in a global function field F/k of genus g over the exact constant field of q elements (of the global function field F, of the constant field extension of degree m of F). Contrary to the `Degree` function the degree is here taken over the respective exact constant fields.

`NumberOfPlacesOfDegreeOneECFBound(F)`

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F)`

`NumberOfPlacesOfDegreeOneECFBound(F, m)`

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F, m)`

`NumberOfPlacesOfDegreeOneECFBound(q, g)`

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(q, g)`

The minimum of the Serre and Ihara bound. Contrary to the `Degree` function the degree is here taken over the respective exact constant fields.

`LPolynomial(F)`

The L-polynomial of the global function field F/k (with respect to the exact constant field).

`LPolynomial(F, m)`

The L-polynomial of the constant field extension of degree m of the global function field F/k (with respect to the exact constant field).

`ZetaFunction(F)`

The Zeta function of the global function field F/k (with respect to the exact constant field).

`ZetaFunction(F, m)`

The Zeta function of the constant field extension of degree m of the global function field F/k (with respect to the exact constant field).

42.8.2 Functions Relative to the Constant Field

`Places(F, m)`

A sequence containing the places of degree m of the global function field F/k.

`HasPlace(F, m)`

Returns `true` and a place of degree m if and only if there exists such a place in the global function field; `false` otherwise.

`HasRandomPlace(F, m)`

Returns `true` and a random place of degree m in the global function field (`false` if there are none).

`RandomPlace(F, m)`

Returns a random place of degree m in the global function field or throws an error if there is none.

Example H42E19

```
> Y<t> := PolynomialRing(Integers());
> R<x> := FunctionField(GF(9));
> P<y> := PolynomialRing(R);
> f := y^3 + y + x^5 + x + 1;
> F<alpha> := FunctionField(f);
> Genus(F);
4
> NumberOfPlacesDegECF(F, 1);
22
> NumberOfPlacesOfDegreeOneECFBound(F);
32
> HasRandomPlace(F, 2);
true (x^2 + $.1*x + 2, alpha + $.1^2*x + $.1^5)
> LPolynomial(F);
6561*t^8 + 8748*t^7 + 7290*t^6 + 3888*t^5 + 1539*t^4 + 432*t^3 + 90*t^2
    + 12*t + 1
```

Example H42E20

Some of the above functions are demonstrated for a global relative field.

```
> PF<x> := PolynomialRing(GF(13, 2));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b>;
> RER_FF2 := RationalExtensionRepresentation(FF2);
> NumberOfPlacesOfDegreeOneECF(FF2) eq NumberOfPlacesOfDegreeOneECF(RER_FF2);
```

```
true
> SerreBound(FF2);
170
> NumberOfPlacesDegECF(FF2, 1);
170
> _, P := HasPlace(FF2, 1);
> P;
(x, (($.1^44*x + $.1^100)*b + ($.1^82*x + $.1^10))*d^2 + (($.1^85*x + $.1^67)*b
    + ($.1^107*x + $.1^130))*d + ($.1^26*x + $.1^69)*b + $.1^149*x)
> Degree(P) eq 1;
true
> LPolynomial(FF2, 2) eq LPolynomial(RER_FF2, 2);
true
```

42.8.3 Functions related to Class Group

`UnitRank(O)`

Given a maximal 'finite' order O in a global function field, return the unit rank of O.

`UnitGroup(O)`

The unit group of a 'finite' maximal order O as an Abelian group and the map from the unit group into O. Also see IsUnitWithPreimage on page 1133.

`Regulator(O)`

The regulator of the unit group of the 'finite' maximal order O.

`PrincipalIdealMap(O)`

The map from the multiplicative group of the field of fractions of O to the group of fractional ideals of O where O is a 'finite' maximal order.

Example H42E21

Following on from the last example,

```
> EFF2F := EquationOrderFinite(FF2);
> G, m := UnitGroup(EFF2F);
> G;
Abelian Group isomorphic to Z/168
Defined on 1 generator
Relations:
    168*G.1 = 0
> m(Random(G));
[ [ $.1^120, 0 ], [ 0, 0 ], [ 0, 0 ] ]
> IsUnit($1);
true
```

```
> Regulator(EFF2F);
1
```

`ClassGroup(F :parameters)`

`DegreeBound`	RngIntElt	*Default :*
`SizeBound`	RngIntElt	*Default :*
`ReductionDivisor`	DivFunElt	*Default :*
`Proof`	BoolElt	*Default :*

The divisor class group of F/k as an Abelian group, a map of representatives from the class group to the divisor group and the homomorphism from the divisor group onto the divisor class group. For a detailed description see `ClassGroup` on page 1172.

`ClassGroup(O)`

The ideal class group of the 'finite' maximal order O as an Abelian group, a map of representatives from the ideal class group to the group of fractional ideals and the homomorphism from the group of fractional ideals onto the ideal class group.

`ClassGroupExactSequence(O)`

Returns the maps in the center of the exact sequence

$$0 \to U \to F^\times \to Id \to Cl \to 0$$

where U is the unit group of O, $F^\times$ is the multiplicative group of the field of fractions of O, Id is the group of fractional ideals of O and Cl is the class group of O for a 'finite' maximal order O.

`ClassGroupAbelianInvariants(F :parameters)`

`DegreeBound`	RngIntElt	*Default :*
`SizeBound`	RngIntElt	*Default :*
`ReductionDivisor`	DivFunElt	*Default :*
`Proof`	BoolElt	*Default :*

Computes a sequence of integers containing the Abelian invariants of the divisor class group of F/k. For a detailed description see `ClassGroupAbelianInvariants` on page 1172.

`ClassGroupAbelianInvariants(O)`

Computes a sequence of integers containing the Abelian invariants of the ideal class group of the 'finite' maximal order O.

`ClassNumber(F)`

The order of the group of divisor classes of degree zero of F/k.

`ClassNumber(O)`

The order of the ideal class group of the 'finite' maximal order O.

Example H42E22

An example of class groups of relative fields is shown.

```
> PF<x> := PolynomialRing(GF(13, 2));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b : Check := false>;
> MFF2I := MaximalOrderInfinite(FF2);
> G, m, mi := ClassGroup(FF2);
> m(Random(G));
Divisor in reduced representation:
Divisor in ideal representation:
Fractional ideal of Maximal Equation Order of FF2 over Maximal Equation Order of
FF1 over Univariate Polynomial Ring in x over GF(13^2)
Generators:
1
($.1^60/x*b + $.1^57/x)*d^2 + ($.1^141/x*b + $.1^4/x)*d + $.1^80/x*b, Ideal of
MFF2I
Generators:
1
1,
-2,
6*(1/x, (($.1^132*x^2 + $.1^164*x + 12)/x^3*b + ($.1^85*x^2 + $.1^155*x +
    12)/x^3)*d^2 + (($.1^75*x^2 + $.1^81*x + 12)/x^3*b + ($.1^29*x^2 + $.1^155*x
    + 12)/x^3)*d + ($.1^163*x^2 + $.1^29*x + 12)/x^3*b + ($.1^141*x + 12)/x^2),
    (x)^2 * (1/x)^2
> mi(&+[Divisor(Random(FF2, 3)) : i in [1 .. 3]]);
0
> ClassNumber(FF2);
1
```

`GlobalUnitGroup(F)`

The group of global units of F/k, i. e. the multiplicative group of the exact constant field, as an Abelian group, together with the map into F. Also see `IsGlobalUnit` on page 1133 and `IsGlobalUnitWithPreimage` on page 1133.

`ClassGroupPRank(F)`

Compute the p-rank of the class group of F/k where p is the characteristic of F/k. For a detailed description see ClassGroupPRank on page 1175.

`HasseWittInvariant(F)`

Return the Hasse–Witt invariant of F/k. See HasseWittInvariant on page 1175 for a detailed description.

`IndependentUnits(O)`

A sequence of independent units of the 'finite' maximal order O.

`FundamentalUnits(O)`

A sequence of fundamental units of the 'finite' maximal order O.

Example H42E23

```
> R<x> := FunctionField(GF(3));
> P<y> := PolynomialRing(R);
> f := y^4 + x*y + x^4 + x + 1;
> F<a> := FunctionField(f);
> O := MaximalOrderFinite(F);
> Basis(O);
[ 1, a, a^2, a^3 ]
> Discriminant(O);
x^12 + x^3 + 1
> UnitRank(O);
1
> U := FundamentalUnits(O);
> U;
[ [ x^33 + x^31 + 2*x^30 + 2*x^28 + 2*x^27 + x^25 + 2*x^24 + x^22 + 2*x^19 +
    2*x^15 + x^10 + 2*x^9 + 2*x^7 + x^6 + 2*x + 2, x^32 + 2*x^30 + x^29 + 2*x^28
    + 2*x^27 + 2*x^26 + x^22 + x^21 + 2*x^19 + x^18 + x^17 + x^16 + x^13 + x^11
    + 2*x^10 + 2*x^9 + 2*x^3 + 1, x^29 + x^27 + 2*x^25 + 2*x^23 + x^22 + 2*x^21
    + x^20 + x^18 + 2*x^17 + x^16 + x^15 + 2*x^14 + x^11 + 2*x^10 + 2*x^4 + x,
    x^30 + 2*x^27 + x^24 + x^21 + 2*x^18 + x^9 + 2*x^6 + 2 ] ]
> Norm(U[1]);
1
> Regulator(O);
33
```

42.9 Structure Predicates

`IsField(R)` `IsEuclideanDomain(R)`
`IsPID(R)` `IsUFD(R)`
`IsDivisionRing(R)` `IsEuclideanRing(R)`
`IsPrincipalIdealRing(R)` `IsDomain(R)`
`F eq G` `F ne G` `O1 eq O2` `O1 ne O2`

`O1 subset O2`

Return whether $O1$ is a subset of $O2$.

`IsGlobal(F)`

Returns `true` if and only if the algebraic function field F/k is global, i.e. the constant field is a finite field; `false` otherwise.

`IsRationalFunctionField(F)`

Return `true` if the function field F is isomorphic to a rational function field, (i.e. F is only trivially algebraic).

`IsFiniteOrder(O)`

Given an order O of a function field, return `true` if and only if the bottom coefficient ring of O is a polynomial ring.

`IsEquationOrder(O)`

Given an order O of a function field, return `true` if and only if the order O is an equation order (i.e. it has been defined by a polynomial and so has a power basis).

`IsAbsoluteOrder(O)`

Return `false` if the order O is an extension of another order, otherwise `true`.

`IsMaximal(O)`

Given an order O of a function field, return `true` if and only if the order O is maximal in its field of fractions.

`IsTamelyRamified(O)`

Return whether the order O is tamely ramified, i.e. no prime ideal of O has residue field with characteristic dividing its ramification index.

`IsTotallyRamified(O)`

Return whether there is an ideal of the order O which is totally ramified, i.e. its ramification index is equal to the degree of O over its coefficient ring.

`IsUnramified(O)`

Return whether a finite order O is unramified at the finite places and whether an infinite order O is unramified at the infinite places.

`IsWildlyRamified(O)`

Return whether there is a prime ideal of the order O which is wildly ramified, i.e. its ramification index is divisible by the characteristic of its residue class field.

`IsInKummerRepresentation(K)`

Tests if the global function field K is, in its current representation, a Kummer extension. More specific, this function tests if the defining polynomial is of the form $x^r - a$ for some r coprime to the characteristic and if r divides the order of the multipicative group of the constant field, ie. if the coefficient ring of K contains a primitive r-th root of unity. In case K is in Kummer representation, the element a is returned as a second return value.

`IsInArtinSchreierRepresentation(K)`

Tests if a global function field K is, in its current representation, a Artin-Schreier extension, ie. if the defining polynomial of K is of the form $x^p - x - a$ where p is the characteristic of K. In this case, the element a is returned as a second return value.

42.10 Homomorphisms

`hom< F -> R | g >`

`hom< F -> R | cf, g >`

The homomorphism from the function field F to any ring R where g is the image of the generator of F in R and cf is a map from the coefficient field of F into R.

`hom< O -> R | g >`

`hom< O -> R | cf, g >`

Create the map from the order O of an algebraic function field to R using g as the image of the primitive element of O. If the map cf is given it should be from the coefficient ring of O into R, otherwise the coefficient ring of O should be automatically coercible into R.

`IsRingHomomorphism(m)`

Return whether the vector space homomorphism m is a homomorphism of rings.

Example H42E24

A simple use of homomorphisms is shown.

```
> PR<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(PR);
> FR1<a> := FunctionField(y^3 - x*y + 1);
> P<y> := PolynomialRing(FR1);
> FR2<c> := FunctionField(y^2 - a^5*x^3*y + 1);
> EFR2F := EquationOrderFinite(FR2);
> cf := hom<FR1 -> EFR2F | a + 1>;
> h := hom<FR2 -> EFR2F | cf, c + 1>;
> h(c) eq c + 1;
true
> h(a*c) eq a*c + a + c + 1;
true
```

`hom< O -> R | `$b_1, ..., b_n$` >`

`hom< O -> R | m, `$b_1, ..., b_n$` >`

Return the map from the order O of an algebraic function field into the ring R which maps the basis elements of O to $b_1, ..., b_n$. The map m, if given, should be from the coefficient ring of O into R and will be used to map the coefficients of the basis elements. If not given, the coefficient ring of O should automatically coerce into R.

42.11 Elements

The function field $F = k(x, \alpha_1, \ldots, \alpha_r, \alpha)$ may be viewed as n-dimensional vector space over $k(x, \alpha_1, \ldots, \alpha_r)$, where n is the degree of the field extension $F/k(x, \alpha_1, \ldots, \alpha_r)$. Note that F is spanned by the powers $1, \alpha, \ldots, \alpha^{n-1}$. Within MAGMA, function field elements are printed as linear combinations of these powers of α over the coefficient field.

An order can be viewed as a free R-module of rank n where R is its coefficient ring (a polynomial ring or the degree valuation ring of $k(x)$ or an order which is a lesser degree extension of $k[x]$) and n equals the degree $F/k(x, \alpha_1, \ldots, \alpha_r)$. It has a basis consisting of n elements. Within MAGMA, function field order elements are printed as a sequence of coefficients of the R-linear combination of such a basis.

Elements can also be represented as a product of other function field or order elements. This is referred to as the product representation. Product representations can be useful for large elements, however, it is expensive to put such elements in a set or to test them for equality as this involves finding coefficients for the element.

42.11.1 Creation of Elements

```
F . 1
F . 2
```

(i) Return the generator for the function field F over $k(x, \alpha_1, \ldots, \alpha_r)$, that is, $\alpha \in F$ such that $F = k(x, \alpha_1, \ldots, \alpha_r, \alpha)$.

(ii) Return the first and second generators for the function field F over k, that is, $\alpha \in F$ and $x \in F$ such that $F = k(x, \alpha)$.

```
Name(F, i)
```

Given a function field F, return the i-th generator, i.e. return the element `F.1` or `F.2` of F.

```
O . i
FF . i
```

Return the ith basis element of the order O or its field of fractions FF.

```
F ! a
elt< F | a >
```

Create the element of the function field F specified by a; here a is allowed to be an element coercible into F, which means that a may be:

(i) an element of F,

(ii) an element of the coefficient field of F,

(iii) an element of another representation of F.

For F an extension of $k(x)$ we additionally have

(iv) an element of an order of F,

(v) an element being coercible into $k(x)$,

(vi) a sequence of elements being coercible into the coefficient field of F of length equal to the degree of F over its coefficient field. In this case the element $a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1}$ is created, where $a = [a_0, \ldots, a_{n-1}]$ and α is the generator `F.1` of F over its coefficient field.

```
O ! a
elt< O | a >
```

Create the element of the order O specified by a; here a is allowed to be an element coercible into O, which means that mathematically $a \in O$ and that a may be any of:

(i) an element of the function field F,

(ii) an element of an order of the function field F,

(iii) an element that can be coerced into $k(x)$,

(iv) an element that can be coerced into its coefficient field,

(v) a sequence of elements being coercible into the coefficient field of O of length equal to the rank of O over its coefficient field. In this case the element $a_1\omega_1 + a_2\omega_2 + \ldots + a_n\omega_n$ is created, where $a = [a_1, \ldots, a_n]$ and $\omega_1, \omega_2, \ldots, \omega_n$ is the basis of O as returned by `Basis(O)`.

`FF ! a`
`elt< FF | a >`

Create the element of the field of fractions FF of an order O specified by a, where a may be any of the above such that $d * a$ is mathematically in O for some $d \in O$.

`elt< F | a0, a1, ..., an-1 >`

Create the element $a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1}$ where $a_0, \ldots, a_{n-1}$ are coercible into the coefficient field of the function field F, n equals the degree of F over its coefficient field and α is the generator `F.1` of F over $k(x)$.

`elt< O | a1, a2, ..., an >`
`elt< FF | a1, a2, ..., an >`

Create the element $a_1\omega_1 + a_2\omega_2 + \ldots + a_n\omega_n$ where $a_1, \ldots, a_n$ are coercible into the coefficient ring of the order O, n equals the rank of O over its coefficient ring and $\omega_1, \omega_2, \ldots, \omega_n$ is the basis of O as returned by `Basis(O)`, (where O is the ring of integers of the field of fractions FF).

`One(F)` `One(O)`
`Identity(F)` `Identity(O)`
`Zero(F)` `Zero(O)`
`Representative(F)` `Representative(O)`

These generic functions (cf. Chapter 17) create 1, 1, 0, and 0 respectively in the function field F or order O.

`Random(F, m)`
`Random(O, m)`

A "random" element of the global function field F or one of its orders O. The size of the coefficients of the element are determined by m.

42.11.2 Parent and Category

`Parent(a)` `Category(a)`

42.11.3 Sequence Conversions

The sequence conversions refer to the function field F as a vector space of dimension n over the coefficient field of F where F is a finite degree extension (degree n) of its coefficient field.

`ElementToSequence(a)`

`Eltseq(a)`

The sequence $[a_1, \ldots, a_n]$ of elements of the coefficient field of the parent of the function field or order element a such that $a = a_1\omega_1 + a_2\omega_2 + \ldots + a_n\omega_n$ where $\omega_1, \omega_2, \ldots, \omega_n$ is a basis of the parent of a.

`Eltseq(a, R)`

A sequence of coefficients of the function field element a in the coefficient field R.

`Flat(a)`

A sequence of coefficients of the function field element a in the bottom coefficient field of the parent of a.

`F ! [ a0, a1, ..., an-1 ]`

The element $a = a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1}$ where the function field $F = k(x, \alpha_1, \ldots, \alpha_r, \alpha)$ and the a_i may be coerced into $k(x, \alpha_1, \ldots, \alpha_r)$.

`O ! [ a1, a2, ..., an ]`

The element $a = a_1\omega_1 + a_2\omega_2 + \ldots + a_n\omega_n$ where $\omega_1, \omega_2, \ldots, \omega_n$ is a basis of the order O and the a_i are coercible into the coefficient ring of O.

Example H42E25

```
> R<x> := FunctionField(GF(5));
> P<y> := PolynomialRing(R);
> f := y^3 + (4*x^3 + 4*x^2 + 2*x + 2)*y^2 + (3*x + 3)*y + 2;
> F<alpha> := FunctionField(f);
> Evaluate(f, alpha);
0
> F.1;
alpha
> b := x + alpha + 1/x*alpha^2;
> b;
1/x*alpha^2 + alpha + x
> b eq F ! [x, 1, 1/x];
true
```

42.11.4 Arithmetic Operators

The following binary arithmetic operations can also be performed in the case where one operand is an element of the function field F or an order O and the other operand is a ring element which can naturally be mapped into F or O.

`+ a` `- a`

`a + b` `a - b` `a * b` `a div b` `a / b` `a ^ k`

`Modexp(a, k, m)`

Return $a^k \ mod \ m$ where m is an element of $k[x]$ or o_∞ according to whether the parent of a is a finite or infinite order.

`a mod I`

Return the element a belonging to the order O as an element of O/I.

`Modinv(a, m)`

Return the inverse of the element a of an order of a function field modulo m where m is an element of $k[x]$ or o_∞ according to whether the order of a is a finite or infinite order or an ideal of the order of a.

42.11.5 Equality and Membership

The following binary arithmetic operations can also be performed in the case where one operand is an element of the function field F or an order O and the other operand is a ring element which can naturally be mapped into F or O.

`a eq b` `a ne b`

`a in F` `a in O` `a in FF`

`a notin F` `a notin O` `a notin FF`

42.11.6 Predicates on Elements

The functions in this section list the general ring element predicates that apply to function fields and orders of a function field.

`IsDivisibleBy(a, b)`

Given elements a and b belonging to a function field F or an order O, returns `true` if there exists $c \in F$ or $c \in O$ such that $a = bc$ and returns c as well, provided that $b \neq 0$.

`IsZero(a)` `IsOne(a)` `IsMinusOne(a)`

`IsNilpotent(a)` `IsIdempotent(a)`

`IsUnit(a)` `IsZeroDivisor(a)` `IsRegular(a)`
`IsIrreducible(a)` `IsPrime(a)`

`IsSeparating(a)`

Returns `true` if the function field element a is a separating element (has a non zero differential).

`IsConstant(a)`

Whether the algebraic function a is constant; if so it is returned as an element of the exact constant field.

`IsGlobalUnit(a)`

Whether the function field element a is a global unit, i.e. a constant (equivalent to `IsConstant`)

`IsGlobalUnitWithPreimage(a)`

Returns `true` and the preimage of the function field element a in the global unit group, `false` otherwise. The function field must be global.

`IsUnitWithPreimage(a)`

Returns `true` and the preimage of the order element a in the unit group of O if a is a unit, `false` otherwise. The function field has to be global.

42.11.7 Functions related to Norm and Trace

Multiplication by $a \in F$ or $a \in O$ defines a linear map of the vector space F over its coefficient field where F is a finite extension of its coefficient field. The following functions work with respect to this mapping.

`Trace(a)` `Norm(a)`
`MinimalPolynomial(a)`
`CharacteristicPolynomial(a)`

`RepresentationMatrix(a)`

Returns the matrix $M \in R^{n \times n}$ such that $a\,(\omega_1, \omega_2, \ldots, \omega_n) = (\omega_1, \omega_2, \ldots, \omega_n)\,M$, where $\omega_1, \omega_2, \ldots, \omega_n$ is a R-basis of the parent of the function field or order element a and R is the coefficient ring of the parent of a.

`Trace(a, R)`

The trace of the function field or order element a over R, a coefficient ring or field of the parent of a.

`Norm(a, R)`

The norm of the order or algebraic function field element a over R, a coefficient ring or field of the parent of a.

`CharacteristicPolynomial(a, R)`

The characteristic polynomial of the order or algebraic function field element a over R, a coefficient ring or field of the parent of a.

`MinimalPolynomial(a, R)`

The minimal polynomial of the order or algebraic function field element a over R, a coefficient ring or field of the parent of a.

`AbsoluteMinimalPolynomial(a)`

The minimal polynomial of the function field element a over the rational function field.

`RepresentationMatrix(a, R)`

Returns the matrix $M \in R^{n\times n}$ such that $a\,(\omega_1, \omega_2, \ldots, \omega_n) = (\omega_1, \omega_2, \ldots, \omega_n)\,M$, where $\omega_1, \omega_2, \ldots, \omega_n$ is a R-basis of the parent of the function field or order element a and R is a coefficient ring of the parent of a.

Example H42E26

```
> P<x> := PolynomialRing(Integers());
> N<n> := NumberField(x^6 - 6);
> P<x> := PolynomialRing(N);
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(y^8 - x^3*N.1^5);
> P<y> := PolynomialRing(F);
> F2<d> := FunctionField(y^3 + N.1*F!x - c);
> d^10;
(c^3 - 3*n*x*c^2 + 3*n^2*x^2*c - n^3*x^3)*d
> Norm(d^10);
-120*n^3*x^3*c^7 + 210*n^4*x^4*c^6 - 252*n^5*x^5*c^5 + 1260*x^6*c^4 -
    720*n*x^7*c^3 + (270*n^2*x^8 + n^5*x^3)*c^2 + (-60*n^3*x^9 - 60*x^4)*c +
    6*n^4*x^10 + 270*n*x^5
> Norm(d^10, CoefficientField(F));
13060694016*n^2*x^80 - 21767823360*n^5*x^75 + 97955205120*n^2*x^70 -
    43535646720*n^5*x^65 + 76187381760*n^2*x^60 - 15237476352*n^5*x^55 +
    12697896960*n^2*x^50 - 1209323520*n^5*x^45 + 453496320*n^2*x^40 -
    16796160*n^5*x^35 + 1679616*n^2*x^30
> Trace(d^10, CoefficientField(F));
0
> Trace(d^10);
0
```

42.11.8 Functions related to Orders and Integrality

`IntegralSplit(a, O)`

Split the element function field or order element a into a numerator and denominator with respect to the order O.

`Numerator(a, O)`

The numerator of the function field element a with respect to the order O.

`Numerator(a)`

Given an element a in a field of fractions of an order O return the numerator of a with respect to O.

`Numerator(a, O)`

Given an element a in a field of fractions of an order and an order O of a function field return the numerator of a with respect to O.

`Denominator(a, O)`

The denominator of the function field element a with respect to the order O.

`Denominator(a)`

Given an element a in a field of fractions of an order O return the denominator of a with respect to O.

`Denominator(a, O)`

Given an element a in a field of fractions of an order and an order O of a function field return the denominator of a with respect to O.

`Min(a, O)`
`Minimum(a, O)`

A generator of the ideal $R \cap (d \times a \times O)$ where R is the coefficient ring of the order O and d is the denominator of the function field or order element a wrt O (d is the second return value).

42.11.9 Functions related to Places and Divisors

`Evaluate(a, P)`

Evaluate the algebraic function a at the place P. If it is not defined at P, infinity is returned.

`Lift(a, P)`

Lift the element a of the residue class field of the place P (including infinity) to an algebraic function.

`Valuation(a, P)`

The valuation of the function field or order element a at the place P.

`Expand(a, P)`

`RelPrec`	RngIntElt	*Default :* 10
`AbsPrec`	RngIntElt	*Default :*

Expand the algebraic function a to a series of given precision at the place P and return the local parameter.

`Divisor(a)`
`PrincipalDivisor(a)`

The (principal) divisor (a) of the function field or order element a.

`Zeros(a)`
`Zeroes(a)`

A sequence containing the zeros of the algebraic function a.

`Zeros(F, a)`
`Zeroes(F, a)`

The zeros of the function field element a in the function field F.

`Poles(a)`

A sequence containing the poles of the algebraic function a.

`Poles(F, a)`

A sequence containing the poles of the function field element a in the function field F.

`Degree(a)`

The degree of the algebraic function a, being defined as the degree of the pole (or zero) divisor of a.

`CommonZeros(L)`

Return the common zeros of the function field elements in the sequence L.

`CommonZeros(F, L)`

Return the common zeros in the function field F of the function field elements in the sequence L.

Example H42E27

```
> R<x> := FunctionField(GF(9));
> P<y> := PolynomialRing(R);
> f := y^3 + y + x^5 + x + 1;
> F<a> := FunctionField(f);
> MinimalPolynomial(a);
y^3 + y + x^5 + x + 1
> RepresentationMatrix(a);
[                  0                   1                   0]
[                  0                   0                   1]
[    2*x^5 + 2*x + 2                   2                   0]
> O := IntegralClosure(ValuationRing(R), F);
> Denominator(a, O);
1/x^2
> O := IntegralClosure(PolynomialRing(GF(9)), F);
> Denominator(a, O);
1
> Zeros(a);
[ (x + 2, a), (x^3 + 2*x^2 + 1, a + x^3 + 2*x^2 + 1) ]
> Degree(a);
5
> P := RandomPlace(F, 2);
> P;
(x^2 + $.1^2*x + $.1^6, a + x^2 + $.1^5*x + 1)
> b := Evaluate(a, P);
> b;
$.1^3*$.1 + $.1^3
> c := Lift(b, P);
> c;
$.1^3*x + $.1^3
> Valuation(a, P);
0
> Valuation(a-c, P);
1
```

`Module(L, R)`

`IsBasis`	BoolElt	*Default :* `false`
`PreImages`	BoolElt	*Default :* `false`

The R-module generated by the function field elements in the sequence L as an abstract module, together with the map into the algebraic function field. The resulting modules can be used for intersection and inner sum computations.

If the optional parameter `IsBasis` is set `true` the function assumes that the given elements form a basis of the module to be computed.

If the optional parameter `PreImages` is set `true` then the preimages of the given elements under the map are returned as the third return value.

Both optional parameters are mainly used to save computation time.

`Relations(L, R)`

`Relations(L, R, m)`

The module of R-linear relations between the function field elements of the sequence L. The argument m is used for the following: Let the elements of L be $a_1, \ldots, a_n$, V be the relation module $\subseteq R^n$ and define $M := \{ \sum_{i=1}^{m} v_i a_i \,|\, v = (v_i)_i \in V \}$. The function tries to compute a generating system of V such that the corresponding generating system of M consists of "small" elements.

`Roots(f, D)`

Compute the roots of the polynomial f which lie in the Riemann-Roch space of the divisor D.

Example H42E28

This example shows the capability of `Module` and `Relations` with relative function fields.

```
> PR<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(PR);
> FR1<a> := FunctionField(y^3 - x);
> P<y> := PolynomialRing(FR1);
> FR2<c> := FunctionField(y^2 - a);
> MFR1F := MaximalOrderFinite(FR1);
> m, f := Module([c, c + a], MFR1F);
> f(m.1);
1
> f(m.2);
c
> m, f := Module([c, c + a], FR1);
> f(m.1);
c
> f(m.2);
1
> m;
KModule m of dimension 2 over FR1
```

```
> Relations([c, c + a], FR1, 1);
Vector space of degree 2, dimension 0 over FR1
User basis:
Matrix with 0 rows and 2 columns
```

42.11.10 Other Operations on Elements

`ProductRepresentation(a)`

Return a product representation for the function field or order element a.

`ProductRepresentation(Q, S)`

`PowerProduct(Q, S)`

Return the element given by the product representation of function field elements in the sequence Q and exponents in the sequence S. It is expensive to put large elements in product representation into sets or to test for them for equality.

`RationalFunction(a)`

`RationalFunction(a, R)`

Return the algebraic function a as a rational function in free variables with respect to the defining polynomial over the coefficient field.

If the ring R is provided it must appear in the tower of coefficient fields of the parent of a and the result is a polynomial over R with respect to all the defining polynomials of the extensions in between.

`Differentiation(x, a)`

The first differentiation resp. derivative of the function field element a with respect to the separating element x.

`Differentiation(x, n, a)`

The nth differentiation of the function field element a with respect to the separating element x. In characteristic zero the nth differentiation equals the nth derivative times $1/n!$.

`DifferentiationSequence(x, n, a)`

The 0-th up to the n-th differentiation of the function field element a with respect to the separating element x.

`PrimePowerRepresentation(x, k, a)`

Return the coefficients of the representation of the function field element a as a linear combination of k-th prime powers and powers of the function field element x. More precisely, let $p > 0$ be the characteristic of F. Then F^{p^k} is a subfield of F of index p^k and F can be viewed as a F^{p^k}-vector space. A basis is given by $1, x, \ldots, x^{p^k-1}$ for x a separating element. The function returns $\lambda_1, \ldots, \lambda_{p^k-1} \in F^{p^k}$ such that $a = \sum_i \lambda_i x^i$.

`Different(a)`

The different of the element a of an order of an algebraic function field.

`RationalReconstruction(e, f)`

For an element e of some some function field K with integral coefficients $e = \sum e_i \alpha^i$, $e_i \in k[x]$ and some polynomial $f \in k[x]$ find the (essentially) unique $E = \sum E_i \alpha_i$ with $E_i \in k(x)$ and $E_i = e_i \bmod f$, where the numerator and denominator of the E_i have degree bounded by half the degree of f. If such E_i exists they are unique, the corresponding element $\sum E_i \alpha^i$ for the function field will be returned as a second return value, the first being `true` to indicate success. If no such element exists, `false` will be returned.

`CoefficientHeight(a)`

`CoefficientHeight(a)`

The (naive) height of the element as defined as the largest degree of any coefficient or denominator polynomial occurring in the coefficients of a.

`CoefficientLength(a)`

`CoefficientLength(a)`

The (naive) length or size of the element, defined as the sum of the degrees of all polynomials occurring as coefficients or denominators in the coefficient representation of a.

Example H42E29

```
> F<z> := GF(13, 3);
> PF<x> := PolynomialRing(F);
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - ConstantField(FF1).1>;
> ProductRepresentation([Random(FF2, 2) : i in [1 .. 3]], [2, 3, 2]);
(((z^476*x + z^319)*b + (z^1861*x + z^439))*d^2 + ((z^348*x + z^931)*b +
    (z^328*x + z^2076))*d + (z^152*x + z^1723)*b + z^1044*x + z^1119)^2 *
(((z^1024*x + z^2085)*b + (z^798*x + z^335))*d^2 + ((z^310*x + z^932)*b +
    (z^281*x + z^1393))*d + (z^1844*x + z^66)*b + z^2127*x + z^1788)^3 *
```

```
(((z^1478*x + z^1782)*b + (z^687*x + z^1898))*d^2 + ((z^560*x + z^425)*b +
    (z^2081*x + z^164))*d + (z^60*x + z^890)*b + z^258*x + z^1739)^2
> ProductRepresentation($1);
[
    ((z^476*x + z^319)*b + (z^1861*x + z^439))*d^2 + ((z^348*x + z^931)*b +
        (z^328*x + z^2076))*d + (z^152*x + z^1723)*b + z^1044*x + z^1119,
    ((z^1024*x + z^2085)*b + (z^798*x + z^335))*d^2 + ((z^310*x + z^932)*b +
        (z^281*x + z^1393))*d + (z^1844*x + z^66)*b + z^2127*x + z^1788,
    ((z^1478*x + z^1782)*b + (z^687*x + z^1898))*d^2 + ((z^560*x + z^425)*b +
        (z^2081*x + z^164))*d + (z^60*x + z^890)*b + z^258*x + z^1739
]
[ 2, 3, 2 ]
> r := Random(FF2, 3);
> RationalFunction(r);
((z^1568*x^2 + z^1591*x + z^1260)*b + (z^746*x^2 + z^1405*x + z^1721))*y^2 +
    ((z^990*x^2 + z^689*x + z^470)*b + (z^1324*x^2 + z^195*x + z^1082))*y +
    (z^331*x^2 + z^1995*x + z^1521)*b + z^1323*x^2 + z^852*x + z^2162
> RationalFunction(r, CoefficientField(FF2));
((z^1568*x^2 + z^1591*x + z^1260)*b + (z^746*x^2 + z^1405*x + z^1721))*$.1^2 +
    ((z^990*x^2 + z^689*x + z^470)*b + (z^1324*x^2 + z^195*x + z^1082))*$.1 +
    (z^331*x^2 + z^1995*x + z^1521)*b + z^1323*x^2 + z^852*x + z^2162
> RationalFunction(r, PF);
(z^1568*x^2 + z^1591*x + z^1260)*$.1^2*$.2 + (z^746*x^2 + z^1405*x +
    z^1721)*$.1^2 + (z^990*x^2 + z^689*x + z^470)*$.1*$.2 + (z^1324*x^2 +
    z^195*x + z^1082)*$.1 + (z^331*x^2 + z^1995*x + z^1521)*$.2 + z^1323*x^2 +
    z^852*x + z^2162
> Differentiation(FF2!x, r);
((z^836*x^2 + z^2140*x + z^1077)/x*b + (z^929*x + z^1405))*d^2 + ((z^258*x^2 +
    z^1238*x + z^287)/x*b + (z^1507*x + z^195))*d + (z^1795*x^2 + z^348*x +
    z^1338)/x*b + z^1506*x + z^852
> Differentiation(FF2!b, r);
((z^1112*x + z^1588)*b + (z^1019*x^2 + z^127*x + z^1260))*d^2 + ((z^1690*x +
    z^378)*b + (z^441*x^2 + z^1421*x + z^470))*d + (z^1689*x + z^1035)*b +
    z^1978*x^2 + z^531*x + z^1521
>  Differentiation(FF2!d, r);
>> Differentiation(FF2!d, r);
                  ^
Runtime error in 'Differentiation': First element must be a separating element
> MFR2I := MaximalOrderInfinite(FF2);
> Different(Numerator(r, MFR2I));
[ [ (z^1095*x^5 + z^849*x^4 + z^111*x^3 + z^853*x^2 + z^224*x + z^1340)/x^6,
    (z^666*x^4 + z^1902*x^3 + z^2086*x^2 + z^1119*x + z^1973)/x^5 ], [
    (z^1673*x^5 + z^699*x^4 + z^1465*x^3 + z^1060*x^2 + z^761*x + z^1979)/x^6,
    (z^1034*x^4 + z^1185*x^3 + z^833*x^2 + z^2044*x + z^1701)/x^5 ], [
    (z^516*x^5 + z^1545*x^4 + z^509*x^3 + z^832*x^2 + z^606*x + z^700)/x^6,
    (z^1033*x^4 + z^983*x^3 + z^1375*x^2 + z^89*x + z^271)/x^5 ] ]
```

42.12 Ideals

Ideals for function field orders O are O-modules $I \subseteq F$ for which there is a $d \in F$ such that $dI \subseteq O$ is a non-zero ideal of O, that is they are fractional ideals of O. Over the coefficient ring of O they are also free modules of rank n, where n equals the degree $[F : k(x, \alpha_1, \ldots, \alpha_r)]$.

42.12.1 Creation of Ideals

`ideal< O | a1, a2, ... , am >`

Given an order O, as well as elements $a_1, a_2, \ldots, a_m$ coercible into the field of fractions F of O, create the fractional ideal of O generated by these elements.

Note that, contrary to the general case for the constructors, the right hand side elements are not necessarily contained in the left hand side.

`ideal< O | T, d >`

The ideal of the order O of an algebraic function field whose basis is the matrix or dedekind module T over the coefficient ring of O divided by the element d of the denominator ring of O.

`ideal< O | T, S >`

`ideal< O | T, I1,...,In >`

The ideal of the order O of an algebraic function field whose basis is the matrix T over the coefficient ring of O along with the coefficient ideals $I_1, ..., I_n$ or those in S.

`x * O`

`O * x`

Create the ideal $x * O$ where x is coercible into the function field of the order O.

`Ideal(P)`

Create a prime ideal corresponding to the place P.

`Ideals(D)`

Create two ideals of the 'finite' and 'infinite' maximal order respectively corresponding to the divisor D.

`O !! I`

Return the ideal I as an ideal of the order O.

42.12.2 Parent and Category

`Parent(I)` `Category(I)`

42.12.3 Arithmetic Operators

`I + J` `I * J`
`I / J` `I ^ k`

The ideal J is required to be invertible. The ideal I is required to be invertible for negative k.

`c * I` `I * c` `I / c`

`c / I`

The principal ideal generated by the ring element c divided by the ideal I.

`IdealQuotient(I, J)`
`ColonIdeal(I, J)`

The colon ideal $[I : J]$ of elements which multiply all elements of the ideal J into the ideal I.

`ChineseRemainderTheorem(I1, I2, e1, e2)`
`CRT(I1, I2, e1, e2)`

Returns an element e of the order O such that $(e_1 - e)$ is in the ideal I_1 of O and $(e_2 - e)$ is in the ideal I_2.

42.12.4 Roots of Ideals

`IsPower(I, n)`

Return whether the ideal I has an nth root and if so return an nth root.

`Root(I, n)`

Return the nth root of the ideal I.

`IsSquare(I)`

Return whether the ideal I is a square and if so return a square root.

`SquareRoot(I)`
`Sqrt(I)`

Return a square root of the ideal I.

Example H42E30

A simple creation of an ideal and the use of `IsSquare` is shown below.

```
> P<x> := PolynomialRing(GF(79));
> P<y> := PolynomialRing(P);
> Fa<a> := FunctionField(y^2 - x);
> P<y> := PolynomialRing(Fa);
> Fb<b> := FunctionField(y^2 - a);
> P<y> := PolynomialRing(Fb);
> Fc<c> := FunctionField(y^2 + a*b);
> I := a*b*c*MaximalOrderInfinite(Fc);
> IsSquare(I^2);
true Fractional ideal of Maximal Order of Fc over Maximal Order of Fb over
Maximal Equation Order of Fa over Valuation ring of Univariate rational function
field over GF(79) with generator 1/x
Basis:
Pseudo-matrix over Maximal Order of Fb over Maximal Equation Order of Fa over
Valuation ring of Univariate rational function field over GF(79) with generator
1/x
Fractional ideal of Maximal Order of Fb over Maximal Equation Order of Fa over
Valuation ring of Univariate rational function field over GF(79) with generator
1/x
Generators:
1
((78*x^2 + 71*x + 78)/x^2*a + (23*x^2 + 15*x + 78)/x^2)*b + (67*x^2 + 60*x +
    78)/x^2*a + (3*x^2 + 47*x + 78)/x * ( 1 0 )
Fractional ideal of Maximal Order of Fb over Maximal Equation Order of Fa over
Valuation ring of Univariate rational function field over GF(79) with generator
1/x
Generators:
1
((59*x^2 + 44*x + 78)/x^2*a + (4*x^2 + 48*x + 78)/x^2)*b + (29*x^2 + 46*x +
    78)/x^2*a + (71*x + 78)/x * ( 0 1 )
> _, II := $1;
> II eq I;
true
> MaximalOrderFinite(Fc)!!I;
Ideal of Maximal Order of Fc over Maximal Equation Order of Fb over Maximal
Equation Order of Fa over Univariate Polynomial Ring in x over GF(79)
Generators:
a*b*c
a*b*c
```

42.12.5 Equality and Membership

`I eq J` `I ne J` `I in S` `I notin S`

42.12.6 Predicates on Ideals

`IsZero(I)`

Returns `true` if and only if the ideal I is the zero ideal of the order O.

`IsOne(I)`

Returns `true` if and only if the ideal I is the identity ideal of the order O, i.e. $I = O$.

`IsIntegral(I)`

Returns `true` if and only if the ideal I is integral (a true ideal of its order).

`IsPrime(I)`

Returns `true` if and only if the ideal I is prime.

`IsPrincipal(I)`

Returns `true` and a generator if the fractional ideal I is principal, `false` otherwise. The function field has to be global.

42.12.6.1 Predicates on Prime Ideals

`IsInert(P)`

Return `true` if the inertia degree of the prime ideal P is the degree of its order.

`IsInert(P, O)`

Return `true` if there is an inert ideal in the order O above the prime ideal P.

`IsRamified(P)`

Return `true` if the ramification index of the prime ideal P is not 1.

`IsRamified(P, O)`

Return `true` if there is a ramified ideal in the order O above the prime ideal P.

`IsSplit(P)`

Return `true` if the prime ideal P is not the only ideal lying above the prime ideal it lies above.

`IsSplit(P, O)`

Return `true` if there are at least 2 distinct ideals which lie in the order O above the prime ideal P.

`IsTamelyRamified(P)`

Return whether the prime ideal P is not wildly ramified.

`IsTamelyRamified(P, O)`

Return whether the prime ideal P is not wildly ramified in the order O.

`IsTotallyRamified(P)`

Return whether the ramification index of the prime ideal P is the same as the degree of its order over its coefficient order.

`IsTotallyRamified(P, O)`

Return whether there are any totally ramified ideals in the order O lying above the prime ideal P.

`IsTotallySplit(P)`

Return whether there are as many ideals as the degree of the order of the prime ideal P lying over the prime P lies over.

`IsTotallySplit(P, O)`

Return whether there are as many ideals of the order O which lie above the prime ideal P as the degree of O.

`IsUnramified(P)`

Return whether the ramification index of the prime ideal P is 1.

`IsUnramified(P, O)`

Return whether all the ideals of the order O which lie above the prime ideal P are unramified.

`IsWildlyRamified(P)`

Return whether the ramfication index of the prime ideal P is a multiple of the characteristic of the residue field of P.

`IsWildlyRamified(P, O)`

Return whether any of the ideals of the order O which lie above the prime ideal P are wildly ramified.

42.12.7 Further Ideal Operations

`I meet J`

The intersection of the ideals I and J.

`Gcd(I, J)`

Given invertible ideals of an order O, returns the greatest common divisor of the ideals I and J.

`Lcm(I, J)`

Given invertible ideals of an order O, returns the least common multiple of the ideals I and J.

`Factorization(I)`
`Factorisation(I)`

Factorization of the ideal I (as sequence of prime ideal, exponent pairs). The order must be maximal.

`Decomposition(O, p)`

A sequence containing all prime ideals of the order O lying above the prime element or ideal p of the coefficient ring of O.

`Decomposition(O)`

A sequence containing all prime ideals of the 'infinite' maximal order O.

`DecompositionType(O, p)`

Sequence of tuples of residue degrees and ramification indices of the prime ideals of the order O lying over p, a prime polynomial or ideal or element of valuation ring of valuation 1.

`DecompositionType(O)`

Sequence of tuples of residue degrees and ramification indices of the prime ideals of the 'infinite' maximal order O.

`MultiplicatorRing(I)`

Returns the multiplicator ring of the ideal I of the order O, that is, the subring of elements of the field of fractions of O that multiply I into itself.

`pMaximalOrder(O, p)`

The p-maximal over order of the order O where p is a prime ideal of the coefficient ring of O. See also the description in Section 42.2.3.

`pRadical(O, p)`

Returns the p-radical of an order O for a prime ideal p of the coefficient ring of O, defined as the ideal consisting of elements of O for which some power lies in the ideal pO.

It is possible to call this function even if p is not prime. In this case the p-trace-radical will be computed, i.e.

$$\{x \in F \mid \mathrm{Tr}(xO) \subseteq C\}$$

for F the field of fractions of O and C the order of p (if p is an ideal) or the parent of p otherwise. If p is square free and all divisors are larger than the field degree, this is the intersection of the radicals for all l dividing p.

`Valuation(a, P)`

`Valuation(I, P)`

The valuation of a or the ideal I at the prime ideal P. The element a must be coercible into the field of fractions of P's order.

`Order(I)`

The order of the ideal I.

`Denominator(I)`

The "smallest" element d of the coefficient ring of the order O of the ideal I such that $dI \subseteq O$.

`Minimum(I)`

A generator m of the ideal $R \cap dI$ where R is the coefficient ring of the ideal's order and d is the denominator of the ideal I (d is the second return value).

`I meet R`

The intersection of the ideal I with a coefficient ring R of its order.

`IntegralSplit(I)`

The integral ideal dI and d, where d is the denominator of the ideal I.

`Norm(I)`

The norm of the ideal I, as element of the coefficient field of the algebraic function field to which I belongs.

`TwoElement(I)`

Given an ideal I with function field F as the function field of its order O, returns two elements $a, b \in F$ such that $I = aO + bO$.

`Generators(I)`

Given a (fractional) ideal I of the order O, return a sequence of elements of the function field F that generate I as an ideal.

`Basis(I)`

`Basis(I, R)`

A basis of the ideal I as a free module over the coefficient ring of its order, coerced into the ring R if given.

`BasisMatrix(I)`

Let $(b_1, \ldots, b_n)$ be the basis of the ideal I and let $(\omega_1, \ldots, \omega_n)$ be the basis of the order O. A matrix B with coefficients in the rational function field is returned such that $(b_1, \ldots, b_n) = (\omega_1, \ldots, \omega_n)B^t$.

`TransformationMatrix(I)`

Let $(b_1, \ldots, b_n)$ be the basis of the ideal I and let $(\omega_1, \ldots, \omega_n)$ be the basis of the order O. A matrix T with coefficients in the coefficient ring of O and a denominator d are returned such that $(b_1, \ldots, b_n) = (\omega_1, \ldots, \omega_n)T^t/d$.

`CoefficientIdeals(I)`

The coefficient ideals of the ideal I in a relative extension. These are the ideals $\{A_i\}$ of the coefficient ring of the order of I such that for every element $e \in I$, $e = \sum_i a_i * b_i$ where $\{b_i\}$ is the basis returned for I and each $a_i \in A_i$.

`Different(I)`

The different of the (possibly fractional) ideal I of an order of an algebraic function field.

`Codifferent(I)`

The codifferent of the ideal I. This will be the inverse of the different of I if I is an ideal of a maximal order.

`Divisor(I)`

The divisor corresponding to the ideal factorization of the ideal I.

`Divisor(I, J)`

The divisor corresponding to the ideal factorization of the ideals I and J belonging to the 'finite' and 'infinite' maximal order.

Example H42E31

```
> PR<x> := FunctionField(Rationals());
> P<y> := PolynomialRing(PR);
> FR1<a> := FunctionField(y^3 - x + 1/x^3);
> P<y> := PolynomialRing(FR1);
> FR2<c> := FunctionField(y^2 - a/x^3*y + 1);
> I := ideal<MaximalOrderFinite(FR2) |
> [ x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
> 468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237, x^9 + 1639*x^8 +
> 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 + 468363837*x^3 +
> 242625823*x^2 + 68744019*x + 8052237, (x^15 + 1639*x^14 + 863249*x^13 +
> 148609981*x^12 + 404988066*x^11 + 567876948*x^10 + 468363837*x^9 +
> 242625823*x^8 + 68744019*x^7 + 8052237*x^6)*c, (x^15 + 1639*x^14 +
> 863249*x^13 + 148609981*x^12 + 404988066*x^11 + 567876948*x^10 +
> 468363837*x^9 + 242625823*x^8 + 68744019*x^7 + 8052237*x^6)*c ]>;
> I;
Ideal of Maximal Order of FR2 over Maximal Order of FR1 over Univariate
Polynomial Ring in x over Rational Field
Basis:
Pseudo-matrix over Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x over Rational
Field
Generator:
x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237 * ( 1 0 )
Fractional ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Generator:
(x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237)/x^2 * ( 0 1 )
> J := ideal<MaximalOrderFinite(FR2) |
> [ x^3 + 278*x^2 + 164*x + 742, x^3 + 278*x^2 + 164*x + 742, (x^9 + 278*x^8 +
> 164*x^7 + 742*x^6)*c, (x^9 + 278*x^8 + 164*x^7 + 742*x^6)*c ]>;
> J;
Ideal of Maximal Order of FR2 over Maximal Order of FR1 over Univariate
Polynomial Ring in x over Rational Field
Basis:
Pseudo-matrix over Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x over Rational
Field
Generator:
x^3 + 278*x^2 + 164*x + 742 * ( 1 0 )
Fractional ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Generator:
```

```
(x^3 + 278*x^2 + 164*x + 742)/x^2 * ( 0 1 )
> Generators(J);
[
    x^3 + 278*x^2 + 164*x + 742,
    x^3 + 278*x^2 + 164*x + 742,
    (x^7 + 278*x^6 + 164*x^5 + 742*x^4)*c,
    (x^7 + 278*x^6 + 164*x^5 + 742*x^4)*c
]
> TwoElement(J);
x^3 + 278*x^2 + 164*x + 742
((3/2*x^10 + 419*x^9 + 802*x^8 + 1441*x^7 + 1484*x^6)*a^2 + (3/2*x^8 + 417*x^7 +
    246*x^6 + 1113*x^5)*a + (3/2*x^8 + 837/2*x^7 + 663*x^6 + 1359*x^5 +
    1113*x^4))*c + (x^6 + 277*x^5 - 114*x^4 + 578*x^3 - 742*x^2)*a^2 + (3*x^5 +
    834*x^4 + 492*x^3 + 2226*x^2)*a - x^3 - 278*x^2 - 164*x - 742
> Minimum(I);
Ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x over Rational
Field
Generator:
x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237 1
> Basis(J);
[ 1, x^6*c ]
> Basis(I);
[ 1, x^6*c ]
> I eq J;
false
> II, d := IntegralSplit(I);
> II;
Ideal of Maximal Order of FR2 over Maximal Order of FR1 over Univariate
Polynomial Ring in x over Rational Field
Basis:
Pseudo-matrix over Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x over Rational
Field
Generator:
x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237 * ( 1 0 )
Fractional ideal of Maximal Order of FR1 over Univariate Polynomial Ring in x
over Rational Field
Generator:
(x^9 + 1639*x^8 + 863249*x^7 + 148609981*x^6 + 404988066*x^5 + 567876948*x^4 +
    468363837*x^3 + 242625823*x^2 + 68744019*x + 8052237)/x^2 * ( 0 1 )
> d;
1
> IsIntegral(I);
true
> GCD(I, J)*LCM(I, J) eq I*J;
```

```
true
```

42.12.7.1 Functions on Prime Ideals

`RamificationIndex(I)`

`RamificationDegree(I)`

The ramification index of the prime ideal I over the corresponding prime of its coefficient ring.

`Degree(I)`

`InertiaDegree(I)`

`ResidueClassDegree(I)`

The residue class degree (inertia degree) of the prime ideal I over the corresponding prime of its coefficient ring.

`ResidueClassField(I)`

The residue class field of the prime ideal I and the residue class mapping.

`Place(I)`

The place corresponding to the prime ideal I, where I is defined over the 'finite' or 'infinite' maximal order.

`SafeUniformizer(P)`

For an ideal I of a maximal order in a function field, this returns an element which has valuation 1 at the given prime and which has valuation 0 at all other primes lying over the same prime of the underlying rational function field. See also `LocalUniformizer` (for places) below.

Example H42E32

```
> R<x> := FunctionField(GF(3));
> P<y> := PolynomialRing(R);
> f := y^4 + x*y + x^4 + x + 1;
> F<a> := FunctionField(f);
> O := MaximalOrderFinite(F);
> x*O;
Ideal of O
Generator:
x
> L := Factorization(x*O);
> L;
[ <Ideal of O
Generators:
```

```
x
a^2 + a + 2, 1>, <Ideal of O
Generators:
x
a^2 + 2*a + 2, 1> ]
> P1 := L[1][1];
> P2 := L[2][1];
> BasisMatrix(P1);
[x 0 0 0]
[0 x 0 0]
[2 1 1 0]
[1 1 0 1]
> P1 meet P2 eq x*O;
true
> IsPrime(P1);
true
> Place(P1);
(x, a^2 + a + 2)
```

42.13 Places

42.13.1 Creation of Structures

`Places(F)`

The set of places of the algebraic function field F/k.

42.13.2 Creation of Elements

42.13.2.1 General Function Field Places

`Decomposition(F, P)`

A sequence containing all places of F/k lying above the place P of the coefficient field $k(x)$ of F. The function field F must be a finite extension of $k(x)$.

`DecompositionType(F, P)`

Sequence of tuples of residue degrees and ramification indices of the places of F/k lying over the place P of the coefficient field $k(x)$ of F. The function field F must be a finite extension of $k(x)$.

`Zeros(a)`

A sequence containing all zeros of the algebraic function a.

`Poles(a)`

A sequence containing all poles of the algebraic function a.

`S ! I`
`Place(I)`

The place corresponding to the prime ideal I, where I is defined over the 'finite' or 'infinite' maximal order and S is the set of places of a function field.

`Support(D)`

Sequences containing the places and exponents occurring in the divisor D.

`AssignNames(~P, s)`

Change the print name employed when displaying P to be the first element in the sequence of strings s which must have length 1.

`InfinitePlaces(F)`

The infinite places of the function field F.

42.13.2.2 Global Function Field Places

F/k denotes a global function field in this section.

`HasPlace(F, m)`

Returns `true` and a place of degree m if and only if there exists such in the function field F/k; `false` otherwise.

`HasRandomPlace(F, m)`

Returns `true` and a random place of degree m in the function field F/k or (`false` if there are none).

`RandomPlace(F, m)`

Returns a random place of degree m in the function field F/k or throws an error if there is none.

`Places(F, m)`

A sequence containing the places of degree m of the function field F/k.

Example H42E33

Some creation of places is illustrated below.

```
> P<t> := PolynomialRing(Integers());
> N := NumberField(t^2 + 2);
> P<x> := PolynomialRing(N);
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(y^4 + x^5 - N.1^7);
> F;
Algebraic function field defined over Univariate rational function field over N
by
y^4 + x^5 + 8*N.1
> Zeros(c);
[ (x^5 + 8*N.1, c + x^5 + 8*N.1) ]
> P<y> := PolynomialRing(F);
> F2<d> := FunctionField(y^2 + F!N.1);
> Decomposition(F2, $1[1]);
[ (x^5 + 8*N.1, c + 2*x^5 + 16*N.1) ]
> DecompositionType(F2, $2[1]);
[ <2, 1> ]
> Places(F2)!$3[1];
(x^5 + 8*N.1, c + 2*x^5 + 16*N.1)
```

42.13.3 Related Structures

42.13.3.1 Parent and Category

The sets of function field places form the MAGMA category `PlcFun`. The notional power structure exists as parent but allows no operations.

`FunctionField(S)`

The corresponding function field of the set of places S.

`DivisorGroup(F)`

The group of divisors of the algebraic function field F/k, which is the free abelian group generated by the elements of the set of places of F/k.

42.13.4 Structure Invariants

42.13.4.1 General function fields

`WeierstrassPlaces(F)`

`SeparatingElement` FLDFUNGELT *Default :*

The Weierstrass places of the function field F/k. The semantics of calling `WeierstrassPlaces()` with F/k or the zero divisor of F/k are identical. See the description of `WeierstrassPlaces` on page 1170.

42.13.4.2 Global Function Fields

F/k denotes a global function field in this section.

`NumberOfPlacesOfDegreeOneOverExactConstantField(F, m)`

`NumberOfPlacesOfDegreeOneECF(F, m)`

The number of places of degree one in the constant field extension of degree m of the function field F/k. Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(F, m)`

`NumberOfPlacesOfDegreeOneECFBound(F, m)`

The minimum of the Serre and Ihara bound on the number of places of degree one in the constant field extension of degree m of the function field F/k. Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

`NumberOfPlacesOfDegreeOverExactConstantField(F, m)`

`NumberOfPlacesDegECF(F, m)`

The number of places of degree m of the function field F/k. Contrary to the `Degree()` function the degree is here taken over the respective exact constant fields.

42.13.5 Structure Predicates

`S1 eq S2` `S1 ne S2`

42.13.6 Element Operations

42.13.6.1 Parent and Category

`Parent(P)` `Category(P)`

42.13.6.2 Arithmetic Operators

`- P`

`P1 + P2` `P1 - P2` `k * P` `P div k` `P mod k`

`Quotrem(P, k)`

Returns divisors D_1, D_2 such that the place $P = kD_1 + D_2$ and the exponents in D_2 are of absolute value less than $|k|$. The operations `div` and `mod` yield D_1 resp. D_2.

42.13.6.3 Equality and Membership

`P1 eq P2` `P1 ne P2` `P in S` `P notin S`

42.13.6.4 Predicates on Elements

`IsFinite(P)`

Returns `true` if the place P is a 'finite' place.

`IsWeierstrassPlace(P)`

Whether the degree one place P is a Weierstraß place of its function field F. See the description of `WeierstrassPlaces` on page 1170.

42.13.6.5 Other Element Operations

`FunctionField(P)`

The function field that corresponds to the place P.

`Degree(P)`

The degree of the place P over the constant field of definition k.

`RamificationIndex(P)`
`RamificationDegree(P)`

The ramification index of the place P over its subplace of the rational function field $k(x)$ (the function field of P must be a finite extension of $k(x)$).

`InertiaDegree(P)`
`ResidueClassDegree(P)`

The degree of inertia (or residue class degree) of a place P over the corresponding subplace of the rational function field (the function field of P must be a finite extension of $k(x)$)

`Minimum(P)`

A monic prime polynomial in $k[x]$ or $1/x$ or an ideal, corresponding to the place of the coefficient field of the function field of the place P which P lies above (the function field of P must be a finite extension of $k(x)$).

`ResidueClassField(P)`

The residue class field of the place P and the map from the order of the place into the field.

`Evaluate(a, P)`

Evaluate the algebraic function a at the place P. If it is not defined at P, infinity is returned.

`Lift(a, P)`

Lift the element a of the residue class field of the place P (including infinity) to an algebraic function.

`TwoGenerators(P)`

Two algebraic functions having the place P as their unique common zero.

`LocalUniformizer(P)`

`UniformizingElement(P)`

A local uniformizing parameter at the place P.

`SafeUniformizer(P)`

The valuation of the element a at the place P.

`Ideal(P)`

Create a prime ideal corresponding to the place P.

`Norm(P)`

The divisor of the norm of the ideal of the place P.

Example H42E34

```
> R<x> := FunctionField(GF(9));
> P<y> := PolynomialRing(R);
> f := y^4 + (2*x^5 + x^4 + 2*x^3 + x^2)*y^2 + x^8
>      + 2*x^6 + x^5 +x^4 + x^3 + x^2;
> F<a> := FunctionField(f);
> Genus(F);
7
> NumberOfPlacesDegECF(F, 2);
28
> P := RandomPlace(F, 2);
> P;
(x^2 + $.1^2*x + $.1^7, a + $.1^5*x + $.1^5)
> LocalUniformizer(P);
x^2 + $.1^2*x + $.1^7
> TwoGenerators(P);
x^2 + $.1^2*x + $.1^7 a + $.1^5*x + $.1^5
> ResidueClassField(P);
Finite field of size 3^4
> Evaluate(1/LocalUniformizer(P), P);
Infinity
> Valuation(1/LocalUniformizer(P), P);
-1
```

42.13.7 Completion at Places

```
Completion(F, p)
Completion(O, p)
```

`Precision` RngIntElt *Default :* 20

The completion of the algebraic function field F or an order O of such at the place p of F or the function field of O. The map from F or O into the series ring is returned also.

The series ring returned is an infinite precision ring whose default precision for elements is given by the `Precision` parameter.

42.14 Divisors

42.14.1 Creation of Structures

```
DivisorGroup(F)
```

Create the group of divisors of the algebraic function field F/k.

42.14.2 Creation of Elements

```
Divisor(P)
Div ! P
1 * P
```

Given a place P in a function field, return the prime divisor $1 * P$.

```
Div ! a
Divisor(a)
```

Given an algebraic function a, return the principal divisor (a).

```
Div ! I
Divisor(I)
```

The divisor corresponding to the factorization of the ideal I.

```
Divisor(I, J)
```

The divisor corresponding to the ideal factorization of the ideals I and J belonging to the 'finite' and 'infinite' maximal order.

```
Identity(G)
Id(G)
```

Given the group G of divisors of a function field, return the zero divisor.

`CanonicalDivisor(F)`

A canonical divisor of the function field F/k.

`DifferentDivisor(F)`

The different divisor of the underlying extension of the function field $F/k(x)$.

`AssignNames(~D, s)`

Change the print name employed when displaying D to be the contents of s which must have length 1 in this case.

42.14.3 Related Structures

42.14.3.1 Parent and Category

The group of divisors form the MAGMA category `DivFun`. The notional power structure exists as parent but allows no operations.

`FunctionField(G)`

Given the group G of divisors of a function field F/k, return F.

`Places(F)`

The set of places of the algebraic function field F/k.

42.14.4 Structure Invariants

`NumberOfSmoothDivisors(n, m, P)`

The number of effective divisors of degree less equal n who consist of places of degree less equal m only. The sequence element $P[i]$ contains the (generic) number of places of degree $1 \leq i \leq \min\{n, m\}$. The formula used is described in [Heß99].

`DivisorOfDegreeOne(F)`

A divisor of degree one over the exact constant field of the global function field F/k.

42.14.5 Structure Predicates

`Div1 eq Div2` `Div1 ne Div2`

42.14.6 Element Operations

42.14.6.1 Arithmetic Operators

`- D`

`D1 + D2` `D1 - D2` `k * D` `D div k` `D mod k`

`P + D` `D + P` `D - P` `P - D`

`Quotrem(D, k)`

Returns divisors D_1, D_2 such that the divisor $D = kD_1 + D_2$ and the exponents in D_2 are of absolute value less than $|k|$. The operations `div` and `mod` yield D_1 resp. D_2.

`GCD(D1, D2)`
`Gcd(D1, D2)`
`GreatestCommonDivisor(D1, D2)`

The greatest common divisor of the divisors $D1$ and $D2$.

`LCM(D1, D2)`
`Lcm(D1, D2)`
`LeastCommonMultiple(D1, D2)`

The least common multiple of the divisors $D1$ and $D2$.

42.14.6.2 Equality, Comparison and Membership

`D1 eq D2` `D1 ne D2`

`D1 le D2` `D1 lt D2` `D1 ge D2` `D1 gt D2`

`D in Div` `D notin Div`

42.14.6.3 Predicates on Elements

`IsZero(D)`

`IsEffective(D)` `IsPositive(D)`

`IsSpecial(D)` `IsPrincipal(D)`

`IsCanonical(D)`

Returns `true` iff the divisor D is canonical and a differential having D as its divisor.

Example H42E35

We show some simple creations and operations on divisors.

```
> PF<x> := PolynomialRing(GF(13, 2));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b>;
> CanonicalDivisor(FF2);
Complementary divisor of Divisor in ideal representation:
Ideal of Maximal Equation Order of FF2 over Maximal Equation Order of FF1 over
Univariate Polynomial Ring in x over GF(13^2)
Generator:
1, Fractional ideal of Maximal Order of FF2 over Maximal Equation Order of FF1
over Valuation ring of Univariate rational function field over GF(13^2) with
generator 1/x
Generator:
x^2
> IsCanonical($1);
true
> D := Divisor(b) + Divisor(d);
> E := Divisor(Random(FF2, 2)*MaximalOrderFinite(FF2),
> Random(FF2, 2)*MaximalOrderInfinite(FF2));
> d := D + E;
> d;
Divisor in reduced representation:
Dtilde :
Divisor in ideal representation:
Fractional ideal of Maximal Equation Order of FF2 over Maximal Equation Order of
FF1 over Univariate Polynomial Ring in x over GF(13^2)
Basis:
Pseudo-matrix over Maximal Equation Order of FF1 over Univariate Polynomial Ring
in x over GF(13^2)
Ideal of Maximal Equation Order of FF1 over Univariate Polynomial Ring in x over
GF(13^2)
Generator:
1 * ( 1 0 0 )
Ideal of Maximal Equation Order of FF1 over Univariate Polynomial Ring in x over
GF(13^2)
Generator:
1 * ( 0 1 0 )
Fractional ideal of Maximal Equation Order of FF1 over Univariate Polynomial
Ring in x over GF(13^2)
Generators:
1
($.1^137*x^12 + $.1^80*x^11 + $.1^22*x^10 + $.1^79*x^9 + $.1^88*x^8 +
   $.1^138*x^7 + $.1^130*x^6 + $.1^127*x^5 + $.1^163*x^4 + $.1^78*x^3 + 6*x^2 +
   $.1^41*x + $.1^146)/(x^12 + $.1^166*x^11 + $.1^50*x^10 + $.1^136*x^9 +
```

```
   $.1^32*x^8 + $.1^46*x^7 + $.1^134*x^6 + $.1^64*x^5 + 8*x^4 + $.1^93*x^3 +
   $.1^153*x^2 + $.1^162*x)*b + ($.1^24*x + $.1^153)/(x^11 + $.1^166*x^10 +
   $.1^50*x^9 + $.1^136*x^8 + $.1^32*x^7 + $.1^46*x^6 + $.1^134*x^5 +
   $.1^64*x^4 + 8*x^3 + $.1^93*x^2 + $.1^153*x + $.1^162) * ( $.1^161*x^11 +
   $.1^74*x^10 + $.1^145*x^9 + $.1^72*x^8 + $.1^122*x^7 + $.1^123*x^6 + 3*x^5 +
   $.1^133*x^4 + 2*x^3 + $.1^105*x^2 + $.1^102*x $.1^48*x^11 + $.1^82*x^10 +
   4*x^9 + $.1^102*x^8 + $.1^145*x^7 + $.1^118*x^6 + $.1^129*x^5 + $.1^102*x^4
   + $.1^138*x^3 + $.1^146*x^2 + $.1^134*x 1 ) , Ideal of Maximal Order of FF2
over Maximal Equation Order of FF1 over Valuation ring of Univariate rational
function field over GF(13^2) with generator 1/x
Basis:
Pseudo-matrix over Maximal Equation Order of FF1 over Valuation ring of
Univariate rational function field over GF(13^2) with generator 1/x
Ideal of Maximal Equation Order of FF1 over Valuation ring of Univariate
rational function field over GF(13^2) with generator 1/x
Generator:
1/x^2 * ( 1 0 0 )
Ideal of Maximal Equation Order of FF1 over Valuation ring of Univariate
rational function field over GF(13^2) with generator 1/x
Generators:
1/x^3
($.1^21*x^3 + $.1^86*x^2 + $.1^151*x + $.1^48)/x^6*b + $.1^79/x^3 * ( 0 1 0 )
Ideal of Maximal Equation Order of FF1 over Valuation ring of Univariate
rational function field over GF(13^2) with generator 1/x
Generator:
1/x^3*b * ( 0 0 1 ) ,
r : 0,
A :
Divisor in ideal representation:
Ideal of Maximal Equation Order of FF2 over Maximal Equation Order of FF1 over
Univariate Polynomial Ring in x over GF(13^2)
Generator:
1, Fractional ideal of Maximal Order of FF2 over Maximal Equation Order of FF1
over Valuation ring of Univariate rational function field over GF(13^2) with
generator 1/x
Generators:
x
x,
a :
(x)^-1 * (b)
```

A nicer (but potentially more expensive) way to print, would be to ensure the divisor had a representation as a linear combination of places and exponents.

```
> p, e := Support(d);
> d;
4*(x, (($.1^24*x + 9)*b + ($.1^133*x + $.1^117))*d^2 + (($.1^83*x + $.1^36)*b +
    ($.1^97*x + $.1^2))*d + ($.1^101*x + $.1^165)*b + $.1^108*x) + (x + $.1^102,
    (($.1^141*x + $.1^113)*b + ($.1^157*x + $.1^48))*d^2 + (($.1^94*x + $.1^92)*b
```

```
    + ($.1^167*x + $.1^79))*d + ($.1^36*x + $.1^85)*b + $.1^18*x + 6) + (x^2 +
    $.1^47*x + 8, (($.1^19*x^3 + $.1^155*x^2 + $.1^75*x + $.1^106)*b + (8*x^3 +
    $.1^131*x^2 + $.1^125*x + $.1^46))*d^2 + (($.1^86*x^3 + $.1^11*x^2 +
    $.1^141)*b + ($.1^94*x^3 + $.1^127*x^2 + 6*x + $.1^57))*d + ($.1^68*x^3 +
    $.1^82*x^2 + $.1^52*x + $.1^69)*b + $.1^95*x^3 + $.1^55*x^2 + $.1^30*x +
    $.1) + (x^8 + $.1^138*x^7 + $.1^91*x^6 + $.1^59*x^5 + $.1^25*x^4 +
    $.1^74*x^3 + 6*x^2 + $.1^153*x + 5, (($.1^86*x^10 + 12*x^9 + $.1^5*x^8 +
    $.1^7*x^7 + $.1^123*x^6 + $.1^8*x^5 + $.1^77*x^4 + $.1^43*x^3 + $.1^110*x^2
    + $.1^124*x + $.1^51)*b + ($.1^78*x^9 + $.1^105*x^8 + $.1^153*x^7 + 6*x^6 +
    $.1^142*x^5 + $.1^152*x^4 + $.1^54*x^3 + $.1^9*x^2 + $.1^43*x + $.1^37))*d^2
    + (($.1^63*x^10 + $.1^125*x^9 + $.1^156*x^8 + $.1^44*x^7 + $.1^27*x^6 +
    $.1^127*x^5 + $.1^160*x^4 + $.1^46*x^3 + 9*x^2 + 8*x + $.1^37)*b +
    ($.1^99*x^10 + $.1^119*x^9 + $.1^103*x^8 + $.1^25*x^7 + $.1*x^6 +
    $.1^114*x^5 + $.1^133*x^4 + $.1^34*x^3 + $.1^4*x^2 + $.1^40*x + $.1^71))*d +
    ($.1^86*x^10 + $.1^7*x^9 + $.1^142*x^8 + 4*x^7 + $.1^161*x^6 + 2*x^5 +
    $.1^17*x^4 + $.1^50*x^3 + $.1^100*x^2 + $.1^144*x + $.1^12)*b + $.1^31*x^10
    + $.1^40*x^9 + 8*x^8 + 9*x^7 + $.1^39*x^6 + $.1^120*x^5 + $.1^114*x^4 +
    $.1^116*x^3 + $.1^43*x^2 + $.1^103*x + $.1^93) - 15*(1/x, (($.1^114*x^2 +
    $.1^96*x + 12)/x^3*b + ($.1^153*x^2 + 4*x + 12)/x^3)*d^2 + (($.1^17*x^2 +
    $.1^124*x + 12)/x^3*b + ($.1^159*x^2 + $.1^124*x + 12)/x^3)*d + ($.1^159*x^2
    + 6*x + 12)/x^3*b + ($.1^21*x + 12)/x^2)
> g := GCD(D, E);
> l := LCM(D, E);
> g + l eq d;
true
> g le D;
true
> l ge E;
true
```

42.14.6.4 Other Element Operations

`FunctionField(D)`

Given a divisor D, return the function field.

`Degree(D)`

The degree of the divisor D over k, the constant field of definition.

`Support(D)`

A sequence containing the places occurring in the divisor D.

`Numerator(D)`

`ZeroDivisor(D)`

The numerator of the divisor D.

`Denominator(D)`
`PoleDivisor(D)`

The denominator of the divisor D.

`Ideals(D)`

Create two ideals of the 'finite' and 'infinite' maximal order respectively corresponding to the divisor D.

`Norm(D)`

The divisor of the norms of the ideals of the divisor D.

`FiniteSplit(D)`
`FiniteDivisor(D)`
`InfiniteDivisor(D)`

Split the divisor D into its finite and infinite part, returning either 2 divisors which are the sum of the finite places in D and the sum of the infinite places in D or the appropriate one of these.

`Dimension(D)`

The dimension of the Riemann-Roch space $\mathcal{L}(D)$ of the divisor D over k, the constant field of definition.

`IndexOfSpeciality(D)`

The index of speciality of the divisor D, which equals the dimension of $\mathcal{L}(W - D)$ where W is a canonical divisor.

`ShortBasis(D :`*parameters*`)`

`Reduction`	BoolElt	*Default :* `true`
`Simplification`	MonStgElt	*Default :* "*Full*"

Compute a basis for the Riemann-Roch space of D in short form:

Let $F = k(x, y)$ be an algebraic function field defined by $f(x, y) = 0$ over k. Given a divisor D of F/k this function returns a basis of the k-vector space

$$\mathcal{L}(D) = \{a \in F^\times \mid (a) \geq -D\} \cup \{0\}$$

in the short form

$$B = [\, b_1 \ldots, b_n \,],\ [\, d_1, \ldots, d_n \,]$$

with $b_i \in F^\times$ and $d_i \in \mathbf{Z}$ for all $1 \leq i \leq n$, where n denotes the degree in y of the defining equation f of F, such that

$$\mathcal{L}(D) = \left\{ \sum_{i=1}^{n} \lambda_i b_i \mid \lambda_i \in k[x] \text{ with } \deg \lambda_i \leq d_i \text{ for } 1 \leq i \leq n \right\}.$$

The optional argument `Reduction` controls whether to use divisor reduction internally or not; it defaults to `true`. For small divisors this is sometimes faster.

The optional argument `Simplification` controls whether the resulting basis is simplified or not; it defaults to `"Full"`. Simplification sometimes is not insignificantly expensive and can be avoided by setting the parameter to `"None"`.

The algorithm is described in [Heß99].

`Basis(D :parameters)`

`Reduction`	BoolElt	*Default :* `true`
`Simplification`	MonStgElt	*Default :* "*Full*"

A sequence containing a basis of the Riemann-Roch space $\mathcal{L}(D)$, for the divisor D.

The optional argument `Reduction` controls whether to use divisor reduction internally or not; it defaults to `true`. For small divisors this is sometimes faster.

The optional argument `Simplification` controls whether the resulting basis is simplified or not; it defaults to `"Full"`. Simplification sometimes is not insignificantly expensive and can be avoided by setting the parameter to `"None"`.

`RiemannRochSpace(D)`

Given a function field F/k and a divisor D belonging to F/k, return a vector space V and a k-linear mapping $h : V \longrightarrow F$ such that V is isomorphic to the Riemann-Roch space $\mathcal{L}(D) \subset F$ under h.

`Valuation(D, P)`

The exponent of the place P in the divisor D.

`Reduction(D)`

`Reduction(D, A)`

Let D be a divisor. Denote the result of both functions by $\tilde{D}$, r, A and a (for the second function the input A always equals the output A). The divisor A has (must have) positive degree and the following holds:

(i) $D = \tilde{D} + rA - (a)$,

(ii) $\tilde{D} \geq 0$ and $\deg(\tilde{D}) < g + \deg(A)$ (over the exact constant field),

(iii) $\tilde{D}$ has minimal degree among all such divisors satisfying (i), (ii).

`GapNumbers(D, P)`

The sequence of gap numbers of the divisor D at P where P must be a place of degree one:

Let F/k be an algebraic function field, D a divisor and P a place of degree one. An integer $m \geq 1$ is a gap number of D at P if $\dim\big(D+(m-1)P\big) = \dim(D+mP)$ holds. The gap numbers m of D satisfy $1 \leq m \leq 2g-1-\deg(D)$ and their cardinality equals the index of speciality $i(D)$. `GapNumbers(D, P)` returns such a particular sequence. The sequences of gap numbers of D at various P are independent of constant field extensions for perfect k and are the same for all but a finite number of places P of degree one (consider e.g. k algebraically closed). If P is omitted in the function call, this uniform sequence is returned by `GapNumbers(D)`. The places P where D has different sequences of gap numbers are called Weierstraß places of D and are returned by `WeierstrassPlaces(D)`. In the above mentioned functions it is equivalent to replace D by either F or the zero divisor.

`GapNumbers(D)`

`SeparatingElement` FLDFUNGELT *Default :*

The sequence of global gap numbers of the divisor D. A separating element used internally for the computation can be specified, it defaults to `SeparatingElement(F)`. See the description of GapNumbers on page 1167.

Example H42E36

Consider the function field F defined by the curve of genus 7 defined by

$$y^4 + (2*x^5 + x^4 + 2*x^3 + x^2)*y^2 + x^8 + 2*x^6 + x^5 + x^4 + x^3 + x^2$$

We construct the function field $F/\mathbf{F}_9$ and compute the Riemann-Roch space corresponding to a certain divisor.

```
> k<w> := GF(9);
> R<x> := FunctionField(k);
> P<y> := PolynomialRing(R);
> f := y^4 + (2*x^5 + x^4 + 2*x^3 + x^2)*y^2 + x^8
>      + 2*x^6 + x^5 +x^4 + x^3 + x^2;
> F<a> := FunctionField(f);
> Genus(F);
7
> P1 := RandomPlace(F, 1);
> P2 := RandomPlace(F, 1);
> D := P1 - P2;
> D;
(1/x, w^7/x^7*a^3 + w^5/x^5*a^2 + w^3/x^2*a + w) - (x, 2/(x^4 + x^2 + 2*x)*a^3 +
    w^3/x*a^2 + (w^5*x^3 + w^3*x + w^7)/(x^3 + x + 2)*a + w^5)
> IsPrincipal(336*D);
true
> infty := Poles(F!x)[1];
```

```
> V, h := RiemannRochSpace(11*infty);
> V;
KModule V of dimension 5 over GF(3^2)
> h;
Mapping from: ModFld: V to FldFun: F
> B := h(Basis(V));
> B;
[
    x/(x^3 + x + 2)*a^3 + (2*x^4 + 2*x^3 + x)/(x^3 + x + 2)*a,
    1/(x^3 + x + 2)*a^3 + (2*x^3 + 2*x^2 + 1)/(x^3 + x + 2)*a,
    a^2 + 2*x^3 + 2*x^2,
    1/x*a^2 + 2*x^2 + 2*x,
    1
]
> (B[2] + 2*B[3])@@h;
(    0     1     2     0     0)
```

Example H42E37

As a trivial but illustrative example we consider the algebraic function field generated by $\sin(x)$ and $\cos(x)$ over Q and construct a single function $a(x)$ such that $\sin(x)$ and $\cos(x)$ can be expressed in terms of $a(x)$:

```
> Qc<c> := PolynomialRing(RationalField());
> Qcs<s> := PolynomialRing(Qc);
> F<s> := FunctionField(s^2 + c^2 - 1);
> c := F!c;
> Genus(F);
0
> Zeros(s);
[ (c - 1, s), (c + 1, s) ]
> Zeros(c-1);
[ (c - 1, s) ]
> P := Zeros(c-1)[1];
> Degree(P);
1
> Dimension(1*P);
2
> Basis(1*P);
[ 1/(c - 1)*s, 1 ]
> a := Basis(1*P)[1];
> Degree(a);
1
> MinimalPolynomial(a);
$.1^2 + (c + 1)/(c - 1)
> (a^2 - 1)/(a^2 + 1);
c
> a * ((a^2 - 1)/(a^2 + 1) - 1);
```

```
s
```

Example H42E38

Over $Q(i)$ the familiar identities

$$\cos(x) = (\exp(ix) + \exp(-ix))/2$$

$$\sin(x) = (\exp(ix) - \exp(-ix))/(2i).$$

hold. In MAGMA one can proceed as follows:

```
> Qx<x> := PolynomialRing(RationalField());
> k<i> := NumberField(x^2 + 1);
> kc<c> := PolynomialRing(k);
> kcs<s> := PolynomialRing(kc);
> F<s> := FunctionField(s^2 + c^2 - 1);
> c := F!c;
> Genus(F);
0
> e := c + i*s;
> ebar := c - i*s;
> Degree(e);
1
> c eq (e + ebar) / 2;
true
> s eq (e - ebar) / (2*i);
true
```

`RamificationDivisor(D)`

`SeparatingElement` FLDFUNGELT *Default :*

The ramification divisor of the divisor D (using `SeparatingElement` for the computation which defaults to `SeparatingElement(F)` for F/k the function field of D):

Let F/k be an algebraic function field, x a separating variable and D a divisor. The ramification divisor of D is defined to be

$$i(D)\,(W - D) + \big(W_x(D)\big) + \nu\,(dx),$$

where W is a canonical divisor of F/k, $W_x(D)$ is the determinant of the Wronksian matrix of D with respect to x and ν is the sum of the Wronskian orders of D with respect to x. It is effective and consists of the Weierstraß places of D. The constant field k is required to be exact.

`WeierstrassPlaces(D)`

`SeparatingElement` FldFunGElt *Default :*

The Weierstrass places of the divisor D (using `SeparatingElement` for the computation which defaults to `SeparatingElement(F)` for F/k the function field of D):

Let F/k be an algebraic function field, D a divisor and P a place of degree one. An integer $m \geq 1$ is a gap number of D at P if $\dim\big(D+(m-1)P\big) = \dim(D+mP)$ holds. The gap numbers m of D at P satisfy $1 \leq m \leq 2g-1-\deg(D)$ and their cardinality equals the index of speciality $i(D)$. The sequences of gap numbers of D are independent of constant field extensions for perfect k and are the same for all but a finite number of places P of degree one (consider e.g. k algebraically closed). The places P of degree one at which D has different sequences of gap numbers are called Weierstraß places of D.

This function returns a list of all places of F/k (having not necessarily degree one) which are lying below Weierstraß places of D viewed in $F\bar{k}/\bar{k}$ (k perfect). The constant field k is required to be exact. Note that if the characteristic of F is positive this function is currently quite slow for large genus because of `Differentiation()`.

`IsWeierstrassPlace(D, P)`

Given a divisor D and a degree 1 place P of a function field, return whether P is a weierstrass place of D.

`WronskianOrders(D)`

`SeparatingElement` FldFunGElt *Default :*

Let D be a divisor of an algebraic function field F/k with separating element x and let $v_1, \ldots v_l$ be a basis of $\mathcal{L}(D)$. For the differentiation D_x with respect to x consider the successively smallest $\nu_1 \leq \ldots \leq \nu_l \in \mathbf{Z}^{\geq 0}$ such that the rows $D_x^{(\nu_i)}(v_1), \ldots, D_x^{(\nu_i)}(v_l)$, $1 \leq i \leq l$ are F-linearly independent. The numbers $\nu_1, \ldots, \nu_l$ are the Wronskian orders of D with respect to x and are returned. If D has dimension zero, the empty list is returned. The constant field k is required to be exact.

The separating element can be given by setting the `SeparatingElement` parameter appropriately.

`ComplementaryDivisor(D)`

Return the complementary divisor $D^{\#}$ of the divisor D. The function field F/k of D must be a finite extension of a rational function field $k(x)$. The divisor $D^{\#}$ equals $\mathrm{Diff}(F/k(x)) - D$ for F the function field of D and $\mathrm{Diff}(F/k(x))$ the different divisor of $F/k(x)$.

`DifferentialBasis(D)`

A basis of the space of differentials of the divisor D. See `DifferentialBasis` on page 1177 for details.

`DifferentialSpace(D)`

A vector space and the isomorphism from this space to the differential space of the divisor D.

`Parametrization(F, D)`

An element x in F which is a non constant element of the basis of the divisor D having degree one and a sequence of elements L in the rational function field are returned such that x generates the function field F over the constant field and L contains the images of the generators of F over its constant field in the rational function field.

42.14.7 Functions related to Divisor Class Groups of Global Function Fields

Let F/k be a global function field. The group of divisor classes is isomorphic to the product of a copy of $\mathbf{Z}$ and the group of divisors classes of degree zero which is a finite abelian group. MAGMA features an algorithm to compute the divisor class group by computing an abelian group G in the form $\mathbf{Z}/c_1\mathbf{Z} \times \ldots \times \mathbf{Z}/c_{2g}\mathbf{Z} \times \mathbf{Z}$ with integers $c_1|\ldots|c_{2g}$ and a surjective homomorphism $f : Div(F) \to G$ from the divisor group to G whose kernel consists precisely of the principal divisors.

The algorithm employed is a randomized index calculus style method of expected subexponential running time for "small" constant field size and "large" genus. A description of this and other algorithms of this section can be found in [Heß99].

Elements in product representation may result from applying the maps returned by some of the computations below. It can be expensive to put these elements into sets and to test them for equality.

`ClassGroupGenerationBound(q, g)`

A bound B such that the places of degree (over the exact constant field) less than or equal to B, taken together with the places of a divisor of degree one, generate the whole divisor class group of any global function field of genus g over the exact constant field of q elements.

`ClassGroupGenerationBound(F)`

A bound B such that all places of degree (over the exact constant field) less than or equal to B, taken together with the places of a divisor of degree one, generate the whole divisor class group of the function field F. Particular properties of the function field are taken into account.

`ClassNumberApproximation(F, e)`

An approximation of the class number of the global function field F/k with multiplicative error less than $1 + e$ for $e > 0$. The formula

$$\left| \log\big(h \,/\, q^g\big) - \sum_{r=1}^{b} q^{-r}/r\big(N_r - (q^r + 1)\big) \right| \;\le\; 2gq^{-b/2}/\big((q^{1/2} - 1)(b + 1)\big)$$

is used where N_r denotes the number of places of degree one in the constant field extension of degree r of F/k.

`ClassNumberApproximationBound(q, g, e)`

Returns an integer B such that all places of degree less than or equal to B of a global function field of genus g over the exact constant field of q elements have to be considered in order to approximate the class number with multiplicative error less than $1 + e$ for $e > 0$.

`ClassGroup(F :`*parameters*`)`

`DegreeBound`	RngIntElt	*Default :*
`SizeBound`	RngIntElt	*Default :*
`ReductionDivisor`	DivFunElt	*Default :*
`Proof`	BoolElt	*Default :*

The divisor class group of the function field F/k as an abelian group, a map of representatives from the class group to the divisor group and the homomorphism from the divisor group onto the divisor class group.

The optional parameter `DegreeBound` allows to control the size of the factor basis which consists of all places of degree less equal `DegreeBound` (plus a small additional amount; the degree is taken over the exact constant field). If not provided the algorithm tries to choose an appropriate value.

The optional parameter `SizeBound` bounds the size of the factor basis to not exceed `SizeBound` places. Every time the factor basis has to be enlarged during the computation it will be by no more than `SizeBound` additional places. If not provided there is no bound on the size of the factor basis. Every enlargement of the factor basis will append all places of the next degree.

The optional parameter `ReductionDivisor` contains the reduction divisor used in the relation search stage. Reasonable choices are divisors of small positive degree. If not provided the algorithm tries to choose an appropriate reduction divisor.

The optional parameter `Proof` indicates whether the computed result should be proven in a proof step. If a small degree bound for the factor basis is used and the divisor class group happens to be a product of a large number of cyclic groups the proof step can be very time consuming and `Proof := false` might be helpful. Once a value is given for `Proof` it remains the default value until set differently. The initial value of `Proof` for every function field is `true`.

`ClassGroupAbelianInvariants(F :`*parameters*`)`

`DegreeBound`	RngIntElt	*Default :*
`SizeBound`	RngIntElt	*Default :*
`ReductionDivisor`	DivFunElt	*Default :*
`Proof`	BoolElt	*Default :*

Computes a sequence of integers containing the Abelian invariants of the divisor class group of the function field F/k.

The optional parameters are the same as for `ClassGroup`.

`ClassNumber(F)`

The order of the group of divisor classes of degree zero of the function field F/k.

Example H42E39

Some class group calculations :

```
> Y<t> := PolynomialRing(Integers());
> R<x> := FunctionField(GF(9));
> P<y> := PolynomialRing(R);
> f := y^3 + y + x^5 + x + 1;
> F<alpha> := FunctionField(f);
> ClassNumberApproximation(F, 1.3);
24890.25505701632912193514
> ClassGroup(F);
Abelian Group isomorphic to Z/13 + Z/13 + Z/13 + Z/13 + Z
Defined on 5 generators
Relations:
    13*$.1 = 0
    13*$.2 = 0
    13*$.3 = 0
    13*$.4 = 0
Mapping from: Abelian Group isomorphic to Z/13 + Z/13 + Z/13 + Z/13 + Z
Defined on 5 generators
Relations:
    13*$.1 = 0
    13*$.2 = 0
    13*$.3 = 0
    13*$.4 = 0 to Divisor group of F
Mapping from: Divisor group of F to Abelian Group isomorphic to Z/13 + Z/13 +
Z/13 + Z/13 + Z
Defined on 5 generators
Relations:
    13*$.1 = 0
    13*$.2 = 0
    13*$.3 = 0
    13*$.4 = 0 given by a rule
> ClassNumber(F);
28561
> Evaluate(LPolynomial(F), 1);
28561
```

`GlobalUnitGroup(F)`

The group of global units of the function field F/k, i. e. the multiplicative group of the exact constant field, as an Abelian group, together with the map into F.

`IsGlobalUnit(a)`

Whether the function field element a is a global unit, i.e. a constant (equivalent to `IsConstant`).

`IsGlobalUnitWithPreimage(a)`

Returns `true` and the preimage of the function field element a in the global unit group, `false` otherwise.

`PrincipalDivisorMap(F)`

The map from the multiplicative group of the function field to the group of divisors.

`ClassGroupExactSequence(F)`

Returns the three maps in the center of the exact sequence

$$0 \to k^\times \to F^\times \to Div \to Cl \to 0$$

where $k^\times$ is the global unit group of the function field, $F^\times$ is the multiplicative group of the function field, Div is the divisor group and Cl is the divisor class group.

`SUnitGroup(S)`

The group of S-units as an Abelian group and the map into the function field, where S is a sequence of places of a function field.

`IsSUnit(a, S)`

Returns `true` if the function field element a is an S-unit for the sequence of places S, `false` otherwise.

`IsSUnitWithPreimage(a, S)`

Returns `true` and the preimage of the function field element a in the S-unit group if a is an S-unit for the sequence of places S, `false` otherwise.

`SRegulator(S)`

The S-Regulator for the sequence of places S.

`SPrincipalDivisorMap(S)`

The map from the multiplicative group of the function field to the group of divisors (mod places in the sequence S).

`IsSPrincipal(D, S)`

Returns `true` and a generator if the divisor D is principal modulo places in the sequence S, `false` otherwise

`SClassGroup(S)`

The S-class group for the sequence of places S as an Abelian group, a map of representatives from the S-class group to the group of divisors (mod places in S) and the homomorphism from the group of divisors (mod places in S) onto the S-class group.

`SClassGroupExactSequence(S)`

Returns the three maps in the center of the exact sequence

$$0 \to U(S) \to F^\times \to Div(S) \to Cl(S) \to 0$$

where $U(S)$ is the S-unit group, $F^\times$ is the multiplicative group of the function field, $Div(S)$ is the group of divisors (mod places in the sequence S) and $Cl(S)$ is the S-class group.

`SClassGroupAbelianInvariants(S)`

Computes a sequence of integers containing the Abelian invariants of the S-class group for the sequence of places S.

`SClassNumber(S)`

The order of the torsion part of the S-class group for the sequence of places S.

`ClassGroupPRank(F)`

Compute the p-rank of the class group of F/k where p is the characteristic of F/k. More precisely: Let F/k be a function field of characteristic p. Consider the subgroup $Cl^0(F/k)[p]$ of p-torsion elements of the group of divisor classes of degree zero. This function returns its dimension as an $\mathbf{F}_p$-vector space. Possible values range from 0 to g, where g is the genus of F/k. The field k is currently required to be a finite field.

`HasseWittInvariant(F)`

Return the Hasse–Witt invariant of F/k. More precisely: Let F/k be a function field of characteristic p. Let $F\bar{k}/\bar{k}$ be the constant field extension by the algebraic closure $\bar{k}$ of k within an algebraic closure $\bar{F}$ of F/k. Consider the subgroup $Cl^0(F\bar{k}/\bar{k})[p]$ of p-torsion elements of the group of divisor classes of degree zero. This function returns its dimension as an $\mathbf{F}_p$-vector space. Possible values range from 0 to g, where g is the genus of F/k. k is required to be perfect.

`TateLichtenbaumPairing(D1, D2, m)`

The Tate–Lichtenbaum pairing $Cl_0[m] \times Cl_0/mCl_0 \to k$ for coprime divisors $D1$ and $D2$.

Example H42E40

```
> k<w> := GF(9);
> R<x> := FunctionField(k);
> P<y> := PolynomialRing(R);
> f := y^4 + (2*x^5 + x^4 + 2*x^3 + x^2)*y^2 +
>             x^8 + 2*x^6 + x^5 + x^4 + x^3 + x^2;
> F<a> := FunctionField(f);
> D1 := Zeros(a)[1] - Poles(F!x)[1];
> D2 := Zeros(a)[4] - Poles(F!x)[2];
> G,mapfromG,maptoG:=ClassGroup(F : Proof:=false);
> Order(maptoG(D1));
48
> Order(maptoG(D2));
336
> TateLichtenbaumPairing(D1,D2,48);
w^7
> TateLichtenbaumPairing(D2,D1,336);
w^3
```

42.15 Differentials

Spaces of differentials of function fields can be created and the differentials belonging to them manipulated. Divisors can be created from differentials and modules generated by a collection of differentials can be formed.

42.15.1 Creation of Structures

`DifferentialSpace(F)`

The space of differentials of the algebraic function field F/k.

42.15.2 Creation of Elements

The simplest ways of creating a differential are given below.

`Differential(a)`

Create the differential $d(a)$ of the function field or order element a.

`Identity(D)`

The identity differential of the space of differentials D.

`IsCanonical(D)`

Returns `true` iff the divisor D is canonical and a differential having D as its divisor.

42.15.3 Related Structures

`FunctionField(D)`

The function field of the differentials in the space D.

`FunctionField(d)`

The function field of the differential d.

42.15.4 Subspaces

`SpaceOfDifferentialsFirstKind(F)`

`SpaceOfHolomorphicDifferentials(F)`

A vector space and the isomorphism from this space to the space of differentials of the first kind (holomorphic differentials) of the function field F/k.

`BasisOfDifferentialsFirstKind(F)`

`BasisOfHolomorphicDifferentials(F)`

A basis of the space of differentials of the first kind (holomorphic differentials) of the function field F/k.

`DifferentialBasis(D)`

Computes a basis for the space of differentials

$$\Omega(D) := \{ \omega \in \Omega(F/k) \mid (\omega) \geq D \}$$

for a divisor D of an algebraic function field F/k.

`DifferentialSpace(D)`

A vector space and the isomorphism from this space to the differential space of the divisor D.

Example H42E41

This example illustrates the differential space of a divisor and some of the operations that can be done with it.

```
> Q := Rationals();
> Qx<x> := PolynomialRing(Q);
> Qxy<y> := PolynomialRing(Qx);
> f1 := y^2 - (x-1)*(x-2)*(x-3)*(x-5)*(x-6);
> F := FunctionField(f1);
> d := Divisor(F.1) + Divisor(F!BaseRing(F).1);
> V1 := DifferentialSpace(d);
> d := 2*Divisor(F.1) - Divisor(F!BaseRing(F).1);
> V2 := DifferentialSpace(d);
> V1;
```

```
KModule V1 of dimension 2 over Rational Field
> V2;
KModule V2 of dimension 2 over Rational Field
> V1 meet V2;
KModule of dimension 0 over Rational Field
> D := DifferentialSpace(F);
> v := V1 ! [2/9, 4/9]; v;
V1: (2/9 4/9)
> D!v;
(2/9*x^2 + 4/9*x) d(x)
> V1!$1;
V1: (2/9 4/9)
> BasisOfDifferentialsFirstKind(F);
[ (x/(x^5 - 17*x^4 + 107*x^3 - 307*x^2 + 396*x - 180)*F.1) d(x), (1/(x^5 -
    17*x^4 + 107*x^3 - 307*x^2 + 396*x - 180)*F.1) d(x) ]
```

42.15.5 Structure Predicates

`D1 eq D2`

Return `true` if the spaces of differentials $D1$ and $D2$ are the same.

42.15.6 Operations on Elements

A number of general operations for elements are also provided for differentials as well as a number of specific functions for differentials.

42.15.6.1 Arithmetic Operators

`r * x`
`x * r`
`x + y`
`-x`
`x - y`
`x / y`
`x / r`

The operations on differentials are inherited from the vector space structure of the space of differentials. Additionally, this space is one–dimensional as a vector space over the function field itself. The quotient x/y of two differentials x and y then gives the unique $r \in F$ such that $x = ry$.

42.15.6.2 Equality and Membership

`x eq y`

Returns `true` if x and y are the same differential.

`x in D`

Returns `true` if x is in the space of differentials D.

42.15.6.3 Predicates on Elements

`IsExact(d)`

Return whether d is known to be an exact differential. If `true` additionally return a generator. If d is not already known to be exact then no attempts to determine whether d is exact or not are currently undertaken.

`IsZero(d)`

Return `true` if d is the zero differential.

42.15.6.4 Functions on Elements

`Valuation(d, P)`

The valuation of the differential d at the place P.

`Divisor(d)`

The divisor (d) of the differential d.

`Residue(d, P)`

The residue of the differential d at the place P, where P must (currently) have degree one.

Example H42E42

```
> PF<x> := PolynomialRing(GF(31, 3));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b : Check := false>;
> Differential(d);
(26/x*d) d(x)
> I := Random(FF2, 2)*MaximalOrderInfinite(FF2);
> P := Place(Factorization(I)[1][1]);
> Valuation(Differential(d), P);
-2
> IsExact(Differential(Random(FF2, 2)));
true
```

`Module(L, R)`

`IsBasis`	BoolElt	*Default :* `false`
`PreImages`	BoolElt	*Default :* `false`

The R-module generated by the differentials in the sequence L as an abstract module, together with the map into the space of differentials. The resulting modules can be used for intersection and inner sum computations.

If the optional parameter `IsBasis` is set `true` the function assumes that the given elements form a basis of the module to be computed.

If the optional parameter `PreImages` is set `true` then the preimages of the given elements under the map are returned as the third return value.

Both optional parameters are mainly used to save computation time.

`Relations(L, R)`

`Relations(L, R, m)`

The module of R-linear relations between the differentials of the sequence L. The argument m is used for the following: Let the elements of L be $a_1, \ldots, a_n$, V be the relation module $\subseteq R^n$ and define $M := \{ \sum_{i=1}^{m} v_i a_i \mid v = (v_i)_i \in V \}$. The function tries to compute a generating system of V such that the corresponding generating system of M consists of "small" elements.

Example H42E43

This example shows some of the conversions and operations possible with the results of `Module`.

```
> Q := Rationals();
> Qx<x> := PolynomialRing(Q);
> Qxy<y> := PolynomialRing(Qx);
> f1 := y^2 - (x-1)*(x-2)*(x-3)*(x-5)*(x-6);
> F := FunctionField(f1);
> D := DifferentialSpace(F);
> M7 := Module([Differential(3*F.1)], FieldOfFractions(Qx));
> M8 := Module([Differential(F.1), Differential(F!BaseRing(F).1)],
> FieldOfFractions(Qx));
> M12 := M7 meet M8;
> M16 := M7 + M8;
> assert M12 subset M7;
> assert M12 subset M8;
> assert M12 subset M16;
> r := M7![&+[Random([-100, 100])/Random([1, 100])*x^i : i in [1 .. 5]]/
> &+[Random([-100, 100])/Random([1,
> 100])*x^i : i in [1 .. 5]] : j in [1 .. Dimension(M7)]];
> assert M7!D!r eq r;
> r;
M7: ((1/100*x^4 + x^3 - x^2 + 1/100*x - 1/100)/(x^4 - 1/100*x^3 - 1/100*x^2 +
    1/100*x - 1/100))
> D!r;
```

```
((3/40*x^8 + 162/25*x^7 - 20937/200*x^6 + 114873/200*x^5 - 279531/200*x^4 +
    304167/200*x^3 - 24321/40*x^2 + 303/20*x - 297/50)/(x^9 - 1701/100*x^8 +
    2679/25*x^7 - 30789/100*x^6 + 19891/50*x^5 - 3593/20*x^4 - 63/10*x^3 +
    883/100*x^2 - 144/25*x + 9/5)*F.1) d(x)
```

`Cartier(b)`

`Cartier(b, r)`

The result of applying the Cartier operator r times to b. More precisely, let F/k be a function field over the perfect field k, $x \in F$ be a separating variable and $b = g\,dx \in \Omega(F/k)$ with $g \in F$ be a differential. The Cartier operator is defined by

$$C(b) = \left(-d^{p-1}g/dx^{p-1}\right)^{1/p} dx.$$

This function computes the r-th iterated application of C to b.

42.15.6.5 Other

`CartierRepresentation(F)`

`CartierRepresentation(F, r)`

Compute a row representation matrix of the Cartier operator on a basis of the space of holomorphic differentials of F/k (applied r times). More precisely, let F/k be a function field over the perfect field k, $\omega_1, \ldots, \omega_g \in \Omega(F/k)$ be a basis for the holomorphic differentials and $r \in \mathbf{Z}^{\geq 1}$. Let $M = (\lambda_{i,j})_{i,j} \in k^{g \times g}$ be the matrix such that

$$C^r(\omega_i) = \sum_{m=1}^{g} \lambda_{i,m}\omega_m$$

for all $1 \leq i \leq g$. This function returns M and $(\omega_1, \ldots, \omega_g)$.

Example H42E44

An example of a trivial cartier action.

```
> PF<x> := PolynomialRing(GF(31, 3));
> P<y> := PolynomialRing(PF);
> FF1<b> := ext<FieldOfFractions(PF) | y^2 - x>;
> P<y> := PolynomialRing(FF1);
> FF2<d> := ext<FF1 | y^3 - b : Check := false>;
> Cartier(Differential(d), 4);
(0) d(x)
> CartierRepresentation(FF2, 3);
Matrix with 0 rows and 0 columns
[]
```

42.16 Weil Descent

Weil Descent is a technique that can be applied to cryptosystem attacks amongst other things. The general idea is as follows. If C is a curve defined over field L and K is a subfield of L, the group of points of the Jacobian of C, $Jac(C)$, over L is isomorphic to the group of points of a higher-dimensional abelian variety, the *Weil restriction* of $Jac(C)$, over K. If a curve D over K can be found along with an algebraic homomorphism defined over K from $Jac(D)$ to this Weil restriction, it is usually possible to transfer problems about $Jac(C)(L)$ (like discrete log problems) to $Jac(D)(K)$. D will have larger genus than C, but we have descended to a smaller field. The advantage is that techniques like Index calculus, where the time complexity depends more strongly on the field size than the genus, can now be applied.

An important special case is where C is an ordinary elliptic curve over a finite field of characteristic 2. In this case, Artin-Schreier theory gives a very concrete algorithm for computing a suitable D. This is the GHS attack (see [Gau00] or the chapter by F. Heß in [Bla05]). There is a corresponding function `WeilDescent` in the Elliptic Curve chapter that works with curves rather than function fields.

The functions below implement the GHS Weil Descent in a more general characteristic p setting.

`WeilDescent(E,k)`

The main function. E must be an "Artin-Schreier" function field over finite field K, an extension of k, of the following form. If p is the characteristic of K, then the base field of E is a rational function field $K(x)$ and $E.1$, the generator of E over $K(x)$ has minimal polynomial of the form $y^p - y = c/x + a + b * x$ for $a, b, c \in K$. These parameters must satisfy $b, c \neq 0$ and at least one of the following conditions
i) $Trace_{K/k}(b) \neq 0$
ii) $Trace_{K/k}(c) \neq 0$
iii) $Trace_{K/\mathbf{F}_p}(a) = 0$

Then, if E_1 is the Galois closure of the separable field extension E/k and $[K : k] = n$, there is an extension of the Frobenius generator of $G(K/k)$ to an automorphism of E_1 which fixes x and is also of order n. If F is the fixed field of this extension, then it has exact constant field k and so it is the function field of a (geometrically irreducible) curve over k. This is the WeilDescent curve. It's algebraic function field F is the first return value.

Note that when $p = 2$, E is the function field of an ordinary elliptic curve in characteristic 2 (multiply the defining equation by x^2 and take xy as the new y variable) and conversely, any such elliptic curve can be put into this form in multiple ways.

The second return value is a map from the divisors of E to the divisors of F which represents the homomorphism $Jac(Curve(E))(K) \to Jac(Curve(F))(k)$. This map, however does not attempt any divisor reduction. For the characteristic 2 `WeilDescent` function on elliptic curves mentioned in the introduction, if F is hyperelliptic, the divisor map returned is the actual homomorphism from the elliptic

curve (with function field E) to the Jacobian of the hyperelliptic curve with function field F. This may be more convenient for the user.

There are functions below that may be called to return the genus and other attributes of F before actually going through with its construction.

One important special case, worth noting here, is that when $p = 2$ and b or c is in k, then the degree of F is also 2 and so the descent curve is a hyperelliptic curve.

`ArtinSchreierExtension(c,a,b)`

A convenience function to generate the function field which is an extension of $K(x)$ by the (irreducible) polynomial $y^p - y - c/x - a - bx$ where K is a finite field which is the parent of a, b, c and p is the characteristic of K.

`WeilDescentDegree(E,k)`

With E and k as in the main `WeilDescent` function, returns the degree of the descended function field F over its rational base field $k(x)$. The computation only involves the Frobenius action on a, b, c and the descent isn't actually performed.

`WeilDescentGenus(E,k)`

With E and k as in the main `WeilDescent` function, returns the genus of the descended function field F over its rational base field $k(x)$. The computation only involves the Frobenius action on a, b, c and the descent isn't actually performed.

`MultiplyFrobenius(b,f,F)`

This is a simple convenience function, useful for generating elements of K on which the Frobenius has a given minimal polynomial when considered as an $\mathbf{F}_p$-linear map (see the example below).

The argument b should be an element of a ring R, f a polynomial over ring R_0, which is naturally a subring of R and F a map from R to itself.

If $f = a_0 + a_1 x + \ldots$ then the function returns $f(F)(b)$ which is $a_0 + a_1 F(b) + a_2 F(F(b)) + \ldots$.

Example H42E45

We demonstrate with an example of descent over a degree 31 extension in characteristic 2, which results in a genus 31 hyperelliptic function field, and a small characteristic 3 example.

```
> SetSeed(1);
> k<u> := GF(2^5);
> K<w> := GF(2^155);
> Embed(k, K);
> k2<t> := PolynomialRing(GF(2));
> h := (t^31-1) div (t^5 + t^2 + 1);
> frob := map< K -> K | x :-> x^#k >;
> b := MultiplyFrobenius(K.1,h,frob);
> E := ArtinSchreierExtension(K!1, K!0, b);
> WeilDescentGenus(E, k);
```

```
31
> WeilDescentDegree(E,k);
2
> C,div_map := WeilDescent(E, k);
> C;
Algebraic function field defined over Univariate rational function field over
GF(2^5) by
y^2 + u*y + u^18/($.1^32 + u^29*$.1^16 + u^14*$.1^8 + u^2*$.1^4 + $.1^2 +
    u^9*$.1 + u^5)
> // get the image of the place representing a K-rational point
> pl := Zeroes(E!(BaseField(E).1)-w)[1];
> D := div_map(pl);
> Degree(D); //31*32
992
> k := GF(3);
> K<w> := GF(3,4);
> a := w+1;
> c := w;
> b := c^3+c;
> E := ArtinSchreierExtension(c, a, b);
> WeilDescentDegree(E,k);
27
> WeilDescentGenus(E,k);
78
> C := WeilDescent(E, k);
> C;
Algebraic function field defined over Univariate rational function field over
GF(3) by
y^27 + 2*y^9 + y^3 + 2*y + (2*$.1^18 + 2*$.1^12 + 2*$.1^9 + 2)/($.1^9 + $.1^3 +
    1)
```

42.17 Function Field Database

An optional database of function fields may be downloaded from the MAGMA website. This section defines the interface to that database. Each database is associated to a given finite field and extension degree. The supported combinations are:

- $\mathbf{F}_2$: degrees 2, 3
- $\mathbf{F}_3$: degree 2
- $\mathbf{F}_4$: degrees 2, 3, 5
- $\mathbf{F}_5$: degrees 2, 3, 4, 8
- $\mathbf{F}_7$: degree 9
- $\mathbf{F}_{11}$: degree 3
- $\mathbf{F}_{13}$: degree 3

For each function field in the database, the following information is stored and may be used to limit the function fields of interest via the sub constructor: The genus; the number of degree one places; the class number; and the class group.

42.17.1 Creation

`FunctionFieldDatabase(q, d)`

Returns a database object for the function fields of degree d over $\mathbf{F}_q$.

`sub< D | : parameters >`

`Genus` RngIntElt *Default :*

`NumberOfDegreeOnePlaces`

RngIntElt *Default :*

Returns a sub-database of D, restricting (or further restricting, if D is already a sub-database of the full database) the contents to those function fields satisfying the specified conditions. Note that it is not possible to "undo" restrictions with this constructor — the results are always at least as limited as D is.

The parameter `Genus` may be used to restrict the search to fields with the specified genus.

The parameter `NumberOfDegreeOnePlaces` may be used to restrict the search to only those fields with the specified number of places of degree one.

42.17.2 Access

`BaseField(D)`

`CoefficientField(D)`

Returns the finite field underlying each function field in the database.

`Degree(D)`

Returns the degree of each function field in the database.

`#D`

`NumberOfFields(D)`

Returns the number of function fields stored in the database.

`FunctionFields(D)`

Returns the sequence of function fields stored in the database.

Example H42E46

The genus of a degree four function field is at most 6. We can see the distribution in a database by counting the size of appropriate sub-databases:

```
> D := FunctionFieldDatabase(5, 4);
> #D;
196380
> [ #sub<D |: Genus := g> : g in [0..6] ];
[ 60, 480, 960, 12960, 35040, 63120, 83760 ]
```

42.18 Bibliography

[**Bj94**] Johannes A. Buchmann and Hendrik W. Lenstra jr. Approximating rings of integers in number fields. *J. Théor. Nombres Bordx.*, 6(2):221–260, 1994.

[**Bla05**] I. Blake, G. Serrousi and N. Smart, editor. *Advances in Elliptic Curve Cryptography*, volume 317 of *LMS LNS*. Cambridge University Press, Cambridge, 2005.

[**Bue04**] D. Buell, editor. *ANTS VI*, volume 3076 of *LNCS*. Springer-Verlag, 2004.

[**CHM98**] John H. Conway, Alexander Hulpke, and John McKay. On transitive permutation groups. *LMS Journal of Computation and Mathematics*, 1:1–8, 1998.

[**FK12**] C. Fieker and J. Klüners. Computational Galois Theory I: Invariants and Computations over **Q**. submitted, http://arxiv.org/abs/1211.3588, 2012.

[**Gau00**] P. Gaudry, F. Heß and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. Cryptology*, 15(1):19–46, 2000.

[**Gei03**] Katharina Geißler. *Berechnung von Galoisgruppen über Zahl- und Funktionenkörpern*. PhD Thesis, TU-Berlin, 2003. available at URL:http://www.math.tu-berlin.de/~kant/publications/diss/geissler.pdf.

[**GK00**] Katharina Geißler and Jürgen Klüners. The determination of Galois Groups. *J. Symbolic Comp.*, 30(6):653–674, 2000.

[**Heß99**] Florian Heß. *Zur Divisorenklassengruppenberechnung in globalen Funktionenkörpern*. Dissertation, Technische Universität Berlin, 1999. URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_FH.ps.gz.

[**Heß04**] Florian Heß. An algorithm for computing isomorphisms of algebraic function fields. In Buell [Bue04], pages 263–271.

[**Klü02**] Jürgen Klüners. Algorithms for Function Fields. *Exp. Math.*, (101):171–181, 2002.

[**SM85**] Leonhard H. Soicher and John McKay. Computing Galois Groups over the rationals. *J. Number Th.*, 20:273–281, 1985.

[**Sta73**] Richard P. Stauduhar. The determination of Galois Groups. *Math. Comp.*, 27:981–996, 1973.

[**Sti93**] Henning Stichtenoth. *Algebraic function fields and codes.* Springer-Verlag, Berlin, 1993.

[**Sut**] Nicole Sutherland. Computing Galois Groups of Polynomials (especially over Function Fields of Prime Characteristic). in preparation.

[**Sut12**] Nicole Sutherland. Efficient Computation of Maximal Orders of Radical (including Kummer) Extensions. *Journal of Symbolic Computation*, 47(5):552–567, 2012.

[**Sut13**] Nicole Sutherland. Efficient Computation of Maximal Orders of Artin–Schreier Extensions. *Journal of Symbolic Computation*, 53(1):26–39, 2013.

[**vHKN11**] M. van Hoeij, J. Klüners, and A. Novocin. Generating Subfields. In Anton Leykin, editor, *Proceedings ISSAC 2011*, 2011.

43 CLASS FIELD THEORY FOR GLOBAL FUNCTION FIELDS

Chapter 43
CLASS FIELD THEORY FOR GLOBAL FUNCTION FIELDS

Global function fields admit a class field theory in the same way as number fields do (Chapter 39). From a computational point of view the main difference is the use of divisors rather than ideals and the availability in general of analytical methods; see Section 43.6.

Class field theory deals with the abelian extensions of a given field. In the number field case, all abelian extensions can be parameterized using more general class groups, in the case of global function fields, the same will be achieved using the divisor class group and extensions of it.

43.1 Ray Class Groups

The ray divisor class group Cl_m modulo a divisor m of a global function field K is defined via the following exact sequence:

$$1 \to k^\times \to O_m^\times \to \mathrm{Cl}_m \to \mathrm{Cl} \to 1$$

where O_m^* is the group of units in the "residue ring" mod m, k is the exact constant field of K and Cl is the divisor class group of K. This follows the methods outlined in [HPP97].

`RayResidueRing(D)`

Let $D = \sum n_i P_i$ be an effective divisor for some places P_i (so $n_i > 0$). The ray residue ring R is the product of the unit groups of the local rings:

$$R = O_m^\times := \prod (O_{P_i}/P_i^{n^i})^\times$$

where O_{P_i} is the valuation ring of P_i.

The map returned as the second return value is from the ray residue ring R into the function field and admits a pointwise inverse for the computation of discrete logarithms.

`RayClassGroup(D)`

Let D be an effective (positive) divisor. The ray class group modulo D is a quotient of the group of divisors that are coprime to D modulo certain principal divisors. It may be computed using the exact sequence:

$$1 \to k^\times \to O_m^\times \to \mathrm{Cl}_m \to \mathrm{Cl} \to 1$$

Note that in contrast to the number field case, the ray class group of a function field is infinite.

The map returned as the second return value is the map from the ray class group into the group of divisors and admits a pointwise inverse for the computation of discrete logarithms.

Since this function uses the class group of the function field in an essential way, it may be necessary for large examples to precompute the class group. Using ClassGroup directly gives access to options that may be necessary for complicated fields.

RayClassGroupDiscLog(y, D)

RayClassGroupDiscLog(y, D)

Return the discrete log of the place or divisor y in the ray class group modulo the divisor D. This is a version of the pointwise inverse of the map returned by RayClassGroup. The main difference is that using the intrinsics the values are cached, ie. if the same place (or divisor) is decomposed twice, the second time will be instantaneous.

A disadvantage is that in situations where a great number of discrete logarithms is computed for pairwise different divisors, a great amount of memory is wasted.

Example H43E1

We will demonstrate the creation of some ray class and ray residue groups. First we have to create a function field:

```
> k<w> := GF(4);
> kt<t> := PolynomialRing(k);
> ktx<x> := PolynomialRing(kt);
> K := FunctionField(x^3-w*t*x^2+x+t);
```

Now, to create some divisors:

```
> lp := Places(K, 2);
> D1 := 4*lp[2]+2*lp[6];
> D2 := &+Support(D1);
```

And now the groups:

```
> G1, mG1 := RayResidueRing(D1); G1;
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/2 + Z/2 + Z/2 +
Z/2 + Z/4 + Z/60 + Z/60
Defined on 10 generators
Relations:
    2*G1.1 = 0
    2*G1.2 = 0
    2*G1.3 = 0
    2*G1.4 = 0
    2*G1.5 = 0
    2*G1.6 = 0
```

```
    2*G1.7 = 0
    4*G1.8 = 0
    60*G1.9 = 0
    60*G1.10 = 0
> G2, mG2 := RayResidueRing(D2); G2;
Abelian Group isomorphic to Z/15 + Z/15
Defined on 2 generators
Relations:
    15*G2.1 = 0
    15*G2.2 = 0
```

$G_1 = O_{D_1}^{\times}$ should surject onto $D_2 = O_{D_2}^{\times}$ since $D_2 | D_1$:

```
> h := hom<G1 -> G2 | [G1.i@mG1@@mG2 : i in [1..Ngens(G1)]]>;
> Image(h) eq G2;
true
> [ h(G1.i) : i in [1..Ngens(G1)]];
[
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    G2.2,
    G2.1
]
```

Ray class groups are similar and can be mapped in the same way:

```
> R1, mR1 := RayClassGroup(D1);
> R2, mR2 := RayClassGroup(D2); R2;
Abelian Group isomorphic to Z/5 + Z/15 + Z
Defined on 3 generators
Relations:
    5*R2.1 = 0
    15*R2.2 = 0
> hR := hom<R1 -> R2 | [R1.i@mR1@@mR2 : i in [1..Ngens(R1)]]>;
> Image(hR);
Abelian Group isomorphic to Z/5 + Z/15 + Z
Defined on 3 generators in supergroup R2:
    $.1 = R2.1
    $.2 = R2.2
    $.3 = R2.3
Relations:
    5*$.1 = 0
    15*$.2 = 0
> $1 eq R2;
```

```
true
```

Note that the missing C_3 part in comparing G_2 and R_2 corresponds to factoring by $k^\times$. The free factor comes from the class group.
Now, let us investigate the defining exact sequence for R_2:

```
> C, mC := ClassGroup(K);
> h1 := map<K -> G2 | x :-> (K!x)@@mG2>;
> h2 := hom<G2 -> R2 | [ G2.i@mG2@@mR2 : i in [1..Ngens(G2)]]>;
> h3 := hom<R2 -> C | [ R2.i@mR2@@mC : i in [1..Ngens(R2)]]>;
> sub<G2 | [h1(x) : x in k | x ne 0]> eq Kernel(h2);
> Image(h2) eq Kernel(h3);
> Image(h3) eq C;
```

So indeed, the exact sequence property holds.

43.2 Creation of Class Fields

Since the beginning of class field theory one of the core problems has been to find defining equations for the fields. Although most of the original proofs are essentially constructive, they rely on complicated and involved computations and thus were not suited for hand computations.

The method used here to compute defining equations is essentially the same as the one used for number fields. The main differences are due to the problem of p-extensions in characteristic p where Artin-Schreier-Witt theory is used, and the fact that the divisor class group is infinite.

`AbelianExtension(D, U)`

Given an effective divisor D and a subgroup U of the ray class group, (see RayClassGroup), Cl_D of D such that the quotient Cl_D/U is finite, create the extension defined by this data. Note that, at this point, no defining equations are computed.

`MaximalAbelianSubfield(K)`

For a relative extension K/k of global function fields, compute the maximal abelian subfield $K/A/k$ of K/k as an abelian extension of k. In particular, this function compute the norm group of K/k as a subgroup of a suitable ray class group.

`HilbertClassField(K, p)`

For a global function field K and a place p of K, compute the (a) Hilbert class field of K as an abelian extension of K. This field is characterised by being the maximal abelian unramified extension of K where p is totally split.

`FunctionField(A)`

`WithAut`	BoolElt	*Default :* `false`
Verbose	`ClassField`	*Maximum :* 3

Given an abelian extension of function fields as created by `AbelianExtension`, compute defining equations for the corresponding (ray) class field.

More precisely: Let $\mathrm{Cl}/U = \prod \mathrm{Cl}/U_i$ be a decomposition of the norm group of A such that the Cl/U_i are cyclic of prime power order. Then for each U_i the function will compute a defining equation, thus A is represented by the compositum.

If `WithAut` is `true`, the second sequence returned contains a generating automorphism for each of the fields returned as the first return value. If `WithAut` is `false`, the first (and only) return value is a function field in non-simple representation.

`MaximalOrderFinite(A)`

`MaximalOrderInfinite(A)`

Compute a finite or an infinite maximal order of the function field of the abelian extension A.

Example H43E2

First we have to create a function field, a divisor and a ray class group:

```
> k<w> := GF(4);
> kt<t> := PolynomialRing(k);
> ktx<x> := PolynomialRing(kt);
> K := FunctionField(x^3-w*t*x^2+x+t);
> lp := Places(K, 2);
> D := 4*lp[2]+2*lp[6];
> R, mR := RayClassGroup(D);
```

Let us compute the maximal extension of exponent 5 such that the infinite place is totally split. This means that we

- have to create a subgroup of R containing the image of the infinite place
- compute the fifth power of R
- combine the two groups

and use this as input to the class field computation:

```
> inf := InfinitePlaces(K);
> U1 := sub<R | [x@@ mR : x in inf]>;
> U2 := sub<R | [5*R.i : i in [1..Ngens(R)]]>;
> U3 := sub<R | U1, U2>;
> A := AbelianExtension(D, U3);
> A;
Abelian extension of type [ 5 ]
Defined modulo 4*(t^2 + t + w, $.1 + w^2*t + 1) +
2*(t^2 + w^2*t + w^2, $.1 + w*t + w)
over Algebraic function field defined over
```

```
Univariate rational function field over GF(2^2) by
x^3 + w*t*x^2 + x + t
> FunctionField(A);
Algebraic function field defined over K by
$.1^5 + (w*K.1^2 + (w*t + w^2)*K.1 + (w^2*t^2 + t
    + w^2))*$.1^3 + ((t^2 + 1)*K.1^2 + w*t*K.1 +
    (w*t^4 + w))*$.1 + (t^3 + w*t^2 + w^2*t)*K.1^2
    + (w*t^4 + w^2*t^2 + w^2)*K.1 + t^4 + w^2*t^3
    + w^2*t + w
> E := $1;
>  #InfinitePlaces(E);
10
```

Which shows that all the places in `inf` split completely.
Now, suppose we still want the infinite places to split, but now we are looking for degree 4 extensions such that the quotient of genus by number of rational places is maximal:

```
> q, mq := quo<R | U1>;
> l4 := [ x`subgroup : x in Subgroups(q : Quot := [4])];
> #l4;
768
> A := [AbelianExtension(D, x@@mq) : x in l4];
> s4 := [<Genus(a), NumberOfPlacesOfDegreeOne(a), a>
>                             : a in A];
> Maximum([x[2]/x[1] : x in s4]);
16/5 15
> E := FunctionField(s4[15][3]);
>  l22 := [ x`subgroup : x in Subgroups(q : Quot := [2,2])];
>  #l22;
43435
```

Since this is quite a lot, we won't investigate them here further.

43.3 Properties of Class Fields

Let D be an effective divisor and U be a subgroup of the ray class group Cl_D. The main existence theorem of class field theory asserts that there is exactly one function field corresponding to the quotient Cl_D/U whose Galois group is isomorphic to Cl_D/U in a canonical way.

Since the field is uniquely defined this way so are its invariants. Some of the invariants can easily be read off the groups involved. Therfore none of the functions listed here will compute a set of defining equations. They can therefore be used on very large fields.

`Conductor(m)`

Let m be an effective divisor. This function computes the conductor of Cl_m which is the smallest divisor f such that the projection $\mathrm{Cl}_m \to \mathrm{Cl}_f$ is surjective.

`Conductor(m, U)`

Let m be an effective divisor and U be a subgroup of the ray class group of m. This function computes the conductor of Cl_m/U which is the smallest divisor f such that the projection $\pi : \mathrm{Cl}_m/U \to \mathrm{Cl}_f/\pi(U)$ is an isomorphism.

`Conductor(A)`

For an abelian extension A of global function fields, compute its conductor, ie. the conductor of the norm group of A.

`DiscriminantDivisor(m, U)`

Let m be an effective divisor and U a subgroup of ray class group such that Cl_m/U is finite. The discriminant divisor is defined as the norm of the different divisor.

`DiscriminantDivisor(A)`

For an abelian extension A of a global function field, compute its discriminant divisor, ie. the norm of the different divisor. Note that the discriminant divisor can be computed from the norm group, thus no defining equation is derived.

`DegreeOfExactConstantField(m)`

Let m be an effective divisor. Since the ray class field modulo m is always an infinite field extension containing the algebraic closure of the constant field, this returns ∞.

`DegreeOfExactConstantField(m, U)`

Let m be an effective divisor and U be a subgroup of the ray class group, (see `RayClassGroup`), modulo m. This function computes the degree of the algebraic closure of the constant field in the class field corresponding to Cl_m/U. This can be infinite.

`DegreeOfExactConstantField(A)`

The degree of the exact constant field of the abelian extension A of a global function field. This is the degree of the algebraic closure of the constant field of the base field of A in A. The degree of this field can be computed from the norm group, thus no defining equation is derived.

`Genus(m, U)`

Let m be an effective divisor and U be a subgroup of the ray class group, (see `RayClassGroup`), modulo m. This function computes the genus of the class field corresponding to Cl_m/U.

`Genus(A)`

The genus of the abelian extension A of a global function field.

`DecompositionType(m, U, p)`

Let m be an effective divisor and U be a subgroup of the ray class group, (see RayClassGroup), modulo m such that the quotient Cl_m/U is finite. For a place p this function will determine the decomposition type of the place in the extension defined by Cl_m/U, i.e. it will return a sequence of pairs $\langle f, e\rangle$ containing the inertia degree and ramification index for all places above p.

`DecompositionType(A, p)`

For an abelian extension A of a global function field k and a place p of k, compute the degree and ramification index of all places P lying above p.

`NumberOfPlacesOfDegreeOne(m, U)`

Let m be an effective divisor and U be a subgroup of the ray class group, (see RayClassGroup), modulo m such that the quotient Cl_m/U is finite. This function will compute the number of places of degree 1 that the class field corresponding to Cl_m/U has.

`NumberOfPlacesOfDegreeOne(A)`

For an abelian extension A of global function fields, compute the number of places of A that are of degree one over the constant field of the base field of A.

`Degree(A)`

For an abelian extension A of global function fields, return the degree of A over its base ring.

`BaseField(A)`

For an abelian extension A of global function fields, return the base field, ie, the global field k over which A was created as an extension.

`A eq B`

For two abelian extensions of the same base field, decide if they describe the same field, ie if the norm groups pulled back into a common over group, agree.

`A subset B`

For two abelian extensions of the same base field, test if the first is contained in the second. This is done by comparing the norm groups in a common over group, thus defining equations are not computed.

`A meet B`

Compute the intersection of two abelian extensions of the same base field as an abelian extension.

`A * B`

Compute the compositum of two abelian extensions of the same base field as an abelian extension. Since both fields are normal, the compositum is well defined and can be computed from the norm groups alone.

43.4 The Ring of Witt Vectors of Finite Length

The ring of Witt vectors of length n (type `RngWitt`) over a global function field K parametrizes the cyclic extensions of K of degree p^n where p is the characteristic of K. Witt vectors (type `RngWittElt`) can be defined over any ring with positive characteristic p. The ring of Witt vectors of length n will always be of characteristic p^n. In the case of Witt vectors over finite fields, they can be seen as isomorphic to finite quotients of unramified p-adic rings.

The functionality offered here is mainly motivated by the class field theory which deals with vectors of short length only.

The Witt rings are based on code developed by David Kohel.

`WittRing(F, n)`

Creates the ring of Witt vectors of length n where F must be a field of positive characteristic.

`W ! a`

Witt vectors of the Witt ring W can be constructed from a where a is

- an element of the same ring
- an integer
- an element of the base ring
- a sequence of length `Length(W)` whose universe can be changed to the base ring of W.

`BaseRing(W)`

`BaseField(W)`

The field of coefficients of the Witt ring W.

`Length(W)`

The length (dimension) of the elements of the Witt ring W.

`Eltseq(a)`

The list of coefficients of the Witt vector a.

`Unity(W)`

`Zero(W)`

The one resp. zero in the Witt ring W.

`W . 1`

The first non-trivial basis element of the Witt ring.

`x in W` `a eq b` `a - b` `- a` `a + b` `a * b` `a ^ n`

`FrobeniusMap(W)`

The Frobenius map on the Witt ring W which is defined to be the map sending vectors to vectors where every coefficient is raised to the pth power.

`FrobeniusImage(e)`

Computes the image of the Witt vector e under the Frobenius map.

`VerschiebungMap(W)`

The verschiebungs map of the Witt ring W, i.e. shift all coefficients one position to the right and pad with a zero in front.

`VerschiebungImage(e)`

Computes the image of the Witt vector e under the verschiebung map.

`Random(W)`

For finite Witt rings, i.e. Witt rings defined over finite fields, return a random element.

`Random(W, n)`

For Witt rings where the base field admits a random function with size restriction n.

`TeichmuellerSystem(R)`

A Techmüller system for the local ring R, ie. a system of representatives for the residue class field of R that is closed under multiplication.

`LocalRing(W)`

Any ring W of Witt vectors of finite length over a finite field is isomorphic to some unramified local ring. This intrinsic will create the corresponding local ring and the embedding into it.

`ArtinSchreierMap(W)`

Returns the map $x \mapsto F(x) - x$ where F is the Frobenius map of the Witt ring W.

`ArtinSchreierImage(e)`

This function computes the image of the Witt vector e under the Artin-Schreier map.

`FunctionField(e)`

`WithAut`	BoolElt	*Default :* `true`
`Check`	BoolElt	*Default :* `false`
`Abs`	BoolElt	*Default :* `false`

A Witt vector $e = (e_1, \ldots, e_n)$ of length n over k where e_1 is not in the image of the Artin-Schreier map $e_1 \notin \{x^p - x : x \in k\}$ defines a cyclic extension K/k of degree p^n. This function will compute K.

If `WithAut` is `true` in addition to K it will compute a generating automorphism and return it as second return value.

If `Abs` is `true`, the function will compute a K as a single step extension of k, otherwise, K will be constructed as a series of n Artin-Schreier extensions.

If `Check` is `true`, it will be verified that the extension is of degree p^n. In particular the restrictions on e_1 are tested.

43.5 The Ring of Twisted Polynomials

The ring of twisted polynomials plays a core role in the analytic side of class field theory of global function fields. Twisted polynomials can be viewed as additive polynomials where the multiplication is the composition of two polynomials. Alternatively, they are the ring of polynomials in the Frobenius automorphism of the base field as indeterminate and multiplication defined in terms of application of the corresponding endomorphism. Thus Twisted polynomials have two natural polynomial representations:

- as (arbitrary) polynomials in F, the Frobenius automorphism
- as additive polynomials, ie. as polynomials where the only terms are T^{q^i}.

In Magma the first representation is used (as the degrees are lower), but the second can always be obtained by converting to a polynomial.

Since every endomorphism in positive characteristic can be represented by an additive polynomial (or an additive power series), this section can also be viewed as making the endomorphism ring accessible.

In Magma the ring of twisted polynomials is of type `RngUPolTwst` and individual elements are of type `RngUPolTwstElt`. Twisted polynomial rings can be created over any ring of characteristic $p > 0$. They are left euclidean and therefore left PIR.

43.5.1 Creation of Twisted Polynomial Rings

`TwistedPolynomials(R)`

`q`	RngIntElt	*Default :* `false`

Given a ring R of characteristic $p > 0$, create the ring of twisted polynomials over R. If `q` is given it has to be a power of the characteristic p, it defaults to $q := p$. The multiplication in this ring $R < F >$ is defined by $rF = Fr^q$ for all $r \in R$. Elements if $R < F >$ are represented as polynomials in F.

43.5.2 Operations with the Ring of Twisted Polynomials

`Unity(R)`

For a ring R of twisted polynomials return the polynomial representing 1.

`Zero(R)`

For a ring R of twisted polynomials return the polynomial representing 0.

`R eq S`

For two rings of twisted polynomials R and S test if they are equal, that is if the underlying polynomial ring is considered equal by MAGMA. Generically that means to test if the base rings of R and S coincide.

`BaseRing(R)`

The coefficient ring of the ring R of twisted polynomials.

`R . i`

For a ring of twisted polynomials R and an integer i which should be 1, return the transcendental element of R.

43.5.3 Creation of Twisted Polynomials

Apart from creation from polynomials directly, twisted polynomial can also be created by specifying some finite $\mathbf{F}_q$-vector space where the corresponding additive polynomial will have its roots.

`AdditivePolynomialFromRoots(x, P)`

`InfBound`	RNGINTELT	*Default :* 5
`Map`	MAP	*Default :* id
`Class`	DIVFUNELT	*Default :* 0
`Limit`	RNGINTELT	*Default :* ∞
`Scale`	RNGELT	*Default :* `false`

An additive polynomial is characterized by the fact that its set of zeros forms an $\mathbf{F}_q$ vector space. Conversely, given an $\mathbf{F}_p$-vector space M in R there is essentially one additive polynomial that has M as its set of roots. Given a place P and a ring element x, this function computes the additive polynomial with roots $L(\texttt{InfBound}P)$, evaluated at x, ie. the module M is a Riemann-Roch space. By choosing x to be for example a transcendental element in a polynomial ring, the actual additive polynomial can be computed. If the parameter `Map` is given, the elements of the Riemann-Roch space are first mapped by this map before the polynomial is computed, thus allowing the creation of polynomials over the completion of a function field. If the parameter `Limit` is given, the polynomial is reduced modulo $x^{\mathbf{Limit}}$. If `Class` is set to a non-zero divisor, instead of $L(nP)$, the Riemann-Roch space $L(nP+\texttt{Class})$ is used. If "tt Scale is set to a ring element that is either compatible with elements of the Riemann-Roch space or with elements in the codomain of the map, the module is scaled as well, thus allowing for normalization.

`Random(F, n)`

For F the ring of twisted polynomials over a finite ring (ie. a ring that supports the generation of random elements, this function will return a polynomial of degree $n-1$ with randomly chosen coefficients.

Example H43E3

```
> Fq<w> := GF(4);
> k<t> := RationalFunctionField(Fq);
> R := TwistedPolynomials(k:q := 4);
> R![1,1];
T_4 + 1
> R![w*t, 1];
T_4 + w*t
> $2 * $1;
T_4^2 + (w*t^4 + 1)*T_4 + w*t
> $2 * $3;
T_4^2 + (w*t + 1)*T_4 + w*t
> p := Places(k, 1)[2];
> a := AdditivePolynomialFromRoots(PolynomialRing(k).1, p
>          :InfBound := 2);
> a;
T_4^3 + ($.1^96 + $.1^84 + $.1^81 + $.1^72 + $.1^69 + $.1^66 + $.1^60
+ $.1^57 + $.1^54 + $.1^51 + $.1^48 + $.1^45 + $.1^42 + $.1^39 +
$.1^36 + $.1^30 + $.1^27 + $.1^24 + $.1^15 + $.1^12 + 1)/$.1^96*T_4^2
+ ($.1^96 + $.1^93 + $.1^90 + $.1^87 + $.1^72 + $.1^69 + $.1^66 +
$.1^63 + $.1^33 + $.1^30 + $.1^27 + $.1^24 + $.1^9 + $.1^6 + $.1^3 +
1)/$.1^108*T_4 + ($.1^90 + $.1^87 + $.1^84 + $.1^81 + $.1^78 + $.1^60
+ $.1^57 + $.1^54 + $.1^51 + $.1^48 + $.1^42 + $.1^39 + $.1^36 +
$.1^33 + $.1^30 + $.1^12 + $.1^9 + $.1^6 + $.1^3 + 1)/$.1^108
> R, mR := RiemannRochSpace(2*p);
> b := Polynomial(a);
> [ Evaluate(b, mR(x)) eq 0 : x in R];
[ true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true ]
> AdditivePolynomialFromRoots(PolynomialRing(k).1, p:InfBound := 2,
>   Map := func<x|Expand(x, p:RelPrec := 100)>);
T_4^3 + ($.1^-96 + $.1^-84 + $.1^-81 + $.1^-72 + $.1^-69 + $.1^-66 +
$.1^-60 + $.1^-57 + $.1^-54 + $.1^-51 + $.1^-48 + $.1^-45 + $.1^-42 +
$.1^-39 + $.1^-36 + $.1^-30 + $.1^-27 + $.1^-24 + $.1^-15 + $.1^-12 +
1 + O($.1^4))*T_4^2 + ($.1^-108 + $.1^-105 + $.1^-102 + $.1^-99 +
$.1^-84 + $.1^-81 + $.1^-78 + $.1^-75 + $.1^-45 + $.1^-42 + $.1^-39 +
$.1^-36 + $.1^-21 + $.1^-18 + $.1^-15 + $.1^-12 + O($.1^-8))*T_4 +
```

```
$.1^-108 + $.1^-105 + $.1^-102 + $.1^-99 + $.1^-96 + $.1^-78 + $.1^-75
+ $.1^-72 + $.1^-69 + $.1^-66 + $.1^-60 + $.1^-57 + $.1^-54 + $.1^-51
+ $.1^-48 + $.1^-30 + $.1^-27 + $.1^-24 + $.1^-21 + $.1^-18 +
O($.1^-8)
```

43.5.4 Operations with Twisted Polynomials

`A + B` `A - B` `- A` `A * B` `A ^ n`

`A eq B` `IsZero(A)`

`LeadingCoefficient(F)`

For a twisted polynomial F, return the leading coefficient as an element of the coefficient ring.

`ConstantCoefficient(F)`

For a twisted polynomial F, return the constant coefficient as an element of the coefficient ring.

`Degree(F)`

For a twisted polynomial F, return its degree. The degree of the underlying additive polynomial is q times the degree of the twisted polynomial.

`Quotrem(F, G)`

For twisted polynomials F and G in the same ring, perform a right division with remainder: This function computes Q and R such that $F = Q * G + R$ and the degree of R is less than the degree of G. In general, unless the coefficient ring is algebraically closed or perfect, there is no left quotient, so the ring of twisted polynomials is a left-PID, but no right-PID.

`GCD(F, G)`

For twisted polynomials F and G in the same ring, compute the creates common right divisor of F and G, ie a twisted polynomial H such that $F = f_1 H$ and $G = f_2 H$ for some twisted polynomials f_1 and f_2 and such that H is monic of maximal degree.

`BaseRing(F)`

For a twisted polynomial F, return the coefficient ring of F, ie. the ring where all the coefficients of F are from.

`Polynomial(G)`

For a twisted polynomial G, return the corresponding additive polynomial by replacing the transcendental element F by T^q and in general F^i by T^{q^i} for $i = 0, \ldots,$ degree of G.

`SpecialEvaluate(F, x)`

For a twisted polynomial F, return the result of the evaluation of the corresponding additive polynomial at the point x.

`SpecialEvaluate(F, x)`

For a univariate polynomial F, return the evaluation of F at x. This function in particular is optimized for sparse polynomials (for example additive polynomials) and imprecise coefficients. In the general case, a call to `Evaluate` will be faster.

`Eltseq(F)`

For a twisted polynomial F, return a sequence containing its coefficients.

43.6 Analytic Theory

Probably the most significant difference between the class field theories for number fields and function fields is the fact that function fields allow an analytic description of abelian extensions in general where number fields (currently) only admit the analytical view for extensions of the rationals (cyclotomic fields) and imaginary quadratic fields (CM-theory).

The analytic description is based on Drinfeld-modules of rank 1 (or in the case of the rational function field the Carlitz-module). Informally, a Drinfeld module is a representation of some (infinite) maximal order of a function field into the ring of additive polynomials over some appropriate ring containing the original field. Similar to CM-theory, abelian extensions are then generated by adjoining torsion points under this action.

`CarlitzModule(R, x)`

For a rational function field $k = \mathbf{F}_q(t)$ and a polynomial f in $k[t]$, compute the image of f under the Carlitz module as an element in the ring of twisted polynomials R. Specifically, the Carlitz module is the representation induced by sending t to $F + t$ where F is the transcendental element of R, the Frobenius of k. As $\mathbf{F}_q[t]$ is freely generated by t, this defines a homomorphism (a representation) of $\mathbf{F}_q[t]$ into the twisted polynomials over k.

Example H43E4

We demonstrate how the Carlitz-module can be used to define abelian extensions of the rational function field.

```
> Fq<w> := GF(4);
> k<t> := RationalFunctionField(Fq);
> R<T> := TwistedPolynomials(k:q := 4);
```

Suppose we want to create the Ray-class field modulo $p := t^2 + t + w$, ie we want to find an abelian extension unramified outside p and the infinite place.

```
> p := t^2+t+w;
> P := CarlitzModule(R, p);
> F := Polynomial(P);
```

```
> Factorisation(F);
[
    <T, 1>,
    <T^15 + (t^4 + t + 1)*T^3 + t^2 + t + w, 1>
]
> K := FunctionField($1[2][1]);
> a := Support(DifferentDivisor(K));
> a[1];
(t^2 + t + w, K.1)
> [ IsFinite(x) : x in a];
[ true, false, false, false, false, false ]
> RamificationIndex(a[1]);
15
```

So, this shows that the field has the ramification bahaviour we wanted. However, the Carlitz-module will give us more information: We will demonstrate that the automorphism group of K is isomorphic to the unit group of $\mathbf{F}_q[t]/p$ under this module.

```
> q, mq := quo<Integers(k)|p>;
> au := func<X|Evaluate(Polynomial(CarlitzModule(R, X)), K.1)>;
> [ <IsUnit(x), Evaluate(DefiningPolynomial(K), au(x@@mq)) eq 0>
>                     : x in q];
[ <true, true>, <true, true>, <true, true>, <true, true>, <true,
true>, <true, true>, <true, true>, <true, true>, <true, true>, <true,
true>, <true, true>, <true, true>, <true, true>, <true, true>, <true,
true>, <false, false> ]
```

Now we try to create the same field using the algebraic class field machinery:

```
> D, U := NormGroup(K);
> Conductor(D, U);
($.1^2 + $.1 + w) + (1/$.1)
```

So this also shows that K is ramified exactly at p and the infinite places and, since the multiplicity is one, tamely ramified at both places.

```
> A := AbelianExtension(D, U);
> F := FunctionField(A);
> F;
Algebraic function field defined over Algebraic function field defined
over GF(2^2) by
$.2 + 1 by
$.1^3 + ($.1^2 + $.1 + w)^2
$.1^5 + w^2*$.1^3 + w*$.1 + (w^2*$.1^4 + w^2*$.1^2 + 1)/($.1^6 + $.1^5
    + w^2*$.1^4 + $.1^3 + $.1^2 + w^2*$.1 + 1)
> HasRoot(Polynomial(K, DefiningPolynomials(F)[1]));
true K.1^10 + (t^2 + t + w)*K.1^4 + (t^2 + t + w)*K.1
> HasRoot(Polynomial(K, DefiningPolynomials(F)[2]));
true w/(t^2 + t + w)*K.1^12 + (w*t^2 + w*t + 1)/(t^2 + t + w)*K.1^6 +
    w*K.1^3
```

`AnalyticDrinfeldModule(F, p)`

For F a global function field and p a place, compute an algebraic description of "the" Drinfeld modudule of rank 1 defined for the ring of functions integral outside p. More precisely, let R be the ring of functions integral outside p, ie. functions havinf their only poles at R. In MAGMA this is represented by changing the representation of the function field so that p is the only infinite place. Then R becomes simply the finite maximal order. Futhermore, let C be the completion of F at p and $\tilde{C}$ be the closure of the algebraic closure of C, thus D will play the role of the complex numbers in this theory, while C is closely related to the reals. By means of mapping R to its image in $D \subset C$, we obtain a lattice Λ of rank 1. Related to this lattice are certain exponential functions (analytic functions where the set of zeros is Λ). The transformation behaviour of such functions can be used to define a map from R to the endomorphisms of C, represented as additive (twisted) polynomials. In particular, under suitable normalisation, the lattice Λ as defined above, the transformation behaviour can be realised over the Hilbert-class field of F. Since a Drinfeld module is uniquely defined by specifying a single image of a non-constant element of R, this function returns a non-constant (as first return value) and the image as a twisted polynomial over the Hilbert-class field as a second. In case the place is of degree one, the Drinfeld module will also be sign-normalized.

`Extend(D, x, p)`

Given D, the image of a non-constant element under a Drinfeld module for the ring R of functions integral outside p, an element x in R, compute the image of x.

Example H43E5

Although the code is not restricted to genus 1 (or 2) and even though hyperelliptic function fields can be handled in a more direct fashion, we will demonstrate the computation of a Drinfeld module on an elliptic curve.
We begin by defining a curve.

```
> k<w> := GF(4);
> kt<t> := PolynomialRing(k);
> kty<y> := PolynomialRing(kt);
> F := FunctionField(y^3+w*t^5+w*t);
> Genus(F);
1
> ClassNumber(F);
3
```

Now, to define a Drinfeld module, we need to single out an "infinite" place:

```
> p := InfinitePlaces(F)[1];
> Degree(p);
1
```

Since the place p is of degree 1, our Drinfeld module will be sign-normalised.

```
> A, D := AnalyticDrinfeldModule(F, p);
```

```
> H<h> := CoefficientRing(D);
> A;
w^2/(x + 1)*F.1
> D;
T_4^2 + ((w*x^5 + 1)/(x^5 + x)*$.1^2*h^2 + (w^2*x^5 + w^2*x^4 + w^2*x
+ 1)/(x^4 + x^3 + x^2 + x)*F.1^2*h + (x^4 + x^2 + w)/(x^2 +
1)*$.1^2)*T_4 + w^2/(x + 1)*F.1
```

So, the Drinfeld module is uniquely determined by the image of a single non-constant (A) which is chosen to have maximal valuation at p and sign 1.
The values of the Drinfeld module are defined over H which is the Hilbert-class field of F, ie. the maximal abelian unramified extension where p is completely split.

```
> Sign(A, p);
1
> Valuation(A, p);
-2
```

To compute the value of "the" Drinfeld module of a different element we use `Extend`:

```
> b := F!t;
> Extend(D, b, p);
w*T_4^3 + ((w*x^20 + w*x^19 + w*x^18 + w*x^17 + w*x^16 + w*x^15 +
w*x^14 + w*x^13 + w*x^12 + w*x^11 + w*x^10 + w*x^9 + w*x^8 + w*x^7 +
w*x^6 + w*x^5 + w*x^4 + w*x^3 + w*x^2 + w*x + w)/(x^4 + x^3 + x^2 +
x)*$.1^2*h^2 + (w^2*x^20 + w^2*x^19 + w^2*x^16 + w^2*x^15 + w^2*x^4 +
w^2*x^3 + w^2)/(x^3 + x^2 + x + 1)*$.1^2*h + (x^20 + x^18 + x^12 +
x^10 + w*x^4 + w*x^2 + w^2)/(x^2 + 1)*$.1^2)*T_4^2 + ((w*x^5 + w^2*x^3
+ w^2*x^2 + x + w)/(x^2 + 1)*$.1*h^2 + (w^2*x^5 + w^2*x^4 + x^3 +
w)/(x + 1)*$.1*h + (x^5 + w^2*x^3 + w*x^2 + w^2*x)*$.1)*T_4 + x
```

To compute a "Ray-class field" from here we can use the following:

```
> P := Places(F, 2)[1];
> a,b := TwoGenerators(P);
> GCD(Extend(D, a, p), Extend(D, b, p));
T_4^2 + ((w*x^5 + 1)/(x^5 + x)*$.1^2*h^2 + (w^2*x^5 + w^2*x^4 + w^2*x
+ 1)/(x^4 + x^3 + x^2 + x)*$.1^2*h + (x^4 + x^2 + w)/(x^2 +
1)*$.1^2)*T_4 + w^2/(x + 1)*$.1 + w^2
> Polynomial($1);
T_4^16 + ((w*x^5 + 1)/(x^5 + x)*F.1^2*h^2 + (w^2*x^5 + w^2*x^4 + w^2*x
    + 1)/(x^4 + x^3 + x^2 + x)*F.1^2*h + (x^4 + x^2 + w)/(x^2 +
    1)*F.1^2)*T_4^4 + (w^2/(x + 1)*F.1 + w^2)*T_4
> R := FunctionField($1 div Parent($1).1);
> R;
Algebraic function field defined over H by
T_4^15 + ((w*x^5 + 1)/(x^5 + x)*F.1^2*h^2 + (w^2*x^5 + w^2*x^4 + w^2*x
    + 1)/(x^4 + x^3 + x^2 + x)*F.1^2*h + (x^4 + x^2 + w)/(x^2 +
```

```
    1)*F.1^2)*T_4^3 + w^2/(x + 1)*F.1 + w^2
```

This should be an abelian extension over F, ramified only at p and P. So let's try to verify that:

```
> f := MinimalPolynomial(R.1, F);
> Degree(f);
45
> Ra := FunctionField(f:Check := false);
> NormGroup(Ra:Cond := 1*p+P);
(1/x, 1/x^2*F.1) + (x^2 + x + w^2, F.1 + x + 1)
Abelian Group isomorphic to Z
Defined on 1 generator in supergroup:
    $.1 = 2*$.1 + 3*$.2 + $.3 (free)
> D, sub_group := $1;
> W := AbelianExtension(D, sub_group);
> W;
Abelian extension of type [ 3, 15 ]
Defined modulo (1/x, 1/x^2*$.1) + (x^2 + x + w^2, $.1 + x + 1)
over Algebraic function field defined over Univariate rational
function field over GF(2^2) by
y^3 + w*t^5 + w*t
> FunctionField(W);
Algebraic function field defined over F by
$.1^3 + (x + 1)^-2 * (x^2 + x + w^2)^-1 * (F.1 + w*x + w) * (F.1 +
    w^2*x + w^2)
$.1^3 + (x) * (x + 1)^-4 * (F.1 + x + 1)^2 * (x^2 + x + w^2)^-1 * (F.1
    + w*x + w) * (F.1 + w^2*x + w^2)
$.1^5 + (1/(x^2 + 1)*F.1^2 + w^2/(x + 1)*F.1 + (x^2 + x + 1))*$.1^3 +
    (w/(x^2 + 1)*F.1^2 + w*x*F.1 + (x^4 + x^2 + 1))*$.1 + (x^2 + x +
    1)/(x^2 + 1)*F.1^2 + w^2*x*F.1 + x^4 + w^2*x^2 + w*x + w^2
> WW := $1;
>  [HasRoot(Polynomial(Ra, x)) : x in DefiningPolynomials(WW)];
[ true, true, true]
```

`Exp(x,p)`

`InfBound`	RngIntElt	*Default :* 5
`Map`	Map	*Default :* id
`Class`	DivFunElt	*Default :* 0
`Limit`	RngIntElt	*Default :* ∞
`Scale`	RngElt	*Default :* `false`

In Drinfeld's theory of elliptic modules, one associates an exponential function to a lattice. The transformation of this function under scaling of the lattice gives then rise to the "Drinfeld-module". This function computes an approximation to the exponential of the "standard-lattice". More precisely: let R be the ring of functions integral outside the place p and let C be the completion of the function field at p.

Considered as a subset of C, R is a 1-dimensional lattice Λ in Drinfelds sense. The exponential associated to this lattice is the function

$$\exp : z \to z \prod{}'(1 - z/l)$$

where the product is taken over all non-zero lattice points l. This function can be seen to be an additive analytic function.

For $n > 0$, let now $L(np)$ be the Riemann-Roch spaces and

$$f_n : z \to z \prod{}'(1 - z/l)$$

(the product is over the non-zero elements of $L(np)$). It can be seen that $f_n \to \exp$ as $n \to \infty$. This intrinsic computes f_n for n equal the value of `InfBound`. If `Limit` is given then the twised polynomial representing f_n is truncated at that term. If `Class` is given and contains the divisor d, then instead of $L(np)$ we use $L(np + d)$ which will approximate a (in general) non-isomorphic exponential coming from the lattice from the ideal representing the finite part of d.

If `Map` is given, then the map is applied to each element in $L(np)$ first, thus allowing to compute analytic approximations instead of alegebraic ones. Additionally, if `Scale` is given, the elements of $L(np)$ are multiplied by this value before the functions are formed, corresponding to a scaling of the lattice.

The exponential is evaluated at x, the first argument. Typically, x will be the transcendental element of a polynomial ring, a twisted polynomial ring or a power series ring.

`AnalyticModule(x, p)`

`InfBound`	RngIntElt	*Default :* 5
`Map`	Map	*Default :* id
`Class`	DivFunElt	*Default :* 0
`Limit`	RngIntElt	*Default :* ∞
`Scale`	RngElt	*Default :* `false`

Let Λ be the lattice as described for `Exp` above. By Drinfeld's theory, the exponential functions of Λ and $x\Lambda$ are related through some polynomial. This function computed the polynomial for x, which is "the" image of x under the Drinfeld module defined by Λ. The use of the parameters is as for `Exp` above.

`CanNormalize(F)`

Let F be a twisted polynomial, typically over a completion. This function tries to conjugate F so that the coefficients are integral with small valuations. On success, `true`, the new polynomial and the element used to normalise F is returned.

`CanSignNormalize(F)`

Let F be a twisted polynomial, typically over a completion. This function tries to conjugate F so that the highest coefficient is an element in the residue class field. On success, `true`, the new polynomial and the element used to normalise F is returned.

`AlgebraicToAnalytic(F, p)`

Given a non-trivial image F under a Drinfeld module with the "infinite place" p, compute a basis for a submodule of the lattice underlying F. The parameter `RelPrec` is used to limit the number of coefficients of the exponential that are reconstructed, thus it also limits the dimenstion of the submodule.

43.7 Related Functions

This section list some related functions that are either useful in the context of class fields for function fields or are necessary for their computation. They will most certainly change their appearance.

`StrongApproximation(m, S)`

`Strict`	BoolElt	*Default :* `false`
`Exception`	DivFunElt	*Default :* `false`
`Raw`	BoolElt	*Default :* `false`

Given an effective divisor m and a sequence S of pairs (Q_i, e_i) of places and elements, find an element a and a place Q_0 such that

$$v_{Q_i}(a - e_i) \geq v_{Q_i}(m),$$

and a is integral everywhere outside Q_i $(0 \leq i \leq n)$.

If `Exception` is not `false`, it has to be a place that will be used for Q_0.

If `Strict` is `true`, the element a will be chosen such

$$v_{Q_i}(a - e_i) = v_{Q_i}(m)$$

If `Raw` is `true`, different rather technical return values are computed that are used internally.

Example H43E6

We first have to define a function field and some places:

```
> k<w> := GF(4);
> kt<t> := PolynomialRing(k);
> ktx<x> := PolynomialRing(kt);
> K := FunctionField(x^3-w*t*x^2+x+t);
```

```
> lp := Places(K, 2);
```

We will now try to find an element x in K such that $v_{p_i}(x - e_i) \geq m_i$ for $p_i =$`lp[i]`, $m_i = i$ and random elements e_i:

```
> e := [Random(K, 3) : i in lp];
> m := [i : i in [1..#lp]];
> D := &+ [ m[i]*lp[i] : i in [1..#lp]];
> x := StrongApproximation(D, [<lp[i], e[i]> : i in [1..#lp]]);
> [Valuation(x-e[i], lp[i]) : i in [1..#lp]];
[ 1, 2, 3, 4, 5, 6 ]
```

Note, that we only required $\geq$ for the valuations, to enforce $=$ we would need to pass the `Strict` option. This will double the running time.

`NonSpecialDivisor(m)`

`Exception`	DivFunElt	*Default :*

Given an effective divisor m, find a place P coprime to m and an integer $r \geq 0$ such that $rP - m$ is a non special divisor and return r and P.

If `Exception` is specified, it must be an effective divisor n coprime to m. In this case the function finds $r > 0$ such that $rn - m$ is non special and returns r and n.

`NormGroup(F)`

`Cond`	DivFunElt	*Default :*
`AS`	RngWittElt	*Default :*
`Extra`	RngIntElt	*Default :* 5

Given a global function field, try to compute its norm group. The norm group is defined to be the group generated by norms of unramified divisors. This group can be related to a subgroup of some ray class group.

Provided F is abelian, this function will compute a divisor m and a sub group U of the ray class group modulo m such that F is isomorphic to the ray class field thus defined.

This function uses a heuristic algorithm. It will terminate after the size of the quotient by the norm group is less or equal than the degree for `Extra` many places.

If `Cond` is given, it must be an effective divisor that will be used as the potential conductor of F. Note: if `Cond` is too small, ie. a proper divisor of the true conductor, the result of this function will be wrong. However, if the conductor is not passed in, the discriminant divisor is used as a starting point. As this is in general far too large, the function will be much quicker if a better (smaller) starting point is passed in.

If `AS` is given, it must be a Witt vector e of appropriate length and F should be the corresponding function field. This allows a much better initial guess for the conductor than using the discriminant.

`Sign(a, p)`

Given a function a in some global function field and a place p such that a is integral at p (has non-negative valuation) return the sign of a, ie. the first non-zero coefficient if the expansion of a at p. The sign function is not unique. MAGMA choses a sign function when creating the residue class field map.

`ChangeModel(F, p)`

Given a global function field F and a place p, return a new function field G that is $\mathbf{F}_q$-isomorphic to F and has p as the only infinite place.

43.8 Enumeration of Places

In several situations one needs to loop over the places of a function field until either the one finds a place with special properties or until they generate a certain group. The functions listed here support this.

`PlaceEnumInit(K)`

`Coprime`	ANY	*Default :*
`All`	BOOLELT	*Default :* `false`

Initialises an enumeration process for places of the function field K. The enumeration process will loop over all irreducible polynomials of the underlying finite field and for each polynomial over all primes lying above it.

If `Coprime` is given, it should be either a set of places that should be ignored in the process or a divisor. In case a divisor is passed in only places coprime to the divisor will be returned.

If `All` is `false`, the infinite places won't be considered.

`PlaceEnumInit(P)`

`Coprime`	ANY	*Default :*

Constructs an enumeration process for places starting at the place P.

If `Coprime` is given, it should be either a set of places that should be ignored in the process or a divisor. In case a divisor is passed in only places coprime to the divisor will be returned.

`PlaceEnumInit(K, Pos)`

`Coprime`	ANY	*Default :*

Constructs an enumeration environment that starts at the place of the function field K indexed by Pos as returned from `PlaceEnumPosition`.

`PlaceEnumCopy(R)`

Copies the environment and the current state of the enumeration process for places R.

`PlaceEnumPosition(R)`

Returns a list of integers that acts as an index to the places as enumerated by the environment R.

`PlaceEnumNext(R)`

Returns the "next" place of the process R.

`PlaceEnumCurrent(R)`

Returns the current place pointed to by the environment R, i.e. the last place returned by `PlaceEnumNext`.

43.9 Bibliography

[**HPP97**] Florian Heß, Sebastian Pauli, and Michael E. Pohst. On the computation of the multiplicative group of residue class rings. *Math. Comp.*, 1997.

44 ARTIN REPRESENTATIONS

Chapter 44
ARTIN REPRESENTATIONS

44.1 Overview

An *Artin representation* is a complex representation of $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q})$ that factors through some finite quotient $\mathrm{Gal}(F/\mathbf{Q})$. In MAGMA, Artin representations are represented as characters of $\mathrm{Gal}(F/\mathbf{Q})$, not the actual modules. They are allowed to be virtual, except in the L-function machinery (see Chapter 127).

44.2 Constructing Artin Representations

`ArtinRepresentations(K)`

`f`	RngUPolElt	*Default :*
`Ramification`	BoolElt	*Default :* `false`
`FactorDiscriminant`	BoolElt	*Default :* `false`
`p0`	RngIntElt	*Default :*

Compute all irreducible Artin representations that factor through the normal closure F of the number field K.

The Galois group $G = \mathrm{Gal}(F/K)$ whose representations are constructed is represented as a permutation group on the roots of `f`, which must be a monic irreducible polynomial with integer coefficients that defines K. By default this is the defining polynomial of K represented as an extension of $\mathbf{Q}$. (It is possible to specify any monic integral polynomial whose splitting field is F, even a reducible one, but `PermutationCharacter(K)` and the Dedekind ζ-function of K will not work correctly.)

The `Ramification` parameter specifies whether to pre-compute the inertia groups at all ramified primes and the conductors of all representations.

The parameter `FactorDiscriminant` determines whether to factorize the discriminant of `f` completely, even if it appears to contain large prime factors. The factorization is used to determine which primes ramify in F/K, which is necessary to compute the conductors. If the factorization is incomplete, MAGMA assumes that the primes in the unfactored part of the discriminant are unramified. One may specify `FactorDiscriminant:=`

`<TrialLimit,PollardRhoLimit,ECMLimit,MPQSLimit,Proof>` and these 5 parameters are passed to the `Factorization` function; the default behaviour (`false`) is the same as `<10000,65535,10,0,false>`. When the factorization is incomplete, MAGMA will print "(?)" following the conductor values, when asked to print an Artin representation.

Finally, p0 specifies which p-adic field to use for the roots of f, in particular in Galois group computations. It must be chosen so that GaloisGroup(f:Prime:=p0) is successful. By default it is chosen by the Galois group computation.

```
K !! ch
```

Writing F for the normal closure of $K/\mathbf{Q}$, this function converts an abstract group character of $\mathrm{Gal}(F/\mathbf{Q})$ or the sequence of its trace values into an Artin representation.

```
PermutationCharacter(K)
```

Construct the permutation representation A of the absolute Galois group of $\mathbf{Q}$ on the embeddings of K into $\mathbf{C}$. This is an Artin representation of $\mathrm{Gal}(F/\mathbf{Q})$ of dimension $[K:\mathbf{Q}]$, where F is the normal closure of K, and it is the same as the permutation representation of $\mathrm{Gal}(F/\mathbf{Q})$ on the cosets of $\mathrm{Gal}(F/K)$.

```
Determinant(A)
```

Construct the determinant of a given Artin representation. The result is given as a 1-dimensional Artin representation attached to the same field.

```
ChangeField(A,K)
```

```
K !! A
```

MinPrimes RngIntElt *Default :* 20

Given an Artin representation (attached to some number field) that is known to factor through the Galois closure of K, attempts to recognize it as such. Returns "the resulting Artin representation attached to K", true if successful, and 0, false if it proves that there is no such representation. The parameter MinPrimes specifies the number of additional primes for which to compare traces of Frobenius elements.

Example H44E1

A quadratic field K has two irreducible Artin representations the factor through $\mathrm{Gal}(K/\mathbf{Q})$, the trivial one and the quadratic character of K:

```
> K<i> := QuadraticField(-1);
> triv, sign := Explode(ArtinRepresentations(K));
> sign;
Artin representation of Quadratic Field with defining polynomial x^2 + 1
over the Rational Field with character ( 1, -1 )
```

An alternative way to define them is directly by their character:

```
> triv,sign:Magma;
QuadraticField(-1) !! [1,1]
QuadraticField(-1) !! [1,-1]
```

The regular representation of $\mathrm{Gal}(K/\mathbf{Q})$ is their sum:

```
> PermutationCharacter(K);
```

```
Artin representation of Quadratic Field with defining polynomial z^2 + 1
over the Rational Field with character ( 2, 0 )
> $1 eq triv+sign;
true
```

Next, let $L = K(\sqrt{-2-i})$. Then L has normal closure F with $\mathrm{Gal}(F/\mathbf{Q}) = D_4$, the dihedral group of order 8:

```
> L := ext<K|Polynomial([2+i,0,1])>;
> G := GaloisGroup(AbsoluteField(L));
> IsIsomorphic(G,DihedralGroup(4));
true
> [Dimension(A): A in ArtinRepresentations(L)];
[1, 1, 1, 1, 2 ]
```

We use `ChangeField` to lift Artin representations from $\mathrm{Gal}(K/\mathbf{Q})$ to $\mathrm{Gal}(F/\mathbf{Q})$, and check that it is still the same as an Artin representation.

```
> A := ChangeField(sign,L);
> A;
Artin representation of Number Field with defining polynomial $.1^2 + i + 2
over its ground field with character ( 1, 1, -1, 1, -1 )
> A eq sign;
true;
```

44.3 Basic Invariants

`Field(A)`

Number field K such that A factors through the Galois group of the normal closure of K.

`Degree(A)`

Degree (=dimension) of an Artin representation A.

`Group(A)`

The Galois group of the field through which A factors.

`Character(A)`

Character of an Artin representation A, represented as a complex-valued character of `Group(A)`.

`Conductor(A)`

Conductor of an Artin representation A (which must be a true representation, i.e. its character is not allowed to be a generalized character). Computes all the necessary local information if Artin representations were defined with `Ramification:=false`, so the first call to this function might take some time.

`Decomposition(A)`

Decompose an Artin representation A into irreducible constituents. Returns a sequence of tuples `[...<`A_i`,`n_i`>...]` with A_i irreducible and n_i its exponent in A (nonzero but possibly negative).

`DefiningPolynomial(A)`

Returns the polynomial whose roots `Group(A)` permutes.

`Minimize(A)`

`Optimize` BoolElt *Default :* `true`

Returns A attached to the smallest number field K such that A factors through its Galois closure. If `Optimize := true`, attempts to minimize the defining polynomial of K using `OptimizedRepresentation`.

`Kernel(A)`

Smallest Galois extension K of the rationals through which A factors. Note that this field may be enormous and incomputable.

Example H44E2

We take an S_4-extension of $\mathbf{Q}$ and compute its Artin representations.

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^4+9*x-2);
> A := ArtinRepresentations(K);
> [Dimension(a): a in A];
[ 1, 1, 2, 3, 3 ]
```

Then we minimize the 2-dimensional one, which factors through an S_3-quotient.

```
> B := Minimize(A[3]); B;
Artin Representation of Number Field with defining polynomial x^3 + 8*x - 81
over the Rational Field with character ( 2, 0, -1 )
> Kernel(B);
Number Field with defining polynomial x^6 + 48*x^4 + 576*x^2 + 179195 over the
Rational Field
```

`IsIrreducible(A)`

Return `true` iff a given Artin representation is irreducible as a complex representation.

`IsRamified(A, p)`

Return `true` iff a given Artin representation is ramified at p.

`IsWildlyRamified(A, p)`

Return `true` iff a given Artin representation is wildly ramified at p.

`EulerFactor(A, p)`

`R` FLD *Default :* `ComplexField()`

The local polynomial (Euler factor) of an Artin representation A at the prime p. It is a polynomial with coefficients in the field R, which is complex numbers by default, and it is the inverse characteristic polynomial of (arithmetic) Frobenius at p on the inertia invariant subspace of A.

Example H44E3

Here are the invariants of Artin representations that factor through the splitting field of $x^4 - 3$, a D_4-extension of $\mathbf{Q}$.

```
> R<x> := PolynomialRing(Rationals());
> K := NumberField(x^4-3);
> A := ArtinRepresentations(K);
> Degree(Field(A[1]),Rationals());
8
> [Dimension(a): a in A];
[ 1, 1, 1, 1, 2 ]
> Character(A[5]);
( 2, -2, 0, 0, 0 )
> [Conductor(a): a in A];
[ 1, 3, 4, 12, 576 ]
> [IsRamified(a,3): a in A];
[ false, true, false, true, true ]
> [IsWildlyRamified(a,3): a in A];
[ false, false, false, false, false ]
> EulerFactor(A[5],5);
$.1^2 + 1.00000000000000000000000000000
```

`DirichletCharacter(A)`

Convert a one-dimensional Artin representation to a Dirichlet character.

`ArtinRepresentation(ch)`

`field` FLDNUM *Default :*

Convert a Dirichlet character `ch` to a one-dimensional Artin representation A. To avoid recomputation, the minimal field through which A factors may be supplied by the `field` parameter. This now uses class field theory (thanks to C. Fieker).

Example H44E4

An example that goes back and forth between the Dirichlet character and the Artin representation.

```
> load galpols;
> f := PolynomialWithGaloisGroup(8,46); // order 576
> K := NumberField(f); // octic field
> A := ArtinRepresentations(K);
> [Degree(a) : a in A];
[ 1, 1, 1, 1, 4, 4, 6, 6, 9, 9, 9, 9, 12 ]
> [Order(Character(Determinant(a))) : a in A];
[ 1, 2, 4, 4, 2, 2, 1, 2, 2, 1, 4, 4, 2 ]
> chi := DirichletCharacter(A[3]); // order 4
> Conductor(chi), Conductor(chi^2);
215 5
> Minimize(ArtinRepresentation(chi)); // disc = N(chi)^2*N(chi^2)
Artin representation of Number Field with defining polynomial
 x^4 - x^3 - 54*x^2 + 54*x + 551 with character ( 1, -1, I, -I )
> Factorization(Discriminant(Integers(Field($1))));
[ <5, 3>, <43, 2> ]
```

44.4 Arithmetic

`A1 + A2`

Direct sum of two Artin representations

`A1 - A2`

Direct difference of two Artin representations

`A1 * A2`

Tensor product of two Artin representations

`A1 eq A2`

Returns `true` iff the two Artin representations are equal

`A1 ne A2`

Returns `true` iff the two Artin representations are not equal

Example H44E5

For Artin representations constructed from the same number field, their arithmetic is just arithmetic of characters:

```
> P<x> := PolynomialRing(Rationals());
> K := NumberField(x^3-2);
> A := ArtinRepresentations(K: Ramification:=true);
> triv, sign, rho := Explode(A);
> triv;
Artin representation of Number Field with defining polynomial
x^3 - 2 over the Rational Field with character ( 1, 1, 1 )
and conductor 1
> rho;
Artin representation of Number Field with defining polynomial
x^3 - 2 over the Rational Field with character ( 2, 0, -1 )
and conductor 108
> triv+rho;
Artin representation of Number Field with defining polynomial
x^3 - 2 over the Rational Field with character ( 3, 1, 0 )
and conductor 108
> sign*rho eq rho;
true
```

Example H44E6

When Artin representations factor through different fields, their arithmetic involves the compositum of the fields:

```
> K1 := QuadraticField(2);
> triv1, sign1 := Explode(ArtinRepresentations(K1));
> K2 := QuadraticField(3);
> triv2, sign2 := Explode(ArtinRepresentations(K2));
> twist := sign1*sign2;
> Field(twist);
Number Field with defining polynomial $.1^4 - 10*$.1^2 + 1
over the Rational Field
> sign3 := Minimize(twist);
> sign3;
Artin representation of Number Field with defining polynomial
$.1^2 - 6 over the Rational Field with character ( 1, -1 )
> sign1*sign2*sign3 eq triv1;
true
```

44.5 Implementation Notes

The algorithms for recognizing Frobenius elements in Galois groups are described in [DD10]. They rely on the cycle type identification, Serre's trick for alternating groups and the general machinery from [DD10]. MAGMA is usually able to handle Galois groups of size < 10000 acting on a small number of points easily, and much larger special groups such as A_n and S_n.

44.6 Bibliography

[**DD10**] T. Dokchitser and V. Dokchitser. Identifying conjugacy classes in Galois groups. 2010.

PART VII
LOCAL ARITHMETIC FIELDS

45 VALUATION RINGS

Chapter 45
VALUATION RINGS

45.1 Introduction

MAGMA currently supports basic operations in valuation rings obtained either from the rational field $\mathbf{Q}$ (and a finite prime p), or from a field of rational functions over a field (and an irreducible polynomial, or the infinite prime).

45.2 Creation Functions

45.2.1 Creation of Structures

`ValuationRing(Q, p)`

Given the rational field Q and a rational prime number p, create the valuation ring R corresponding to the discrete non-Archimedean valuation v_p, consisting of rational numbers r such that $v_p(r) \geq 0$, that is, $r = \frac{x}{y} \in \mathbf{Q}$ such that $p \not| y$.

`ValuationRing(F, f)`

Given the rational function field F as a field of fractions of the univariate polynomial ring $K[x]$ over a field K, as well as a monic irreducible polynomial $f \in K[x]$, create the valuation ring R corresponding to the discrete non-Archimedean valuation v_f. Thus R consists of rational functions $\frac{g}{h} \in F$ with $v_f(g/h) \geq 0$, that is, with $f \not| h$.

`ValuationRing(F)`

Given the rational function field F as a field of fractions of the univariate polynomial ring $K[x]$ over a field K, create the valuation ring R corresponding to v_∞, consisting of $\frac{g}{h} \in F$ such that $\deg(h) \geq \deg(g)$.

45.2.2 Creation of Elements

`V ! r`

Given a valuation ring V and an element of the field of fractions F of V (from which V was created), coerce the element r into V. This is only possible for elements $r \in F$ for which the valuation on V is non-negative, an error occurs if this is not the case.

45.3 Structure Operations

45.3.1 Related Structures

`Category(V)` `Parent(V)` `PrimeRing(V)` `Center(V)`

`FieldOfFractions(V)`

Return field of fractions of the valuation ring V, which is the rational field or the function field from which V was created.

45.3.2 Numerical Invariants

`Characteristic(V)`

45.4 Element Operations

45.4.1 Arithmetic Operations

`+ v` `- v`

`v + w` `v - w` `v * w` `v ^ k` `v / w`

`v +:= w` `v -:= w` `v *:= w`

`v div w`

The quotient q of the division with remainder $v = qw + r$ of the valuation ring elements v and w, where the remainder will have valuation less than that of w; if the valuation of v is greater than or equal than that of w, this simply returns the quotient v/w, if the valuation of w exceeds that of v it returns 0.

45.4.2 Equality and Membership

`v eq w` `v ne w`

`v in V` `v notin V`

45.4.3 Parent and Category

`Parent(v)` `Category(v)`

45.4.4 Predicates on Ring Elements

`IsZero(n)` `IsOne(n)` `IsMinusOne(n)`
`IsNilpotent(n)` `IsIdempotent(n)`
`IsUnit(n)` `IsZeroDivisor(n)` `IsRegular(n)`

45.4.5 Other Element Functions

`EuclideanNorm(v)`
`Valuation(v)`

Given an element v of a valuation ring V, return the valuation (associated with V) of v.

`Quotrem(v, w)`

Given two elements v, w of a valuation ring V with associated valuation ϕ, return a quotient and remainder q and r in V such that $v = qw + r$ and $0 \leq \phi(r) < \phi(w)$. If $\phi(v) < \phi(w)$ this simply returns $q = 0$ and $r = v$, and if $\phi(v) \geq \phi(w)$ then it returns $q = v/w$ and $r = 0$.

`GreatestCommonDivisor(v, w)`
`Gcd(v, w)`

This function returns a greatest common divisor of two elements v, w in a valuation ring V. This will return u^m, where $m = \min(\phi(v), \phi(w))$ is the minimum of the valuations of v and w $m = \min(\phi(v), \phi(w))$ and u is the uniformizing element of V (with valuation $\phi(u) = 1$).

`ExtendedGreatestCommonDivisor(v, w)`
`Xgcd(v, w)`
`XGCD(v, w)`

This function returns a greatest common divisor $z \in V$ of two elements v, w in a valuation ring V as well as multipliers $x, y \in V$ such that $xv + yw = z$. The principal return value will be $z = u^m$, where $m = \min(\phi(v), \phi(w))$ is the minimum of the valuations of v and w $m = \min(\phi(v), \phi(w))$ and u is the uniformizing element of V (with valuation $\phi(u) = 1$).

46 NEWTON POLYGONS

Chapter 46
NEWTON POLYGONS

46.1 Introduction

This chapter introduces functions which allow the creation and simple study of Newton polygons. It allows data from a number of different contexts to be used to construct the polygons. It also covers different interpretations of Newton polygons, and translation among these interpretations. Recall that a Newton polygon is the intersection of finitely many rational half spaces in the rational plane. An advantage of this definition is that it emphasizes that Newton polygons are often thought of as being noncompact. However, they are not implemented here in this way. Instead polygons are interpreted as the convex hull of finitely many points of the plane (possibly including some points at $+\infty$ along the axes).

Any Newton polygon is contained in some virtual cartesian product of the rational field with itself (virtual since this plane is not a structure that is intended to be accessible to the user or which is characteristic of the polygon). The first and second coordinate functions of this plane are referred to as the x and y coordinates respectively. Points of the plane will always be written $\langle a, b\rangle$ while faces of polygons will be written $\langle a, b, c\rangle$. Geometrically, a face $\langle a, b, c\rangle$ is a one-dimensional boundary intersection of N with the line $ax + by = c$.

The *standard Newton polygon* of a polynomial $f = f(u, v)$ is, by definition, the convex hull of the points $\langle a, b\rangle$, so-called *Newton points*, ranging over monomials $u^a v^b$ having nonzero coefficient in f together with the points $+\infty$ on the two axes. The infinite points, however, are not listed among the vertices of the polygon; they are simply a convenient way of hiding all Newton points other than those on the 'lower lefthand' faces of the hull of those points. A similar definition applies to polynomials $f(y)$ whose coefficients lie in some field of fractional power series, or *Puiseux field*, $k\langle\langle x\rangle\rangle$. Newton Polygons can also be created for polynomials over local rings (and fields). The defining points are computed as $\langle i, v(a_i)\rangle$ where a_i is the coefficient of the i-th power of the generator and v denotes the valuation function on the ring. Infinite points arise only as the valuation of zero coefficients.

The main intended application of Newton polygons is to the Newton–Puiseux analysis of singular points of plane curves, or put another way, the factorization of polynomials defined over Puiseux fields.

The examples below should be thought of as running consecutively in a single MAGMA session.

```
> R<x,y> := PolynomialRing(Rationals(),2);
> f := x^5 + x^2*y + x^2*y^3 + y^4 + x^3*y^5 + y^2*x^7;
> N := NewtonPolygon(f);
> N;
Newton Polygon of x^7*y^2 + x^5 + x^3*y^5 + x^2*y^3 + x^2*y + y^4 over
```

```
Rational Field
> Faces(N);
[ <3, 2, 8>, <1, 3, 5> ]
> Vertices(N);
[ <0, 4>, <2, 1>, <5, 0> ]
```

This is the standard Newton polygon associated with the polynomial f. Only those vertices and faces of the convex hull of points which correspond to the monomials appearing in f which 'face' the origin are considered to be vertices and faces of the polygon. As already mentioned, one interpretation of this is to think of the points $+\infty$ on each axis as being included among the defining points of N, and then N is the convex hull of its defining points.

To consider N as the compact convex hull of only the monomials of f simply use alternative vertex and face functions.

```
> AllVertices(N);
[ <0, 4>, <2, 1>, <5, 0>, <7, 2>, <3, 5> ]
> AllFaces(N);
[ <3, 2, 8>, <1, 3, 5>, <-1, 1, -5>, <-3, -4, -29>, <1, -3, -12> ]
```

However, where there is a choice of interpretation, MAGMA always interprets N in the way it was defined. So for example, the functions which test for faces and vertices compare a given value with the sequence returned by the functions `Faces(N)` or `Vertices(N)`, which are fixed the first time they are calculated, rather than with any other collections of faces or vertices.

```
> IsFace(N,<3/4,3/6,2>);
true <3, 2, 8>
> IsFace(N,<-3,-4,-29>);
false
```

Notice that faces are reduced to normal integral form and that the correct form is returned as the second return value of the test function.

When a polygon is created using a polynomial, the restriction of the polynomial to faces is important characteristic data.

```
> FaceFunction(Faces(N)[1]);
x^2*y + y^4
```

The face function is simply the sum of those monomial terms of f (they keep their coefficients) whose corresponding Newton point lies on the given face.

46.2 Newton Polygons

All polygons are determined by a finite collection of points in the rational plane. For MAGMA, these points are the most basic attribute of any Newton polygon. They are always determined and recorded on creation of a polygon. Throughout this chapter, for a Newton polygon N, these points are denoted by P_N. As seen in the introduction, the main class of polygons is that comprising polygons in the first quadrant of the plane and including the points $+\infty$ on the two axes. But there are other useful types, especially when calculating factorizations of univariate polynomials over series rings. The data distinguishing the different flavours of Newton polygon is the collection of lines and points that are considered to be faces and vertices.

46.2.1 Creation of Newton Polygons

These are the functions available for constructing Newton polygons and retrieving the points which describe them.

`NewtonPolygon(f)`

`Faces`	MONSTGELT	*Default :* "*Inner*"

The standard Newton polygon of a polynomial f in two variables. This is the hull of the Newton points of the polynomial together with the points $+\infty$ on each axis. The horizontal and vertical 'end' faces are not listed among the faces of the polygon; the points at infinity are not listed among the vertices of the polygon.

The parameter `Faces` can have the value `"Inner"`, `"Lower"` or `"All"`. This determines which faces are returned by the intrinsic `Faces`.

`NewtonPolygon(f)`

`SwapAxes`	BOOLELT	*Default :* `false`
`Faces`	MONSTGELT	*Default :*

The standard Newton polygon of a polynomial in one variable defined over a series ring or a local ring or field. A value of `true` for `SwapAxes` is only valid if the polynomial is over a series ring. If `SwapAxes` is set to `true` then the exponents of the series variable will be plotted on the horizontal axis and the exponents of the polynomial on the vertical axis.

For a polynomial over a series ring, the hull includes the points $+\infty$ on each axis. For a polynomial over a local ring, the infinite points are not included.

The parameter `Faces` can have the value `"All"`, `"Inner"` or `"Lower"`. This determines which faces are returned by the intrinsic `Faces`. The default for series rings is `"Inner"` and for local rings is `"Lower"`.

`NewtonPolygon(f, p)`

`Faces`	MONSTGELT	*Default :* "*Inner*"

The newton polygon of f where p is a prime used for valuations of the coefficients of f. The polynomial f may be over the integers or rationals or a number field or

algebraic function field or an order thereof. The prime p may be an integer or a prime ideal. The newton polygon will have points (i, v_i) where i is the exponent of a term of f and v_i is the valuation of the coefficient of the ith term. The points at $+\infty$ on each axis are included.

The parameter `Faces` can have the value `"Inner"`, `"Lower"` or `"All"`. This determines which faces are returned by the intrinsic `Faces`.

`NewtonPolygon(f, p)`

`Faces` MonStgElt *Default : "Inner"*

The newton polygon of the polynomial f where the place p of an algebraic function field is the prime used for determining the valuations of the coefficients of f. The points at $+\infty$ on each axis are included.

The parameter `Faces` can have the value `"Inner"`, `"Lower"` or `"All"`. This determines which faces are returned by the intrinsic `Faces`.

`NewtonPolygon(C)`

The standard Newton polygon of the defining polynomial of the curve C.

`NewtonPolygon(V)`

`Faces` MonStgElt *Default : "All"*

The Newton polygon that is the compact convex hull of the set or sequence V of points of the form $\langle a, b\rangle$ where a, b are integers or rational numbers.

The parameter `Faces` can have the value `"All"`, `"Lower"` or `"Inner"`. This determines which faces are returned by the intrinsic `Faces`.

`DefiningPoints(N)`

The points of the rational plane used in the initial creation of N. Applying this function to two polygons allows their defining points to be compared. No explicit function is provided for testing whether defining points of two polygons are equal.

Example H46E1

Some ways of creating Newton Polygons from polynomials are shown below.

```
> P<y> := PuiseuxSeriesRing(Rationals());
> R<x> := PolynomialRing(P);
> f := 3*x^4 + (5*y^3 + 4*y^(1/4))*x^3 + (7*y^2 + 1/2*y^(1/3))*x^2 + 6*x + y^(
> 4/5);
> N := NewtonPolygon(f);
> N;
Newton Polygon of 3*x^4 + (4*y^(1/4) + 5*y^3)*x^3 + (1/2*y^(1/3) + 7*y^2)*x^2 +
6*x + y^(4/5) over Puiseux series field in y over Rational Field
> P<x> := PolynomialRing(Integers());
> L := ext<ext<pAdicRing(5, 100) | 3> | x^2 + 5>;
> R<x> := PolynomialRing(L);
> f := 3*x^4 + 75*x^3 + 78*x^2 + 10*x + 750;
```

```
> NR := NewtonPolygon(f);
> NR;
Newton Polygon of 3*x^4 + 75*x^3 + 78*x^2 + 10*x + 750 over L
```

Newton Polygons can also be created by specifying the defining points that the polygon must enclose.

```
> N2 := NewtonPolygon({<2, 0>, <0, 3>, <4, 1>});
> N2;
Newton Polygon with defining points {(0, 3), (2, 0), (4, 1)}
> N6 := NewtonPolygon({<1, 4>, <1, 6>, <2, 4>, <3, 1>, <6, 1>, <5, 2>, <4, 5>,
> <4, 7>, <6, 6>, <7, 7>, <2, 7>, <5, 9>, <8, 4>, <8, 6>, <8, 8>, <7, 9>});
> N6;
Newton Polygon with defining points {(1, 4), (1, 6), (2, 4), (2, 7), (3, 1), (4,
5), (4, 7), (5, 2), (5, 9), (6, 1), (6, 6), (7, 7), (7, 9), (8, 4), (8, 6), (8,
8)}
```

These polygons will be referred to in later examples.

46.2.2 Vertices and Faces of Polygons

Both the vertices $\langle a, b\rangle$ and faces $\langle a, b, c\rangle$ (representing $ax + by = c$) of a given polygon N are computed as needed. As seen above, these will be a particular choice of possible faces and vertices determined by the data used to create the polygon. They can be recovered using the `Faces()` and `Vertices()` intrinsics. A different choice of faces and vertices, those faces and vertices of the compact convex hull of the defining points say, can be made using the other intrinsics below.

Recall that P_N denotes the set of points used in the definition of the Newton polygon N whether they arise as the powers of monomials appearing in a polynomial or have been given explicitly as a sequence of pairs.

`Faces(N)`

The sequence of faces $\langle a, b, c\rangle$ (representing $ax + by = c$) of N listed anticlockwise. How this is interpreted in terms of the points used to create N depends on the creation function used (see Section 46.2.1). The faces are listed anticlockwise starting with the face with its left endpoint being the lowest of the leftmost points.

`InnerFaces(N)`

Those faces of the compact convex hull of P_N starting at the lowest of the leftmost points which have strictly negative gradient.

`LowerFaces(N)`

Those faces of the compact convex hull of P_N which bound it below in the y direction.

`OuterFaces(N)`

The union of lower faces which aren't inner faces and the faces which bound the compact convex hull of P_N above in the y-direction (ignoring infinite points).

`AllFaces(N)`

The faces of the compact convex hull of P_N.

Example H46E2

Using some of the polygons defined before the different types of faces are illustrated.

```
> Faces(N);
[ <4, 5, 4> ]
> InnerFaces(N);
[ <4, 5, 4> ]
> OuterFaces(N);
[ <0, 1, 0>, <-1, -4, -4>, <-11, -60, -48> ]
> AllFaces(N);
[ <4, 5, 4>, <0, 1, 0>, <-1, -4, -4>, <-11, -60, -48> ]
> Faces(NR);
[ <4, 1, 6>, <2, 1, 4>, <0, 1, 0> ]
> InnerFaces(NR);
[ <4, 1, 6>, <2, 1, 4> ]
> LowerFaces(NR);
[ <4, 1, 6>, <2, 1, 4>, <0, 1, 0> ]
```

For the polynomial over the Puiseux Field it is no coincidence that `InnerFaces` and `Faces` return the same sequences. Similarly, for the polynomial over the local ring `Faces` is defined to be `LowerFaces`. For both, this is the category of faces that gives the most information for the purposes that the polygon is used. It can also be noted that combining `InnerFaces` and `OuterFaces` will give `AllFaces` with no repetitions (though repetitions will occur if the polygon has only one face and this face is an inner face).

`Vertices(N)`

The sequence of vertices of N. The vertices will be listed anticlockwise from the lowest of the leftmost points.

`InnerVertices(N)`

The sequence of vertices which arise as endpoints of inner faces.

`LowerVertices(N)`

The sequence of vertices which arise as endpoints of lower faces.

`OuterVertices(N)`

The sequence of vertices which arise as endpoints of outer faces.

`AllVertices(N)`

The sequence of vertices of the compact convex hull of P_N.

Example H46E3

This example illustrates the types of vertices that can be calculated. Note that the printing of these polygons, created from their defining points, changes as more information is calculated. This would occur in the same manner if faces were being calculated instead of vertices.

```
> InnerVertices(N2);
[ <0, 3>, <2, 0> ]
> N2;
Newton Polygon with vertices {(0, 3), (2, 0)} and defining points {(0, 3), (2,
0), (4, 1)}
> InnerVertices(N6);
[ <1, 4>, <3, 1> ]
> N6;
Newton Polygon with vertices {(1, 4), (3, 1)} and defining points {(1, 4), (1,
6), (2, 4), (2, 7), (3, 1), (4, 5), (4, 7), (5, 2), (5, 9), (6, 1), (6, 6), (7,
7), (7, 9), (8, 4), (8, 6), (8, 8)}
> Vertices(N2);
[ <0, 3>, <2, 0>, <4, 1> ]
> Vertices(N6);
[ <1, 4>, <3, 1>, <6, 1>, <8, 4>, <8, 8>, <7, 9>, <5, 9>, <2, 7>, <1, 6> ]
> AllVertices(N2);
[ <0, 3>, <2, 0>, <4, 1> ]
> N2;
Newton Polygon with vertices {(0, 3), (2, 0), (4, 1)} and defining points {(0,
3), (2, 0), (4, 1)}
> AllVertices(N6);
[ <1, 4>, <3, 1>, <6, 1>, <8, 4>, <8, 8>, <7, 9>, <5, 9>, <2, 7>, <1, 6> ]
> N6;
Newton Polygon with vertices {(1, 4), (3, 1), (6, 1), (8, 4), (8, 8), (7, 9),
(5, 9), (2, 7), (1, 6)} and defining points {(1, 4), (1, 6), (2, 4), (2, 7), (3,
1), (4, 5), (4, 7), (5, 2), (5, 9), (6, 1), (6, 6), (7, 7), (7, 9), (8, 4), (8,
6), (8, 8)}
```

Here `Vertices` has been defined to be `AllVertices`. All the known vertices of the polygon are printed when the polygon is printed. There is some overlap between the inner and outer vertices as is shown below. Every vertex is either an inner vertex or an outer vertex with some being both. Not all defining points are vertices.

```
> OuterVertices(N6);
[ <3, 1>, <6, 1>, <8, 4>, <8, 8>, <7, 9>, <5, 9>, <2, 7>, <1, 6>, <1, 4> ]
> OuterVertices(N2);
[ <2, 0>, <4, 1>, <0, 3> ]
```

`EndVertices(F)`

A sequence containing the two end vertices of the face $F = \langle a, b, c \rangle$.

`FacesContaining(N,p)`

Those faces of the polygon N returned by `Faces` on which the point $p = \langle a, b \rangle$ lies.

Example H46E4

Using some of the example polygons that have been created above, we illustrate the simple use of `EndVertices` and `FacesContaining`.

```
> AN := AllFaces(N);
> AN;
[ <4, 5, 4>, <0, 1, 0>, <-1, -4, -4>, <-11, -60, -48> ]
> A6 := AllFaces(N6);
> A6;
[ <3, 2, 11>, <0, 1, 1>, <-3, 2, -16>, <-1, 0, -8>, <-1, -1, -16>, <0, -1, -9>,
<2, -3, -17>, <1, -1, -5>, <1, 0, 1> ]
> AllVertices(N);
[ <0, 4/5>, <1, 0>, <4, 0>, <3, 1/4> ]
> AllVertices(N6);
[ <1, 4>, <3, 1>, <6, 1>, <8, 4>, <8, 8>, <7, 9>, <5, 9>, <2, 7>, <1, 6> ]
> EndVertices(AN[1]);
[ <0, 4/5>, <1, 0> ]
> EndVertices(AN[4]);
[ <0, 4/5>, <3, 1/4> ]
> EndVertices(A6[1]);
[ <1, 4>, <3, 1> ]
> EndVertices(A6[5]);
[ <7, 9>, <8, 8> ]
> EndVertices(A6[9]);
[ <1, 4>, <1, 6> ]
> FacesContaining(N, <1, 0>);
[ <4, 5, 4> ]
> FacesContaining(N6, <1, 0>);
[]
> FacesContaining(N6, <4, 1>);
[ <0, 1, 1> ]
> FacesContaining(N, <4, 1>);
[]
> FacesContaining(N6, <3, 1>);
[ <3, 2, 11>, <0, 1, 1> ]
```

`GradientVector(F)`

The a and b values of the line describing the face F of the form $a * x + b * y = c$ where a, b and c are integers.

`GradientVectors(N)`

A sequence containing the gradient vectors of the faces of the newton polygon N.

`Weight(F)`

The c value of the line describing the face F of the form $a * x + b * y = c$ where a, b and c are integers.

`Slopes(N)`

The slopes of the faces of the newton polygon N.

`InnerSlopes(N)`

`LowerSlopes(N)`

`AllSlopes(N)`

The slopes of the polygon N corresponding of `InnerFaces`, `LowerFaces` and `AllFaces` respectively.

Example H46E5

In this example `GradientVector` and `Weight` can be seen to be access functions on the components of a face of a polygon.

```
> A := AllFaces(N);
> A;
[ <4, 5, 4>, <0, 1, 0>, <-1, -4, -4>, <-11, -60, -48> ]
> f := A[3];
> GradientVector(f);
<-1, -4>
> Weight(f);
-4
```

The gradient of the face can now be easily computed as shown.

```
> a := GradientVector(f)[1];
> b := GradientVector(f)[2];
> -a/b;
-1/4
```

46.2.3 Tests for Points and Faces

Once more, recall that P_N denotes the finite set of points in the plane used to define the Newton polygon N. Whether or not a point is considered to lie in a polygon depends on what are considered to be its faces. MAGMA always uses the list of faces returned by `Faces(N)` when testing points. Of course, this is not always the case in applications. One must to perform other tests explicitly when there is doubt.

`IsFace(N, F)`

Return `true` if and only if the tuple $F = \langle a, b, c \rangle$ describes a line coinciding with a face of the polygon N as returned by `Faces`.

`IsVertex(N, p)`

Return `true` if and only if the point $p = \langle a, b \rangle$ of the rational plane (given as a tuple) is a vertex of the polygon N as returned by `Vertices`.

`IsInterior(N,p)`

Return `true` if and only if the point $p = \langle a, b \rangle$ given as a tuple lies strictly in the interior of the polygon N.

`IsBoundary(N, p)`

Return `true` if and only if the point $p = \langle a, b \rangle$ given as a tuple lies on the boundary of the polygon N, that is, the point is contained in a face of N.

`IsPoint(N,p)`

Return `true` if and only if the point $p = \langle a, b \rangle$ (given as a tuple) lies on the polygon N.

46.3 Polynomials Associated with Newton Polygons

The polynomial used to define a polygon can be recovered, but more usefully so can those restrictions of that polynomial to parts of the polygon, the so-called face functions in particular.

Note that most of these functions will return an error if N was not defined in terms of a polynomial.

`HasPolynomial(N)`

Return `true` if and only if the polygon N was defined as the Newton polygon of some polynomial.

`Polynomial(N)`

The polynomial used to define the polygon N.

`ParentRing(N)`

The parent ring of the polynomial of the polygon N.

`IsNewtonPolygonOf(N, f)`

Return whether the newton polygon N is defined by the polynomial f.

`FaceFunction(F)`

If the polygon N is defined by a polynomial in two variables f this returns those monomial terms of f whose corresponding Newton points lie on the face F. On the other hand, if N is determined by a univariate polynomial over a series ring, this returns the univariate polynomial supported on the face F.

`IsDegenerate(F)`

Return `true` if the face function along F is not squarefree.

`IsDegenerate(N)`

Return `true` if a face function on some face of N is degenerate.

46.4 Finding Valuations of Roots of Polynomials from Newton Polygons

Newton polygons can be used to find the valuations of roots of the polynomial from which the polygon was created at the prime used in the creation (given implicitly or explicitly). The following functions use Newton polygons to calculate the valuations of the roots of the polynomial paired with the number of roots with that valuation.

`ValuationsOfRoots(f)`

The valuations of the roots of f, where f is a polynomial over a local ring or a series ring.

`ValuationsOfRoots(f, p)`

The valuations of the roots of f with respect to p where p may be either a prime integer, a prime ideal of a number field or a place of a function field.

46.5 Using Newton Polygons to Find Roots of Polynomials over Series Rings

The operations described in this section are relevant for polynomials over series rings. There are two main algorithms involved.

`SetVerbose("Newton", v)`

Set the verbose printing to level v for `PuiseuxExpansion`, `ExpandToPrecision`, `DuvalPuiseuxExpansion`, `Roots` and `ImplicitFunction`. A level of 1 will mean that any partial solutions that could not be expanded to the precision requested will be printed before an error is returned except for `Roots`. The polynomials used in forming extensions will also be printed before the extension is computed. In `Roots`, the algorithm used to compute the expansions will be printed. When Walker's algorithm is being used the current value of the denominator will be printed. For `ImplicitFunction` a warning about a potentially bad value of d will be printed if the value of d given is not divisible by the exponent denominator of some coefficient of f. A level of 2 will print the last polynomials calculated during the newton polygon part of `PuiseuxExpansion` and `DuvalPuiseuxExpansion` and some evaluated polynomials during `ImplicitFunction`.

46.5.1 Operations not associated with Duval's Algorithm

`PuiseuxExpansion(f, n)`

`PreciseRoot`	BoolElt	*Default :* `false`
`TestSquarefree`	BoolElt	*Default :* `true`
`NoExtensions`	BoolElt	*Default :* `false`
`LowerFaces`	BoolElt	*Default :* `true`
`OneRoot`	BoolElt	*Default :* `false`
Verbose	`Newton`	*Maximum :* 2

This function implements the algorithm described in [Wal78].

Return a sequence of partial expansions of the roots of the polynomial f over a series ring as puiseux series. The roots are returned with relative precision at least n/d where d is the least common multiple of exponent denominator for the series expansion and the exponent denominators of the coefficients of the f. An input of $n = 0$ will return the expansions calculated by the newton polygon part of the algorithm and these will be to the precision of what is known. The coefficient ring of the series ring containing the coefficients of f must always be a field and unless extensions are not required it must be able to be extended.

If the coefficient ring of the series ring is a finite field whose characteristic is less than or equal to the degree of the polynomial then the denominators computed in the newton polygon part of the algorithm may not be bounded and the function will return an error. However, it is possible for some polynomials that the denominators will be bounded. This is stated by [Gri95], pg 269 – 272.

Care needs to be taken with polynomials whose coefficients have low precision. The algorithm must extract from f the squarefree part and in doing so lose even more precision. The result is that the algorithm may not have enough precision to calculate the expansions correctly. A solution is to set `TestSquarefree` to `false` if the polynomial is known to have no multiple roots. However this will not solve all precision problems and the answer is only as good as the precision allows it to be.

If `PreciseRoot` is set to `true` then the partial expansions to be returned are checked and if any are exact roots of f they are returned with full precision. If `NoExtensions` is set to `true` then expansions are found within the puiseux series ring only. By default, the coefficient ring of the series ring is extended to find all the partial expansions. If `LowerFaces` is set to `false` then expansions with negative valuations will not be found. If `OneRoot` is set to true then representatives of conjugate roots only will be found instead of each of the roots individually.

Note that it may useful to define the polynomial over an algebraically closed field (via `AlgebraicClosure`), so that all roots may be found.

`ExpandToPrecision(f, c, n)`

`PreciseRoot`	BoolElt	*Default :* `false`
`TestSquarefree`	BoolElt	*Default :* `true`
Verbose	`Newton`	*Maximum :* 2

Given a polynomial f over a Puiseux series ring and a partial root c of that polynomial (found by `PuiseuxExpansion` for example) continue to expand that root until it has relative precision n/d where d is the least common multiple of the exponent denominator of c and the exponent denominators of the coefficients of f. If c is given to greater precision (or length greater than n if its precision is infinite), the relative precision of c is reduced to n/d. An error results if c is not a partial expansion to precision n/d. If `PreciseRoot` is true then the partial expansion to be returned is checked and if it is an exact root of f it is returned with full precision. All input is checked for being an exact root regardless. If `TestSquarefree` is `false` the polynomial will not be made squarefree. This may avoid some loss of precision but may result in some unique partial roots not being recognized as unique partial roots and as such they cannot be expanded.

An error may result if c is a partial root of f but the exponent denominator of the full expansion is greater than that of c. Therefore for c to be expanded it must have the same denominator as the expansion it is part of. This will rarely be an issue for those partial expansions resulting from `PusieuxExpansion` which did not encounter problems with precision in the newton polygon part since the algorithm for this will find at least as much of the expansion as necessary to compute the exponent denominator.

`ImplicitFunction(f, d, n)`

Verbose	`Newton`	*Maximum :* 2

Return a root of the polynomial f over a series ring. The input d is the denominator (or a multiple of) the exponent denominator of the root. The root is given to absolute precision n/d. The evaluation of f at zero (polynomial) evaluated at zero (series) must be zero but that of its derivative must be nonzero.

Example H46E6

This example illustrates the joint use of `PuiseuxExpansion` and `ExpandToPrecision` which can be used together to gain and improve partial roots of a polynomial. The use of `ExpandToPrecision` following `PuiseuxExpansion` avoids the recalculation of information already known.

```
> P<x> := PuiseuxSeriesRing(Rationals());
> R<y> := PolynomialRing(P);
> f := y^3 + 2*x^-1*y^2 + 1*x^-2*y + 2*x;
> c := PuiseuxExpansion(f, 0);
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> Q<q> := PolynomialRing(A);
> c;
```

```
[
    -2*a^3 + O(a^4),
    -a^-1 + n*a + O(a^2),
    -a^-1 - n*a + O(a^2)
]
> [ExpandToPrecision(f, c[i], 10) : i in [1 .. #c]];
[
    -2*a^3 - 8*a^7 - 56*a^11 + O(a^13),
    -a^-1 + n*a + a^3 + 5/4*n*a^5 + 4*a^7 + O(a^9),
    -a^-1 - n*a + a^3 - 5/4*n*a^5 + 4*a^7 + O(a^9)
]
```

The same results could have been gained using `PuiseuxExpansion` with the required precision in the first place.

```
> c := PuiseuxExpansion(f, 10);
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> c;
[
    -2*a^3 - 8*a^7 - 56*a^11 + O(a^13),
    -a^-1 + n*a + a^3 + 5/4*n*a^5 + 4*a^7 + O(a^9),
    -a^-1 - n*a + a^3 - 5/4*n*a^5 + 4*a^7 + O(a^9)
]
```

However, asking for more precision requires time so that if it is not necessary the extra calculation can be avoided and if more precision happens to be required then it can be gained without recalculation. `ExpandToPrecision` is also called on only one root so that if only one expansion is required using `PusieuxExpansion` and then `ExpandToPrecision` will not calculate any unnecessary information.

```
> time c := PuiseuxExpansion(f, 100);
Time: 2.810
> time c := PuiseuxExpansion(f, 10);
Time: 0.060
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> time ExpandToPrecision(f, c[1], 100);
-2*a^3 - 8*a^7 - 56*a^11 - 480*a^15 - 4576*a^19 - 46592*a^23 -
    496128*a^27 - 5457408*a^31 - 61529600*a^35 - 707266560*a^39 -
    8257566720*a^43 - 97654702080*a^47 - 1167349284864*a^51 -
    14082308833280*a^55 - 171221451538432*a^59 -
    2096081963188224*a^63 - 25814314231136256*a^67 -
    319605795242639360*a^71 - 3975750610806374400*a^75 -
    49666299938073477120*a^79 - 622818862289639178240*a^83 -
    7837247078959687925760*a^87 - 98931046460491133091840*a^91 -
    1252424949872174982758400*a^95 - 158971065678060806587023 36*a^99
    + O(a^103)
```

```
Time: 0.410
```

`IsPartialRoot(f, c)`

Return `true` if the series c can be expanded to at least one root of the polynomial f.

`IsUniquePartialRoot(f, c)`

`TestSquarefree` BoolElt *Default :* `true`

Return `true` if the series c can be expanded to exactly one distinct root of the polynomial f. By default f will have multiple factors removed to allow partial expansions of multiple roots to be recognized as being unique. If `TestSquarefree` is set to `false` then f will be taken as given which may avoid errors due to lost precision but may not pick partial expansions of multiple roots as being unique and as such is best used when f is squarefree or the expansion is known to be of a single root.

Example H46E7

The above 2 functions can be used to reduce the occurrence of errors from `ExpandToPrecision` by checking that the input can be expanded. Errors resulting from a lack of precision which means that the expansion cannot be calculated to the requested precision are the only errors that cannot be removed. Only unique partial roots can be expanded. If a partial root is not unique then calling `PuiseuxExpansion` will provide several further partial expansions of the partial root that will themselves be unique and so can be used to calculate several expansions of the original.

```
> P<x> := PuiseuxSeriesRing(Rationals());
> R<y> := PolynomialRing(P);
> f := (y^2 - x^3)^2 - y*x^6;
> IsPartialRoot(f, x^(3/2));
true
>  ExpandToPrecision(f, x^(3/2), 10);
>> ExpandToPrecision(f, x^(3/2), 10);
                    ^
Runtime error in 'ExpandToPrecision': Element is not a unique partial
root of the polynomial
> IsUniquePartialRoot(f, x^(3/2));
false
> c := PuiseuxExpansion(f, 0);
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> Q<q> := PolynomialRing(A);
> c;
[
    a^(3/2) + 1/2*a^(9/4) + O(a^(5/2)),
    a^(3/2) - 1/2*a^(9/4) + O(a^(5/2)),
    -a^(3/2) + 1/2*n*a^(9/4) + O(a^(5/2)),
```

```
    -a^(3/2) - 1/2*n*a^(9/4) + O(a^(5/2))
]
> IsUniquePartialRoot(f, x^(3/2) + 1/2*x^(9/4));
true
> ExpandToPrecision(f, x^(3/2) + 1/2*x^(9/4), 10);
x^(3/2) + 1/2*x^(9/4) - 1/64*x^(15/4) + O(x^4)
>  ExpandToPrecision(f, x^(3/2) + x^2, 30);
>> ExpandToPrecision(f, x^(3/2) + x^2, 30);
                     ^
Runtime error in 'ExpandToPrecision': Element is not a partial root of
the polynomial
> IsPartialRoot(f, x^(3/2) + x^2);
false
```

So if `IsPartialRoot` returns `false` then no expansion can be made. If `IsUniquePartialRoot` returns `false` (but `IsPartialRoot` returns `true`) then several expansions can be made after calling `PuiseuxExpansion`.

`PuiseuxExponents(p)`

Given a series expansion return the sequence of exponents $[a/b]$ of the non zero terms of the series p up to and including the first one where b is the global denominator for the series.

`PuiseuxExponentsCommon(p, q)`

Given two series return the sequence of exponents $[a/b]$ of the non zero initial terms of the series p and q which are equal up to but not including the first unequal terms.

Example H46E8

This example illustrates how `PuiseuxExponents` and `PuiseuxExponentsCommon` can be used on output from `PusieuxExpansion`. (Similar can be done with related functions and general series).

```
> P<x> := PuiseuxSeriesRing(FiniteField(5, 3));
> R<y> := PolynomialRing(P);
> f := (1+x)*y^4 - x^(-1/3)*y^2 + y + x^(1/2);
> time c := PuiseuxExpansion(f, 5);
Time: 0.030
> c;
[
    4*x^(1/2) + x^(2/3) + 3*x^(5/6) + x^(7/6) + O(x^(4/3)),
    x^(1/3) + x^(1/2) + 4*x^(2/3) + 2*x^(5/6) + O(x^(7/6)),
    4*x^(-1/6) + 2*x^(1/3) + O(x^(2/3)),
    x^(-1/6) + 2*x^(1/3) + O(x^(2/3))
]
> PuiseuxExponents(c[1]);
[ 1/2, 2/3, 5/6 ]
> PuiseuxExponents(c[3]);
```

```
[ -1/6 ]
> P<x> := PuiseuxSeriesRing(FiniteField(5, 3));
> R<y> := PolynomialRing(P);
> f := ((y^2 - x^3)^2 - y*x^6)^2 - y*x^15;
> c := PuiseuxExpansion(f, 0);
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> Q<q> := PolynomialRing(A);
> c;
[
    4*a^(3/2) + 4*a^(9/4) + 4*a^3 + O(a^(13/4)),
    4*a^(3/2) + 4*a^(9/4) + a^3 + O(a^(13/4)),
    4*a^(3/2) + a^(9/4) + 4*a^3 + O(a^(13/4)),
    4*a^(3/2) + a^(9/4) + a^3 + O(a^(13/4)),
    a^(3/2) + 3*a^(9/4) + 4*a^3 + O(a^(13/4)),
    a^(3/2) + 3*a^(9/4) + a^3 + O(a^(13/4)),
    a^(3/2) + 2*a^(9/4) + 4*a^3 + O(a^(13/4)),
    a^(3/2) + 2*a^(9/4) + a^3 + O(a^(13/4))
]
> PuiseuxExponentsCommon(c[1], c[1]);
[ 3/2, 9/4, 3 ]
> PuiseuxExponentsCommon(c[1], c[2]);
[ 3/2, 9/4 ]
> PuiseuxExponentsCommon(c[1], c[3]);
[ 3/2 ]
> PuiseuxExponentsCommon(c[1], c[8]);
[]
```

46.5.2 Operations associated with Duval's algorithm

The following functions have a similar use to those given above but implement a different algorithm, namely that of [Duv89] which is faster and can handle larger degree polynomials. However, it can only be used with polynomials which are essentially over a laurent series ring and the coefficient ring of that laurent series ring has either characteristic zero or characteristic greater than the degrees of the squarefree factors of the polynomial.

`DuvalPuiseuxExpansion(f, n)`

`Version`	MonStgElt	*Default :* "*Rational*"
`TestSquarefree`	BoolElt	*Default :* `true`
`NoExtensions`	BoolElt	*Default :* `false`
`LowerFaces`	BoolElt	*Default :* `true`
`OneRoot`	BoolElt	*Default :* `false`
Verbose	`Newton`	*Maximum :* 2

A sequence of parametrizations of puiseux expansions of roots of f, as puiseux series, where f is a polynomial over a series ring. The expansions will have at least n non zero terms (unless the expansion is finite and has less than n non zero terms), with more than n occurring only if n is less than the number of terms returned by the newton polygon part of the algorithm. The coefficients of f must have exponent denominator 1.

This algorithm is faster than that given by Walker and implemented in `PuiseuxExpansion`, since it doesn't calculate non zero terms explicitly and doesn't make all necessary extensions during the algorithm leaving some to be made when the series is computed from the parametrizations.

If f has coefficients with finite precision then the expansions can only be computed to as many non zero terms as can be known for that expansion. After this limit has been reached, an error results since the next non zero term is not known. If f has roots with finite puiseux expansions then if n is greater than the number of non zero terms in the expansion the expansion is returned with infinite precision.

If `Version` is set to `"Classical"` then the (slower) classical branch of the algorithm will be run which makes all extensions necessary for the computation of the expansions. It is still faster than `PusieuxExpansion` since it does not iterate through and calculate zero terms but will encounter the same problems that `PuiseuxExpansion` does with field extensions over the rationals. The classical version will return as many parametrizations as there are expansions and some of these parametrizations will give the same set of expansions.

If `NoExtensions` is set to `true` then only the expansions which lie in the puiseux series ring corresponding to the coefficient ring of f are calculated. Otherwise, all the expansions of roots of f are calculated regardless of where they lie. `LowerFaces` and `OneRoot` work as for `PuiseuxExpansion`.

This algorithm works with the squarefree part of f only. If any coefficient of f has low precision then this step may make it impossible for any information about the expansions to be gained due to a loss of further precision. A way around this is to set `TestSquarefree` to `false` if the polynomial is known to be squarefree. This may result in some information being returned but such information is only as good as the precision it was allowed.

`ParametrizationToPuiseux(T)`

The series that satisfy the parametrization `T`. These are found by evaluating `T[2]` at t where `T[1]` $= \lambda t^e$.

`PuiseuxToParametrization(S)`

A parametrization of the series S. It is the simplest one which takes the denominator out of S and makes it the exponent of the first entry in the parametrization.

Example H46E9

This example illustrates the use of `DuvalPuiseuxExpansion` and `ParametrizationToPuiseux` to gain the information given by `PuiseuxExpansion` and also compares the performance of the two

algorithms. It also highlights some of the anomalies that may be encountered due to precision concerns.

```
> P<x> := PuiseuxSeriesRing(Rationals());
> R<y> := PolynomialRing(P);
> f := (y^2 - x^3)^2 - y*x^6;
> time D := DuvalPuiseuxExpansion(f, 0);
Time: 0.000
> D;
[
    <16*x^4, 64*x^6 + 256*x^9 + O(x^10)>
]
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.060
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(3/2) + 1/2*a^(9/4) + O(a^(5/2)),
    a^(3/2) - 1/2*a^(9/4) + O(a^(5/2)),
    -a^(3/2) + 1/2*n*a^(9/4) + O(a^(5/2)),
    -a^(3/2) - 1/2*n*a^(9/4) + O(a^(5/2))
]
> time c := PuiseuxExpansion(f, 0);
Time: 0.050
```

Here it can be seen that the newton polygon part of the algorithm is substantially faster using Duval's method, though the converting of the parametrization to a series is not as fast. Asking for more terms shows this more substantially.

```
> time D := DuvalPuiseuxExpansion(f, 10);
Time: 0.020
> D;
[
    <16*x^4, 64*x^6 + 256*x^9 - 512*x^15 + 2048*x^18 - 4608*x^21 + 56320*x^27 -
        294912*x^30 + 792064*x^33 - 12082176*x^39 + 68157440*x^42 + O(x^43)>
]
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.129
> time c := PuiseuxExpansion(f, 10);
Time: 0.100
> A<a> := Parent(c[1]);
> N<n> := CoefficientRing(A);
> c;
[
    a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + O(a^4),
    a^(3/2) - 1/2*a^(9/4) + 1/64*a^(15/4) + O(a^4),
    -a^(3/2) + 1/2*n*a^(9/4) + 1/64*n*a^(15/4) + O(a^4),
    -a^(3/2) - 1/2*n*a^(9/4) - 1/64*n*a^(15/4) + O(a^4)
```

```
]
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + 1/128*a^(9/2) - 9/4096*a^(21/4) +
        55/131072*a^(27/4) - 9/32768*a^(15/2) + 1547/16777216*a^(33/4) -
        11799/536870912*a^(39/4) + 65/4194304*a^(21/2) + O(a^(43/4)),
    a^(3/2) - 1/2*a^(9/4) + 1/64*a^(15/4) + 1/128*a^(9/2) + 9/4096*a^(21/4) -
        55/131072*a^(27/4) - 9/32768*a^(15/2) - 1547/16777216*a^(33/4) +
        11799/536870912*a^(39/4) + 65/4194304*a^(21/2) + O(a^(43/4)),
    -a^(3/2) + 1/2*n*a^(9/4) + 1/64*n*a^(15/4) - 1/128*a^(9/2) -
        9/4096*n*a^(21/4) - 55/131072*n*a^(27/4) + 9/32768*a^(15/2) +
        1547/16777216*n*a^(33/4) + 11799/536870912*n*a^(39/4) -
        65/4194304*a^(21/2) + O(a^(43/4)),
    -a^(3/2) - 1/2*n*a^(9/4) - 1/64*n*a^(15/4) - 1/128*a^(9/2) +
        9/4096*n*a^(21/4) + 55/131072*n*a^(27/4) + 9/32768*a^(15/2) -
        1547/16777216*n*a^(33/4) - 11799/536870912*n*a^(39/4) -
        65/4194304*a^(21/2) + O(a^(43/4))
]
```

It can be seen that the computation of the information is a lot faster using Duval's method. It is only the cosmetic of converting this information into series that could make this algorithm seem slow. But also note that there is much greater information given by Duval's algorithm. The equivalent information is given below.

```
> time D := DuvalPuiseuxExpansion(f, 3);
Time: 0.009
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.049
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + O(a^4),
    a^(3/2) - 1/2*a^(9/4) + 1/64*a^(15/4) + O(a^4),
    -a^(3/2) + 1/2*n*a^(9/4) + 1/64*n*a^(15/4) + O(a^4),
    -a^(3/2) - 1/2*n*a^(9/4) - 1/64*n*a^(15/4) + O(a^4)
]
```

One thing that may be taken for granted from `PuiseuxExpansion` is that all the expansions lie in the same puiseux series ring. However, for `DuvalPuiseuxExpansion` this may not be the case. It will always be true that each of the parametrizations will lie in the same puiseux series ring but series resulting from different parametrizations may not. This occurs since some extensions are left to the stage of calculating the series from the parametrization to be made and for different parametrizations these extensions may be different.

```
> f := (-x^3 + x^4) - 2*x^2*y - x*y^2 + 2*x*y^4 + y^5;
> time D := DuvalPuiseuxExpansion(f, 0);
Time: 0.010
```

```
> D;
[
    <x^2, -x^2 - x^3 + O(x^4)>,
    <x^3, x + O(x^2)>
]
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.000
> P;
[
    -x - x^(3/2) + O(x^2),
    -x + x^(3/2) + O(x^2)
]
> Parent(P[1]);
Puiseux series field in x over Rational Field
> time P := ParametrizationToPuiseux(D[2]);
Time: 0.030
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(1/3) + O(a^(2/3)),
    n*a^(1/3) + O(a^(2/3)),
    (-n - 1)*a^(1/3) + O(a^(2/3))
]
> Parent(P[1]);
Puiseux series field in a over N
> N;
Number Field with defining polynomial $.1^2 + $.1 + 1 over the Rational Field
```

`DuvalPuiseuxExpansion` reacts differently to `PuiseuxExpansion` when given input which has finite expansions either due to finite precision or exact roots. These differences are shown below and are due to the fact that `DuvalPuiseuxExpansion` always looks for the next non zero term in an expansion whereas `PuiseuxExpansion` will calculate zero terms.

```
> f := y - x^3 - x^7 - x^76 + O(x^200);
> D := DuvalPuiseuxExpansion(f, 0);
> D;
[
    <x, x^3 + O(x^4)>
]
> D := DuvalPuiseuxExpansion(f, 3);
> D;
[
    <x, x^3 + x^7 + x^76 + O(x^77)>
]
>  D := DuvalPuiseuxExpansion(f, 4);
>> D := DuvalPuiseuxExpansion(f, 4);
                             ^
Runtime error in 'DuvalPuiseuxExpansion': Insufficient precision to calculate to
```

```
requested precision
> c := PuiseuxExpansion(f, 197);
> c;
[
    x^3 + x^7 + x^76 + O(x^200)
]
>  c := PuiseuxExpansion(f, 200);
>> c := PuiseuxExpansion(f, 200);
                         ^
Runtime error in 'PuiseuxExpansion': Insufficient precision to calculate to
requested precision
> f := y - x^3 - x^7 - x^76;
> D := DuvalPuiseuxExpansion(f, 0);
> D;
[
    <x, x^3 + O(x^4)>
]
> D := DuvalPuiseuxExpansion(f, 3);
> D;
[
    <x, x^3 + x^7 + x^76 + O(x^77)>
]
> D := DuvalPuiseuxExpansion(f, 4);
> D;
[
    <x, x^3 + x^7 + x^76>
]
> c := PuiseuxExpansion(f, 10);
> c;
[
    x^3 + x^7 + O(x^13)
]
> c := PuiseuxExpansion(f, 100);
> c;
[
    x^3 + x^7 + x^76 + O(x^103)
]
> c := PuiseuxExpansion(f, 200);
> c;
[
    x^3 + x^7 + x^76 + O(x^203)
]
> c := PuiseuxExpansion(f, 200 : PreciseRoot := true);
> c;
[
    x^3 + x^7 + x^76
```

```
]
```

The two methods can be combined. Given a series there is no way that Duval's REGULAR algorithm can be used to further the precision of an expansion. But `ExpandToPrecision` can be used to gain the extra precision. Using the REGULAR algorithm would be preferable since it is faster but this is not possible for the type of input that is available. This is explored in the case of our first example.

```
> f := (y^2 - x^3)^2 - y*x^6;
> time D := DuvalPuiseuxExpansion(f, 0);
Time: 0.009
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.050
> A<a> := Parent(P[1]);
> N<n> := CoefficientRing(A);
> P;
[
    a^(3/2) + 1/2*a^(9/4) + O(a^(5/2)),
    a^(3/2) - 1/2*a^(9/4) + O(a^(5/2)),
    -a^(3/2) + 1/2*n*a^(9/4) + O(a^(5/2)),
    -a^(3/2) - 1/2*n*a^(9/4) + O(a^(5/2))
]
> time ExpandToPrecision(f, P[1], 20);
a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + 1/128*a^(9/2) - 9/4096*a^(21/4) +
    O(a^(13/2))
Time: 0.070
> time D := DuvalPuiseuxExpansion(f, 5);
Time: 0.010
> time P := ParametrizationToPuiseux(D[1]);
Time: 0.059
```

It can be seen that using `ExpandToPrecision` is slower even than rerunning Duval's algorithm from the beginning. Even more so when it is remembered that running Duval's algorithm with the extra precision will give parametrizations of all the expansions of the roots of f and not just one expansion. This seems to still be the case when larger examples are considered so that if more terms of an expansion are required it is probably best to start from the beginning asking for these extra terms.

```
> time p1 := ExpandToPrecision(f, P[1], 50);
> A<a> := Parent(p1);
> N<n> := CoefficientRing(A);
> p1;
a^(3/2) + 1/2*a^(9/4) - 1/64*a^(15/4) + 1/128*a^(9/2) - 9/4096*a^(21/4) +
    55/131072*a^(27/4) - 9/32768*a^(15/2) + 1547/16777216*a^(33/4) -
    11799/536870912*a^(39/4) + 65/4194304*a^(21/2) - 189805/34359738368*a^(45/4)
    + 1584999/1099511627776*a^(51/4) - 2261/2147483648*a^(27/2) + O(a^14)
Time: 0.439
> time D := DuvalPuiseuxExpansion(f, 13);
Time: 0.009
> time P := ParametrizationToPuiseux(D[1]);
```

```
Time: 0.200
```

46.5.3 Roots of Polynomials

This section describes two similar functions that can be used for finding roots of polynomials over series rings in a similar way to finding roots of polynomials over any other ring for which roots can be computed in MAGMA.

```
Roots(f)
```

```
Roots(f, n)
```

Verbose `Newton` *Maximum :* 2

Find the roots of the polynomial f which lie in the coefficient ring of f. The first form of this function can be used on any polynomial over any ring for which MAGMA can compute roots. Since precision is an issue in series rings and some roots may have infinite expansions, the second version of this function which is specific to series rings allows a lower bound for the precision to which these roots will be known to be specified. The first computes the roots to at least the default precision of the ring if that ring has infinite precision otherwise it computes them to at least the precision of the ring. The first version will be enough when all the coefficients of the polynomial have infinite precision. The second version may be required when a precision other than that assumed by the first is sought which may be due to the impossibility of computing the roots to such a high precision as the default. The precision is relative to the least common multiple of the exponent denominators of the coefficients of f and the exponent denominator of the root. Roots which are known to be different but are identical to the precision specified will be returned as two distinct roots.

Duval's algorithm as implemented in `DuvalPuiseuxExpansion` will usually be used. Walker's algorithm as implemented in `PuiseuxExpansion` will be used if the polynomial has coefficients involving fractional powers or the characteristic of the coefficient ring of the series ring is less than the degree of a squarefree factor.

If Walker's algorithm is used and the characteristic of the field is less than the degree of the polynomial then the computation may not finish (see remarks under `PuiseuxExpansion`) and control will return to the user when interrupted.

If verbose printing of partial output that doesn't have enough precision is required the functions `PuiseuxExpansion` and `DuvalPuiseuxExpansion` should be used with the appropriate precision. `Roots` also requires that there is enough precision in the roots so that the multiplicities can be calculated correctly and the parts of the roots that are returned are distinct. Therefore an error will be given if there is not enough precision to calculate the part of the root that results from the newton polygon part of the algorithm and the root is not known to be a single root since the multiplicity may not be able to be calculated correctly. Information at such a low precision can be gained correctly by using the `PuiseuxExpansion` functions and determining the multiplicities manually.

`HasRoot(f)`

Return `true` if the polynomial f has a root in its coefficient ring and that root can be found to the fixed or default precision of the ring as applicable. A root is also returned in this case. If f is irreducible over its coefficient ring then return `false`.

Example H46E10

Below are some examples of the use of the `Roots` function.

```
> SetVerbose("Newton", 1);
> P<x> := PuiseuxSeriesRing(Rationals());
> R<y> := PolynomialRing(P);
> f := y^3 + 2*x^-1*y^2 + 1*x^-2*y + 2*x;
> Roots(f);
DUVAL :
[
    <-2*x^3 - 8*x^7 - 56*x^11 - 480*x^15 - 4576*x^19 + O(x^23), 1>
]
> f := f^2;
> Roots(f);
DUVAL :
[
   <-2*x^3 - 8*x^7 - 56*x^11 - 480*x^15 - 4576*x^19 + O(x^23), 2>
]
> f := y^3 + 2*x^-1*y^2 + 1*x^-2*y + 2*x;
> f +:= O(x^20)*(y^3 + y^2 + y + 1);
> f;
(1 + O(x^20))*y^3 + (2*x^-1 + O(x^20))*y^2 + (x^-2 + O(x^20))*y + 2*x + O(x^20)
>  Roots(f);
DUVAL :
>> Roots(f);
        ^
Runtime error in 'Roots': Roots not calculable to default precision
> Roots(f, 10);
DUVAL :
[
   <-2*x^3 - 8*x^7 - 56*x^11 + O(x^13), 1>
]
> f := f^2;
> f;
(1 + O(x^20))*y^6 + (4*x^-1 + O(x^19))*y^5 + (6*x^-2 + O(x^18))*y^4 + (4*x^-3 +
   4*x + O(x^18))*y^3 + (x^-4 + 8 + O(x^18))*y^2 + (4*x^-1 + O(x^18))*y + 4*x^2
   + O(x^21)
> Roots(f, 10);
DUVAL :
[
   <-2*x^3 - 8*x^7 - 56*x^11 + O(x^13), 2>
]
```

```
> f := (y - x^(1/4))*(y - x^(1/3));
> Roots(f);
WALKER :
[
    <x^(1/3) + O(x^2), 1>,
    <x^(1/4) + O(x^(23/12)), 1>
]
```

46.6 Bibliography

[**Duv89**] Dominique Duval. Rational Puiseux Expansions. *Compositio Mathematica*, 70:119 – 154, 1989.

[**Gri95**] Deryn Griffiths. Series Expansions of Algebraic Functions. In W. Bosma and A. van der Poorten, editors, *Computational Algebra and Number Theory*, pages 267 – 277. Kluwer Academic Publishers, Netherlands, 1995.

[**Wal78**] Robert J. Walker. *Algebraic Curves*, pages 98 – 99. Springer-Verlag, 1978.

47 p-ADIC RINGS AND THEIR EXTENSIONS

Chapter 47
p-ADIC RINGS AND THEIR EXTENSIONS

47.1 Introduction

MAGMA supports finite extensions of the ring $\mathbf{Z}_p$ of p-adic integers or the field $\mathbf{Q}_p$ of p-adic numbers. Within this chapter, we mean these objects if we refer to local rings or fields. Section 47.2 provides more background information on the theory behind the p-adics.

MAGMA has two different models for working with these locals: fixed precision rings (`RngPadRes` and `RngPadResExt`, with element types `RngPadResElt` and `RngPadResExtElt`) and free precision rings (`RngPad` and `FldPad`, with element types `RngPadElt` and `FldPadElt`). The merits of each model are discussed in Section 47.3.1.

MAGMA also contains a type of local field where extensions can be made by any irreducible polynomial. For more information on these local fields, see Chapter 51.

47.2 Background

The p-adic field $\mathbf{Q}_p$ arises naturally as the completion of $\mathbf{Q}$ with respect to an absolute value function $|x|_p = p^{-v_p(x)}$, where $v_p(x)$ is the p-adic *valuation* of x (that is, a power of p such that $x = p^{v_p(x)}\frac{a}{b}$ but $p \not| ab$). The *ring of integers* of $\mathbf{Q}_p$, denoted $\mathbf{Z}_p$, is the set of all elements of non-negative valuation. The ring $\mathbf{Z}_p$ has a unique maximal ideal, generated by the prime p; the *residue class field* K is the quotient $\mathbf{Z}_p/p\mathbf{Z}_p$. Any element x of $\mathbf{Q}_p$ can be expressed as a power series in the prime p, so that $x = \sum_{i=v}^{\infty} a_i p^i$, where v is the valuation of x, a_v is non-zero, and each a_i is a lift of an element from the residue class field. In more general terms, $\mathbf{Q}_p$ is a local field, with its ring of integers $\mathbf{Z}_p$ being a local ring. A uniformizer π of $\mathbf{Q}_p$ is the prime p.

More generally, consider an irreducible polynomial over some local field L_1 (such as $\mathbf{Q}_p$). Then the extension given by adjoining a root α of this polynomial to L_1, $L_2 = L_1[\alpha]$, is also a local field. Let π_1 and π_2 be uniformizers of L_1 and L_2, respectively. Then $\pi_2^e = \pi_1 u$, where u is a unit of L_2. The number e is the *ramification degree* of L_2 over L_1, and divides the degree n of the extension. If $e = n$, we say L_2 is *totally ramified* over L_1; if $e = 1$, we say L_2 is *unramified* over L_1. The degree of the residue class field K_2 of L_2 over the residue class field K_1 of L_1 is $f = \frac{n}{e}$. Finite extensions in MAGMA must be either *unramified* or *totally ramified*; MAGMA also allows towers of extensions to be built.

It is well known that up to isomorphism there is only one degree n unramified extension of L, which can be obtained by adjoining a $p^n - 1$-th root of unity ζ to L. This extension is Galois, with Galois group isomorphic to the cyclic group of order n. The Galois group is generated by the Frobenius automorphism σ, which takes ζ to ζ^p. For some applications, it is necessary to have a fast Frobenius action, so MAGMA supports such a representation. However, the defining polynomial in this representation is particularly dense and can be expensive to construct, hence it is ill-suited for general applications. An unramified extension can only be defined by *inertial* polynomials, which are polynomials that are irreducible

over the residue class field. MAGMA allows an unramified extension to be defined by any inertial polynomial.

All totally ramified extensions contain a root of an Eisenstein polynomial of L. An Eisenstein polynomial $f(x) = \sum_{i=0}^{n} a_i x^i$ satisfies $v_\pi(a_n) = 0$, $v_\pi(a_i) \geq 1$ for all $0 < i < n$, and $v_\pi(a_0) = 1$. MAGMA allows a totally ramified extension to be defined by any Eisenstein polynomial; note that this does not allow arbitrary representations of totally ramified extensions to be constructed.

47.3 Overview of the p-adics in MAGMA

47.3.1 p-adic Rings

Since p-adic rings are completions (like the real numbers), it is difficult to represent their elements in an exact form. As described in Section 47.2, any element x of a local ring L can be expressed as a power series in the uniformizing element of L. Most problems can be solved by computing in a finite quotient of L, which is equivalent to truncating the infinite expansion of x at some point. More precisely, we work in the quotient rings $L/\pi^k L$ for non-negative k, which is called the *precision* of the ring L.

Unfortunately, working with finite approximations to elements does have problems. Some operations on these approximations will yield results with reduced precision. For example, in a ring constructed with precision k the quotient of two elements with valuations $v_1 \geq v_2$ lies in the local ring, but can only be determined up to $k - v_2$ digits. Also, it is only possible in general to construct an approximation to the GCD of two polynomials f and g over a local ring.

MAGMA offers two models for computing in a local ring L. The first model (the *fixed precision model*) allows the user to work with the quotient rings defined above. These quotient rings are finite structures, and their elements can easily be represented exactly; hence, many exact algorithms can be applied to them. On the other hand, precision management is left to the user, since operations which may lose precision, such as division, will still return results to the full precision of the ring. In the second model (the *free precision model*), the user works more directly with L, instead of a finite quotient of L. Each element is still represented by a finite approximation, however, these approximations can be of varying precisions, thus freeing the user from precision management. However, these rings are inexact (for instance, there is no zero element of the ring, only approximations to it), and hence many algorithms in MAGMA which are designed for exact rings may cause significant precision loss. The situation is analogous to the support for real numbers in MAGMA.

47.3.2 p-adic Fields

A local field in MAGMA is the field of fractions of a local ring. Representing a local field on a machine causes some trouble, since there is no algebraic structure in which truncations of the elements could be interpreted (as in the case of the finite quotients for the local rings). Moreover, all structural information of a local field is already contained in its ring

of integers, so that the main motivation for supporting local fields is to provide some basic arithmetic.

An element x of a local field L is stored internally as $x = \pi^v u + O(\pi^{v+k})$, where v is the (possibly negative) valuation of x, and u is a unit known to precision k in the ring of integers of L. It can happen that operations in a field yield elements with no known digits at all, since the unit parts may not overlap.

47.3.3 Free Precision Rings and Fields

Free precision rings and fields can be created with bounded or unbounded precision. If they are created with some bounded precision k, then no element in the structure can have precision greater than k. Thus, bounded free precision rings are "inexact" versions of the fixed precision rings. Unbounded precision structures can have elements with arbitrarily large, but not infinite, precision. In general, it is recommended that free precision structures are utilized unless speed is absolutely critical.

47.3.4 Precision of Extensions

In MAGMA, all measurements of precision are made with respect to a valuation function v_π which takes the uniformizer π of the local ring L to unity (throughout this chapter, the subscript will be dropped on v_π where there is no ambiguity). This can have surprising results. For instance, consider a local ring L_1 which extends L_2, such that the ramification degree of L_1 over L_2 is e. Let π_1 and π_2 be the uniformizing elements of L_1 and L_2, respectively. Then, $v_{\pi_1}(\pi_1) = 1$, and $v_{\pi_1}(\pi_2) = e$. Hence, an element of precision k in L_2 will have precision ek in when mapped into L_1. Thus, it is important to always remember in which ring precision is determined.

47.4 Creation of Local Rings and Fields

A local ring in MAGMA can be constructed in two ways: as either a p-adic ring, or as an extension of another local ring. MAGMA supports the construction of towers of extensions of local rings; the only restriction is that each extension must be either unramified or totally ramified. As discussed in Section 47.2, MAGMA requires that the defining polynomial is either inertial or Eisenstein.

Additionally, the user must specify whether to construct a fixed precision or free precision structure, and, if necessary, assign a precision to the structure. For local rings the precision is interpreted as an absolute precision, specifying to what precision the element is known, but for local fields it is interpreted as a relative precision, specifying to what precision the unit part of the element is known.

47.4.1 Creation Functions for the p-adics

`pAdicRing(p, k)`

`pAdicField(p, k)`

Given a prime integer p and non-negative single-precision integer k, construct the bounded free precision ring (field) of p-adic integers with maximum precision k.

`pAdicRing(p)`

`pAdicField(p)`

`Precision` RngIntElt *Default :* 20

Given a prime integer p, construct the unbounded free precision ring (field) of p-adic numbers. The optional parameter `Precision`, which must be a non-negative single precision integer, controls the default precision to which elements are created, e.g., when coercing precise elements such as integers or rationals into the ring.

`pAdicQuotientRing(p, k)`

Given a prime integer p and non-negative single precision integer k, construct the fixed precision quotient ring $\mathbf{Z}_p/p^k\mathbf{Z}_p$.

`quo< L | x >`

Given a local ring L, construct the quotient ring L/xL, where x is an element of L.

Example H47E1

The creation of p-adic rings using the above functions is illustrated below.

```
> R := pAdicRing(5);
> R;
5-adic ring
> R`DefaultPrecision;
20
> R!1;
1 + O(5^20)
> R := pAdicRing(5 : Precision := 20);
> R!1;
1 + O(5^20)
> Q := quo<R | 5^20>;
> Q;
Quotient of the 5-adic ring modulo the ideal generated by 5^20
> Q!1;
1
> Q eq pAdicQuotientRing(5, 20);
true
```

47.4.2 Creation Functions for Unramified Extensions

`UnramifiedExtension(L, n)`

`ext< L | n >`

`Cyclotomic`	BoolElt	*Default :* `false`
`GNBType`	RngIntElt	*Default :* 0

Given a local ring or field L and a positive single precision integer n, construct the default unramified extension of L of degree n. If K is the residue class field of L, then the defining polynomial of the default degree n extension of K is lifted to be an inertial polynomial of L; this polynomial is used as the defining polynomial of the extension. If `Cyclotomic` is `true`, then the lift of the defining polynomial will be such that $p^n - 1$-st root of unity will be adjoined to L (this representation makes computation of the Frobenius automorphism particularly efficient). The angle bracket notation can be used to assign a name to the generator of the extension, e.g. `K<t> := UnramifiedExtension(L, n)`. If `Cyclotomic` is `false` but `GNBType` is $t > 0$ then a Gaussian normal basis of type t is used. This allows extremely fast multiple Frobenius computations but multiplication is slower than the usual or `Cyclotomic` representation if $t > 2$. Because of this, currently only 1 or 2 are legal values for t. Only certain extension degrees will have a Gaussian normal basis of type 1 or 2. To determine if this is true, the `HasGNB` functions described below may be used.

`UnramifiedQuotientRing(K, k)`

Given a finite field K and a non-negative single precision integer k, construct the fixed precision quotient ring which has residue class field K and precision k. The angle bracket notation can be used to assign a name to the generator of the extension, e.g. `L<t> := UnramifiedExtension(K, f)`.

`UnramifiedExtension(L, f)`

`ext< L | f >`

Given a local ring or field L and a polynomial f with coefficients coercible to L, construct the unramified extension of L defined by f. The polynomial f must be an inertial polynomial over L. The angle bracket notation can be used to assign a name to the generator of the extension, e.g. `K<t> := UnramifiedExtension(L, f)`. Free precision rings can only be extended by a polynomial if they are of bounded precision, in which case f must be specified to the maximum precision of the ring.

`IsInertial(f)`

Given a polynomial f with coefficients over a local ring or field L, return `true` if and only if f is an inertial polynomial. A polynomial is inertial over L if it is irreducible over the residue class field of L.

`HasGNB(R, n, t)`

Given a local ring or field, returns `true` iff the unramified extension of degree n can be generated by a Gaussian Normal Basis (GNB) of Type t. A GNB allows particularly fast multiple Frobenius and Norm computations. Multiplication will tend to be slower though, unless $t = 1$.

`CyclotomicUnramifiedExtension(R, f)`

`CyclotomicUnramifiedExtension(R, f)`

`CyclotomicUnramifiedExtension(R, f)`

`CyclotomicUnramifiedExtension(R, f)`

Given a local ring of field R, construct the unramified degree f extension by adjoining a $p^f - 1$-th root of unity to R. Functionally equivalent to calling `UnramifiedExtension(R, f:Cyclotomic := true)`.

Example H47E2

The creation of unramified extensions of local rings using the above functions is illustrated below.

```
> R1 := pAdicRing(2, 20);
> R2 := ext<R1 | 5>;
> R2;
Unramified extension defined by the polynomial x^5 + x^2 + 1
 over 2-adic ring mod 2^20
> DefiningPolynomial(R2);
x^5 + x^2 + 1
> R3 := ext<R1 | 5 : Cyclotomic>;
> R3;
Cyclotomic unramified extension of degree 5 over 2-adic ring mod 2^20
> DefiningPolynomial(R3);
x^5 + 426248*x^4 - 14172*x^3 - 147105*x^2 + 293314*x - 1
> R3.1^(2^5-1);
1
> P1<x> := PolynomialRing(R1);
> f1 := x^3 + 3*x + 1;
> IsInertial(f1);
true
> R4 := ext<R1 | f1>;
> R4;
Unramified extension defined by the polynomial x^3 + 3*x + 1
 over 2-adic ring mod 2^20
> P2<y> := PolynomialRing(R2);
> f2 := y^3 + 3*y + 1;
> IsInertial(f2);
true
> ext<R2 | f2>;
Unramified extension defined by the polynomial x^3 + 3*x + 1
```

```
over Unramified extension defined by the polynomial x^5 + x^2 + 1
over 2-adic ring mod 2^20
```

47.4.3 Creation Functions for Totally Ramified Extensions

`TotallyRamifiedExtension(L, f)`

`ext< L | f >`

Given a local ring or field L and a polynomial f with coefficients coercible to L, construct the totally ramified extension of L defined by f. The polynomial f must be an Eisenstein polynomial, that is, the leading coefficient is a unit, the constant coefficient has valuation 1 and all other coefficients have valuation greater than or equal to 1. The angle bracket notation can be used to assign a name to the generator of the extension, e.g. `K<t> := TotallyRamifiedExtension(L, f)`. Free precision rings can only be extended by a polynomial if they are of bounded precision, in which case f must be specified to the maximum precision of the ring.

`IsEisenstein(f)`

Given a polynomial f with coefficients over a local ring or field L, return `true` if and only if f is an Eisenstein polynomial over L. An Eisenstein polynomial satisfies the following properties: the leading coefficient is a unit, the constant coefficient has valuation 1 and all other coefficients have valuation greater than or equal to 1.

Example H47E3

The creation of totally ramified extensions of local rings using the above functions is illustrated below.

```
> L1<a> := ext<pAdicRing(5, 20) | 4>;
> L1;
Unramified extension defined by the polynomial x^4 + 4*x^2 + 4*x + 2
 over 5-adic ring mod 5^20
> L2<b> := ext<L1 | x^4 + 125*x^2 + 5>;
> L2;
Totally ramified extension defined by the polynomial x^4 + 125*x^2 + 5
 over Unramified extension defined by the polynomial x^4 + 4*x^2 + 4*x + 2
 over 5-adic ring mod 5^20
> P<y> := PolynomialRing(L2);
> L3<c> := TotallyRamifiedExtension(L2, y^3 + b^4*a^5*y + b*a^2);
> L3;
Totally ramified extension defined by the polynomial x^3 + ((500*a^3 + 500*a^2 +
    250*a)*b^2 + 20*a^3 + 20*a^2 + 10*a)*x + a^2*b
 over Totally ramified extension defined by the polynomial x^4 + 125*x^2 + 5
 over Unramified extension defined by the polynomial x^4 + 4*x^2 + 4*x + 2
```

```
 over 5-adic ring mod 5^20
```

If the precision of the base ring is only 1, then it is not possible to construct a ramified extension, as there is not enough precision to allow the constant coefficient to be non-zero to that precision.

```
> R<x> := PolynomialRing(Integers());
> L<a> := UnramifiedExtension(pAdicRing(5, 1), 3);
>  TotallyRamifiedExtension(L, x^4 + 5);
>> TotallyRamifiedExtension(L, x^4 + 5);
                           ^
Runtime error in 'TotallyRamifiedExtension': Polynomial must be Eisenstein
> L<a> := UnramifiedExtension(pAdicRing(5, 2), x^5 + x^2 + 2);
> TotallyRamifiedExtension(L, x^4 + 5);
Totally ramified extension defined by the polynomial x^4 + 5 over Unramified
extension defined by the polynomial x^5 + x^2 + 2 over 5-adic ring mod 5^2
> ext<L | x^4 + 125*x^2 + 5>;
Totally ramified extension defined by the polynomial x^4 + 5 over Unramified
extension defined by the polynomial x^5 + x^2 + 2 over 5-adic ring mod 5^2
```

47.4.4 Creation Functions for Unbounded Precision Extensions

Suppose we have an unbounded precision local ring or field L, and we wish to create a finite extension of it. If we need the default degree n unramified extension, then we can use the construction functions defined in Section 47.4.2 to construct this extension. However, suppose we wish to define the extension by some polynomial f. As there is no upper bound on the precision of elements of L, it is impossible for us to represent the polynomial f sufficiently precisely, and hence we cannot use the creation functions defined in previous sections for this task. To allow such extensions to be created, MAGMA allows extensions to be defined by a map $\phi : \mathbf{Z}_{\geq 0} \to R[x]$, where R is a ring whose elements are coercible to the quotient rings $L/\pi^k L$ for all $k \in \mathbf{Z}_{\geq 0}$. The map ϕ, given an input precision k, returns the defining polynomial of the extension to precision k. Internally, whenever MAGMA needs to represent an element of the extension to some precision, it will use ϕ to compute the defining polynomial up to this precision. MAGMA may call ϕ on any precision between zero and the precision of the most precise element created by the user.

`ext< L | m >`

Given a free precision local ring or field L and a map m with domain $\mathbf{Z}$ and codomain $R[x]$, where elements of R are coercible to the quotient rings $L/\pi^k L$ for all $k \in \mathbf{Z}_{\geq 0}$, construct an extension of L defined by m. Given a non-negative single precision integer k, the map m must return the defining polynomial of the extension to precision k, as a polynomial over R. The map m's behaviour for other input values is undefined. Internally, MAGMA will coerce the value returned by the map m to be a polynomial over $L/\pi^k L$. Examples of suitable codomains R include the integers, rationals, or L itself.

Example H47E4

The creation of extensions of local rings using maps is illustrated below. We show how it is possible to define an extension of a free precision ring using an "exact" polynomial.

```
> R := pAdicRing(2);
> Z := Integers();
> P<x> := PolynomialRing(Z);
> m := map<Z -> P | k :-> x^3 + x + 1>;
> R2 := ext<R | m>;
> R2;
Unramified extension defined by a map over 2-adic ring
> DefiningPolynomial(R2);
(1 + O(2^20))*$.1^3 + O(2^20)*$.1^2 + (1 + O(2^20))*$.1 + 1 + O(2^20)
> R2`DefaultPrecision := 1000;
> DefiningPolynomial(R2);
(1 + O(2^1000))*$.1^3 + O(2^1000)*$.1^2 + (1 + O(2^1000))*$.1 + 1 + O(2^1000)
```

47.4.5 Miscellaneous Creation Functions

`IntegerRing(F)`

`Integers(F)`

`RingOfIntegers(F)`

Given a local field F, construct the ring of integers R of F. The ring R is the set of elements of F of non-negative valuation.

`RingOfIntegers(R)`

Given a ring R, this function simply returns it, it is provided to support generic functionality for finite extensions of rings and fields.

`FieldOfFractions(R)`

Given a local ring R, construct the field of fractions F of R. The relative precision of F is equal to the precision of R.

`SplittingField(f, R)`

Given a polynomial f over the integers and a p-adic ring R, compute an extension S over R such that f splits into linear factors over S. The algorithms uses the R4-methods as developed by Pauli ([Pau01b]).

`AbsoluteTotallyRamifiedExtension(R)`

Given a tower of ramified extensions over some unramified ring S, compute a more efficient representation of R, ie. an extension of S that is totally ramified and defined by a single Eisenstein polynomial. The map returned allows to convert between the new and old representations.

47.4.6 Other Elementary Constructions

`Composite(R, S)`

For two p-adic fields that are normal over $\mathbf{Q}_p$, compute the compositum of R and S, ie. the smallest field containing both R and S.

47.4.7 Attributes of Local Rings and Fields

There is only one attribute for local rings and fields that is accessible to the user.

`L‘DefaultPrecision`

Used to retrieve or set the default precision of the local ring or field L. This attribute is only relevant if L is an unbounded free precision ring, in which case this will change the precision with which elements are created by default. For bounded precision structures, the default precision of the ring is equal to the upper bound on precision; attempting to set this attribute will result in an error in this case.

47.5 Elementary Invariants

These functions return some simple information partially defining a local ring.

`Prime(L)`

Given a local ring or field L, return the prime p defining the p-adic ring or field underlying L. This is also the characteristic of the residue class field of L.

`InertiaDegree(L)`

Return the inertia degree of the local ring or field L over its coefficient ring.

`InertiaDegree(K, L)`

Return the inertia degree of the local ring or field K relative to its subring L.

`AbsoluteInertiaDegree(L)`

Return the inertia degree of the local ring or field L over the p-adic ring.

`RamificationDegree(L)`

`RamificationIndex(L)`

Return the ramification degree of the local ring or field L over its coefficient ring.

`RamificationDegree(K, L)`

`RamificationIndex(K, L)`

Return the ramification degree of the local ring or field K relative to its subring L.

`AbsoluteRamificationDegree(L)`

`AbsoluteRamificationIndex(L)`

Return the ramification degree of the local ring or field L over the p-adic ring.

`AbsoluteDegree(L)`

The degree of L over $\mathbf{Z}_p$.

`Degree(L)`

Return the degree of the local ring or field L over its coefficient ring.

`Degree(K, L)`

Return the degree of the local ring or field K relative to its subring L.

`DefiningPolynomial(L)`

Return the minimal polynomial of the generator of L over its coefficient ring. If L is p-adic, the polynomial $x - 1$ is returned. For free precision rings and fields, the coefficients of the defining polynomial are given to the default precision of L.

`DefiningMap(L)`

Given a free precision local ring or field L, return the map that was used to define the extension (see Section 47.4.4 for information on defining extension by maps). If a map was not used, then an error is raised.

`HasDefiningMap(L)`

Given a free precision local ring or field L, return `true` if L is defined by a map; if so, the defining map is also returned.

`PrimeRing(L)`

`PrimeField(L)`

`pAdicRing(L)`

`pAdicField(L)`

Given a local ring or field L, return the p-adic ring or field which is a subring of L.

`BaseRing(L)`

`CoefficientRing(L)`

`BaseField(L)`

`CoefficientField(L)`

`BaseRing(L)`

Given a local ring or field L, return the base ring of L.

`ResidueClassField(L)`

Given a local ring or field L, return the residue class field K of L, and a map from L to K.

`ResidueSystem(R)`

Given a p-adic ring or field R, compute a set of representatives of the residue class field of R as elements of R.

`UniformizingElement(L)`

Given a local ring or field L, return the uniformizing element of L.

`L . 1`

Given a local ring or field L, return an element α of L such that if K is L's base ring or field, then the powers of α give a basis of L as a vector space over K.

`Precision(L)`

Given a local ring or field L, return the precision with which L has been created. If L is a local ring this is the maximum absolute precision to which its elements can be created. If L is a local field this is the maximum relative precision to which its elements can be created. If L is an unbounded free precision ring or field, then infinity is returned.

`HasPRoot(R)`

Given a local ring R extending $\mathbf{Z}_p$ for some prime p, decide if R contains a primitive p-th root of unity.

`HasRootOfUnity(L, n)`

Given a local ring L and some positive integer n, decide if L contains a primitive nth root of unity.

`Discriminant(R)`

Compute the discriminant of the local ring R over its coefficient ring. Since R is defined by either an inertial polynomial or an Eisenstein one, this is equivalent to computing the discriminant of the defining polynomial.

`Discriminant(K, k)`

Given p-adic rings K/k, compute the discriminant of K as an extension of k.

`AdditiveGroup(R)`

The additive group of the p-adic quotient R as an abelian group and the isomorphism from this group back to R.

Example H47E5

We illustrate the functions in this section for rings. Similar constructions can be used for fields.

```
> Zp := pAdicRing(5, 20);
> I<a> := UnramifiedExtension(Zp, 3);
> R<x> := PolynomialRing(I);
> L<b> := ext<I | x^3 + 5*a*x^2 + 5>;
> Prime(L);
5
> InertiaDegree(L);
1
```

The inertia degree of L is returned as 1 because L has been defined as a totally ramified extension of I. However, the inertia degree of L over Z_p is 3, because I itself is an unramified extension of Z_p.

```
> InertiaDegree(L, Zp);
3
> Degree(L);
3
> Degree(L, Zp);
9
> DefiningPolynomial(L);
x^3 + 5*a*x^2 + 5
> P<y> := PolynomialRing(Zp);
> DefiningPolynomial(I);
y^3 + 3*y + 3
> BaseRing(L);
Unramified extension defined by the polynomial x^3 + 3*x + 3
 over 5-adic ring mod 5^20
> PrimeRing(L);
5-adic ring mod 5^20
> PrimeRing(I);
5-adic ring mod 5^20
> ResidueClassField(L);
Finite field of size 5^3
Mapping from: RngPad: L to GF(5^3)
> ResidueClassField(I);
Finite field of size 5^3
Mapping from: RngPad: I to GF(5^3)
```

Here, we see that the residue class fields of I and L are identical. This is due to the fact that L is a totally ramified extension of I.

```
> UniformizingElement(L);
b
> Precision(L);
60
> Precision(I);
20
```

```
> R<a> := ext<pAdicRing(2) | 2>;
> DefiningPolynomial(R);
(1 + O(2^20))*$.1^2 + (1 + O(2^20))*$.1 + 1 + O(2^20)
> Precision(R);
Infinity
```

47.6 Operations on Structures

`AssignNames(~L, S)`

`AssignNames(~L, S)`

Assign a name to the generator of L. The sequence must have only one element, which must be a string. This element is assigned to be the name of the generator when L is considered as a linear associative algebra over its base ring.

`Characteristic(L)`

`Characteristic(L)`

`Characteristic(L)`

The characteristic of the local ring or field L.

`#L`

The number of elements in the local ring or field L. The cardinality is finite only if L is a quotient ring or a bounded free precision ring.

Iterating over the elements of a local ring is possible if it is bounded, but it will take time in proportion to the cardinality of L. It is recommended only in the case of "small" local rings (i.e., rings for which the precision is be very small).

`Name(L, k)`

Given a local ring or field L and an integer k, return the generator of L if k is 1; otherwise, raise an error.

`ChangePrecision(L, k)`
`ChangePrecision(~L, k)`
`ChangePrecision(L, k)`
`ChangePrecision(~L, k)`
`ChangePrecision(L, k)`
`ChangePrecision(~L, k)`
`ChangePrecision(L, k)`
`ChangePrecision(~L, k)`
`ChangePrecision(L, k)`
`ChangePrecision(~L, k)`
`ChangePrecision(L, k)`
`ChangePrecision(~L, k)`

Given a local ring or field L and a non-negative single precision integer k, change the maximum precision with which elements can be created to be k. Depending on how L and its subrings have been constructed, there may be an upper bound (possibly infinite for free structures) on the precision to which L can be changed. For instance, the precision to which a defining polynomial has been given places a bound on the precision of the extension — no defining polynomial can be expanded beyond the precision with which it was originally specified.

`L eq K`

Given local rings or fields L and K, return whether or not L and K are the same object.

`L ne K`

Given local rings or fields L and K, return whether or not L and K are different objects.

Example H47E6

```
> Zp := pAdicRing(5, 20);
> I<a> := UnramifiedExtension(Zp, 3);
> R<x> := PolynomialRing(I);
> L<b> := ext<I | x^3 + 5*a*x^2 + 5>;
> ChangePrecision(Zp, Infinity());
5-adic ring
> L;
Totally ramified extension defined by the polynomial x^3 + 5*a*x^2 + 5
 over Unramified extension defined by the polynomial x^3 + 3*x + 3
 over 5-adic ring mod 5^20
```

```
> ChangePrecision(~L, 50);
> L;
Totally ramified extension defined by the polynomial x^3 + 5*$.1*x^2 + 5
 over Unramified extension defined by the polynomial x^3 + 3*x + 3
 over 5-adic ring mod 5^17
> #L;
875811540203010669327330989556197582050l\
63713672822357348167677431119426678662870\
59291488677263259887695312 5
> AssignNames(~L, ["t"]);
> L.1;
t
> b;
b
> L eq ChangePrecision(L, 10);
false
```

Note that b is an element of the original ring L with precision 60 which is why it retains its print name.

47.6.1 Ramification Predicates

IsRamified(R)

IsUnramified(R)

IsTotallyRamified(R)

Return whether the local ring or field extension R is ramified, unramified or totally ramified.

IsTamelyRamified(R)

IsWildlyRamified(R)

Return whether the local ring or field extension R is tamely ramified (the prime does not divide the ramification degee) or wildly ramified (the prime does divide the ramification degree).

47.7 Element Constructions and Conversions

Local ring elements are implemented using a balanced mod representation. This allows small negative elements x to be represented as x rather than $p^k - x$ where k is a p-adic precision.

47.7.1 Constructions

To simplify the creation of elements in a local ring, various coercions are provided. The most obvious is to regard the integer ring or rational field as embedded in the p-adic ring or field. But there is a range of coercions available, including that of elements from the residue class field.

To create an element of a local field not lying in the local ring, constructors are provided that create an element coercible into the ring, and which increase or decrease this element's valuation in the field.

`Zero(L)`

Given a local ring or field L, create the additive identity of L. Note that if L is an unbounded precision structure, this will only be the zero element to the default precision of the ring, and hence only an approximation to the additive identity of L.

`One(L)`

Given a local ring or field L, create the multiplicative identity of L. Note that if L is an unbounded precision structure, this will only be the one element to the default precision of the ring, and hence only an approximation to the multiplicative identity of L.

`Random(L)`

Given a local ring or field L, return a random element of L, which must be a quotient ring or bounded precision ring. The element will have the default precision of the ring.

`Representative(L)`

Return an element of the local ring or field L.

`elt< L | u >`

`L ! u`

Coerce the object u into the local ring or field L. The resulting element will have as much precision as possible. The element u is allowed to be one of the following:

(i) An integer.

(ii) An element of $\mathbf{Z}/p^m\mathbf{Z}$; (where m is a precision).

(iii) An element of the residue class field of L.

(iv) An element of a local ring or field with something in common with L.

(v) A rational number. If L is a ring then u must not have valuation in the denominator.

(vi) An element of a valuation ring over the rationals with the same prime as L.

(vii) A sequence s. In this case, the sequence s is coerced to a sequence t over the base ring or field of L. This sequence is coerced to $\sum_{i=1}^{\#t} t[i]L.1^{i-1}$.

`elt< L | u, r >`

Create an element of the local ring or field L by coercing u into L and returning with it precision r.

`elt< L | v, u, r >`

Create an element of the local ring or field L by coercing u into L, multiplying by the v-th power of the uniformizing element and returning it with precision r.

`BigO(x)`

`O(x)`

For an element x of a local ring or field L of valuation v, create an element of valuation v and relative precision 0. For rings this is the zero element in the quotient $L/\pi^v L$.

`UniformizingElement(L)`

Given a local ring or field L, return the uniformizing element of L to the default precision of L.

Example H47E7

Here we illustrate the usage of element constructors for local fields and imprecise zeros.

```
> Zp := pAdicRing(5, 20);
> I<a> := UnramifiedExtension(Zp, 3);
> R<x> := PolynomialRing(I);
> L<b> := ext<I | x^3 + 5*a*x^2 + 5>;
> K<pi> := ext<BaseField(FieldOfFractions(L)) | x^2 + 5>;
> K;
Totally ramified extension defined by the polynomial x^2 + 5
 over Unramified extension defined by the polynomial x^3 + 3*x + 3
 over 5-adic field mod 5^20
> elt<K | 64>;
64 + O(pi^40)
> P := PrimeField(K);
> elt<K | 2, 3/4, 6>;
pi^2*7 + O(pi^6)
> 4*$1;
pi^2*3 + O(pi^6)
> K!3/5;
pi^-2*3 + O(pi^38)
> O(K!40^200);
O(pi^400)
```

```
> Precision($1);
0
> O(K!0);
O(pi^40)
> O(K!1);
O(1)
> R<x> := PolynomialRing(Integers());
> K<pi> := ext<pAdicField(5, 100) | x^2 + 5>;
> elt<K | 64>;
64 + O(pi^200)
> Precision($1);
200
> K!3/10;
-pi^-2*39443045261050590270586428264139311483660321755451150238513946533203 11 +
O(pi^198)
```

Example H47E8

There is a subtle interplay between the default precision of rings and fields and coercion of sequences, which we demonstrate here. We construct the 5-adic field P, and an unramified extension of R. We set the default precision of R to be higher than that of P.

```
> P := pAdicRing(5);
> R := ext<P | 2>;
> P`DefaultPrecision;
20
> R`DefaultPrecision := 40;
> x := Random(R);
> x;
-157980184343196320136914558 7*R.1 - 68057573045897503903339476 9 + O(5^40)
> s := [-68057573045897503903339476 9, -157980184343196320136914558 7];
> R!s;
-13585890629962*R.1 + 47482939261481 + O(5^20)
> P`DefaultPrecision := 40;
> R!s;
-157980184343196320136914558 7*R.1 - 68057573045897503903339476 9 + O(5^40)
```

As can be seen from the above, it is the default precision of the base ring, not the ring itself, which determines the precision of elements created by sequences.

47.7.2 Element Decomposers

`ElementToSequence(x)`

`Eltseq(x)`

`Coefficients(x)`

Given an element x of a degree n extension K of L, these functions return a sequence s of elements of L such that $x = \sum_{i=1}^{n} s[i]K.1^{i-1}$.

`Coefficient(x, i)`

Equivalent to, but more efficient than, `Coefficients(x)[i]`.

Example H47E9

We want to perform a Galois descent for a polynomial, i.e. we interpret the product of the Galois conjugates of a polynomial in a subring.

```
> p := 3;
> L<a> := UnramifiedExtension(pAdicRing(p), 4 : Cyclotomic);
> R<x> := PolynomialRing(L);
> g := x^2 + (a+a^-2)*x + (a^-1+a^3+1);
> g;
(1 + O(3^20))*x^2 + (-151999392*a^3 - 428033534*a^2 + 1509587217*a - 64512399 +
    O(3^20))*x + 2*a^3 + 307453058*a^2 - 1732356354*a - 151999391 + O(3^20)
> a2 := a^(p^2);
> g2 := R ! [ &+[Eltseq(c)[i]*a2^(i-1) : i in [1..4]] : c in Eltseq(g) ];
```

Here `Eltseq` is being used to replace occurrences of a in the element by a_2.

```
> h := g * g2;
> h;
(1 + O(3^20))*x^4 + (774759822*a^3 - 1559939781*a^2 + 644136111*a + 143311364 +
    O(3^20))*x^3 + (73899384*a^3 - 333497478*a^2 + 1655979363*a + 158024680 +
    O(3^20))*x^2 + (989668189*a^3 - 661853000*a^2 - 1685887308*a + 122035547 +
    O(3^20))*x - 1630826887*a^3 - 328410694*a^2 - 175290219*a + 1601599448 +
    O(3^20)
```

The polynomial g_2 is the image of g under the automorphism induced by the square of the Frobenius automorphism. The product $h = gg_2$ has coefficients in the unramified extension of degree 2, which is the fixed field under the square of the Frobenius and is generated by a^{10}. Next, we determine a representation for the polynomial h with coefficients lying in this unramified extension of degree 2.

```
> K<b> := UnramifiedExtension(pAdicRing(p), 2 : Cyclotomic);
> S<y> := PolynomialRing(K);
> M := RMatrixSpace(PrimeRing(L), 2, 4) ! 0;
> V := RSpace(PrimeRing(L), 4);
> M[1] := V ! Eltseq(a^0);
> M[2] := V ! Eltseq(a^10);
> sol := [ Solution(M, V ! Eltseq(c)) : c in Eltseq(h) ];
```

```
> h2 := S ! [ K ! Eltseq(s) : s in sol ];
> h2;
(1 + O(3^20))*y^4 + (-1237301755*b + 505889265 + O(3^20))*y^3 + (-542504216*b +
    1258167831 + O(3^20))*y^2 + (-280210015*b - 1205051127 + O(3^20))*y +
    1213139407*b + 1602993886 + O(3^20)
```

47.8 Operations on Elements

47.8.1 Arithmetic

For local ring elements the usual operations are available. The quotient of two elements can be computed when the result lies in the ring (i.e. the valuation of the dividend is not smaller than that of the divisor). The precision of the result is reduced by the valuation of the divisor. The result of an operation with elements of reduced precision will have as much precision as possible. For addition and subtraction this is the minimum of the precisions of the two elements. For multiplication it is the minimum of $v_1 + k_2$ and $v_2 + k_1$, where v_i and k_i are the valuations and precisions of the two elements, respectively.

For local field elements the operations are performed with the maximum precision possible. For multiplication and division this is the minimum of the (relative) precisions of the two elements. For addition and subtraction of elements x and y with valuations v_x and v_y and precisions k_x and k_y the precision of $x \pm y$ is $\min(v_x + k_x, v_y + k_y) - v_p(x \pm y)$, which may even be 0.

`-x`

The negative of the element x.

`x + y`

The sum of the elements x and y.

`x - y`

The difference of the elements x and y.

`x * y`

The product of the elements x and y.

`x ^ k`

The k-th power of the element x. If k has valuation (when coerced) and x has precision less than that of its parent ring, the power x^k will have more precision than x.

`x div y`

The quotient of the elements x and y. For elements of a local ring, this results in an error if the valuation of x is smaller than that of y.

`x div:= y`

Mutation operation: replace the element x by its quotient upon division by y. For elements of a local ring, this results in an error if the valuation of x is smaller than that of y.

`x / y`

The quotient of the elements x and y. For elements of a local ring, the result will be returned in its field of fractions.

`IsExactlyDivisible(x, y)`

Return `true` if x can be exactly divided by y; in this case also return the quotient.

Example H47E10

Division of non-units in the ring will yield a result in the field of fractions. In that case the result has reduced precision. To ensure that the result is returned as a ring element, the operator `div` should be used.

```
> R := pAdicRing(2);
> pi := UniformizingElement(R);
> pi;
2 + O(2^20)
> 1 / pi;
2^-1 + O(2^18)
> Parent($1);
2-adic field
>  1 div pi;
>> 1 div pi;
     ^
Runtime error in 'div': Division is not exact
> IsExactlyDivisible(1, pi);
false
> IsExactlyDivisible(pi^2, pi);
true 2 + O(2^20)
```

47.8.2 Equality and Membership

It is possible to test whether two local ring or field elements are equal only in quotient rings. Equality testing in free precision rings is disabled, as there are several reasonable definitions of equality in an imprecise ring.

`x eq y`

Given local ring or field elements x and y, return `true` if and only if x and y are identical in value. This operator is only defined in fixed precision rings.

`x ne y`

Given local ring or field elements x and y, return `true` if and only if x and y are not identical in value. This operator is only defined in fixed precision rings.

`x in L`

Return `true` if and only if x lies in the local ring or field L.

`x notin L`

Return `true` if and only if x does not lie in the local ring or field L.

Example H47E11

We demonstrate how an unramified extension can be constructed from a given degree. The idea is to interpret the minimal polynomial of a primitive element in the residue class field as a polynomial over $\mathbf{Z}_p$. The quotient of $\mathbf{Z}_p[x]$ by the ideal generated by this polynomial is isomorphic to the unramified extension and $a := \overline{x}$ is a first approximation for the $p^f - 1$-th root of unity. This is improved by iterating $a \mapsto a^{p^f}$ until this remains fixed.

```
> p := 2;
> f := 5;
> Zp := pAdicRing(p, 25);
> R<x> := PolynomialRing(Zp);
> g := R ! MinimalPolynomial(GF(p,f).1);
> Q<r> := quo<R | g>;
> a := [ r, r^(p^f) ];
> while a[#a] ne a[#a-1] do
>     print a[#a];
>     Append(~a, a[#a]^(p^f));
> end while;
34*r^4 - 44*r^3 + 58*r^2 - 23*r + 36
12522914*r^4 + 12522004*r^3 - 12174790*r^2 - 8200343*r - 10407260
8242594*r^4 + 12409364*r^3 + 5143098*r^2 + 15781737*r - 3636572
5490082*r^4 + 8804884*r^3 - 11109830*r^2 + 11456361*r + 11698852
-15481438*r^4 - 5875180*r^3 + 5667386*r^2 + 7262057*r - 884060
> [ Minimum([ Valuation(c) : c in Eltseq(a[i] - a[i-1]) ]) : i in [2..#a-1] ];
[ 1, 6, 11, 16, 21 ]
```

The last statement demonstrates the convergence of the process. The polynomial defining the unramified extension could now easily be obtained as the minimal polynomial of the fixed element.

```
> U := ext<Zp | f>;
> MinimalPolynomial(U ! Eltseq(a[#a]));
x^5 - 13205240*x^4 - 3159900*x^3 - 13778593*x^2 + 9730498*x - 1
```

47.8.3 Properties

Local ring and field elements can be tested for certain properties. However, note that in free precision rings and fields, exact zero, one, and minus one elements do not exist, only approximations to such elements — in these cases, the predicates will always return `false`.

`IsZero(x)`

Given an element x of a local ring or field, return `true` if and only if x is the zero element of its parent ring or field.

`IsOne(x)`

Given an element x of a local ring or field, return `true` if and only if x is the one element of its parent ring or field.

`IsMinusOne(x)`

Given an element x of a local ring or field, return `true` if and only if x is the minus one element of its parent ring or field.

`IsUnit(x)`

Given an element x of a local ring or field, return `true` if and only if x is a unit, that is, x has valuation 0.

`IsIntegral(x)`

Given an element x of a local ring or field, return `true` if and only if x has non-negative valuation.

47.8.4 Precision and Valuation

The parent of a local element can be retrieved, its precision accessed and changed, its valuation computed and its denominator returned.

`Parent(x)`

Return the parent local ring or field of the element x.

`Precision(x)`

For a local ring element x, returns the precision to which it is known; for a local field element x, returns the precision to which its unit part is known.

`AbsolutePrecision(x)`

Returns the precision of a local ring or field element x.

`RelativePrecision(x)`

Returns the difference between the absolute precision and the valuation of the local ring or field element x.

`ChangePrecision(x, k)`

`ChangePrecision(∼x, k)`

`ChangePrecision(x, k)`

`ChangePrecision(∼x, k)`

Given a local ring or field element or polynomial over a local ring or field x and a non-negative single precision integer k, change the precision of x to k. If k is smaller than the precision of x, x is truncated; if k is larger, then an element y is returned such that $v(x-y) \geq k$. Note that the precision of an element cannot be changed to be greater than its parent. Any attempt to do so will result in the element gaining the precision of its parent.

`Expand(x)`

Change the precision of the local ring or field element x to be the maximum precision allowed by its parent. This results in an error if x is an element of an unbounded free precision ring.

`Valuation(x)`

Return the valuation of the local ring or field element x. This is always bounded by the absolute precision of the element.

Example H47E12

The uses and properties of relative and absolute precision for field elements are illustrated here, as well as the results of `Expand`.

```
> R<x> := PolynomialRing(Integers());
> K<d> := ext<ext<pAdicField(5, 100) | 2> |  x^2 + 5>;
> x := d + d^7;
> x;
-d*124 + O(d^200)
> AbsolutePrecision(x);
200
> RelativePrecision(x);
199
> RelativePrecision(x + O(d^10));
9
> AbsolutePrecision(x + O(d^10));
10
> RelativePrecision(ChangePrecision(x, 19));
19
> RelativePrecision(ChangePrecision(x, 10));
10
> AbsolutePrecision(ChangePrecision(x, 10));
11
> Expand(ChangePrecision(x, 10));
```

```
d*3001 + O(d^201)
> Valuation(x);
1
> ChangePrecision(x, 20) - (x + O(d^21));
O(d^21)
```

The last two lines show that changing the precision of an element is equivalent to adding imprecision to the element effectively cancelling off all terms beyond that of relative valuation 20.

47.8.5 Logarithms and Exponentials

Logarithms and exponentials can also be calculated for certain elements.

`Log(x)`

The logarithm of the local ring or field element x, returned to the precision of x. Note that the power series of the logarithm function only converges if the valuation of $x-1$ is positive. For ring elements x, the answer lies in the ring (and not its field of fractions) only if the valuation of $x-1$ is greater than or equal to the ramification degree of the ring divided by the prime.

The rate of convergence of `Log` is dependent on the valuation of $x-1$. The greater the valuation the faster the convergence, as is illustrated in the example below.

`Exp(x)`

The exponential of the local ring or field element x, returned to the precision of x. Note that the power series of the exponential function only converges if the valuation of x is strictly larger than $e/(p-1)$, where e is the ramification degree of the parent ring of x.

Example H47E13

This example illustrates the relative timings of the `Log` function and the `Exp` function for rings and fields and for various valuations of $x-1$ or x as appropriate.

```
> K := ext<pAdicField(3, 20) | x^3 + x^2 + x + 2>;
> K<d> := ext<K | x^3 + 3*x^2 + 3*x + 3>;
> L<b> := IntegerRing(K);
> x := 1 + b;
> time Log(x);
Time: 0.070
> x := 1 + b^5;
> time Log(x);
874050819*b^2 + 1624571442*b - 914550768
Time: 0.010
> Valuation(Exp(Log(x)) - x);
60
> x := b^2;
```

```
> time Exp(x);
1418565628*b^2 + 1334033745*b + 905945461
Time: 0.170
> Valuation(Log(Exp(x)) - x);
60
> x := b^6;
> time Exp(x);
1700312013*b^2 - 72781965*b + 1129064707
Time: 0.020
> Valuation(Log(Exp(x)) - x);
60
```

47.8.6 Norm and Trace Functions

The norm, trace and minimal polynomial of an element can be calculated. A function returning the image of an element under a power of the Frobenius automorphism is also provided.

`Norm(x)`

Given a local ring or field element x, return the norm of x over the base ring of its parent. The norm is the product of all conjugates of x.

`Norm(x, R)`

Given a local ring or field element x, return the norm of x over the local ring or field R. The ring R must be a subring of the parent of x. The norm is the product of all conjugates of x.

`Trace(x)`

Given a local ring or field element x, return the trace of x over the base ring of its parent. The trace is the product of all conjugates of x.

`Trace(x, R)`

Given a local ring or field element x, return the trace of x over the local ring or field R. The ring R must be a subring of the parent of x. The trace is the sum of all conjugates of x.

`MinimalPolynomial(x)`

Given a local ring or field element x, return the minimal polynomial of x over the base ring of its parent.

`MinimalPolynomial(x, R)`

Given a local ring or field element x, return the minimal polynomial of x over the local ring or field R. The ring R must be a subring of the parent of x.

`CharacteristicPolynomial(x)`

Given a local ring element x, return the characteristic polynomial of x over the base ring of its parent.

`CharacteristicPolynomial(x, R)`

Given a local ring or field element x, return the characteristic polynomial of x over the local ring or field R. The ring R must be a subring of the parent of x.

`GaloisImage(x, i)`

Given a local ring or field element x, return the image of x under the Frobenius automorphism composed with itself i times, that is, $a \mapsto a^{(p^i)}$ where a is a $p^f - 1$st root of unity of the parent of x.

Example H47E14

One important application of the p-adics is to counting points on elliptic curves over finite fields. This example demonstrates how a simple version of the Arithmetic-Geometric Mean (AGM) algorithm could be implemented in MAGMA.

```
> d := 50;
> FF := GF(2^d);
> E := EllipticCurve([FF | 1, 0, 0, 0, Random(FF)]);
> assert Degree(sub<BaseRing(E) | jInvariant(E)>) gt 2;
>
> n := (d + 1) div 2 + 3;
> R := ext<pAdicRing(2) | d>;
> R`DefaultPrecision := n;
>
> a6 := elt<R | jInvariant(E)^-1, 1>;
> lambda := 1 + 8 * a6;
>
> for k in [4..n] do
>     ChangePrecision(~lambda, k + 2);
>     lambda := (1 + lambda) * InverseSqrt(lambda) div 2;
> end for;
> lambda := 2 * lambda div (1 + lambda);
> Exp(Trace(Log(lambda)));
32196193 + O(2^26)
> Trace(E) mod 2^26;
32196193
```

47.8.7 Teichmüller Lifts

Another important operation is the determination of the canonical *Teichmüller lift* of an element u in the residue field of a p-adic ring. By definition this is the unique root of unity of order prime to p which reduces to u modulo the maximal ideal.

`TeichmuellerLift(u, R)`

For a fixed precision quotient ring R of the p-adics or an unramified extension and a finite field element u, the function attempts to coerce u into the residue class field of R and then computes and returns its Teichmüller lift to R. This uses a fast iterative lifting method of Harley described in Chapter 12 of [C$^+$05]. For unramified extensions this works most efficiently with `Cyclotomic` or `Gaussian Normal` bases since it uses the Frobenius operation.

47.9 Linear Algebra

All linear algebra over local rings and fields in MAGMA is implemented in terms of the fixed precision model. Internally, linear algebra over free precision models is performed over an appropriate quotient ring. The advantage of this method is that exact algorithms can be used to solve linear systems; the disadvantage is that the answer may be returned to less precision than is theoretically possible.

47.10 Roots of Elements

Roots of local ring and field elements can be found to some precision.

`SquareRoot(x)`

`Sqrt(x)`

Given a local ring or field element x, return a square root of x. An error results if x is not a square. The result may have less precision than x.

`IsSquare(x)`

Return whether the local ring or field element x is the square of an element in its parent and if it is, the square root is returned. The result may have less precision than x.

`InverseSquareRoot(x)`

`InverseSqrt(x)`

Given a local ring or field element x, return an inverse square root of x. The element x must be a unit. An error results if x is not a square. The result may have less precision than x.

`InverseSquareRoot(x, y)`

`InverseSqrt(x, y)`

Given local ring or field elements x and y, return an inverse square root of x lifted from an initial approximation y. The element x must be a unit. An error results if x is not a square, or if y is not a valid initial approximation to an inverse square root of x. The result may have less precision than x.

`Root(x, n)`

Return an n-th root of x if one exists. An error results if x is not an n-th power. The result may have less precision than x.

`IsPower(x, n)`

Return whether x is an n-th power of some element belonging to its parent and if it is return an n-th root. The result may have less precision than x.

`InverseRoot(x, n)`

Given a local ring or field element x, return an inverse n-th root of x. The element x must be a unit. An error results if x is not an n-th power. The result may have less precision than x.

`InverseRoot(x, y, n)`

Given local ring or field elements x and y, return an inverse n-th root of x lifted from an initial approximation y. The element x must be a unit. An error results if x is not an n-th power, or if y is not a valid initial approximation to an inverse n-th root of x. The result may have less precision than x.

47.11 Polynomials

Various simple operations for polynomials as well as root finding and factorization functions have been implemented for polynomials over local rings and fields.

47.11.1 Operations for Polynomials

A number of functions for polynomials in general are applicable for polynomials over local rings and fields. Arithmetic functions, including `div` and `mod` can be used with such polynomials (though there may be some precision loss), as well as all the elementary functions to access coefficients and so forth. Derivatives can be taken and polynomials over local rings and fields can be evaluated at elements coercible into the coefficient ring. Along with GCD for these polynomials, the LCM of two polynomials can also be found.

Although the ring of polynomials over a local ring is not a principal ideal domain, it is useful to have a GCD function available. For example, for polynomials which are coprime over the local field, the ideal generated by the two polynomials contains some power of the uniformizing element of the local ring. This power determines whether an approximate factorization can be lifted to a proper factorization.

`GreatestCommonDivisor(f, g)`

`Gcd(f, g)`

`GCD(f, g)`

Determine the greatest common divisor of polynomials f and g where f and g are over a free precision local ring or field. The GCD returned is such that the cofactors of the polynomials will have coefficients in the ring (if the polynomial is not over a field). The process of computing the GCD of two polynomials may result in some inaccuracy. The GCD is computed by echelonizing the Sylvester matrix of f and g.

`f div g` `f mod g` `LeastCommonMultiple(f, g)`

`Coefficient(f, i)` `LeadingCoefficient(f)`

`Derivative(f)` `Evaluate(f, x)`

Example H47E15

This example illustrates the usage and results of the GCD functions. Note the precision loss in the answer.

```
> L := pAdicRing(5, 20);
> R<x> := PolynomialRing(L);
> elts := [Random(L) : i in [1..3]];
> f := (x - elts[1])^3 * (x - elts[2])^2 * (x - elts[3]);
> f;
x^6 + 31722977012336*x^5 - 34568128687249*x^4 +
    4655751767246*x^3 + 11683626356181*x^2 -
    29833674388290*x + 32360011367900
> GCD(f, Derivative(f));
(1 + O(5^18))*x^3 - (934087632277 + O(5^18))*x^2 -
    (89130566204 + O(5^18))*x + 1178670674955 + O(5^18)
> f mod $1;
O(5^18)*x^5 + O(5^18)*x^4 + O(5^18)*x^3 + O(5^18)*x^2 +
    O(5^18)*x + O(5^19)
> (x - elts[1])^2 * (x - elts[2]);
x^3 + 14324701430223*x^2 + 26613750293171*x +
    31696248799955
> ChangePrecision($1, 18);
(1 + O(5^18))*x^3 - (934087632277 + O(5^18))*x^2 -
    (89130566204 + O(5^18))*x + 1178670674955 + O(5^18)
```

`ShiftValuation(f, n)`

Shifts the valuation of each coefficient of f by n, ie. scales the polynomial by π^n.

47.11.2 Roots of Polynomials

Roots of polynomials can be found to some precision by applying the Hensel lift to an approximation or by factorizing the polynomial and looking for linear factors from which the roots can be read off.

47.11.2.1 Hensel Lifting of Roots

Roots of a polynomial f defined over a local ring or field can be found by first determining their valuation (using the Newton polygon), finding a first approximation over the finite field and finally improving this approximation until the Hensel condition $v(f(a)) > 2v(f'(a))$ is satisfied.

`NewtonPolygon(f)`

The Newton polygon of a polynomial $f = \sum_{i=0}^{n} c_i x^i$ (over a local ring or field) is the lower convex hull of the points $(i, v(c_i))$. The slopes of the Newton polygon determine the valuations of the roots of f in a splitting ring and the number of roots with that valuation. The faces of the Newton polygon can be determined using the function `Faces` which returns the faces expressed as the line $mx + ny = c$ which coincides with the face. The function `GradientVector` will return the m and n values from the line so that the valuation (gradient) can be calculated. The function `EndVertices` will return the end points of the face, the x coordinates of which will give the number of roots with valuation equal to the gradient of the face.

Newton polygons are discussed in greater detail in Chapter 46 and are illustrated below.

`ValuationsOfRoots(f)`

Return a sequence containing pairs which are valuations of roots of f and the number of roots of f which have that valuation.

Example H47E16

For a polynomial of the form $g := \prod(x - r_i)$ we demonstrate that the Newton polygon determines the valuations of the roots of g.

```
> Z3 := pAdicRing(3, 30);
> R<y> := PolynomialRing(Z3);
> pi := UniformizingElement(Z3);
> roots := [ pi^Random([0..3]) * Random(Z3) : i in [1..10] ];
> [ Valuation(r) : r in roots ];
[ 3, 1, 6, 3, 0, 3, 2, 3, 3, 2 ]
> g := &* [ y - r : r in roots ];
> N := NewtonPolygon(g);
> N;
Newton Polygon of y^10 + 44594997030169*y^9 - 85346683389318*y^8 +
    76213593390537*y^7 + 74689026811236*y^6 - 48671968754502*y^5 -
    58608670426020*y^4 - 63609139981179*y^3 + 77334553491246*y^2 +
    39962036019861*y - 94049035648173 over pAdicRing(3, 30)
```

```
> F := Faces(N);
> F;
[ <6, 1, 26>, <3, 1, 23>, <2, 1, 17>, <1, 1, 9>, <0, 1, 0> ]
> [GradientVector(F[i]) : i in [1 .. #F]];
[ <6, 1>, <3, 1>, <2, 1>, <1, 1>, <0, 1> ]
> [$1[i][1]/$1[i][2] : i in [1 ..#$1]];
[ 6, 3, 2, 1, 0 ]
> [EndVertices(F[i]) : i in [1 .. #F]];
[
    [ <0, 26>, <1, 20> ],
    [ <1, 20>, <6, 5> ],
    [ <6, 5>, <8, 1> ],
    [ <8, 1>, <9, 0> ],
    [ <9, 0>, <10, 0> ]
]
> [$1[i][2][1] - $1[i][1][1] : i in [1 .. #$1]];
[ 1, 5, 2, 1, 1 ]
```

So there is one root of valuation 6, five of valuation 3, two of valuation 2, one of valuation 1 and one root with valuation zero. This information could also have been gained using `ValuationsOfRoots`.

```
> ValuationsOfRoots(g);
[ <6, 1>, <3, 5>, <2, 2>, <1, 1>, <0, 1> ]
```

`HenselLift(f, x)`

`HenselLift(f, x, k)`

Return a root of the polynomial f over a local ring or field by lifting the approximate root x to a root with precision k (or the default precision of the structure if not specified). This results in an error, if the Hensel condition $v(f(x)) > 2v(f'(x))$ is not satisfied.

Example H47E17

This examples illustrates how Hensel lifting is used to compute square roots.

```
> Zx<x> := PolynomialRing(Integers());
> L1<a> := ext<pAdicRing(3, 20) | 2>;
> L2<b> := ext<L1 | x^2 + 3*x + 3>;
> R<y> := PolynomialRing(L2);
> c := (a+b)^42;
> r := L2 ! Sqrt(ResidueClassField(L2) ! c);
> r;
a
> rr := HenselLift(y^2-c, r);
> rr;
(-1513703643*a - 1674232545)*b - 1219509587*a + 760894776
> Valuation(rr^2 - c);
```

```
40
> ChangePrecision(rr, 1);
a + O(b)
```

For $p = 2$ the situation is a bit more difficult, since the derivative of $y^2 - c$ is not a unit at this point.

```
> Zx<x> := PolynomialRing(Integers());
> L1<a> := ext<pAdicRing(2, 20) | 2>;
> L2<b> := ext<L1 | x^2 + 2*x + 2>;
> R<y> := PolynomialRing(L2);
> c := (a+b)^42;
> r := L2 ! Sqrt(ResidueClassField(L2) ! c);
> r;
1
>  HenselLift(y^2-c, r);
>> HenselLift(y^2-c, r);
             ^
Runtime error in 'HenselLift': Hensel lift condition is not satisfied
```

We have to find a better approximation for the square root first.

```
> for d in GF(2,2) do
>     if Valuation((r + b*L2!d)^2 - c) gt 4 then
>         print L2!d;
>     end if;
> end for;
a + 1
> r +:= b * (a+1);
> HenselLift( y^2-c, r );
(-199021*a + 100463)*b + 204032*a - 31859 + O(b^38)
> ChangePrecision($1, 1);
1 + O(b)
> ChangePrecision($2, 2);
(a + 1)*b + 1 + O(b^2)
```

The square root is an element of reduced precision, since the proper root is only guaranteed to coincide with the approximation up to valuation 18.

47.11.2.2 Functions returning Roots

These functions determine the roots of a polynomial from the factorization of the polynomial.

`Roots(f)`

`Roots(f, R)`

`IsSquarefree` BoolElt *Default :* `false`

Return the roots of the polynomial f over a local ring or field R as a sequence of tuples of elements in R and multiplicities. If R is not specified it is taken to be the

coefficient ring of f. If the polynomial is known to be squarefree, the root-finding algorithm may run considerably faster.

`HasRoot(f)`

Try to find a root of the polynomial f over a local ring or field. If a root is found, this function returns `true` and a root as a second value; otherwise it returns `false`.

Example H47E18

We generate the ramified extensions of $\mathbf{Z}_2$ of degree 2 by looping over some Eisenstein polynomials with small coefficients and checking whether a new polynomial has a root in one of the already known rings.

```
> Zx<x> := PolynomialRing(Integers());
> RamExt := [];
> for c0 in [2, -2, 6, -6] do
> for c1 in [0, 2, -2, 4, -4, 6, -6] do
>     g := x^2 + c1*x + c0;
>     new := true;
>     for L in RamExt do
>         if HasRoot(PolynomialRing(L)!g) then
>             new := false;
>             break L;
>         end if;
>     end for;
>     if new then
>         print "new field with polynomial", g;
>         Append(~RamExt, ext<pAdicRing(2, 20) | g>);
>     end if;
> end for;
> end for;
new field with polynomial x^2 + 2
new field with polynomial x^2 + 2*x + 2
new field with polynomial x^2 + 4*x + 2
new field with polynomial x^2 + 2*x - 2
new field with polynomial x^2 + 4*x - 2
new field with polynomial x^2 + 6
```

These are all such extensions, since the extensions of $\mathbf{Q}_2$ of degree 2 are in bijection to the 7 non-trivial classes of $\mathbf{Q}_2^*/(\mathbf{Q}_2^*)^2$ and one of these classes yields the unramified extension.

47.11.3 Factorization

It is possible to factorize (to some precision) polynomials over a local ring or field. Approximate factorizations can also be lifted to a factorization to greater precision.

`HenselLift(f, s)`

Given a sequence s of polynomials with coefficients that can be coerced into the coefficient ring of f such that $f \equiv \prod_{i=1}^{\#s} s[i]$ modulo π, $s[i]$ and $[j]$ are co–prime modulo π for all i and j, and $s[i]$ is monic for all i, find a more accurate factorization $t[1], t[2], \ldots t[\#s]$ such that $f \equiv \prod_{i=1}^{\#s} t[i]$ modulo π^k, where k is the minimum precision of the coefficients of f.

Example H47E19

If the reduction of a polynomial over the residue class field is not a power of an irreducible polynomial, the factorization into powers of different irreducibles can be lifted to a factorization over the local ring.

```
> Z2 := pAdicRing(2, 25);
> R<x> := PolynomialRing(Z2);
> f := &* [ x - i : i in [1..8] ];
> F2 := ResidueClassField(Z2);
> Factorization( PolynomialRing(F2)!f );
[
    <$.1, 4>,
    <$.1 + 1, 4>
]
> h1 := x^4;
> h2 := (x+1)^4;
> h := HenselLift(f, [h1, h2]);
> h[1], h[2], f - h[1]*h[2];
x^4 - 20*x^3 + 140*x^2 - 400*x + 384
x^4 - 16*x^3 + 86*x^2 - 176*x + 105
0
```

`IsIrreducible(f)`

Given a polynomial f with coefficients lying in a local ring or field L, return `true` if and only if f is irreducible over L. Currently, this only works over p-adic rings, unramified extensions of p-adic rings, totally ramified extensions of p-adic rings, and totally ramified extension of unramified extensions of p-adic rings.

`SquareFreeFactorization(f)`

Return a sequence of tuples of polynomials and multiplicities where the polynomials are not divisible by any square of a polynomial. The product of the polynomials to the corresponding multiplicities is the polynomial f (to some precision). Currently, this only works over p-adic rings, unramified extensions of p-adic rings, totally ramified extensions of p-adic rings, and totally ramified extension of unramified extensions of p-adic rings.

`Factorization(f)`

`LocalFactorization(f)`

`Certificates`	BoolElt	*Default :* `false`
`IsSquarefree`	BoolElt	*Default :* `false`
`Ideals`	BoolElt	*Default :* `false`
`Extensions`	BoolElt	*Default :* `false`

Return the factorization of the polynomial f over a local ring or field into irreducible factors as a sequence of pairs, the first entry giving the irreducible factor and the second its multiplicity.

Precision is important since for polynomials over rings of relatively small precision a correct factorization may not be possible and an error will result. A lower bound on the precision needed for the factorization to succeed is given by `SuggestedPrecision`; this precision may still be insufficient, however.

The precision the factorization is returned to is reduced for multiple factors.

The optional parameter `IsSquarefree` can be set to `true`, if the polynomial is known to be square-free. The `Certificates` parameter can be set to `true` to compute certificates for the irreducibility of the individual factors returned. This information can be used to compute the p-maximal order of the equation order defined by the factor. If the `Extensions` parameter is set to `true` then certificates will be returned which will include an extension given by each factor.

`SuggestedPrecision(f)`

For a polynomial f over a local ring or field, return a precision at which the factorization of f as given by `Factorization(f)` will be Hensel liftable to the correct factorization.

The precision returned is not guaranteed to be enough to obtain a factorization of the polynomial. It may be that a correct factorization cannot be found at that precision but may be possible with a little more precision.

Currently, this only works over p-adic rings, unramified extensions of p-adic rings, totally ramified extensions of p-adic rings, and totally ramified extension of unramified extensions of p-adic rings.

`IsIsomorphic(f, g)`

Given two irreiducible polynomials over the same p-adic field, test if the extensions defined by them are isomorphic.

`Distance(f, g)`

Given two Eisenstein polynomials of the same degree and discriminant compute their distance, i.e. the minimal weighted valuation of the difference of the coefficients.

Example H47E20

The use of `SuggestedPrecision` along with `Factorization` is illustrated below for a few different cases.

```
> L<b> := ext<ext<pAdicRing(5, 20) | 2> |  y^2 + 5*y + 5>;
> R<x> := PolynomialRing(L);
> f := (x - 1)^3*(x - b)^2*(x - b^2 + b - 1);
> SuggestedPrecision(f);
80
> Factorization(f);
[
    <x - b + O(b^40), 2>,
    <x - 1 + O(b^40), 3>,
    <x + 6*b + 4 + O(b^40), 1>
]
1 + O(b^40)
> f := (x + 2)^3*(x + b)*(x + 3)^4;
> f;
x^8 + (b + 18 + O(b^40))*x^7 + (18*b + 138 + O(b^40))*x^6 + (138*b + 584 +
    O(b^40))*x^5 + (584*b + 1473 + O(b^40))*x^4 + (1473*b + 2214 + O(b^40))*x^3
    + (2214*b + 1836 + O(b^40))*x^2 + (1836*b + 648 + O(b^40))*x + 648*b +
    O(b^40)
> SuggestedPrecision(f);
80
> Precision(L);
80
> P<y> := PolynomialRing(Integers());
> R<b> := ext<ext<pAdicRing(3, 20) | 2> |  y^2 + 3*y + 3>;
> P<x> := PolynomialRing(R);
> f := x^12 + 100*x^11 + 4050*x^10 + 83700*x^9 + 888975*x^8 + 3645000*x^7 -
> 10570500*x^6 - 107163000*x^5 + 100875375*x^4 + 1131772500*x^3 -
> 329614375*x^2 + 1328602500*x + 332150625;
> SuggestedPrecision(f);
48
> Factorization(f);
[
    <x + 153560934 + O(b^40), 1>,
    <x - 1595360367 + O(b^40), 1>,
    <x + 1273329451*$.1 - 1037496066 + O(b^40), 1>,
```

```
    <x + -1273329451*$.1 - 97370567 + O(b^40), 1>,
    <x^4 + (-1598252738*$.1 - 309919655 + O(b^40))*x^3 + (-280421715*$.1 -
        102998055 + O(b^40))*x^2 + (1263867503*$.1 + 923047935 + O(b^40))*x +
        -1252549104*$.1 - 786365260 + O(b^40), 1>,
    <x^4 + (1598252738*$.1 - 600198580 + O(b^40))*x^3 + (280421715*$.1 +
        457845375 + O(b^40))*x^2 + (-1263867503*$.1 - 1604687071 + O(b^40))*x +
        1252549104*$.1 + 1718732948 + O(b^40), 1>
]
1 + O(b^40)
> R<b> := ext<ext<pAdicRing(3, 25) | 2> |  y^2 + 3*y + 3>;
> P<x> := PolynomialRing(R);
> f := x^12 + 100*x^11 + 4050*x^10 + 83700*x^9 + 888975*x^8 + 3645000*x^7 -
> 10570500*x^6 - 107163000*x^5 + 100875375*x^4 + 1131772500*x^3 -
> 329614375*x^2 + 1328602500*x + 332150625;
> Factorization(f);
[
    <x + 143905694229 + O(b^50), 1>,
    <x - 399882754935 + O(b^50), 1>,
    <x + 46601526664*$.1 + 229090274400 + O(b^50), 1>,
    <x + -46601526664*$.1 + 135887221072 + O(b^50), 1>,
    <x^4 + (242476655332*$.1 + 187976437999 + O(b^50))*x^3 + (372805509192*$.1 -
        38457626466 + O(b^50))*x^2 + (126788105939*$.1 + 60198382752 +
        O(b^50))*x + 347425890996*$.1 + 215394267602 + O(b^50), 1>,
    <x^4 + (-242476655332*$.1 - 296976872665 + O(b^50))*x^3 + (-372805509192*$.1
        + 63219964593 + O(b^50))*x^2 + (-126788105939*$.1 - 193377829126 +
        O(b^50))*x + -347425890996*$.1 + 367831095053 + O(b^50), 1>
]
1 + O(b^50)
```

Note that the polynomial itself must have coefficients with precision at least that given by `SuggestedPrecision` (and not just the coefficient ring) for `Factorization` to succeed. Sometimes this will not be possible if the coefficients of the polynomial are not known to sufficient precision.

Example H47E21

In this example we demonstrate how factorizations of a rational polynomial over some local rings can give information on the Galois group.

```
> Zx<x> := PolynomialRing(Integers());
> g := x^5 - x + 1;
> Factorization(Discriminant(g));
[ <19, 1>, <151, 1> ]
> g2 := Factorization( PolynomialRing(pAdicRing(2, 10)) ! g );
> g2;
[
    <$.1^2 + 367*$.1 - 93, 1>,
    <$.1^3 - 367*$.1^2 - 386*$.1 + 11, 1>
]
```

```
> g3 := Factorization( PolynomialRing(pAdicRing(3, 10)) ! g );
> g3;
[
    <$.1^5 - $.1 + 1, 1>
]
> g7 := Factorization( PolynomialRing(pAdicRing(7, 10)) ! g );
> g7;
[
    <$.1^2 + 138071312*$.1 + 2963768, 1>,
    <$.1^3 - 138071312*$.1^2 + 132202817*$.1 - 84067650, 1>
]
```

This shows that the Galois group of g contains elements of orders 2, 3, 5 and 6, and therefore is isomorphic to S_5.

47.12 Automorphisms of Local Rings and Fields

The automorphisms of a local ring or field are determined by their images on the generators of the ring. All computations necessary to determine the automorphism group can be performed in the local ring.

`Automorphisms(L)`

Given a local ring or field L, returns the automorphisms of L over its p-adic sub-field $\mathbf{Q}_p$ as a sequence of maps of L into L.

`Automorphisms(K, k)`

Given a local ring or field K over k, returns the k-automorphisms of K as a sequence of maps of K into K.

`AutomorphismGroup(L)`

Return the automorphism group acting on L over its p-adic sub-field $\mathbf{Q}_p$ as a permutation group (representing the regular action). Also return the map from the permutation group to the group of automorphisms represented explicitly (i.e. like returned from the function above).

`AutomorphismGroup(K, k)`

Return the automorphism group acting on K over its p-adic k as a permutation group (representing the regular action). Also return the map from the permutation group to the group of automorphisms represented explicitly (i.e. like returned from the function above).

`IsNormal(K)`

Given a p-adic ring or field K, test if K is normal over it's prime field $\mathbf{Q}_p$, ie. if K admits exactly n automorphisms where n is the degree of K.

`IsNormal(K, k)`

Given a p-adic ring or field K, test if K is normal over the subfield k.

`IsAbelian(K, k)`

Given p-adic fields K/k, test if the automorphism group of K over k is abelian.

`Continuations(m, L)`

For an automorphism m of the p-adic ring L, compute all possible extensions of m to L.

`IsIsomorphic(E, K)`

For two p-adic rings or fields, test if they are isomorphic over $\mathbf{Q}_p$.

Example H47E22

We define an extension of $\mathbf{Z}_2$ with ramification degree 2 and inertia degree 2 and compute automorphisms.

```
> I<a> := ext<pAdicRing(2, 10) | 2>;
> R<x> := PolynomialRing(I);
> L<b> := ext<I | x^2 + 2*a*x + 2*a^2>;
> L;
Totally ramified extension defined by the polynomial
x^2 + (2*a)*x + -2*a - 2 over Unramified extension
defined by the polynomial x^2 + x + 1 over 2-adic ring
mod 2^10
> A := Automorphisms(L);
> [<A[i](a), A[i](b)> : i in [1 .. #A]];
[ <a, b + O(b^18)>, <a, -b + -2*a + O(b^18)>, <-a - 1,
a*b + O(b^18)>, <-a - 1, -a*b + 2*a + 2 + O(b^18)> ]
> AutomorphismGroup(L);
Permutation group acting on a set of cardinality 4
    Id($)
    (1, 2)(3, 4)
    (1, 3)(2, 4)
Mapping from: GrpPerm: $, Degree 4 to Power Structure
of Map given by a rule
```

47.13 Completions

Local Rings can be obtained by completing an order at a prime ideal (see Chapter 34 and Completion on page 891).

```
Completion(O, P)
Completion(K, P)
```

Precision RngIntElt *Default :* 20

The completion (as an unbounded precision local ring or field with default precision given by Precision) of the order or number field at the prime ideal P, and the embedding of the order or number field into the resulting local ring.

```
LocalRing(P, k)
```

The completion (as a local ring) of the order of the prime ideal P at P with precision k and the embedding of the order into the resulting local ring.

Example H47E23

Here we demonstrate the use of Completion.

```
> K := NumberField(x^6 - 5*x^5 + 31*x^4 - 85*x^3 + 207*x^2 - 155*x + 123);
> lp := Decomposition(K, 7);
> C, mC := Completion(K, lp[2][1]);
> C;
Totally ramified extension defined by a map over Unramified extension defined by
a map over 7-adic field
> mC;
Mapping from: FldNum: K to FldPad: C given by a rule
> mC(K.1);
(46564489*$.1 - 47959419)*C.1 - 116434149*$.1 - 61099304 + O(C.1^20)
> mC(K.1)@@mC - K.1;
8337821493402521350488*K.1^5 - 690735069600568964644432*K.1^4 +
    189847416443444330877726*K.1^3 - 45336153029197695133787 6*K.1^2 +
    336979647814116799276099*K.1 - 26752086971419700257907 1
> Valuation($1, lp[2][1]);
18
> C`DefaultPrecision := 30;
> mC(K.1);
(1090965976127*$.1 - 1208477074641)*C.1 - 589359803563*$.1 + 288063654676 +
O(C.1^30)
> mC(K.1)@@mC - K.1;
-619800242441603716728687734334907 83*K.1^5 +
    1189796803064803092593291088768968754*K.1^4 -
    320235394693319058886430918065386895 7*K.1^3 +
    753738692804616458073114503187201704 9*K.1^2 -
    551129700293668257996421058601330881 0*K.1 +
    443809944480643131358253343594109872 2
```

```
> Valuation($1, lp[2][1]);
28
> C`DefaultPrecision := 10;
> mC(K.1);
(-7708*$.1 + 7759)*C.1 + 4747*$.1 - 5859 + O(C.1^10)
> mC(K.1)@@mC - K.1;
1908210240*K.1^5 - 7326424608*K.1^4 + 16701662320*K.1^3 - 35965440540*K.1^2 +
    41324075079*K.1 - 30476856505
> Valuation($1, lp[2][1]);
8
```

47.14 Class Field Theory

The class field theory of local fields classifies abelian extensions of local field in a way similar to the way global class field theory deals with extensions of number fields and global function fields.

While the origins of local class field theory are, via completions and localisations, in the global case, today it is a theory in its own. Although local class field theory can be used to obtain global results, it has very powerful generalisations that the global case (currently) does not allow.

Local class fields are classified in terms of the norm group, ie. the multiplicative group of norms of elements, rather than some ideal or divisor class group as in the global case. Since the multiplicative group of a local field is far better understood than the ideal group of a global field, the theory is much more explicit and easier in the local case.

47.14.1 Unit Group

In contrast to the case of global fields, the multiplicative group of both p-adic rings and fields has a well understood structure which can be computed by algorithms developed and implemented by S. Pauli [Pau06]. It should be noted that all the unit group related functions operate on fixed-precision rings only.

`PrincipalUnitGroupGenerators(R)`

The principal units of a p-adic ring or field R are elements of the form $1 + \pi\mathbf{Z}_R$ where π is a uniformizing element of R and $\mathbf{Z}_R$ is the ring of integers. This function returns a sequence of generators for this group.

`PrincipalUnitGroup(R)`

The principal units of a p-adic ring or field R are elements of the form $1 + \pi\mathbf{Z}_R$ where π is a uniformizing element of R and $\mathbf{Z}_R$ is the ring of integers. This function returns an abstract abelian group isomorphic to the group of principal units and an explicit isomorphism, ie. a map between the abstract group and the p-adic ring or field.

`UnitGroup(R)`

Given a p-adic ring R of fixed precision, this function computes an abstract abelian group isomorphic to the unit group as well as an explicit map between the abstract group and R.

`UnitGroup(F)`

Given a p-adic field F of fixed precision, this function computes an abstract abelian gorup isomorphic to the multiplicative group of F as well as an explicit map between the abstract group and F.

`UnitGroupGenerators(R)`

Given a p-adic ring with fixed precision, this function computes generators for its unit group.

`UnitGroupGenerators(F)`

Given a p-adic field with fixed precision, this function computes generators for its multiplicative group.

`pSelmerGroup(p,F)`

Given a l-adic field F, return the p-Selmer group, i.e., the group F^*/F^{*p}, as an abstract group, as well as the map from F^* to the abstract group.

47.14.2 Norm Group

Given two p-adic field F/k the norm group of F in k, ie. the image of the norm map from F to k is the central object of local class field theory. Since the norm map will always operate on some multiplicative group, all functions in this section will take the map returned by `UnitGroup` as an argument as this then allows the convenient way of describing the norm group as a subgroup of some explicit finitely generated abelian group.

`NormGroup(R, m)`

Given a p-adic ring or field R extending S and a description of the unit group of S encoded by a map m from some abstract abelian group to S as computed by `UnitGroup`, compute the image of the norm map as a subgroup. The map returned is the embedding map returned form the subgroup constructor.

`NormEquation(R, m, b)`

Given a p-adic ring R defined over S, the unit group of S encoded by the map m as computed by `UnitGroup(S)` and some element $b \in S$, try to compute an element $a \in R$ such that the norm of a equals b. In case such an element exists, it is returned as a second value.

`NormEquation(m1, m2, G)`

Given two p-adic rings R and S and their unit groups U_R and U_S as parameterized by the maps $m_1 : U_R \to R$ and $U_S \to S$ as well as a subgroup $G < U_S$, compute the preimage of G under the norm map operating on the unit groups.

`Norm(m1, m2, G)`

Given two p-adic rings R and S and their unit groups U_R and U_S as parameterized by the maps $m_1 : U_R \to R$ and $U_S \to S$ as well as a subgroup $G < U_R$, compute the image of G under the norm map operating on the unit groups.

`NormKernel(m1, m2)`

Given two p-adic rings R and S and their unit groups U_R and U_S as parameterized by the maps $m_1 : U_R \to R$ and $U_S \to S$ compute the kernel of the norm map from U_R to U_S as a subgroup of U_R.

47.14.3 Class Fields

Class fields, that is abelian extensions are parameterized by their norm groups. Pauli, in [Pau06] gave explicit algorithms to solve the reverse problem of class field theory: given a suitable subgroup of some (abstractly given) multiplicatively group of some p-adic field, compute explicit defining equations for the class field.

`ClassField(m, G)`

Given a p-adic field S and its multiplicative group U_S specified by the map $m : U_S \to S$ and a suitable subgroup $G < U_S$, this function computes for each cyclic factor of U_S/G an explicit defining equation for the class field corresponding to this factor.

`NormGroupDiscriminant(m, G)`

Given a p-adic field S and its multiplicative group U_S specified by the map $m : U_S \to S$ and a suitable subgroup $G < U_S$, this function computes the valuation of the discriminant of the extension parameterized by G without computing explicit equations for it.

47.15 Extensions

It is a well known classical theorem that p-adic fields admit only finitely many different extensions of bounded degree (in contrast to number fields which have an infinite number of extensions of any degree). In his thesis, Pauli [Pau01a] developed explicit methods to enumerate those extensions.

`AllExtensions(R, n)`

`E`	RngIntElt	*Default :* 0
`F`	RngIntElt	*Default :* 0
`Galois`	BoolElt	*Default :* `false`
`D`	RngIntElt	*Default :* 0
`j`	RngIntElt	*Default :* -1

Given a p-adic ring or field R and some positive integer n, compute defining equations for all extensions of R of degree n. The optional parameters can be used to limit the extensions in various ways:

`E`	specifies the ramification index. 0 implies no restriction.
`F`	specifies the inertia degree, 0 implies no restriction.
`D`	specified the valuation of the discriminant, 0 implies no restriction.
`j`	specifies the valuation of the discriminant via the formula $D := d+j-1$ where d is the degree of R.
`Galois`	when set to `true`, limits the extensions to only list normal extensions.

`NumberOfExtensions(R, n)`

`E`	RngIntElt	*Default :* 0
`F`	RngIntElt	*Default :* 0
`Galois`	BoolElt	*Default :* `false`
`D`	RngIntElt	*Default :* 0
`j`	RngIntElt	*Default :* -1

Given a p-adic ring or field R and some positive integer n, compute the number of extensions of R of degree n. Similarly to the above function, the optional parameters can be used to impose restrictions on the fields returned.

`OreConditions(R, n, j)`

Given a p-adic ring or field R and positive integers n and j, test if there exists extensions of R of degree n with valuation of the discriminant being $n + j - 1$.

47.16 Bibliography

[**C$^+$05**] H. Cohen et al. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Discrete Mathematics and Its Applications. CRC Pr Llc, 2005.

[**Pau01a**] Sebastian Pauli. *Efficient Enumeration of Extensions of Local Fields with Bounded Discriminant.* PhD thesis, Concordia University, 2001.

[**Pau01b**] Sebastian Pauli. Factoring polynomials over local fields. *Journal of Symbolic Computation*, 32(5):533–547, 2001.

[**Pau06**] Sebastian Pauli. Constructing Class Fields over Local Fields. *J. Théorie des Nombres de Bordeaux*, 18(3):627–652, 2006.

48 GALOIS RINGS

Chapter 48
GALOIS RINGS

48.1 Introduction

MAGMA provides facilities for computing with Galois rings. The features are currently very basic, but advanced features will be available in the near future, including support for the creation of subrings and appropriate embeddings, allowing lattices of compatible embeddings, just as for finite fields.

A Galois ring R in MAGMA is considered as a finite algebraic extension of $\mathbf{Z}_{p^a}$ (where p is prime) by a monic polynomial $D \in \mathbf{Z}[x]$ which is irreducible modulo p. Thus R is presented as the polynomial quotient ring

$$\mathbf{Z}_{p^a}[x]/\langle D \rangle$$

and is usually written as $\mathbf{GR}(p^a, d)$, where d is the degree of D. The cardinality of R is easily seen to be p^{ad}.

R has a unique maximal ideal generated by p, and the quotient ring $R/\langle p \rangle$ is a finite field isomorphic to $\mathbf{Z}_p[x]/\langle D \rangle$, where D is here considered as a polynomial in $\mathbf{Z}_p[x]$ (the coefficients are reduced modulo p). This finite field is called the *residue field* of R. In the following, we will also call the integer residue ring $\mathbf{Z}_{p^a}$ the *base ring* of R, because this is the subring of R generated by 1 and we can think of R as an extension of $\mathbf{Z}_{p^a}$.

For an non-zero element x of R, the valuation of x is defined to be the largest power of p which divides the coefficients of x, where x is considered as a polynomial in $\mathbf{Z}_p[x]/\langle D \rangle$. x is a unit if and only if the valuation of x is zero.

Because of the valuation defined on them, Galois rings are Euclidean rings, so they may be used in MAGMA in any place where general Euclidean rings are valid. This includes many matrix and module functions, and the computation of Gröbner bases. Linear codes over Galois rings will be supported in the near future.

48.2 Creation Functions

48.2.1 Creation of Structures

`GaloisRing(q, d)`

`GR(q, d)`

Given integers q, $d \geq 1$, where $q = p^a$ for prime p and $a \geq 1$, create the default Galois ring $\mathbf{GR}(p^a, d)$. The defining polynomial used to construct the ring will be that used for $\mathbf{F}_{p^d}$, lifted to $\mathbf{Z}_{p^a}$. If p is very large, it is advised to use the next function instead, because MAGMA must first factor q completely.

The angle bracket notation can be used to assign names to the generator; e.g.: `R<w> := GaloisRing(2, 3)`.

`GaloisRing(p, a, d)`

`GR(p, a, d)`

`Check` BoolElt *Default :* `true`

Given a prime p and integers $a, d \geq 1$, create the default Galois ring $\mathbf{GR}(p^a, d)$. The defining polynomial used to construct the ring will be that used for $\mathbf{F}_{p^d}$, lifted to $\mathbf{Z}_{p^a}$.

By default p is checked to be a strong pseudoprime for 20 random bases b with $1 < b < p$; if the parameter `Check` is false, then no check is done on p at all (this is useful when p is very large and one does not wish to perform an expensive primality test on p).

`GaloisRing(q, D)`

`GR(q, D)`

Given an integer q, where $q = p^a$ for prime p and $a \geq 1$, and a monic polynomial D over $\mathbf{Z}$ such that D is irreducible mod p, create the Galois ring $R = \mathbf{GR}(p^a, D)$. The coefficients of D are reduced modulo p^a, and R is constructed to behave like the polynomial quotient ring $\mathbf{Z}_{p^a}[x]/\langle D \rangle$. If p is very large, it is advised to use the next function instead, because Magma must first factor q completely.

`GaloisRing(p, a, D)`

`GR(p, a, D)`

`Check` BoolElt *Default :* `true`

Given a prime p, an integer $a \geq 1$, and a monic polynomial D over $\mathbf{Z}$ such that D is irreducible mod p, create the Galois ring $R = \mathbf{GR}(p^a, D)$. The coefficients of D are reduced modulo p^a, and R is constructed to behave like the polynomial quotient ring $\mathbf{Z}_{p^a}[x]/\langle D \rangle$. The parameter `Check` is as above.

48.2.2 Names

`AssignNames(~R, [f])`

Procedure to change the name of the generating element in the Galois ring R to the contents of the string f. When R is created, the name will be `R.1`.

This procedure only changes the name used in printing the elements of R. It does *not* assign to an identifier called f the value of the generator in R; to do this, use an assignment statement, or use angle brackets when creating the ring.

Note that since this is a procedure that modifies R, it is necessary to have a reference `~R` to R in the call to this function.

`Name(R, 1)`

Given a Galois ring R, return the element which has the name attached to it, that is, return the element `R.1` of R.

48.2.3 Creation of Elements

`R . 1`

`Generator(R)`

The generator for R as an algebra over its base ring $\mathbf{Z}_{p^a}$. Thus, if R is viewed as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$, then `R.1` corresponds to x in this presentation.

`R ! a`

Given a Galois ring R create the element specified by a; here a is allowed to be an element coercible into R, which means that a may be

(i) An element of R;

(ii) An integer, to be identified with a modulo the characteristic p^a of R;

(iii) An element of the base ring $\mathbf{Z}_{p^a}$ of R, to be identified with the corresponding element of R.

(iv) A sequence of elements of the base ring $\mathbf{Z}_{p^a}$ of R. In this case the element $a_0 + a_1 w + \cdots + a_{n-1}w^{n-1}$ is created, where $a = [a_0, \ldots a_{n-1}]$ and w is the generator `R.1` of R over $\mathbf{Z}_{p^a}$.

`One(R)` `Identity(R)`

`Zero(R)` `Representative(R)`

These generic functions create 1, 1, 0, and 0 respectively, in any Galois ring.

`Random(R)`

Create a pseudo-random element of Galois ring R.

48.2.4 Sequence Conversions

`ElementToSequence(a)`

`Eltseq(a)`

Given an element a of the Galois ring R, return the sequence of coefficients $[a_0, \ldots, a_{n-1}]$ in the base ring $\mathbf{Z}_{p^a}$ of R (where R is $\mathbf{Z}_{p^a}[x]/\langle D\rangle$), such that $a = a_0 + a_1 w + \cdots + a_{n-1}w^{d-1}$, with w the generator of R, and d the degree of D.

Example H48E1

We can define the Galois ring $\mathbf{GR}(2^3, 2)$ using the default function:

```
> R<w> := GaloisRing(2^3, 2);
> R;
GaloisRing(2, 3, 2)
```

We note that R has characteristic 8 and that $w^2 + w + 1 = 0$ in R.

```
> R!8;
```

```
0
> R!9;
1
> w;
w
> 4*w;
4*w
> 4*w + 4*w;
0
> w^2;
7*w + 7
> w^2 + w + 1;
0
```

We can list all the elements of R by simply looping over R:

```
> [x: x in R];
[ 0, 1, 2, 3, 4, 5, 6, 7, w, w + 1, w + 2, w + 3, w + 4, w + 5, w + 6,
    w + 7, 2*w, 2*w + 1, 2*w + 2, 2*w + 3, 2*w + 4, 2*w + 5, 2*w + 6, 2*w
    + 7, 3*w, 3*w + 1, 3*w + 2, 3*w + 3, 3*w + 4, 3*w + 5, 3*w + 6,
    3*w + 7, 4*w, 4*w + 1, 4*w + 2, 4*w + 3, 4*w + 4, 4*w + 5, 4*w +
    6, 4*w + 7, 5*w, 5*w + 1, 5*w + 2, 5*w + 3, 5*w + 4, 5*w + 5, 5*w
    + 6, 5*w + 7, 6*w, 6*w + 1, 6*w + 2, 6*w + 3, 6*w + 4, 6*w + 5,
    6*w + 6, 6*w + 7, 7*w, 7*w + 1, 7*w + 2, 7*w + 3, 7*w + 4, 7*w +
    5, 7*w + 6, 7*w + 7 ]
```

We see that the elements of R can be considered as polynomials of degree at most 1, with coefficients in the range $\{0 \dots 7\}$.
We can easily create elements of R also using the `!` operator, and use the `Eltseq` function to recover the corresponding sequence of coefficients.

```
> R ! [1, 2];
2*w + 1
> Eltseq(2*w + 1);
[ 1, 2 ]
> Eltseq(w);
[ 0, 1 ]
```

48.3 Structure Operations

48.3.1 Related Structures

`Category(R)` `Parent(R)` `Centre(R)`
`PrimeRing(R)` `PrimeField(R)`
`FieldOfFractions(R)`

48.3.2 Numerical Invariants

`Characteristic(R)`

The characteristic of R, which is p^a where R is considered as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$.

`#R`

The cardinality of R, which is p^{ad} where R is considered as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$ and d is the degree of D.

`Degree(R)`

The degree of R, which is the degree of D, where R is considered as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$.

`ResidueField(R)`

The residue field of R, which is the finite field $\mathbf{Z}_p[x]/\langle D\rangle$, where R is considered as $\mathbf{Z}_{p^a}[x]/\langle D\rangle$.

48.3.3 Ring Predicates and Booleans

The following functions are described for rings in general in Section 17.4.3.

`IsCommutative(R)` `IsUnitary(R)`
`IsFinite(R)` `IsOrdered(R)`
`IsField(R)` `IsEuclideanDomain(R)`
`IsPID(R)` `IsUFD(R)`
`IsDivisionRing(R)` `IsEuclideanRing(R)`
`IsPrincipalIdealRing(R)` `IsDomain(R)`
`R eq G` `R ne G`

48.4 Element Operations

See also Section 17.5.

48.4.1 Arithmetic Operators

`+ a` `- a`
`a + b` `a - b` `a * b` `a ^ k`
`a +:= b` `a -:= b` `a *:= b`

48.4.2 Euclidean Operations

a div b | a mod b | Quotrem(a, b)
GCD(a, b) | LCM(a, b) | XGCD(a, b)

48.4.3 Equality and Membership

a eq b | a ne b
a in R | a notin R

48.4.4 Parent and Category

Parent(a) | Category(a)

48.4.5 Predicates on Ring Elements

IsZero(a) | IsOne(a) | IsMinusOne(a)
IsNilpotent(a) | IsIdempotent(a)
IsUnit(a) | IsZeroDivisor(a)

Example H48E2

This simple example shows how one can compute Gröbner bases over Galois rings. We create an ideal in 3 variables over a Galois ring and compute its Gröbner basis.

```
> R<w> := GaloisRing(3, 3, 2);
> #R;
729
> P<x,y,z> := PolynomialRing(R, 3);
> I := ideal<P | [x^2 - w*y, 3*y^3 - 3*w*x*z, 9*z^5 - 9*w]>;
> GroebnerBasis(I);
[
    x^2 + 26*w*y,
    3*x*y^3 + (6*w + 6)*y*z,
    3*x*z + (15*w + 3)*y^3,
    9*x + (9*w + 9)*y^3*z^4,
    3*y^6 + (21*w + 15)*y*z^2,
    9*z^5 + 18*w
]
```

Notice that the leading coefficients include 3 and 9, which are not units in the ring. See Chapter 105 for much more information on such Gröbner bases.

49 POWER, LAURENT AND PUISEUX SERIES

Chapter 49
POWER, LAURENT AND PUISEUX SERIES

49.1 Introduction

This chapter describes the operations on formal power, Laurent and Puiseux series available in MAGMA.

49.1.1 Kinds of Series

Internally MAGMA has only one kind of formal series, where general fractional exponents are allowed, but externally there are three kinds of series: power, Laurent and Puiseux. Power series must have a non-negative integral valuation and integral exponents, Laurent series must have an integral valuation and integral exponents (possibly negative), while Puiseux series may have a general rational valuation and general rational exponents. The main reason for the three kinds is simply to supply error checking for operations illegal to a specific kind, so the user will not move into the next kind of exponents inadvertently. Apart from these error checks, there is no difference at all between the different kinds of series, so one can simply use Puiseux series always if the restrictions on exponents are not desired, and this will cause no loss of efficiency or functionality at all even if all exponents remain integral or integral and non-negative.

In MAGMA, the set of all power series over a given base ring R is called a power series ring (denoted by $R[[x]]$) and the category names are `RngSerPow` for the ring and `RngSerPowElt` for the elements. The set of all Laurent series over a given base ring R is called a Laurent series ring (denoted by $R((x))$) and the category names are `RngSerLaur` for the ring and `RngSerLaurElt` for the elements. Finally, the set of all Puiseux series over a given base ring R is called a Puiseux series ring (denoted by $R\langle\langle x\rangle\rangle$) and the category names are `RngSerPuis` for the ring and `RngSerPuisElt` for the elements. For the rest of this chapter, the term "series ring" refers to any of the above rings and the corresponding virtual category name is `RngSer`; the term "series" refers to any of the above series kinds and the corresponding virtual category name is `RngSerElt`.

49.1.2 Puiseux Series

Puiseux series in a variable x are often mathematically defined to be Laurent series in another variable y say, where $y = x^{1/d}$, for a fixed positive integer d; this d is usually fixed for all the series under consideration. MAGMA is more general, in that although each series is internally represented in this way (i.e., its valuation, exponents and precision are thought to be divided by a single denominator associated with it), different Puiseux series may have different exponent denominators and may be freely mixed (for example, in addition, where the exponent denominator of the result will be derived from that of the arguments). Thus there is no restriction whatsoever to a fixed exponent denominator for a given Puiseux series ring.

49.1.3 Representation of Series

Formal series are stored in truncated form, unless only finitely many terms are non-zero. If a series has precision p, that means that its coefficients are unknown from exponent p onwards. This truncated form is similar to the floating point representation of real numbers except for one significant difference: because for series terms do not "carry" in arithmetic operations like they do in integer and real arithmetic, error propagation common in floating point methods does not occur. Consequently, given a result of any sequence of arithmetic operations with formal series, the coefficients of the known terms (the ones up to the given precision) will always be *exactly correct*, with no error (assuming the coefficient ring has exact arithmetic for its elements of course).

Elements

$$c_v x^v + c_{v+1} x^{v+1} + \ldots$$

(with $v \in \mathbf{Q}$ and $r_v \neq 0$) of series ring over a commutative ring R are stored in the form of approximations

$$c_v x^v + c_{v+1} x^{v+1} + \ldots + O(x^p)$$

to a certain *precision* $p \geq v$. The $O(x^p)$ notation is used to refer to terms of degree at least the precision p in x. For Laurent series v must be an integer and for power series v must be a non-negative integer. Note that for Puiseux series the above element is actually internally stored in the form

$$c_w x^{w/d} + c_{w+1} x^{(w+1)/d} + \ldots + O(x^{q/d})$$

where $v = w/d$ and $p = q/d$ and d (the exponent denominator) is minimal.

49.1.4 Precision

Associated with any series there are two types of precision: the *absolute precision* and the *relative precision*. For the element

$$c_v x^v + c_{v+1} x^{v+1} + \cdots + O(x^p)$$

(with $v \in \mathbf{Q}$ and $r_v \neq 0$), the absolute precision is p and the relative precision is $p - v$.

The absolute precision indicates the largest known term, and the relative precision indicates the size of the range over which terms are known. The two types of precision and the *valuation* of the series, which equals v in the above, are therefore always related via $p = v + r$.

49.1.5 Free and Fixed Precision

For each type of series ring, there are two sub-kinds: a *free precision* ring (the usual case), where elements of the ring have arbitrary precision, and a *fixed precision* ring, where all elements of the ring have a fixed precision. In the latter case, for power series rings the fixed precision is absolute, while for Laurent and Puiseux series ring the fixed precision is relative.

The free precision rings most closely resemble the mathematical objects $R[[x]]$ and $R((x))$; elements in these free rings and fields carry their own precision with them. Operations usually return results to a precision that is maximal given the input (and the nature of the operation). Operations which are given infinite precision series but which must return finite precision series (e.g., division) return series whose precision is the *default* precision for the series ring (this is different from the fixed precision of fixed precision rings); the default precision is 20 by default but may be set to another value at creation by a parameter (see below). Elements of free structures that have finite series expansion (i.e., polynomials) can be created and stored exactly, with infinite (absolute and relative) precision. Also note that the relative precision will be 0 for approximations to 0.

The structures with fixed precision, which we will sometimes refer to as *quotient* structures, behave differently. All elements in a *power* series ring of fixed precision have the same fixed *absolute* precision p, and the relative precision may be anything from 0 to p. This means that the ring with fixed precision p behaves like a quotient of a polynomial ring, $R[x]/x^p$. All elements in a *Laurent* or *Puiseux* series ring of fixed precision have the same fixed *relative* precision; the only exception to this rule is that 0 in the ring is stored as zero with infinite absolute precision. The absolute precision of elements in a free Laurent or Puiseux series ring can be anything.

49.1.6 Equality

Since a given series ring will contain series truncated at arbitrary precision, care has to be taken as to the meaning of equality of two elements. Two series are considered equal iff they are identical (the equality operator `eq` follows this convention). But we call two series A and B in the same ring *weakly equal* if and only if their coefficients are the same whenever both are defined, that is, if and only if for every $n \leq p$ the coefficients A_n and B_n are equal, where p is the minimum of the precisions of A and B. Thus, for example, $A = 3+x+O(x^2)$ and $B = 3+x+17x^2+O(x^4)$ are the same, but $C = 3+x+17x^2+x^3+O(x^4)$ is different from B. Note that A and C are equal under this definition, and hence weak equality is not transitive!

49.1.7 Polynomials over Series Rings

For a discussion of operations for polynomials over series rings see Chapter 46 and Section 49.7.

49.2 Creation Functions

49.2.1 Creation of Structures

`PowerSeriesRing(R)`

`PowerSeriesRing(R, p)`

`Global`	BoolElt	*Default :* `true`
`Precision`	RngIntElt	*Default :* 20

Given a commutative ring R, create the ring $R[[x]]$ of formal power series over R. If a second integer argument p is given, the resulting ring is a fixed precision series ring with fixed precision p; otherwise the resulting ring is a free precision series ring and the optional argument `Precision` may be used to set the default precision for elements of the power series ring (it will be 20 otherwise; see the section above on free and fixed precision). By default, a global series ring will be returned; if the parameter `Global` is set to `false`, a non-global series ring will be returned (to which a separate name for the indeterminate can be assigned). The angle bracket notation can be used to assign a name to the indeterminate: `S<x> := PowerSeriesRing(R)`.

`LaurentSeriesRing(R)`

`LaurentSeriesRing(R, p)`

`Global`	BoolElt	*Default :* `true`
`Precision`	RngIntElt	*Default :* 20

Given a commutative ring R, create the ring $R((x))$ of formal Laurent series over R. If a second integer argument p is given, the resulting ring is a fixed precision series ring with fixed precision p; otherwise the resulting ring is a free precision series ring and the optional argument `Precision` may be used to set the default precision for elements of the power series ring (it will be 20 otherwise; see the section above on free and fixed precision). By default, a global series ring will be returned; if the parameter `Global` is set to `false`, a non-global series ring will be returned (to which a separate name for the indeterminate can be assigned). The angle bracket notation can be used to assign a name to the indeterminate: `S<x> := LaurentSeriesRing(R)`.

`PuiseuxSeriesRing(R)`

`PuiseuxSeriesRing(R, p)`

`Global`	BoolElt	*Default :* `true`
`Precision`	RngIntElt	*Default :* 20

Given a commutative ring R, create the ring $R\langle\langle x\rangle\rangle$ of formal Puiseux series over R. If a second integer argument p is given, the resulting ring is a fixed precision series ring with fixed precision p; otherwise the resulting ring is a free precision series ring and the optional argument `Precision` may be used to set the default precision for elements of the power series ring (it will be 20 otherwise; see the section above on free and fixed precision). The optional argument `Precision` may be used to set the default precision for elements of the power series ring. By default, a global series ring will be returned; if the parameter `Global` is set to `false`, a non-global series ring will be returned (to which a separate name for the indeterminate can be assigned). The angle bracket notation can be used to assign a name to the indeterminate: `S<x> := PuiseuxSeriesRing(R)`.

Example H49E1

We demonstrate the difference between global and non-global rings. We first create the global power series ring over **Q** twice.

```
> Q := RationalField();
> P<x> := PowerSeriesRing(Q);
> PP := PowerSeriesRing(Q);
> P;
Power series ring in x over Rational Field
> PP;
Power series ring in x over Rational Field
> PP.1;
x
```

PP is identical to P. We now create non-global series rings (which are also different to the global series ring). Note that elements of all the rings are mathematically equal by automatic coercion.

```
> Pa<a> := PowerSeriesRing(Q: Global := false);
> Pb<b> := PowerSeriesRing(Q: Global := false);
> Pa;
Power series ring in a over Rational Field
> Pb;
Power series ring in b over Rational Field
> a;
a
> b;
b
> P;
Power series ring in x over Rational Field
> x;
x
> x eq a; // Automatic coercion
true
> x + a;
2*x
```

49.2.2 Special Options

`AssertAttribute(S,"DefaultPrecision", n)`

Procedure to change the default precision on a free series ring series S; the default precision will be set to n, which must be a non-negative integer.

`HasAttribute(S, "DefaultPrecision")`

Function that returns a Boolean indicating whether a default precision has been set on the free series ring S (which will always be `true`), as well as its (non-negative) integer value (which is 20 by default).

```
AssignNames(~S, ["x"])
```

Procedure to change the name of the 'indeterminate' transcendental element generating the series ring or field S; the name (used in printing elements of S) is changed to the string x. Note that no assignment to the identifier x is made (so x cannot be used for the specification of elements of S without further assignment).

```
Name(S, 1)
```

```
S . 1
```

Return the element of the series ring or field with a name attached to it, that is, return the 'indeterminate' transcendental element generating S over its coefficient ring.

49.2.3 Creation of Elements

The easiest way to create power and Laurent series in a given ring is to use the angle bracket construction to attach names to the indeterminate, and to use these names to express the series (see the examples). Below we list other options.

```
R . 1
```

```
UniformizingElement(R)
```

Return the generator (indeterminate) for the series ring R.

```
elt< R | v, [ a_1, ..., a_d], p >
```

Given a series ring R, integers v and p (where $p > 0$ or $p = \infty$), and a sequence $a = [a_1, \ldots, a_d]$ of elements of R, create the element in R with valuation v, known coefficients given by a and relative precision p. That is, this function returns the series $a_1x^v + \cdots + a_dx^{v+d-1} + O(x^{v+p})$, or, if $p = -1$, the exact series $a_1x^v + \cdots + a_dx^{v+d-1}$. If R is a power series ring, then v must be non-negative.

The integer v or the integer p or both may be omitted. If v is omitted, it will be set to zero by default; if p is omitted it will be taken to be $v + d$, where d is the length of the sequence a.

```
R ! s
```

Coerce s into the series ring R. Here s is allowed to be a sequence of elements from (or coercible into) the coefficient ring of R, or just an element from (or coercible into) R. A sequence $[a_1, \ldots, a_d]$ is converted into the series $a_1+a_2x^1+\cdots+a_dx^{d-1}+O(x^d)$.

`BigO(f)`

`O(f)`

Create the series $O(x^v)$ where x is the generator of the parent of f and v is the valuation of f. The most typical usage of this function is the expression $O(x^n)$ where x is the generator of a series ring, but a general series f is actually allowed.

`One(Q)` `Identity(Q)`

`Zero(Q)` `Representative(Q)`

49.3 Structure Operations

49.3.1 Related Structures

`Parent(R)` `Category(R)`

`BaseRing(R)`

`CoefficientRing(R)`

Return the coefficient ring of the series ring R.

`IntegerRing(R)`

`Integers(R)`

`RingOfIntegers(R)`

Return the power series ring which is the integer ring of the laurent series ring R.

`FieldOfFractions(R)`

Return the laurent series ring which is the field of fractions of the series ring R.

`ChangePrecision(R, r)`

`ChangePrecision(~R, r)`

Return a series ring identical to the series ring R but having precision r.

`ChangeRing(R, C)`

Return the series ring identical to the series ring R but having coefficient ring C.

`ResidueClassField(R)`

Return the residue class field of the series ring R (which will be the same as the coefficient ring of R) and the map from R into the residue class field.

49.3.2 Invariants

`Characteristic(R)`

`Precision(R)`
`GetPrecision(R)`

Return the precision of the fixed precision series ring R. If R is a fixed precision power series ring, then this is the fixed absolute precision for all elements of the ring. If R is a fixed precision Laurent series ring, then this is the maximum relative precision for all elements of the ring.

49.3.3 Ring Predicates and Booleans

`IsCommutative(Q)` `IsUnitary(Q)`
`IsFinite(Q)` `IsOrdered(Q)`
`IsField(Q)` `IsEuclideanDomain(Q)`
`IsPID(Q)` `IsUFD(Q)`
`IsDivisionRing(Q)` `IsEuclideanRing(Q)`
`IsPrincipalIdealRing(Q)` `IsDomain(Q)`
`R eq S` `R ne S`

49.4 Basic Element Operations

49.4.1 Parent and Category

`Parent(r)` `Category(r)`

49.4.2 Arithmetic Operators

`+ b` `- b`
`a + b` `a - b` `a * b` `a ^ k`
`a div b` `a / b`

49.4.3 Equality and Membership

`a eq b` `a ne b`
`a in R` `a notin R`

49.4.4 Predicates on Ring Elements

Note the definition of equality in the introduction to this Chapter. This not only affects the result of the application of `eq` and `ne`, but also that of `IsOne, IsZero` and `IsMinusOne`.

`IsZero(a)` `IsOne(a)` `IsMinusOne(a)`
`IsNilpotent(x)` `IsIdempotent(x)`
`IsUnit(a)` `IsZeroDivisor(x)` `IsRegular(x)`
`IsIrreducible(x)` `IsPrime(x)`

`IsWeaklyZero(f)`

Given a series f, return whether f is weakly zero, which is whether f is exactly zero or of the form $O(x^p)$ for some p.

`IsWeaklyEqual(f, g)`

Given series f and g, return whether f is weakly equal to g, which is whether $(f-g)$ is weakly zero (see `IsWeaklyZero`).

`IsIdentical(f, g)`

Given series f and g, return whether f is identical to g, which is whether f and g have exactly the same valuation, precision, and coefficients.

49.4.5 Precision

`AbsolutePrecision(f)`

Given a series f, this returns the absolute precision that is stored with f. If f is a series in x, the absolute precision of f is the exponent p such that x^p is the first term of f of which the coefficient is not known, that is, it is the least p such that $f \in O(x^p)$. If f is known exactly (in a free ring), the absolute precision is infinite and an error occurs. Note that the absolute precision may be a non-integral rational number if f is a Puiseux series.

`RelativePrecision(f)`

Given a series f, this returns the relative precision that is stored with f. The relative precision counts the number of coefficients of f that is known, starting at the first non-zero term. Hence the relative precision is the difference between the absolute precision and the valuation of f, and is therefore always non-negative; however, if f is exact, the relative precision is infinite and the value ∞ is returned. Note that the relative precision may be a non-integral rational number if f is a Puiseux series.

`ChangePrecision(f, r)`

`ChangePrecision(~f, r)`

The (non puiseux) series f with absolute precision r (which can be positive infinity).

49.4.6 Coefficients and Degree

`Coefficients(f)`

`ElementToSequence(f)`

`Eltseq(f)`

Let f be a series with coefficients in a ring R and with indeterminate x. This function returns the sequence Q of coefficients of f, the unscaled valuation v and the exponent denominator d of f (v is the true valuation of f multiplied by d). The i-th entry $Q[i]$ of Q equals the coefficient of

$$x^{\frac{v+i-1}{d}}$$

in f. Thus the first entry of Q is the 'first' (lowest order) non-zero coefficient of f, i.e., the coefficient of x^w where w is the true valuation of f.

`Coefficient(f, i)`

Given a series f with coefficients in a ring R, and a rational or integer i, return the coefficient of the i-th power of the indeterminate x of f as an element of R. If f is a Puiseux series i may be a (non-integral) rational; otherwise i must be an integer (and also must be non-negative if f is a power series). Also, i must be less than p, the precision of f.

`LeadingCoefficient(f)`

Given a series f with coefficients in a ring R, return the leading coefficient of f as an element of R, which is the first non-zero coefficient of f (i.e., the coefficient x^v in f, where x is the indeterminate of f and v is the valuation of f).

`LeadingTerm(f)`

Given a series f with coefficients in a ring R, return the leading term of f, which is the first non-zero term of f (i.e., the term of f whose monomial is x^v, where x is the indeterminate of f and v is the valuation of f).

`Truncate(f)`

Given a series f, return the exact series obtained by truncating f after the last known non-zero coefficient.

`ExponentDenominator(f)`

Given a series f, return the exponent denominator of f, i.e., the lowest common denominator of all the exponents of the non-zero terms of f (always an integer). For power series and Laurent series, this will always be 1 of course.

`Degree(f)`

Given a series f, return the degree of the truncation of f, that is, the exponent of the last known non-zero term. Note that this may be a non-integral rational number if f is a Puiseux series.

`Valuation(f)`

Given a series f, return the smallest integer v (possibly negative for Laurent series) such that the coefficient of x^v in f is not known to be zero. For the exact 0 element (in a free ring), the valuation is ∞. Note that the valuation may be a non-integral rational number if f is a Puiseux series.

`ExponentDenominator(f)`

The exponent denominator of the series f. This is the lowest common denominator of the exponents of the non-zero terms of f.

49.4.7 Evaluation and Derivative

`Derivative(f)`

Given a series $f \in R$, return the derivative of f with respect to its indeterminate, as an element of R. Note that the precision decreases by 1 (unless f has infinite precision).

`Derivative(f, n)`

Given a series $f \in R$ and an integer $n > 0$, return the n-th derivative of f with respect to its indeterminate, as an element of R. Note that the precision decreases by n (unless f has infinite precision).

`Integral(f)`

Given a series $f \in R$, return an anti-derivative F of f with respect to its indeterminate, which is an element of R which has derivative f. The coefficient of x^{-1} in f must be zero. Note that the precision of F will be exceeding that of f by 1 (unless f has infinite precision).

`Evaluate(f, s)`

Given an element f of a series ring over the coefficient ring R, and an element s of the ring S, return the value of $f(s)$ when the indeterminate x is evaluated at s. The result will be an element of the common overstructure over R and S.

`Laplace(f)`

The Laplace transform of the series f; if f has expansion $\sum_{i\geq 0} a_i x^i$, its Laplace transform has expansion $\sum_{i\geq 0}(i!a_i)x^i$. The valuation of f must be integral and non-negative.

49.4.8 Square Root

`SquareRoot(f)`

`Sqrt(f)`

Return the square root of the series f, f must have even valuation if it is a power or Laurent series.

49.4.9 Composition and Reversion

`Composition(f, g)`

Given elements f and g from the same series ring P, return their composition, defined by

$$f \circ g = \sum_{i<p} f_i(g^i),$$

where $f = \sum_{i<p} f_i x^i$.

`Reversion(f)`

`Reverse(f)`

Given a series f (in x, say), this returns the inverse of f under composition, that is, an element g of the same power series ring such that its composition with f equals x to the best possible precision. If f is a power or Laurent series, the valuation of f must be 1. If f is a Puiseux series, the valuation of f must be positive (but need not equal 1), and if the valuation of f is not 1, the leading coefficient of f must be 1.

`Convolution(f, g)`

Given elements f and g from the same series ring P, return their convolution $f * g$, defined by

$$f * g = \sum_{i<\min(p,q)} f_i g_i x^i,$$

where $f = \sum_{i<p} f_i x^i + O(x^p)$ and $g = \sum_{i<q} g_i x^i + O(x^q)$.

Example H49E2

We demonstrate the functions `Composition` and `Reversion`. First we check that `Arcsin` is the reversion of `Sin`.

```
> S<x> := PowerSeriesRing(RationalField());
> f := Sin(x);
> g := Arcsin(x);
> f;
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 -
    1/39916800*x^11 + 1/6227020800*x^13 - 1/1307674368000*x^15 +
    1/355687428096000*x^17 - 1/121645100408832000*x^19 + O(x^21)
> g;
x + 1/6*x^3 + 3/40*x^5 + 5/112*x^7 + 35/1152*x^9 + 63/2816*x^11 +
    231/13312*x^13 + 143/10240*x^15 + 6435/557056*x^17 +
    12155/1245184*x^19 + O(x^21)
> Composition(f, g);
x + O(x^21)
> Composition(g, f);
x + O(x^21)
> Reversion(f) - g;
O(x^21)
> Reversion(g) - f;
O(x^21)
```

Next we compute the reversion of a series whose valuation is not 1.

```
> S<x> := PuiseuxSeriesRing(RationalField());
> f := x^3 - x^5 + 2*x^8;
> r := Reversion(f);
> f;
x^3 - x^5 + 2*x^8
> r;
x^(1/3) + 1/3*x + 4/9*x^(5/3) - 2/3*x^2 + 65/81*x^(7/3) -
    22/9*x^(8/3) + 5/3*x^3 - 208/27*x^(10/3) + 5005/729*x^(11/3)
    - 70/3*x^4 + 206264/6561*x^(13/3) - 50830/729*x^(14/3) +
    134*x^5 - 498674/2187*x^(16/3) + 31389020/59049*x^(17/3) +
    O(x^6)
> Composition(r, f);
x + O(x^18)
> Composition(f, r);
x + O(x^(20/3))
```

Finally we compute the reversion of a proper Puiseux series.

```
> f := x^(2/5) - x^(2/3) + x^(3/2) + O(x^2);
> r := Reversion(f);
> r;
x^(5/2) + 5/2*x^(19/6) + 145/24*x^(23/6) + 715/48*x^(9/2) +
    389795/10368*x^(31/6) - 5/2*x^(21/4) + O(x^(11/2))
> Composition(f, r);
```

```
x + O(x^4)
> Composition(r, f);
x + O(x^(11/5))
```

49.5 Transcendental Functions

In each of the functions below, the precision of the result will be approximately equal to the precision of the argument if that is finite; otherwise it will be approximately equal to the default precision of the parent of the argument. An error will result if the coefficient ring of the series is not a field. If the first argument has a non-zero constant term, an error will result unless the coefficient ring of the parent is a real or complex domain (so that the transcendental function can be evaluated in the constant term).

See also the chapter on real and complex fields for elliptic and modular functions which are also defined for formal series.

49.5.1 Exponential and Logarithmic Functions

`Exp(f)`

Given a series f defined over a field, return the exponential of f.

`Log(f)`

Given a series f defined over a field of characteristic zero, return the logarithm of f. The valuation of f must be zero.

Example H49E3

In this example we show how one can compute the Bernoulli number B_n for given n using generating functions. The function `BernoulliNumber` now actually uses this method (since V2.4). Using this method, the Bernoulli number B_{10000} has been computed, taking about 14 hours on a 250MHz Sun Ultrasparc (the computation depends on the new asymptotically fast methods for series division).

The exponential generating function for the Bernoulli numbers is:

$$E(x) = \frac{x}{e^x - 1}.$$

This means that the n-th Bernoulli number B_n is $n!$ times the coefficient of x^n in $E(x)$. The Bernoulli numbers $B_0, \ldots, B_n$ for any n can thus be calculated by computing the above power series and scaling the coefficients.

In this example we will compute B_{500}. We first set the final coefficient index n we desire to be 500. We then create the denominator $D = e^x - 1$ of the exponential generating function to precision $n+2$ (we need $n+2$ since we lose precision when we divide by the denominator and the valuation changes). For each series we create, we print the sum of it and $O(x^{20})$, which will only print the terms up to x^{19}.

```
> n := 500;
```

```
> S<x> := LaurentSeriesRing(RationalField(), n + 2);
> time D := Exp(x) - 1;
Time: 0.040
> D + O(x^20);
x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + 1/720*x^6 + 1/5040*x^7 +
    1/40320*x^8 + 1/362880*x^9 + 1/3628800*x^10 + 1/39916800*x^11 +
    1/479001600*x^12 + 1/6227020800*x^13 + 1/87178291200*x^14 +
    1/1307674368000*x^15 + 1/20922789888000*x^16 + 1/355687428096000*x^17 +
    1/6402373705728000*x^18 + 1/121645100408832000*x^19 + O(x^20)
```

We then form the quotient $E = x/D$ which gives the exponential generating function.

```
> time E := x / D;
Time: 5.330
> E + O(x^20);
1 - 1/2*x + 1/12*x^2 - 1/720*x^4 + 1/30240*x^6 - 1/1209600*x^8 +
    1/47900160*x^10 - 691/1307674368000*x^12 + 1/74724249600*x^14 -
    3617/10670622842880000*x^16 + 43867/5109094217170944000*x^18 + O(x^20)
```

We finally compute the Laplace transform of E (which multiplies the coefficient of x^i by $i!$) to yield the generating function of the Bernoulli numbers up to the x^n term. Thus the coefficient of x^n here is B_n.

```
> time B := Laplace(E);
Time: 0.289
> B + O(x^20);
1 - 1/2*x + 1/6*x^2 - 1/30*x^4 + 1/42*x^6 - 1/30*x^8 + 5/66*x^10 -
    691/2730*x^12 + 7/6*x^14 - 3617/510*x^16 + 43867/798*x^18 + O(x^20)
> Coefficient(B, n);
-1659638064056855722985212308807713420665866430280667189235265099315533164122\
0960084014956088135770921465025323942809207851857992860213463783252745409096420\
9325099531654667356754859790348176199837272098442910819081455978296749801598899\
9762442406337466011207033006983290297104826000697178669172291137497976329300339\
5597947178384074157727965044194649323374986427142260817436887069719900107342629\
0768812383228675592757482195884044880230345282960230516388584671851732024838889\
7943427208374137376444107655632132200434773968878128912429523363013448081657579\
9421098878036925794394279735614878635245562568694033843064339220490783007204809\
3617576807141980442305220157754752870753156688862999789581507566774171800043629\
9814543966136466123270197841417404998354619/8365830
> n;
500
```

49.5.2 Trigonometric Functions and their Inverses

`Sin(f)`

Given a series f defined over a field of characteristic zero, return the sine of f. The valuation of f must be zero.

`Cos(f)`

Given a series f defined over a field of characteristic zero, return the cosine of f. The valuation of f must be zero.

`Sincos(f)`

Given a series f defined over a field of characteristic zero, return both the sine and cosine of f. The valuation of f must be zero.

`Tan(f)`

Return the tangent of the series f defined over a field.

`Arcsin(f)`

Given a series f defined over a field of characteristic zero, return the inverse sine of f.

`Arccos(f)`

Given a series f defined over the real or complex field, return the inverse cosine of f.

`Arctan(f)`

Given a series f defined over a field of characteristic zero, return the inverse tangent of f.

49.5.3 Hyperbolic Functions and their Inverses

`Sinh(f)`

Given a series f defined over a field, return the hyperbolic sine of f.

`Cosh(f)`

Given a series f defined over a field, return the hyperbolic cosine of f.

`Tanh(f)`

Given a series f defined over a field, return the hyperbolic tangent of f.

`Argsinh(f)`

Given a series f defined over a field of characteristic zero, return the inverse hyperbolic sine of f.

`Argcosh(f)`

Given a series f defined over the real or complex field, return the inverse hyperbolic cosine of f.

`Argtanh(f)`

Given a series f defined over a field of characteristic zero, return the inverse hyperbolic tangent of f.

49.6 The Hypergeometric Series

For more information on the Hypergeometric Series, see [Hus87], page 176.

`HypergeometricSeries(a,b,c, z)`

Return the hypergeometric series $F(a, b, c; z)$ defined by

$$F(a, b, c; z) = \sum_{0 \leq n} \frac{(a)_n (b)_n}{n!(c)_n z^n}$$

where $(a)_n = a(a + 1) \cdots (a + n - 1)$.

49.7 Polynomials over Series Rings

Factorization is available for polynomials over series rings defined over finite fields. We recommend constructing polynomials from sequences rather than by addition of terms, especially over fields, to avoid some precision loss. For example, $x^4 + t$ will not have full precision in the constant coefficient as there will be a O term in the constant coefficient of x^4 which will reduce its precision as a field element. The equivalent polynomial `Polynomial([t, 0, 0, 0, 1])` will have more precision than that constructed as $x^4 + t$.

`HenselLift(f, L)`

Given a polynomial f over a series ring over a finite field or an extension of a series ring and a factorization L of f to precision 1 return a factorization of f known to the full precision of the coefficient ring of f.

`Factorization(f)`

`Certificates`	BoolElt	*Default :* `false`
`Ideals`	BoolElt	*Default :* `false`
`Extensions`	BoolElt	*Default :* `false`

The factorization of the polynomial f over a power series ring, laurent series field over a finite field or an extension of either into irreducibles.

If `Certificates` is set to `true`, a two-element certificate for each factor, proving its irreducibility, is returned.

If `Ideals` is set to `true`, two generators of some ideal for each factor are returned within the certificates.

If `Extensions` is set to `true`, an extension for each factor is returned within the certificates.

Example H49E4

We illustrate factorizations over series rings and the extensions which can be gained through them.

```
> P<t> := PowerSeriesRing(GF(101));
> R<x> := PolynomialRing(P);
> Factorization(x^5 + t*x^4 - t^2*x^3 + (1 + t^20)*x^2 + t*x + t^6);
[
    <(1 + O(t^20))*x + t^5 + O(t^8), 1>,
    <(1 + O(t^20))*x + t + 100*t^4 + 100*t^5 + t^7 + O(t^8), 1>,
    <(1 + O(t^20))*x + 1 + 34*t^2 + 34*t^3 + 34*t^4 + 90*t^5 + 29*t^6 + 16*t^8 +
        32*t^9 + 6*t^10 + 66*t^11 + 41*t^12 + 93*t^13 + t^14 + 69*t^15 + 8*t^16
        + 61*t^17 + 86*t^18 + 19*t^19 + O(t^20), 1>,
    <(1 + O(t^20))*x^2 + (100 + 67*t^2 + 67*t^3 + 68*t^4 + 11*t^5 + 72*t^6 +
        100*t^7 + 85*t^8 + 69*t^9 + 94*t^10 + 31*t^11 + 59*t^12 + 16*t^13 +
        14*t^14 + 35*t^15 + 77*t^16 + 28*t^17 + 33*t^18 + 72*t^19 + O(t^20))*x +
        1 + 67*t^2 + 68*t^3 + 11*t^4 + 67*t^5 + 57*t^6 + 16*t^7 + 57*t^8 +
        21*t^9 + 68*t^10 + 68*t^11 + 61*t^12 + 98*t^13 + 4*t^14 + 21*t^15 +
        7*t^16 + 95*t^17 + 23*t^18 + 76*t^19 + O(t^20), 1>
]
1 + O(t^20)
> P<t> := PowerSeriesRing(GF(101), 50);
> R<x> := PolynomialRing(P);
> Factorization(x^5 + t*x^4 - t^2*x^3 + (1 + t^20)*x^2 + t*x + t^6 :
> Extensions);
[
    <x + t^5 + t^9 + 2*t^13 + t^16 + 5*t^17 + 6*t^20 + 14*t^21 + 26*t^24 +
        42*t^25 + 3*t^27 + 4*t^28 + 32*t^29 + 35*t^31 + 10*t^32 + 29*t^33 +
        46*t^35 + t^36 + 31*t^37 + O(t^38), 1>,
    <x + t + 100*t^4 + 100*t^5 + t^7 + 100*t^9 + t^10 + 4*t^11 + t^12 + 91*t^13
        + 86*t^14 + 98*t^15 + 15*t^16 + 7*t^17 + 83*t^18 + 10*t^19 + 23*t^20 +
        54*t^21 + 47*t^22 + 72*t^23 + 87*t^24 + 55*t^25 + 4*t^26 + 15*t^27 +
        90*t^28 + 82*t^29 + 97*t^30 + 15*t^31 + 23*t^32 + 86*t^33 + 57*t^34 +
        10*t^35 + 79*t^36 + 52*t^37 + O(t^38), 1>,
    <x + 1 + 34*t^2 + 34*t^3 + 34*t^4 + 90*t^5 + 29*t^6 + 16*t^8 + 32*t^9 +
        6*t^10 + 66*t^11 + 41*t^12 + 93*t^13 + t^14 + 69*t^15 + 8*t^16 + 61*t^17
        + 86*t^18 + 19*t^19 + 47*t^20 + 11*t^21 + 42*t^22 + 38*t^23 + 46*t^24 +
        90*t^25 + 14*t^26 + 7*t^27 + 89*t^28 + 85*t^29 + 70*t^30 + 24*t^31 +
        28*t^32 + 71*t^33 + 53*t^34 + 55*t^35 + 90*t^36 + 26*t^37 + 4*t^38 +
        56*t^39 + 44*t^40 + 8*t^41 + 25*t^42 + 94*t^43 + 14*t^44 + 92*t^45 +
        56*t^46 + 83*t^47 + 26*t^48 + 41*t^49 + O(t^50), 1>,
    <x^2 + (100 + 67*t^2 + 67*t^3 + 68*t^4 + 11*t^5 + 72*t^6 + 100*t^7 + 85*t^8
        + 69*t^9 + 94*t^10 + 31*t^11 + 59*t^12 + 16*t^13 + 14*t^14 + 35*t^15 +
        77*t^16 + 28*t^17 + 33*t^18 + 72*t^19 + 25*t^20 + 22*t^21 + 12*t^22 +
        92*t^23 + 43*t^24 + 15*t^25 + 83*t^26 + 76*t^27 + 19*t^28 + 3*t^29 +
        35*t^30 + 27*t^31 + 40*t^32 + 16*t^33 + 92*t^34 + 91*t^35 + 32*t^36 +
        93*t^37 + 84*t^38 + 98*t^39 + 85*t^40 + 54*t^41 + 25*t^42 + 88*t^43 +
        35*t^44 + 17*t^45 + t^46 + 39*t^47 + 89*t^48 + 67*t^49 + O(t^50))*x + 1
        + 67*t^2 + 68*t^3 + 11*t^4 + 67*t^5 + 57*t^6 + 16*t^7 + 57*t^8 + 21*t^9
```

```
        + 68*t^10 + 68*t^11 + 61*t^12 + 98*t^13 + 4*t^14 + 21*t^15 + 7*t^16 +
        95*t^17 + 23*t^18 + 76*t^19 + 62*t^20 + 59*t^21 + 66*t^22 + 35*t^23 +
        41*t^24 + 45*t^25 + 32*t^26 + 56*t^27 + 35*t^28 + 19*t^29 + 21*t^30 +
        59*t^31 + 50*t^32 + 72*t^33 + 58*t^34 + 75*t^35 + 59*t^36 + 76*t^37 +
        83*t^38 + 66*t^39 + 6*t^40 + 8*t^41 + 94*t^42 + 77*t^43 + 100*t^44 +
        30*t^45 + 72*t^46 + 26*t^47 + 54*t^48 + 21*t^49 + O(t^50), 1>
]
1 + O(t^50)
[
    rec<recformat<F: RngIntElt, Rho: RngUPolElt, E: RngIntElt, Pi: RngUPolElt,
    IdealGen1, IdealGen2: RngUPolElt, Extension> |
        F := 1,
        Rho := 1 + O($.1^50),
        E := 1,
        Pi := $.1 + O($.1^50),
        Extension := Power series ring in t over GF(101) with fixed absolute
            precision 50>,
    rec<recformat<F: RngIntElt, Rho: RngUPolElt, E: RngIntElt, Pi: RngUPolElt,
    IdealGen1, IdealGen2: RngUPolElt, Extension> |
        F := 1,
        Rho := 1 + O($.1^50),
        E := 1,
        Pi := $.1 + O($.1^50),
        Extension := Power series ring in t over GF(101) with fixed absolute
            precision 50>,
    rec<recformat<F: RngIntElt, Rho: RngUPolElt, E: RngIntElt, Pi: RngUPolElt,
    IdealGen1, IdealGen2: RngUPolElt, Extension> |
        F := 1,
        Rho := 1 + O($.1^50),
        E := 1,
        Pi := $.1 + O($.1^50),
        Extension := Power series ring in t over GF(101) with fixed absolute
            precision 50>,
    rec<recformat<F: RngIntElt, Rho: RngUPolElt, E: RngIntElt, Pi: RngUPolElt,
    IdealGen1, IdealGen2: RngUPolElt, Extension> |
        F := 2,
        Rho := (1 + O($.1^50))*$.1 + O($.1^50),
        E := 1,
        Pi := $.1 + O($.1^50),
        Extension := Extension of Power series ring in t over GF(101) with fixed
            absolute precision 50 by x^2 + (100 + O(t^50))*x + 1 + O(t^50)>
]
```

49.8 Extensions of Series Rings

Extensions of series rings are either unramified or totally ramified. Only series rings defined over finite fields can be extended. We recommend constructing polynomials from sequences rather than by addition of terms, especially over fields, to avoid some precision loss. For example, $x^4 + t$ will not have full precision in the constant coefficient as there will be a O term in the constant coefficient of x^4 which will reduce its precision as a field element. Extensions require full precision polynomials and some polynomials such as $x^4 + t$ may not have enough precision to be used to construct an extension whereas the equivalent `Polynomial([t, 0, 0, 0, 1])` will.

49.8.1 Constructions of Extensions

`UnramifiedExtension(R, f)`

Construct the unramified extension of R, a series ring or an extension thereof, defined by the inertial polynomial f, that is, adjoin a root of f to R.

`TotallyRamifiedExtension(R, f)`

`MaxPrecision` RngIntElt *Default :*

Construct a totally ramified extension of R, a series ring or an extension thereof, defined by the eisenstein polynomial f, that is, adjoin a root of f to R.

The parameter `MaxPrecision` defaults to the precision of R. It can be set to the maximum precision the coefficients of f are known to, which must not be less than the precision of R. This allows the precision of the result to be increased to the degree of f multiplied by this maximum precision.

The polynomial f may be given over a series ring or an extension of a series ring having a higher precision than R. This allows the precision of the result to be increased up to the precision the polynomial is known to (or `MaxPrecision` if set) without losing any of the polynomial known past the precision of R.

The precision of a ramified extension cannot be increased unless the defining polynomial is known to more precision than the coefficient ring, indicated either by providing the polynomial to greater precision or by setting `MaxPrecision`. This should be taken into account when constructing ramified extensions, especially if polynomials are to be factored over the extension.

`ChangePrecision(E, r)`

`ChangePrecision(~E, r)`

The extension of a series ring E with precision r. It is not possible to increase the precision of a ramified extension unless the parameter `MaxPrecision` was set on construction of the extension and r is less than or equal to this value multiplied by the ramification degree or the polynomial used in creating the extension was given to more precision than the coefficient ring of E.

`FieldOfFractions(E)`

The field of fractions of the extension of a series ring E.

Example H49E5

We show a simple creation of a two-step extension and change its precision.

```
> P<t> := PowerSeriesRing(GF(101), 50);
> PP<tt> := PowerSeriesRing(GF(101));
> R<x> := PolynomialRing(PP);
> U := UnramifiedExtension(P, x^2 + 2);
> T := TotallyRamifiedExtension(U, x^2 + tt*x + tt); T;
Extension of Extension of Power series ring in t over GF(101) with fixed
absolute precision 50 by x^2 + 2 + O(t^50) by x^2 + (t + O(t^50))*x + t +
    O(t^50)
> Precision($1);
100
> ChangePrecision($2, 200);
Extension of Extension of Power series ring in $.1 over GF(101) with fixed
absolute precision 100 by x^2 + 2 + O($.1^100) by x^2 + ($.1 + O($.1^100))*x +
    $.1 + O($.1^100)
> ChangePrecision($1, 1000);
Extension of Extension of Power series ring in $.1 over GF(101) with fixed
absolute precision 500 by x^2 + 2 + O($.1^500) by x^2 + ($.1 + O($.1^500))*x +
    $.1 + O($.1^500)
> ChangePrecision($1, 20);
Extension of Extension of Power series ring in $.1 over GF(101) with fixed
absolute precision 10 by x^2 + 2 + O($.1^10) by x^2 + ($.1 + O($.1^10))*x + $.1
    + O($.1^10)
```

Both U and T have a field of fractions.

```
> FieldOfFractions(U);
Extension of Laurent series field in $.1 over GF(101) with fixed relative
precision 50 by (1 + O($.1^50))*x^2 + O($.1^50)*x + 2 + O($.1^50)
> FieldOfFractions(T);
Extension of Extension of Laurent series field in $.1 over GF(101) with fixed
relative precision 50 by (1 + O($.1^50))*x^2 + O($.1^50)*x + 2 + O($.1^50) by (1
    + O($.1^50))*x^2 + ($.1 + O($.1^50))*x + $.1 + O($.1^50)
```

49.8.2 Operations on Extensions

`Precision(E)`

`GetPrecision(E)`

The maximum precision elements of the extension of a series ring E may have.

`CoefficientRing(E)`

`BaseRing(E)`

The ring the extension of a series ring E was defined over.

`DefiningPolynomial(E)`

The polynomial used to define the extension E.

`InertiaDegree(E)`

The degree of the extension E if E is an unramified extension, 1 otherwise.

`RamificationIndex(E)`
`RamificationDegree(E)`

The degree of the extension E if E is a totally ramified extension, 1 otherwise.

`ResidueClassField(E)`

The residue class field of the extension E, that is, $E/\pi * E$.

`UniformizingElement(E)`

A uniformizing element for the extension E, that is, an element of E of valuation 1.

`IntegerRing(E)`
`Integers(E)`
`RingOfIntegers(E)`

The ring of integers of the extension E of a series ring if E is a field (extension of a laurent series ring).

`E1 eq E2`

Whether extensions $E1$ and $E2$ are considered to be equal.

`E . i`

The primitive element of the extension E, that is, the root of the defining polynomial of E adjoined to the coefficient ring of E to construct E.

`AssignNames(~E, S)`

Assign the string in the sequence S to be the name of the primitive element of E, that is, the root of the defining polynomial adjoined to the coefficient ring of E to construct E.

Example H49E6

A number of the operations above are applied to a two-step extension

```
> L<t> := LaurentSeriesRing(GF(53), 30);
> P<x> := PolynomialRing(L);
> U := UnramifiedExtension(L, x^3 + 3*x^2 + x + 4);
> P<y> := PolynomialRing(U);
> T := TotallyRamifiedExtension(U, Polynomial([t, 0, 0, 0, 1]));
> Precision(U);
30
> Precision(T);
120
> CoefficientRing(U);
Laurent series field in t over GF(53) with fixed relative precision 30
> CoefficientRing(T);
Extension of Laurent series field in t over GF(53) with fixed relative precision
30 by (1 + O(t^30))*x^3 + (3 + O(t^30))*x^2 + (1 + O(t^30))*x + 4 + O(t^30)
> DefiningPolynomial(U);
(1 + O(t^30))*x^3 + (3 + O(t^30))*x^2 + (1 + O(t^30))*x + 4 + O(t^30)
> DefiningPolynomial(T);
(1 + O(t^30))*y^4 + O(t^30)*y^3 + O(t^30)*y^2 + O(t^30)*y + t + O(t^30)
> InertiaDegree(U);
3
> InertiaDegree(T);
1
> RamificationDegree(U);
1
> RamificationDegree(T);
4
> ResidueClassField(U);
Finite field of size 53^3
Mapping from: RngSerExt: U to GF(53^3)
> ResidueClassField(T);
Finite field of size 53^3
Mapping from: RngSerExt: T to GF(53^3)
> UniformizingElement(U);
t + O(t^31)
> UniformizingElement(T);
(1 + O(t^30))*$.1 + O(t^30)
> Integers(T);
Extension of Extension of Power series ring in $.1 over GF(53) with fixed
absolute precision 30 by x^3 + (3 + O($.1^30))*x^2 + x + 4 + O($.1^30) by x^4 +
    $.1 + O($.1^30)
> U.1;
(1 + O(t^30))*$.1 + O(t^30)
> T.1;
```

```
(1 + O(t^30))*$.1 + O(t^30)
```

49.8.3 Elements of Extensions

`x * y` `x + y` `x - y` `- x` `x ^ n` `x div y` `x / y`

`x eq y` `IsZero(e)` `IsOne(e)` `IsMinusOne(e)` `IsUnit(e)`

`Valuation(e)`

The valuation of the element e of an extension of a series ring. This is the index of the largest power of π which divides e.

`RelativePrecision(e)`

The relative precision of the element e of an extension of a series ring. This is the number of digits of e (in π) which are known.

`AbsolutePrecision(e)`

The absolute precision of the element e of an extension of a series ring. This is the same as the sum of the relative precision of e and the valuation of e.

`Coefficients(e)`
`Eltseq(e)`
`ElementToSequence(e)`

Given an element e of an extension of a series ring, return the coefficients of e with respect to the powers of the uniformizing element of the extension.

Example H49E7

We show some simple arithmetic with some elements of some extensions.

```
> P<t> := PowerSeriesRing(GF(101), 50);
> R<x> := PolynomialRing(P);
> U<u> := UnramifiedExtension(P, x^2 + 2*x + 3);
> UF := FieldOfFractions(U);
> R<y> := PolynomialRing(U);
> T<tt> := TotallyRamifiedExtension(U, y^2 + t*y + t);
> TF<tf> := FieldOfFractions(T);
> UF<uf> := FieldOfFractions(U);
> u + t;
u + t + O(t^50)
> uf * t;
($.1 + O($.1^50))*uf + O($.1^51)
> tf eq tt;
false
> tf - tt;
```

```
O($.1^50)*tf + O($.1^50)
> IsZero($1);
false
> Valuation($2);
100
> Valuation(tt);
1
> Valuation(U!t);
1
> Valuation(T!t);
2
> RelativePrecision(u);
50
> AbsolutePrecision(u);
50
> AbsolutePrecision(uf);
50
> RelativePrecision(uf);
50
> RelativePrecision(u - uf);
0
> AbsolutePrecision(u - uf);
50
> u^7;
(13 + O(t^50))*u + 71 + O(t^50)
> Coefficients($1);
[
    71 + O(t^50),
    13 + O(t^50)
]
> tt^8;
(4*t^4 + 91*t^5 + 6*t^6 + 100*t^7 + O(t^50))*tt + t^4 + 95*t^5 + 5*t^6 + 100*t^7
    + O(t^50)
> Coefficients($1);
[ t^4 + 95*t^5 + 5*t^6 + 100*t^7 + O(t^50), 4*t^4 + 91*t^5 + 6*t^6 + 100*t^7 +
    O(t^50) ]
```

49.8.4 Optimized Representation

OptimizedRepresentation(E)
OptimisedRepresentation(E)

An optimized representation for the unramified extension E of a series ring. The defining polynomial of E must be coercible into the residue class field. The result is a series ring over the residue class field of E and a map from E to this series ring.

Example H49E8

We give a simple example of the use of `OptimizedRepresentation`.

```
> P<t> := PowerSeriesRing(GF(101), 50);
> R<x> := PolynomialRing(P);
> U<u> := UnramifiedExtension(P, x^2 + 2*x + 3);
> U;
Extension of Power series ring in t over GF(101) with fixed absolute precision
50 by x^2 + (2 + O(t^50))*x + 3 + O(t^50)
> OptimizedRepresentation(U);
Power series ring in $.1 over GF(101^2) with fixed absolute precision 50
Mapping from: RngSerExt: U to Power series ring in s over GF(101^2) with fixed
absolute precision 50 given by a rule
```

49.9 Bibliography

[**Hus87**] Dale Husemöller. *Elliptic Curves*, volume 111 of *Graduate Texts in Mathematics*. Springer, New York, 1987.

50 LAZY POWER SERIES RINGS

Chapter 50
LAZY POWER SERIES RINGS

50.1 Introduction

The lazy series rings `RngLaz` and their elements `RngLazElt` allow the creation of infinite precision series by providing series for which all coefficients can be calculated by some given formula. Only finitely many coefficients of such a series can be known at any one time but all infinitely many of the coefficients are knowable. Any coefficient of a lazy series can be generated and once a coefficient is computed it will be stored in the series for quick retrieval.

The simplest implementation of this idea is creating a series by providing a map. This map is a formula for computing coefficients of a series. Given the exponents of the variables of a term it will give you the coefficient of that term. Series with finitely many non–zero terms can be created in special ways and several usual arithmetic operations can be applied to lazy series. Such constructions yield lazy series with more complicated formulas for their coefficients.

Consider the series $\sum_{i=0}^{n} i * x^i$. A formula for the coefficients is given by $i \mapsto i$. This series can be created as a lazy series as follows:

```
> L<x> := LazyPowerSeriesRing(Integers(), 1);
> m := map<Integers() -> Integers() | i :-> i>;
> s := elt<L | m>;
> s;
Lazy power series
```

Currently s does not know what any of its coefficients are yet it is possible to calculate any coefficient of s.

```
> Coefficient(s, 0);
0
> Coefficient(s, 100);
100
> Coefficient(s, 100000000000000000000000);
100000000000000000000000
```

50.2 Creation of Lazy Series Rings

Both univariate and multivariate lazy series rings can be created.

`LazyPowerSeriesRing(C, n)`

The lazy power series ring with coefficient ring C and n variables. Any ring is valid input for C and n can be any positive integer.

`ChangeRing(L, C)`

Given a lazy series ring L defined over a ring R and some ring C, return the lazy series ring with coefficient ring C but the same number of variables as L. A map from L to the new lazy series ring is also returned which takes an series s in L to a series whose coefficients are those of s coerced into C.

Example H50E1

Here we illustrate the creation and printing of lazy power series rings.

```
> L := LazyPowerSeriesRing(Rationals(), 5);
> L;
Lazy power series ring in 5 variables over Rational Field
> ChangeRing(L, MaximalOrder(CyclotomicField(7)));
Lazy power series ring in 5 variables over Maximal Equation Order with defining
polynomial x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 over Z
```

50.3 Functions on Lazy Series Rings

Lazy series rings have variables, names for their variables and a coefficient ring.

`R . i`

The ith variable of the lazy power series ring R, where i is between 1 and the rank of R.

`AssignNames(~R, S)`

Given a lazy series ring R with n indeterminates and a sequence S of n strings, assign the elements of S to the names of the variables of R.

`BaseRing(R)`

`CoefficientRing(R)`

The coefficient ring of the lazy power series ring R. The coefficients of all the series in R will lie in this ring.

`Rank(R)`

The number of variables associated with the lazy power series ring R.

`R1 eq R2`

Return `true` if the lazy series rings $R1$ and $R2$ are the same ring, that is, they have the same coefficient ring and rank.

Example H50E2

The functions on lazy power series rings are illustrated here.

```
> L := LazyPowerSeriesRing(FiniteField(73), 7);
> L.4;
Lazy power series
> AssignNames(~L, ["a", "b", "c", "d", "fifth", "sixth", "seventh"]);
> L.4;
Lazy power series
```

The names for the variables of L are not used in default series printing. However they are used when printing an element to a given precision using PrintToPrecision.

```
> CoefficientRing(L);
Finite field of size 73
> Rank(L);
7
> L eq LazyPowerSeriesRing(CoefficientRing(L), Rank(L));
true
```

50.4 Elements

Lazy series may be created in a number of ways. Arithmetic for ring elements is available as well as a few predicates. Coefficients of the monomials in the series can be calculated.

50.4.1 Creation of Finite Lazy Series

`R ! c`

Return the series in the lazy series ring R with constant term c and every other coefficient 0 where c is any ring element coercible into the coefficient ring of R.

`R ! s`

Return the lazy series in the lazy series ring R whose coefficients are those of the lazy series s coerced into the coefficient ring of R.

`R ! S`

Return the series in the lazy series ring R whose coefficients are the elements of S where S is a sequence of elements each coercible into the coefficient ring of R. The coefficients are taken to be given in the order which Coefficients will return them and PrintToPrecision will print the terms of the series. Any coefficient not given is assumed to be zero so that all coefficients are calculable. The resulting series can only have finitely many non zero terms and all such non zero coefficients must be given in S.

`LazySeries(R, f)`

Create the lazy series in the lazy series ring R with finitely many non zero terms which are the terms of the polynomial f. The number of variables of the parent ring of f must be the same as the number of variables of R. The coefficients of f must be coercible into the coefficient ring of R.

It is also possible for f to be a rational function, p/q. The series created from f will be `LazySeries(R, p)*LazySeries(R, q)`$^{-1}$.

Example H50E3

Creation of series using the above functions is shown here.

```
> L := LazyPowerSeriesRing(AlgebraicClosure(), 3);
> LR := LazyPowerSeriesRing(Rationals(), 3);
> s := L!1;
> s;
Lazy power series
> LR!s;
Lazy power series
> P<x, y, z> := RationalFunctionField(Rationals(), 3);
> LazySeries(L, (x + y + 8*z)^7/(1 + 5*x*y*z + x^8)^3);
Lazy power series
```

50.4.1.1 Creation of Lazy Series from Maps

Creating a lazy series from a map simulates the existence of lazy series as series for which coefficients are not known on creation of the series but calculated by some rule given when creating the series.

This rule can be coded in a map where the input to the map is the exponents of the variables in a term and the output is the coefficient of the term described by the input. So for a lazy series ring with variables x, y and z, a general term of a series will look like $c_{ijk} * x_i^r * y_j^s * z_k^t$. The coefficient of this term, c_{ijk}, in a series defined by a map m is the result of $m(\langle i, j, k\rangle)$.

`elt< R | m >`

Creates a series in the lazy series ring R from the map m. The map m must take as input either an integer (when R is univariate only) or a tuple of integers (of length the number of variables of R) and return an element of the coefficient ring of R. This element will be taken as the coefficient of the term of the series whose variables have the exponents given in the input tuple, as described above.

Example H50E4

We first illustrate the univariate case.

```
> L<x> := LazyPowerSeriesRing(MaximalOrder(QuadraticField(5)), 1);
> Z := Integers();
> m := map<Z -> CoefficientRing(L) | t :-> 2*t>;
> s := elt<L | m>;
> PrintToPrecision(s, 10);
2*x + 4*x^2 + 6*x^3 + 8*x^4 + 10*x^5 + 12*x^6 + 14*x^7 + 16*x^8 + 18*x^9 +
    20*x^10
> Coefficient(s, 34);
68
> m(34);
68
> Coefficient(s, 2^30 + 10);
2147483668
> m(2^30 + 10);
2147483668
```

Example H50E5

And now for the multivariate case.

```
> L<x, y, z> := LazyPowerSeriesRing(AlgebraicClosure(), 3);
> Z := Integers();
> m := map<car<Z, Z, Z> -> CoefficientRing(L) | t :-> t[1]*t[2]*t[3]>;
> s := elt<L | m>;
> PrintToPrecision(s, 5);
x*y*z + 2*x^2*y*z + 2*x*y^2*z + 2*x*y*z^2 + 3*x^3*y*z + 4*x^2*y^2*z +
    4*x^2*y*z^2 + 3*x*y^3*z + 4*x*y^2*z^2 + 3*x*y*z^3
> Coefficient(s, [1, 1, 1]);
1
> m(<1, 1, 1>);
1
> Coefficient(s, [3, 1, 2]);
6
> m(<3, 1, 2>);
6
```

50.4.2 Arithmetic with Lazy Series

All the usual arithmetic operations are possible for lazy series.

`s + t`

The sum of the two lazy series s and t.

`-s`

The negation of the lazy series s.

`s - t`

The difference between lazy series s and t.

`s * t`

The product of the lazy series s and t.

`s + r`
`r + s`

The sum of the lazy series s and the element r of the coefficient ring of the parent ring of s.

`c * s`
`s * c`

The product of the lazy series s and the element c of the coefficient ring of the parent ring of s.

`s * n`

The product of the lazy series s and the monomial x^n, where x^n is $x_1^{n_1} \times \ldots \times x_r^{n_r}$ where r is the number of variables of the parent ring of s, n is the sequence $[n_1, \ldots, n_r]$ and $x = x_1, \ldots, x_r$ are the series variables of the parent ring of s.

`s ^ n`

Given a lazy series s and an integer n, return the nth power of s. It is allowed for n to be negative and inverses will be taken where possible.

Example H50E6

Here we demonstrate the above arithmetic operations.

```
> L<a, b, c, d> := LazyPowerSeriesRing(Rationals(), 4);
> (a + 4*b + (-c)*[8, 9, 2^30 + 10, 2] - d*b + 5)^-8;
Lazy power series
```

50.4.3 Finding Coefficients of Lazy Series

In theory, all the coefficients of most series can be calculated. In practice it is not possible to compute infinitely many. Some problems arise however when a series has been created using multiplication, inversion, evaluation or by taking square roots. For such series the jth coefficient for all $j < i$ must be known for the ith coefficient to be calculated. In such cases the exponents specified for each variable in the monomial whose coefficient is required must be small integers ($< 2^{30}$).

The default ordering of coefficients is by total degree of the corresponding monomials. This is the same order on multivariate polynomials by default. When drawn on paper or imagined in 3 or more dimensions this looks like a spiral if the coordinates representing the monomial exponents are joined. This is the same ordering which is used to compute the valuation of a series.

Once computed, coefficients are stored with the series so any subsequent call to these functions will be faster in non–trivial cases than the first call. It is possible to interrupt any of these functions and return to the prompt.

`Coefficient(s, i)`

Returns the coefficient in the univariate lazy series s of x^i where x is the series variable of the parent of s and i is a non negative integer.

`Coefficient(s, T)`

Returns the coefficient in the multivariate lazy series s of the monomial $x_1^{T_1} * \ldots * x_r^{T_r}$ where T is the sequence $[T_1, \ldots, T_r]$, $x_1, \ldots, x_r$ are the variables of the parent ring of s and r is the rank of the parent of s.

`Coefficients(s, n)`

`Coefficients(s, l, n)`

The coefficients of the lazy series s whose monomials have total degree at least l (0 if not given) and at most n where the monomials are ordered using the default "spiral" degree order. The bounds l and n must be non negative integers.

`Valuation(s)`

The valuation of the lazy series s. This is the exponent of the monomial with the first non–zero coefficient as returned by `Coefficients` above. The return value will be either `Infty`, an integer (univariate case) or a sequence (multivariate case).

`PrintToPrecision(s, n)`

Print the sum of all terms of the lazy series s whose degree is no more than n where n is a non negative integer. The series is printed using the "spiral" ordering.

`PrintTermsOfDegree(s, l, n)`

Print the sum of the terms of the lazy series s whose degree is at least l and at most n. The terms are printed using the spiral ordering. The bounds l and n must be non negative integers.

`LeadingCoefficient(s)`

The coefficient in the lazy series s whose monomial exponent is the valuation of s. That is, the first non–zero coefficient of s where the ordering which determines "first" is the "spiral" ordering used by `Coefficients` and `Valuation`.

`LeadingTerm(s)`

The term in the lazy series s whose monomial exponent is the valuation of s. That is, the first non–zero term of s where the ordering which determines "first" is the "spiral" ordering used by `Coefficients` and `Valuation`.

Example H50E7

In this example we look at some coefficients of an infinite series.

```
> L<a, b, c, d> := LazyPowerSeriesRing(Rationals(), 4);
> s := (1 + 2*a + 3*b + 4*d)^-5;
```

Find the coefficient of $a * b * c * d$.

```
> Coefficient(s, [1, 1, 1, 1]);
0
```

Find the coefficients of all monomials with total degree at most 6.

```
> time Coefficients(s, 6);
[ 1, -10, -15, 0, -20, 60, 180, 0, 240, 135, 0, 360, 0, 0, 240, -280, -1260, 0,
-1680, -1890, 0, -5040, 0, 0, -3360, -945, 0, -3780, 0, 0, -5040, 0, 0, 0,
-2240, 1120, 6720, 0, 8960, 15120, 0, 40320, 0, 0, 26880, 15120, 0, 60480, 0, 0,
80640, 0, 0, 0, 35840, 5670, 0, 30240, 0, 0, 60480, 0, 0, 0, 53760, 0, 0, 0, 0,
17920, -4032, -30240, 0, -40320, -90720, 0, -241920, 0, 0, -161280, -136080, 0,
-544320, 0, 0, -725760, 0, 0, 0, -322560, -102060, 0, -544320, 0, 0, -1088640,
0, 0, 0, -967680, 0, 0, 0, 0, -322560, -30618, 0, -204120, 0, 0, -544320, 0, 0,
0, -725760, 0, 0, 0, 0, -483840, 0, 0, 0, 0, 0, -129024, 13440, 120960, 0,
161280, 453600, 0, 1209600, 0, 0, 806400, 907200, 0, 3628800, 0, 0, 4838400, 0,
0, 0, 2150400, 1020600, 0, 5443200, 0, 0, 10886400, 0, 0, 0, 9676800, 0, 0, 0,
0, 3225600, 612360, 0, 4082400, 0, 0, 10886400, 0, 0, 0, 14515200, 0, 0, 0, 0,
9676800, 0, 0, 0, 0, 0, 2580480, 153090, 0, 1224720, 0, 0, 4082400, 0, 0, 0,
7257600, 0, 0, 0, 0, 7257600, 0, 0, 0, 0, 0, 3870720, 0, 0, 0, 0, 0, 0, 860160 ]
Time: 0.140
> #$1;
210
```

`PrintToPrecision` will display the monomials to which these coefficients correspond.

```
> time PrintToPrecision(s, 6);
1 - 10*a - 15*b - 20*d + 60*a^2 + 180*a*b + 240*a*d + 135*b^2 + 360*b*d +
    240*d^2 - 280*a^3 - 1260*a^2*b - 1680*a^2*d - 1890*a*b^2 - 5040*a*b*d -
    3360*a*d^2 - 945*b^3 - 3780*b^2*d - 5040*b*d^2 - 2240*d^3 + 1120*a^4 +
    6720*a^3*b + 8960*a^3*d + 15120*a^2*b^2 + 40320*a^2*b*d + 26880*a^2*d^2 +
    15120*a*b^3 + 60480*a*b^2*d + 80640*a*b*d^2 + 35840*a*d^3 + 5670*b^4 +
```

```
    30240*b^3*d + 60480*b^2*d^2 + 53760*b*d^3 + 17920*d^4 - 4032*a^5 -
    30240*a^4*b - 40320*a^4*d - 90720*a^3*b^2 - 241920*a^3*b*d - 161280*a^3*d^2
    - 136080*a^2*b^3 - 544320*a^2*b^2*d - 725760*a^2*b*d^2 - 322560*a^2*d^3 -
    102060*a*b^4 - 544320*a*b^3*d - 1088640*a*b^2*d^2 - 967680*a*b*d^3 -
    322560*a*d^4 - 30618*b^5 - 204120*b^4*d - 544320*b^3*d^2 - 725760*b^2*d^3 -
    483840*b*d^4 - 129024*d^5 + 13440*a^6 + 120960*a^5*b + 161280*a^5*d +
    453600*a^4*b^2 + 1209600*a^4*b*d + 806400*a^4*d^2 + 907200*a^3*b^3 +
    3628800*a^3*b^2*d + 4838400*a^3*b*d^2 + 2150400*a^3*d^3 + 1020600*a^2*b^4 +
    5443200*a^2*b^3*d + 10886400*a^2*b^2*d^2 + 9676800*a^2*b*d^3 +
    3225600*a^2*d^4 + 612360*a*b^5 + 4082400*a*b^4*d + 10886400*a*b^3*d^2 +
    14515200*a*b^2*d^3 + 9676800*a*b*d^4 + 2580480*a*d^5 + 153090*b^6 +
    1224720*b^5*d + 4082400*b^4*d^2 + 7257600*b^3*d^3 + 7257600*b^2*d^4 +
    3870720*b*d^5 + 860160*d^6
Time: 0.010
```

The valuation of s can be obtained as follows. The valuation of zero is a special case.

```
> Valuation(s);
[ 0, 0, 0, 0 ]
> Valuation(s*0);
Infinity
```

`CoefficientsNonSpiral(s, n)`

Returns the coefficients of the monomials in the lazy series s whose exponents are given by $[i_1, ..., i_r]$ where each $i_j \leq n_j$. The argument n may either be a non negative integer (univariate case) or a sequence of non negative integers of length r where r is the rank of the parent ring of s. The index of the $[i_1, ..., i_r]$-th coefficient in the return sequence is given by

$$\sum_{j=1}^{r} i_j * (\prod_{k=j+1}^{r} (n_k + 1)).$$

`Index(s, i, n)`

Return the index in the return value of `CoefficientsNonSpiral(s, n)` of the monomial in the lazy series s whose exponents are given by the (trivial in the univariate case) sequence i.

Example H50E8

We find the coefficients of the series used in the last example using the alternative algorithm.

```
> L<a, b, c, d> := LazyPowerSeriesRing(Rationals(), 4);
> s := (1 + 2*a + 3*b + 4*d)^-5;
> time CoefficientsNonSpiral(s, [3, 3, 3, 2]);
[ 1, -20, 240, 0, 0, 0, 0, 0, 0, 0, 0, 0, -15, 360, -5040, 0, 0, 0, 0, 0, 0, 0,
0, 0, 135, -3780, 60480, 0, 0, 0, 0, 0, 0, 0, 0, 0, -945, 30240, -544320, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, -10, 240, -3360, 0, 0, 0, 0, 0, 0, 0, 0, 0, 180, -5040,
80640, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1890, 60480, -1088640, 0, 0, 0, 0, 0, 0, 0,
0, 0, 15120, -544320, 10886400, 0, 0, 0, 0, 0, 0, 0, 0, 0, 60, -1680, 26880, 0,
0, 0, 0, 0, 0, 0, 0, 0, -1260, 40320, -725760, 0, 0, 0, 0, 0, 0, 0, 0, 0, 15120,
-544320, 10886400, 0, 0, 0, 0, 0, 0, 0, 0, 0, -136080, 5443200, -119750400, 0,
0, 0, 0, 0, 0, 0, 0, 0, -280, 8960, -161280, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6720,
-241920, 4838400, 0, 0, 0, 0, 0, 0, 0, 0, 0, -90720, 3628800, -79833600, 0, 0,
0, 0, 0, 0, 0, 0, 0, 907200, -39916800, 958003200, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
Time: 0.370
> #$1;
192
```

It appears that the "spiral" algorithm may be faster since it computed more coefficients in less time. This may be because of the operations (inversion) involved in calculating s and that `CoefficientsNonSpiral` computed coefficients of larger degree monomials than `Coefficients`. In calculating these coefficients for larger degrees, coefficients which were not asked for may have been calculated. `Coefficients` will only need to calculate the coefficients which have been asked for — all intermediate calculations would have been asked for.

50.4.4 Predicates on Lazy Series

`s eq t`

Return `true` if the lazy series s and t are exactly the same series.

`IsZero(s)`

Return `true` if the lazy series s was created as the zero series.

`IsOne(s)`

Return `true` if the lazy series s was created as the one series.

`IsMinusOne(s)`

Return `true` if the lazy series s was created as the minus one series.

`IsUnit(s)`

Return `true` if the lazy series s is a unit.

`IsWeaklyZero(s, n)`

Return `true` if all the terms of the lazy series s with degree at most n are zero. Calling this function without the second argument returns `IsZero(s)`.

`IsWeaklyEqual(s, t, n)`

Return `true` if the terms of the lazy series s and t with degree at most n are the same. Calling this function without the third argument returns `s eq t`.

50.4.5 Other Functions on Lazy Series

`Derivative(s)`

`Derivative(s, v)`

`Derivative(s, v, n)`

Return the (nth) derivative of the lazy series s with respect to the vth variable of the parent ring of s. If v is not given, the parent ring of s must be univariate and the derivative with respect to the unique variable is returned. It is only allowed to give values of v from 1 to the rank of the parent of s and for n to be a positive integer.

`Integral(s)`

`Integral(s, v)`

Return the integral of the lazy series s with respect to the vth variable of the parent ring of s. If v is not given, the parent ring of s must be univariate and the integral with respect to the unique variable is returned. It is only allowed to give values of v from 1 to the rank of the parent of s.

`Evaluate(s, t)`

`Evaluate(s, T)`

Return the lazy series s evaluated at the lazy series t or the sequence T of lazy series. The series t or the series in T must have zero constant term so that every coefficient of the result can be finitely calculated.

Example H50E9

Some usage of `Evaluate` is shown below.

```
> R := LazyPowerSeriesRing(Rationals(), 2);
> AssignNames(~R, ["x","y"]);
> m := map<car<Integers(), Integers()> -> Rationals() | t :-> 1>;
> s := elt<R | m>;
> PrintToPrecision(s, 3);
1 + x + y + x^2 + x*y + y^2 + x^3 + x^2*y + x*y^2 + y^3
> R1 := LazyPowerSeriesRing(Rationals(), 1);
> AssignNames(~R1, ["z"]);
```

```
> m1 := map<car<Integers()> -> Rationals() | t :-> t[1]>;
> s1 := elt<R1 | m1>;
> PrintToPrecision(s1, 3);
z + 2*z^2 + 3*z^3
> e := Evaluate(s, [s1,s1]);
> PrintToPrecision(e, 10);
1 + 2*z + 7*z^2 + 22*z^3 + 67*z^4 + 200*z^5 + 588*z^6 + 1708*z^7 + 4913*z^8 +
    14018*z^9 + 39725*z^10
> Parent(e);
Lazy power series ring in 1 variable over Rational Field
> f := Evaluate(s1, s - 1);
> PrintToPrecision(f, 10);
x + y + 3*x^2 + 5*x*y + 3*y^2 + 8*x^3 + 18*x^2*y + 18*x*y^2 + 8*y^3 + 20*x^4 +
    56*x^3*y + 75*x^2*y^2 + 56*x*y^3 + 20*y^4 + 48*x^5 + 160*x^4*y + 264*x^3*y^2
    + 264*x^2*y^3 + 160*x*y^4 + 48*y^5 + 112*x^6 + 432*x^5*y + 840*x^4*y^2 +
    1032*x^3*y^3 + 840*x^2*y^4 + 432*x*y^5 + 112*y^6 + 256*x^7 + 1120*x^6*y +
    2496*x^5*y^2 + 3600*x^4*y^3 + 3600*x^3*y^4 + 2496*x^2*y^5 + 1120*x*y^6 +
    256*y^7 + 576*x^8 + 2816*x^7*y + 7056*x^6*y^2 + 11616*x^5*y^3 +
    13620*x^4*y^4 + 11616*x^3*y^5 + 7056*x^2*y^6 + 2816*x*y^7 + 576*y^8 +
    1280*x^9 + 6912*x^8*y + 19200*x^7*y^2 + 35392*x^6*y^3 + 47280*x^5*y^4 +
    47280*x^4*y^5 + 35392*x^3*y^6 + 19200*x^2*y^7 + 6912*x*y^8 + 1280*y^9 +
    2816*x^10 + 16640*x^9*y + 50688*x^8*y^2 + 103168*x^7*y^3 + 154000*x^6*y^4 +
    175344*x^5*y^5 + 154000*x^4*y^6 + 103168*x^3*y^7 + 50688*x^2*y^8 +
    16640*x*y^9 + 2816*y^10
> f := Evaluate(s1 + 1, s - 1);
> PrintToPrecision(f, 10);
1 + x + y + 3*x^2 + 5*x*y + 3*y^2 + 8*x^3 + 18*x^2*y + 18*x*y^2 + 8*y^3 + 20*x^4
    + 56*x^3*y + 75*x^2*y^2 + 56*x*y^3 + 20*y^4 + 48*x^5 + 160*x^4*y +
    264*x^3*y^2 + 264*x^2*y^3 + 160*x*y^4 + 48*y^5 + 112*x^6 + 432*x^5*y +
    840*x^4*y^2 + 1032*x^3*y^3 + 840*x^2*y^4 + 432*x*y^5 + 112*y^6 + 256*x^7 +
    1120*x^6*y + 2496*x^5*y^2 + 3600*x^4*y^3 + 3600*x^3*y^4 + 2496*x^2*y^5 +
    1120*x*y^6 + 256*y^7 + 576*x^8 + 2816*x^7*y + 7056*x^6*y^2 + 11616*x^5*y^3 +
    13620*x^4*y^4 + 11616*x^3*y^5 + 7056*x^2*y^6 + 2816*x*y^7 + 576*y^8 +
    1280*x^9 + 6912*x^8*y + 19200*x^7*y^2 + 35392*x^6*y^3 + 47280*x^5*y^4 +
    47280*x^4*y^5 + 35392*x^3*y^6 + 19200*x^2*y^7 + 6912*x*y^8 + 1280*y^9 +
    2816*x^10 + 16640*x^9*y + 50688*x^8*y^2 + 103168*x^7*y^3 + 154000*x^6*y^4 +
    175344*x^5*y^5 + 154000*x^4*y^6 + 103168*x^3*y^7 + 50688*x^2*y^8 +
    16640*x*y^9 + 2816*y^10
```

`SquareRoot(s)`

`Sqrt(s)`

The square root of the lazy series s.

`IsSquare(s)`

Return `true` if the lazy series s is a square and the square root if so.

`PolynomialCoefficient(s, i)`

Given a series s in a lazy series ring whose coefficient ring is a polynomial ring (either univariate or multivariate), consider the polynomial which would be formed if this series was written as a polynomial with series as the coefficients. For i a non negative integer and the coefficient ring of the series ring a univariate polynomial ring this function returns the series which is the ith coefficient of the polynomial resulting from the rewriting of the series.

When the coefficient ring of the parent of s is a multivariate polynomial ring i should be a sequence of non negative integers of length the rank of the coefficient ring. The series returned will be the series which is the coefficient of $x_1^{i_1} * \ldots * x_r^{i_r}$ in the rewritten series where r is the rank of the coefficient ring and x_j are the indeterminates of the coefficient ring.

51 GENERAL LOCAL FIELDS

Chapter 51
GENERAL LOCAL FIELDS

51.1 Introduction

The local fields described in this chapter are extensions of any local field in MAGMA by any irreducible polynomial over that field. They are not constrained by requirements that the polynomial defining the extension be inertial or eisenstein. These local fields allow for ramified and inertial extensions to be made in one step rather than forcing such an extension to be split into two – being a ramified extension and an unramified extension. They are typed `RngLocA` with elements of type `RngLocAElt`. To compare these fields to the local fields where extensions are either totally ramified or unramified, see Chapter 47.

The fields are represented as a polynomial quotient ring. A map into an isomorphic `FldPad` can be constructed and the isomorphic field used for various calculations.

51.2 Constructions

Local fields can be constructed as extensions of other local fields and as subfields of other local fields.

`LocalField(L, f)`

Construct a local field F as an extension of the local field L by the polynomial f over L.

Example H51E1

We can use the 2 step local fields to help us find an irreducible polynomial over a p-adic field which can be used to extend that p-adic field in 1 step.

```
> P<x> := PolynomialRing(Integers());
> Zp := pAdicRing(7, 50);
> U := UnramifiedExtension(Zp, x^2 + 6*x + 3);
> R := TotallyRamifiedExtension(U, x^3 + 7*x^2 + 7*x + 7);
> L<a> := LocalField(pAdicField(7, 50), MinimalPolynomial(R.1 + U.1, Zp));
> L;
Extension of 7-adic field mod 7^50 by x^6 + (32 + O(7^50))*x^5 + (390 +
    O(7^50))*x^4 + (2284 + O(7^50))*x^3 + (6588 + O(7^50))*x^2 + (8744 +
    O(7^50))*x + 5452 + O(7^50)
```

We can also use any irreducible polynomial we can find to define a 1 step extension.

```
> LocalField(pAdicField(7, 50), x^6 - 49*x^2 + 686);
Extension of 7-adic field mod 7^50 by x^6 + O(7^50)*x^5 + O(7^50)*x^4 +
    O(7^50)*x^3 - (7^2 + O(7^52))*x^2 + O(7^50)*x + 2*7^3 + O(7^53)
```

`sub< L | $a_1, ..., a_n$ >`

`sub< L | S >`

Construct the local field F as a subfield of the local field L containing the elements a_i of L or the elements of the sequence S.

Example H51E2

```
> Qp := pAdicField(5, 20);
> P<x> := PolynomialRing(Qp);
> L<a> := LocalField(Qp, x^4 + 4*x^2 + 2);
> P<x> := PolynomialRing(L);
> LL<aa> := LocalField(L, x^4 + 4*L.1);
> r := (10236563738184*a^3 - 331496727861*a^2 + 10714284669258*a +
> 8590525712453*5)*aa^2
> -12574685904653*a^3 + 19786544763736*a^2 + 4956446023134*a + 37611818678747;
> S, m := sub< LL | r >;
> S;
Extension of Extension of 5-adic field mod 5^20 by x^4 + O(5^20)*x^3 + (4 +
    O(5^20))*x^2 + (4 + O(5^20))*x + 2 + O(5^20) by (O(5^20)*a^3 + O(5^20)*a^2 +
    O(5^20)*a + (1 + O(5^20)))*x^2 + (O(5^20)*a^3 + O(5^20)*a^2 + O(5^20)*a +
    O(5^20))*x + O(5^20)*a^3 + O(5^20)*a^2 + (4 + O(5^20))*a + O(5^20)
> m(S.1);
(O(5^20)*a^3 + O(5^20)*a^2 + O(5^20)*a + O(5^20))*aa^3 + (O(5^20)*a^3 +
    O(5^20)*a^2 + O(5^20)*a + (1 + O(5^20)))*aa^2 + (O(5^20)*a^3 + O(5^20)*a^2 +
    O(5^20)*a + O(5^20))*aa + O(5^20)*a^3 + O(5^20)*a^2 + O(5^20)*a + O(5^20)
```

51.3 Operations with Fields

`BaseRing(L)`

`CoefficientRing(L)`

Return the coefficient field of the local field L. This is the field which was extended to construct L.

`DefiningPolynomial(L)`

Return the polynomial used to define the local field L as an extension of its coefficient field.

`Degree(L)`

Return the degree of the local field L, that is, the degree of its defining polynomial.

`Degree(L, R)`

Return the degree of L as an extension of R where R is some coefficient ring of L.

`InertiaDegree(L)`

`RamificationDegree(L)`

`RamificationIndex(L)`

Return the degree of the inertial or totally ramified subfield of the local field L as an extension of the coefficient field of L.

`Precision(L)`

Return the precision of the local field L. This is the maximum number of digits which can occur in an element of L, the difference between the valuation of an element of L and the valuation of the term of highest valuation occurring in that element.

`Prime(L)`

Return the prime of the local field L. This is the same as the prime of the coefficient field of L.

Example H51E3

Continuing from the first example we have :

```
> CoefficientRing(L);
7-adic field mod 7^50
> DefiningPolynomial(L);
$.1^6 + (32 + O(7^50))*$.1^5 + (390 + O(7^50))*$.1^4 + (2284 + O(7^50))*$.1^3 +
    (6588 + O(7^50))*$.1^2 + (8744 + O(7^50))*$.1 + 5452 + O(7^50)
> Precision(L);
150
> Prime(L);
7
> Degree(L); RamificationDegree(L); InertiaDegree(L);
6
3
2
```

`QuotientRepresentation(L)`

Return the polynomial quotient ring which is isomorphic to the local field L and is used to represent L.

`RamifiedRepresentation(L)`

Return the local field isomorphic to the local field L constructed as an unramified then a ramified extension and the map from L into the isomorphic field.

Example H51E4

We create a 1 step local field and compute its representation as a 2 step local field.

```
> P<x> := PolynomialRing(Integers());
> L<a> := LocalField(pAdicField(7, 50), x^6 - 49*x^2 + 686);
> L;
Extension of 7-adic field mod 7^50 by x^6 + O(7^50)*x^5 + O(7^50)*x^4 +
    O(7^50)*x^3 - (7^2 + O(7^52))*x^2 + O(7^50)*x + 2*7^3 + O(7^53)
> QuotientRepresentation(L);
Univariate Quotient Polynomial Algebra in $.1 over 7-adic field mod 7^50
with modulus $.1^6 + O(7^50)*$.1^5 + O(7^50)*$.1^4 + O(7^50)*$.1^3 - (7^2 +
    O(7^52))*$.1^2 + O(7^50)*$.1 + 2*7^3 + O(7^53)
> RR, m := RamifiedRepresentation(L);
> RR;
Totally ramified extension defined by the polynomial x^2 +
    41709273235569453711334820170343703378866 3*$.1^2 +
    58619460221835676233637925289507569853713 7*$.1 -
    13009411322463399816688753375590121409659 7
 over Unramified extension defined by the polynomial x^3 + 6*x + 2
 over 7-adic field mod 7^50
> m(L.1);
RR.1 + O(RR.1^92)
> RR.1 @@ m;
O(7^48)*$.1^5 + O(7^48)*$.1^4 + O(7^49)*$.1^3 + O(7^49)*$.1^2 + $.1 + O(7^50)
> CoefficientRing(RR).1 @@ m;
O(7^34)*$.1^5 - (954564700580430506024960512238*7^-1 + O(7^35))*$.1^4 +
    O(7^36)*$.1^3 + (10312136871155901743985045546 31*7^-1 + O(7^35))*$.1^2 +
    O(7^37)*$.1 + 13128487736606729510635017356 8*7 + O(7^36)
```

`AssignNames(~L, S)`

Assign the name in the sequence S to the generator of the extension defining the local field L.

`Name(L, i)`

Return the generator of the local field L which has assigned to it the name in the sequence S which was input to `AssignNames`. The only valid input for i is 1.

`Discriminant(L)`

Return the discriminant of the local field L.

`ResidueClassField(L)`

Return the residue class field of the maximal order of the local field L and the map between L and its residue class field.

`RelativeField(L, m)`

Return L as an extension of the domain of the map m which should be a map from a subfield of L (having the same coefficient ring as L) into L.

51.3.1 Predicates on Fields

`IsRamified(L)`

Return whether the local field L has an non trivial ramified subfield, that is, the ramification degree of L is greater than 1.

`IsTamelyRamified(L)`

`IsWildlyRamified(L)`

Return whether the local field L is tamely or wildly ramified.

`IsTotallyRamified(L)`

Return whether the local field L is a totally ramified extension, that is, L has a trivial inertial subfield.

`IsUnramified(L)`

Return whether the local field L is equal to its inertial subfield.

51.4 Maximal Order

`IntegralBasis(L)`

Return a basis for the maximal order of the local field L.

`IsIntegral(a)`

Return whether the local field element a lies in the maximal order of its parent L and a sequence giving the coordinates of a with respect to the integral basis of L if so.

Example H51E5

We construct a local field and compute an integral basis for it.

```
> P<x> := PolynomialRing(Integers());
> L := LocalField(pAdicField(7, 50), x^6 - 49*x^2 + 686);
> IntegralBasis(L);
[ 1 + O(7^50), (7^-1 + O(7^49))*$.1^2 + O(7^49)*$.1 + O(7^49), (7^-2 +
    O(7^48))*$.1^4 + O(7^48)*$.1^3 + O(7^48)*$.1^2 + O(7^50)*$.1 + O(7^50), $.1
    + O(7^50), (7^-1 + O(7^49))*$.1^3 + O(7^49)*$.1^2 + O(7^49)*$.1 + O(7^99),
    (7^-2 + O(7^48))*$.1^5 + O(7^48)*$.1^4 + O(7^48)*$.1^3 + O(7^50)*$.1^2 +
    O(7^50)*$.1 + O(7^50) ]
```

51.5 Homomorphisms from Fields

`hom< L → R | a >`

`hom< L → R | cfm, a >`

Return the homomorphism from the local field L into the ring R whose image of the generator of L is a and whose action on the coefficient field of L is given by cfm if given.

51.6 Automorphisms and Galois Theory

`FrobeniusAutomorphism(L)`

Return the automorphism of the unramified extension L which is the lift of the frobenius automorphism on the residue class field of L.

`AutomorphismGroup(L)`

Return the automorphism group of the local field L and a map from the group to the parent of automorphisms of L.

`DecompositionGroup(L)`

`InertiaGroup(L)`

`RamificationGroup(L, i)`

Return the subgroup of the automorphism group of the local field L whose elements are the automorphisms (represented as group elements) σ such that $v(\sigma(z) - z) \geq i+1$. The decomposition group is the -1th ramification group and the inertia group is the 0th ramification group.

`FixedField(L, G)`

Return the subfield of the local field L which is fixed by the automorphisms (represented as group elements) in the subgroup G of the automorphism group of L.

Example H51E6

The automorphism and inertia groups of a local field are computed and their fixed fields examined.

```
> P<x> := PolynomialRing(Integers());
> L := LocalField(pAdicField(7, 50), x^6 - 49*x^2 + 686);
> A, am := AutomorphismGroup(L);
> am(Random(A));
Mapping from: RngLocA: L to RngLocA: L
> $1(L.1);
-(279674609046925265141076018485*7^-2 + O(7^34))*$.1^5 + O(7^35)*$.1^4 +
    (10359052517489881294588814641 23*7^-1 + O(7^35))*$.1^3 + O(7^36)*$.1^2 -
    (10094439077108649085019837355 01 + O(7^36))*$.1 + O(7^37)
> FixedField(L, A);
Extension of 7-adic field mod 7^50 by (1 + O(7^33))*x + O(7^33)
```

```
> InertiaGroup(L);
Permutation group acting on a set of cardinality 6
    Id($)
    (1, 2)(3, 5)(4, 6)
> FixedField(L, InertiaGroup(L));
Extension of 7-adic field mod 7^50 by (1 + O(7^37))*x^3 - (2*7^2 + O(7^37))*x^2
    + (7^4 + O(7^37))*x - 4*7^6 + O(7^37)
```

51.7 Local Field Elements

`L ! r`

Return the element of the local field L described by r where r may be anything which is coercible into the quotient representation of L.

`L . i`

Return the generator of the local field L. The only valid input for i is 1.

`InertialElement(L)`

Return a generator for the inertial subfield of the local field L.

`UniformizingElement(L)`

Return an element of the local field L of valuation 1.

51.7.1 Arithmetic

`a * b` `a + b` `a - b` `- a` `a ^ n` `a / b`

51.7.2 Predicates on Elements

`a eq b`

Return whether the local field elements a and b are considered equal.

`IsOne(a)`
`IsMinusOne(a)`

Return whether the local field element a is known to be 1 or -1 to the precision of the field.

`IsWeaklyZero(a)`

Return whether the local field element a is not known to be non zero.

`IsZero(a)`

Return whether the local field element a is known to be zero.

51.7.3 Other Operations on Elements

`Valuation(a)`

The valuation of the element a in a local field.

`RelativePrecision(a)`

The relative precision of the element a in a local field.

`Eltseq(a)`

Return the coefficients of powers of the generator of the parent of a in a.

`RepresentationMatrix(a)`

The representation matrix of the element a of a local field.

Example H51E7

Continuing from the first example we have :

```
> UniformizingElement(L);
a^2 + (6 + O(7^50))*a + 3 + O(7^50)
> InertialElement(L);
a + O(7^50)
> Valuation(UniformizingElement(L));
1
> Valuation(InertialElement(L));
0
> Eltseq(UniformizingElement(L));
[ 3 + O(7^50), 6 + O(7^50), 1 + O(7^50) ]
```

51.8 Polynomials over General Local Fields

Polynomials over local fields can be factored and their roots computed.

`Factorization(f)`

`Certificates` BoolElt *Default :* `false`

The factorization of the polynomial f over a local field defined by an arbitrary polynomial. The factorization is returned as a sequence of tuples of prime polynomials and exponents along with a scalar factor. If the parameter `Certificates` is `true` then certificates proving the primality of each prime are also returned.

`SuggestedPrecision(f)`

For a polynomial f over a general local field, return a precision at which the factorization of f as given by `Factorization(f)` will be Hensel liftable to the correct factorization.

The precision returned is not guaranteed to be enough to obtain a factorization of the polynomial. It may be that a correct factorization cannot be found at that precision but may be possible with a little more precision.

`Roots(f)`

`Roots(f, R)`

The roots of the polynomial f over the general local field R where R is taken to be the coefficient ring of f if it is not given.

Example H51E8

```
> Q3:=pAdicField(3,40);
> Q3X<x>:=PolynomialRing(Q3);
> L<a>:=LocalField(Q3,x^6-6*x^4+9*x^2-27);
> Factorization(Polynomial(L,x^6-6*x^4+9*x^2-27));
[
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + (5*3^35 + O(3^38))*a^5 + O(3^38)*a^4 - (10*3^36 +
        O(3^39))*a^3 + O(3^39)*a^2 + (2701703435345984179 + O(3^40))*a +
        O(3^40), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + -(29642867960*3^-1 + O(3^23))*a^5 + O(3^24)*a^4 +
        (148214339800*3^-1 + O(3^24))*a^3 + O(3^25)*a^2 - (116512378274*3 +
        O(3^25))*a + O(3^26), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + -(29642867960*3^-1 + O(3^23))*a^5 + O(3^24)*a^4 +
        (148214339800*3^-1 + O(3^24))*a^3 + O(3^25)*a^2 - (349537134821 +
        O(3^25))*a + O(3^26), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + -(5*3^35 + O(3^38))*a^5 + O(3^38)*a^4 + (10*3^36 +
        O(3^39))*a^3 + O(3^39)*a^2 - (2701703435345984179 + O(3^40))*a +
```

```
        O(3^40), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + (29642867960*3^-1 + O(3^23))*a^5 + O(3^24)*a^4 -
        (148214339800*3^-1 + O(3^24))*a^3 + O(3^25)*a^2 + (116512378274*3 +
        O(3^25))*a + O(3^26), 1>,
    <(O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + (1 +
        O(3^40)))*$.1 + (29642867960*3^-1 + O(3^23))*a^5 + O(3^24)*a^4 -
        (148214339800*3^-1 + O(3^24))*a^3 + O(3^25)*a^2 + (349537134821 +
        O(3^25))*a + O(3^26), 1>
]
O(3^38)*a^5 + O(3^38)*a^4 + O(3^39)*a^3 + O(3^39)*a^2 + O(3^40)*a + 1 + O(3^40)
```

52 ALGEBRAIC POWER SERIES RINGS

Chapter 52
ALGEBRAIC POWER SERIES RINGS

52.1 Introduction

Algebraic Power Series are a lazy representation of multivariate power series with fractional exponents, which are roots of univariate polynomials with coefficients in multivariate polynomial rings. The functionality allows the "lazy" computation of the power series expansion to any finite degree, this being well-determined by the defining algebraic equation.

The package was designed with the computation of formal resolutions of singularities of surfaces in mind but should provide a useful general tool for users. As well as allowing definition directly from a polynomial equation, the user can compose algebraic power series and recursively define series which are roots of polynomials whose coefficients are polynomial functions in other algebraic power series. There are also functions for basic arithmetic operations and tests for exact equality and the like.

The defined series must be expandable in fractional (positive) powers of the base variables. This is true for all roots of a *quasi-ordinary* polynomial possibly after a finite extension of the base field.

The package was designed and implemented by Tobias Beck at the RICAM institute in Linz, Austria. Some low-level adaptations for added efficiency and integration were carried out by the MAGMA group. The algorithms are described in [Bec07, Sec. 4].

52.2 Basics

In MAGMA, algebraic power series are represented in a hybrid lazy-exact way. Eventually every power series is given by a defining polynomial and a sufficiently large initial segment. Intermediate operations are represented in a lazy way. This makes it possible to compute both quickly and to high precision if necessary.

Note, however, that decision procedures may be very time intensive. In the sequel we have indicated in each function whether it is fast or whether it has to be used with care.

52.2.1 Data Structures

Algebraic power series are of type `RngPowAlgElt`.

There are two types of algebraic series: `Atomic` and `Substitution` which we sometimes refer to briefly as type A or type B. There is no difference in the functionality available but they are structurally different. The user can determine the type of a series, if (s)he desires, from the attribute `type` which is 0 for type A and 1 for type B.

`Atomic` series are the basic type that are given directly as the root of a univariate polynomial $p(z)$ with given initial expansion. $p(z)$ comes from a polynomial $f(z)$ with coefficients in a multivariate polynomial ring. Either $p = f$ or p is determined from f by

evaluating the coefficients of f at a given array, *subs*, of algebraic power series. This allows the construction of algebraic power series as roots of polynomials over finitely-generated fields of already-constructed series. Most of the constructors return a series of this type.

`Substitution` series allow the composition of algebraic power series. The principal defining data is an algebraic power series s in n variables and an array of n algebraic series that are substituted into this. In fact, the substitution is not necessarily direct, but through n given elements in the dual lattice of the exponent lattice of s as explained in the constructor `EvaluationPowerSeries`.

Both types of series have an associated exponent lattice specified by two components: Γ, a sublattice of a standard integral lattice, and e, a positive integer. The expansion of the series will in general have fractional exponents and e is the LCM of the denominators of these (e may be 1). The finite expansions that are returned are always integral-exponent multivariate polynomials. The *actual* mathematical expansion is derived from this return value by dividing all exponents by e. With this scaling up of fractional exponents by e to get integral exponents, all exponent vectors for monomials occurring in the expansion (up to any degree) will lie in the lattice Γ. So the actual exponent lattice for the series is $(1/e)\Gamma$.

The user doesn't have to worry too much about the lattice Γ. It is automatically computed by most of the constructors for algebraic power series and the default of the standard integral lattice can always be used (assuming e is correct!). It's utility is that, in computing the exponent lattice for composite constructions on series, factors of e may be cancelled out from the resulting lattice. So an algebraic construction involving series with non-integral exponents may produce a result with only integral exponents. Such lattice computations are carried out automatically.

Important note: To speed up some of the basic internal polynomial operations, it is assumed that the domain of an algebraic power series is a multivariate polynomial ring with a *degree* ordering (`glex` or `grevlex`). Attempts to create series using polynomial rings with a non-degree ordering will result in a user error.

52.2.2 Verbose Output

A verbose flag `AlgSeries` exists which can take values `true`, `false`, 0 or 1. Setting to `true` (or 1) will output information on the progress of some of the potentially more time-consuming intrinsics.

52.3 Constructors

Using constructors one can construct power series starting from polynomial data or using other power series recursively.

`PolyToSeries(s)`

Given a multivariate polynomial s, returns the series representation of s.

`AlgebraicPowerSeries(dp, ip, L, e)`

`subs` SeqEnum *Default :* []

Define a power series root of a polynomial p using an initial expansion ip and its exponent lattice $1/\texttt{e}\ \texttt{L}$. The defining polynomial p is either dp when $subs$ is empty or obtained by substituting the elements of $subs$ into the variables of dp. The initial expansion has to be sufficiently long in order to uniquely identify a root, see [Bec07, Cond. 4.3]. In this initial finite expansion and in subsequent ones, the variables occurring actually represent e-th roots, *ie* $x_1x_2^2$ is really $x_1^{1/e}x_2^{2/e}$. All exponents of monomials occurring in these expansions and the coefficients of p should lie in L (although this is not checked in many places), monomials of p giving true values rather than e-th roots.

There are simpler constructors where L is omitted (when it is assumed to be the standard integral lattice) or e is (when it is taken as 1).

As noted earlier, no strong checks are performed at construction time on the correctness of L or e or whether ip is indeed the initial expansion of a unique root in this "raw data" constructor. Incorrect initial data will only be revealed when failure occurs in further expansion of the series. The preferred methods of series creation are `ImplicitFunction`, `EvaluationPowerSeries` and `RationalPuiseux` because in these cases the sanity checks are more easily verified.

`EvaluationPowerSeries(s, nu, v)`

Given a series s, a sequence nu of vectors in the dual of its exponent lattice of s and a sequence v (of the same length) of power series in some other common domain (with compatible coefficient field). Returns the series obtained by substituting $\underline{x}^{\mu} \mapsto \prod_i \texttt{v[i]}^{\langle \texttt{nu[i]},\mu\rangle}$. This requires that nu and v fulfill a certain condition on the orders to guarantee convergence of the resulting series, see [Bec07, Cond. 4.6].

`ImplicitFunction(dp)`

`subs` SeqEnum *Default :* []

The unique series with zero constant term defined by a polynomial $p \in k[x_1, .., x_n][z]$ or $k[[x_1, \ldots, x_n]][z]$, fulfilling the conditions of the implicit function theorem, *i.e.*, $p(0, \ldots, 0) = 0$ and $\partial p/\partial z(0, \ldots, 0) \neq 0$. The polynomial p is equal to dp possibly substituted with the series in $subs$ as in `AlgebraicPowerSeries`. dp should have coefficients in a multivariate polynomial ring.

52.3.1 Rational Puiseux Expansions

Let $p \in k[[x_1, \ldots, x_n]][z]$ be a quasi-ordinary polynomial over a field k of characteristic zero. This means that p is non-zero, squarefree and monic (i.e., its leading coefficient in z is a unit in the power series ring) and if $d \in k[[x_1, \ldots, x_n]]$ denotes its discriminant then $d = x_1^{e_1} \cdots x_n^{e_n} u(x_1, \ldots, x_n)$ where u is a unit in the power series ring.

In this case the Theorem of Jung-Abhyankar states that p has $\deg(p)$ distinct Puiseux series roots, i.e., power series roots with fractionary exponents and coefficients in the algebraic closure of k.

These roots are computed by a generalization of the so called Newton-Puiseux algorithm. Also Duval's extension for computing rational parametrization has been implemented.

`RationalPuiseux(p)`

`Gamma`	LATTICE	*Default :* `StandardLattice`
`subs`	SEQENUM	*Default :* []
`Duval`	BOOLELT	*Default :* `false`
`OnlySingular`	BOOLELT	*Default :* `false`
`ExtName`	MONSTGELT	*Default :* "*gamma*"
`ExtCount`	RNGINTELT	*Default :* 0

We first specify the behavior of this function in the case that no special value of *subs* has been given. This function assumes that p is a univariate polynomial over a multivariate polynomial ring $S = k[x_1, ..., x_r]$ and that p is quasi-ordinary. In this case it will compute a set of rational parametrizations of p. Note that for reasons of efficiency the user has to make sure that p is actually quasi-ordinary! (Otherwise, further processing of the output may result in runtime errors.)

The first return value will be the exponent lattice of the input polynomial in the usual format $< \Gamma_0, e_0 >$. If the parameter `Gamma` has been specified, then Γ_0 = `Gamma` and $e_0 = 1$. In this case `Gamma` has to be an integral 'r'-dimensional lattice of full rank containing all the exponents of p. Otherwise Γ_0 will be set to the r-dimensional standard lattice and again $e_0 = 1$.

As a second value a complete list of rational parametrizations in the format $< \lambda, s, N, E >$ is returned. Here λ is a sequence of r field elements and s is a fractionary algebraic power series of type `RngPowAlgElt`. Let p_1 denote the image of p under the transformation $x^{\mu_i} \mapsto \lambda_i x^{\mu_i}$ where $(\mu_i)_i$ is the basis of the exponent lattice $e_0^{-1}\Gamma_0$ then s is a solution of p_1, i.e., we have $p_1(s) = 0$. Note that if neither `Gamma` nor `subs` have been supplied this just means that x_i is substituted by $\lambda_i x_i$. Finally N is the index of $e_0^{-1}\Gamma_0$ in the exponent lattice of s and E is the degree of the extension of the coefficient field needed for defining s.

The behavior described above corresponds to the Newton-Puiseux algorithm with Duval's trick. The field extensions that are used for expressing the series fulfill a certain minimality condition. If `Duval` is set to `false` then the function returns a complete set of representatives (up to conjugacy) of Puiseux series roots of the original polynomial p, in other words, the λ-vectors will always be vectors of ones.

If `OnlySingular` is set to `true` then only those parametrizations that correspond to singular branches are returned.

If the ground field has to be extended, the algebraic elements will be assigned the name `ExtName_`i where i starts from `ExtCount`. The last return value is the value of `ExtCount` plus the number of field extensions that have been introduced during the computation.

Finally, if the parameter `subs` is passed, then it has to be a sequence of r power series in a common domain and internally the variables in p will be substituted by

the corresponding series. Again the resulting polynomial has to be quasi-ordinary. In this case Γ_0 and e_0 are determined by building the sum of the exponent lattices of all series in `subs`. The parameter `Gamma` then has no effect.

For further details on the algorithm and other references see [Bec07, Sec. 4.3]

Example H52E1

We illustrate the constructors by examples. For displaying results we already use the command `Expand` that will be explained later.

```
> Q := Rationals(); Qs<s> := FunctionField(Q);
> Qxy<x,y> := PolynomialRing(Q, 2, "glex");
> Qxyz<z> := PolynomialRing(Qxy);
> Qst<t> := PolynomialRing(Qs, 1, "glex");
> Qstu<u> := PolynomialRing(Qst);
```

One can consider polynomials as series.

```
> s0 := PolyToSeries(1 - 3*x + x^2*y + y^20);
> Expand(s0, 10);
true x^2*y - 3*x + 1
```

One can define series by the implicit function theorem at the origin.

```
> s1 := ImplicitFunction(z*(1 - x - y) - x - y);
> Expand(s1, 4);
true x^3 + 3*x^2*y + 3*x*y^2 + y^3 + x^2 + 2*x*y + y^2 + x + y
```

One can define a power series if an initial expansion is known. Note that the following power series has exponent lattice $\mathbf{Z}(\frac{1}{5}, -\frac{2}{5}) + \mathbf{Z}(\frac{2}{5}, \frac{1}{5})$ but its "expansions" are polynomials supported on $\mathbf{Z}(1, -2) + \mathbf{Z}(2, 1)$.

```
> defpol := (1+5*y+10*y^3+10*y^2+5*y^4+y^5)*z^5+(-20*y^3*x-
>   30*y^2*x-5*y^4*x-5*x-20*y*x)*z^4+(10*x^2+30*y^2*x^2+10*y^3*x^2+
>   30*x^2*y)*z^3+(-20*y*x^3-10*x^3-10*y^2*x^3)*z^2+
>   (5*y*x^4+5*x^4)*z-x^5-x^2*y;
> Gamma := Lattice(RMatrixSpace(Integers(), 2, 2) ! [1,-2, 2,1]);
> init := x^2*y;
> s2 := AlgebraicPowerSeries(defpol, init, Gamma, 5);
> Expand(s2, 20);
true
-x^2*y^16 + x^5*y^10 + x^2*y^11 - x^5*y^5 - x^2*y^6 + x^5 + x^2*y
```

We can "substitute" series into each other.

```
> X := AlgebraicPowerSeries(u^3-t+s*t^2, t, StandardLattice(1), 3);
> Y := PolyToSeries(t);
> duals := [RSpace(Integers(), 2) | [1, 3], [2, 1]];
> s3 := EvaluationPowerSeries(s2, duals, [X, Y]);
> Expand(s3, 13);
```

```
true (-1/9*s^2 - 1/3*s - 1)*t^10 + (-1/3*s + 1)*t^7 + t^4
```

We can compute all the Puiseux series roots of a quasi-ordinary polynomial up to conjugacy over $\mathbf{Q}$.

```
> qopol := z^6 + 3*x*y^2*z^4 + x*y*z^3 + 3*x^2*y^4*z^2 + x^3*y^6;
> _, prms := RationalPuiseux(qopol : Duval := false); prms;
[*
    <[ 1, 1 ], Algebraic power series -x*y, 3, 1>,
    <[ 1, 1 ], Algebraic power series gamma_0*x*y, 3, 2>,
    <[ 1, 1 ], Algebraic power series -x^2*y^5, 3, 1>,
    <[ 1, 1 ], Algebraic power series gamma_1*x^2*y^5, 3, 2>
*]
> Domain(prms[2][2]); ExponentLattice(prms[2][2]);
Polynomial ring of rank 2 over Number Field with defining
polynomial $.1^2 - $.1 + 1 over the Rational Field
Graded Lexicographical Order
Variables: x, y
<
    Lattice of rank 2 and degree 2
    Basis:
    ( 1  1)
    ( 1 -2),
    3
>
> Expand(prms[2][2], 15);
true x^3*y^9 + (gamma_0 - 1)*x^2*y^5 + gamma_0*x*y
```

We find that the sum over all field extensions $1 + 2 + 1 + 2 = 6$ is equal to the degree of the defining polynomial `qopol`. The third parametrization involves a field extension of $\mathbf{Q}$ by `gamma_0` s.t. $\texttt{gamma\_0}^2 - \texttt{gamma\_0} + 1 = 0$ and an extension of the exponent lattice to $\mathbf{Z}(\frac{1}{3}, \frac{1}{3}) + \mathbf{Z}(\frac{1}{3}, -\frac{2}{3})$. It turns out that the field extension is not necessary if we are only interested in parametrizations.

```
> _, prms := RationalPuiseux(qopol : Duval := true); prms;
[*
    <[ -1, -1 ], Algebraic power series -x*y, 3, 1>,
    <[ -1, 1 ], Algebraic power series -x^2*y^5, 3, 1>
*]
```

No field extensions have been introduced, but this required the application of automorphisms $\mathbf{Q}[[x, y]] \to \mathbf{Q}[[x, y]]$ in advance (more precisely $x \mapsto -x, y \mapsto -y$ resp. $x \mapsto -x, y \mapsto y$). This time we can sum up the overall extension degrees (*i.e.*, for fields and lattices) $3 \cdot 1 + 3 \cdot 1 = 6$ to the degree of `qopol`.

52.4 Accessors and Expansion

The following functions provide an interface to conveniently extract information from a power series defined as above.

`Domain(s)`

Return the multivariate polynomial ring that is used for approximating the series s by its truncations.

`ExponentLattice(s)`

Return the exponent lattice $(1/e)\Gamma$ of the series as tuple (Γ, e) where Γ is an integral lattice and e is an integer.

`DefiningPolynomial(s)`

Return a defining polynomial of the series which is a squarefree univariate polynomial over the multivariate polynomial domain `Domain(s)`. In the case of series defined with substitutions, the computation may be expensive and can involve recursive resultant computations.

`Order(s)`

`TestZero` BoolElt *Default :* `false`

Given a series s, return the integral order (total degree of smallest non-zero term occurring) of its expansion as returned by `Expand`, *i.e.*, its fractionary order times the exponent denominator. If s is zero, this function will **not terminate**. Set `TestZero` to `true` to get a return value -1 in this case, but note that this involves the computationally complex call `IsZero`.

`Expand(s,ord)`

Given the power series β which is represented by s, let α be the result of substituting variables $x_i \mapsto x_i^e$ where e is taken from the output of `ExponentLattice(s)` (*i.e.*, α is β without exponent denominators). Returns `true` and the truncation of α modulo terms of order greater or equal `ord`. A return of `false` indicates that the representation is inconsistent (which should only happen when `RationalPuiseux` is called with non quasi-ordinary input or `AlgebraicPowerSeries` is used inconsistently).

Example H52E2

We can study the series `s3`.

```
> Domain(s3);
Polynomial ring of rank 1 over Univariate rational function
field over Rational Field
Graded Lexicographical Order
Variables: t
> ExponentLattice(s3);
<
    Standard Lattice of rank 1 and degree 1,
```

```
    3
>
> DefiningPolynomial(s3);
(s^45*t^45 - ... - 15*s^30*t^2 - s^30)*u^15 + ... +
(5*s^37*t^41 - ... + 120*s^31*t^19 - 30*s^30*t^18)*u^3 -
s^35*t^40 + ... - 5*s^31*t^21 + s^30*t^20
> Order(s3);
4
```

These commands reveal the following about `s3`: It is a power series in $\mathbf{Q}(s)[[t^{1/3}]]$, because it is approximated in $\mathbf{Q}(s)[t]$ and has exponent lattice $\frac{1}{3}\mathbf{Z}$. A defining polynomial in $\mathbf{Q}(s)[t][u]$ was also computed. (Recall that `s3` has been defined recursively.) The order is $\frac{4}{3}$ which we know already from a previous expansion.

52.5 Arithmetic

There are functions to perform basic arithmetic operations (addition, subtraction etc.) on power series.

`AlgComb(c,ss)`

Given a polynomial c in r variables and a sequence ss of r power series (in a common domain with compatible coefficient field) return the series obtained by substituting the elements of ss for the variables of c. This allows the construction of completely arbitrary algebraic combinations.

`s + t`
`s - t`
`s * t`

Add, subtract or multiply two power series.

Example H52E3

One can easily substitute power series into polynomials.

```
> // construct the series s0^2+s1^2
> h0 := AlgComb(x^2 + y^2, [s0,s1]);
> Expand(h0, 3);
true 10*x^2 + 2*x*y + y^2 - 6*x + 1
```

This includes of course the ring operations.

```
> h1 := Add(s1, PolyToSeries(One(Qxy)));
> Expand(h1, 4);
true
x^3 + 3*x^2*y + 3*x*y^2 + y^3 + x^2 + 2*x*y + y^2 + x + y + 1
> h2 := Mult(h1, PolyToSeries(1 - x - y));
> Expand(h2, 4);
true 1
```

```
> h3 := Add(h2, PolyToSeries(-One(Qxy)));
> Expand(h3, 4);
true 0
```

52.6 Predicates

The following functions provide decision algorithms for algebraic power series. They may involve **recursive resultant computations**, hence, have a high complexity and should be used with care.

`IsZero(s)`

Decides if the series is zero.

`s eq t`

Decides if two series are equal.

`IsPolynomial(s)`

Decides whether the series is actually a polynomial (with integral exponents) in the multivariate polynomial domain as returned by `Domain(s)`. In the positive case also returns that polynomial. This function relies on `SimplifyRep`.

Example H52E4

The previous computations suggest that `h2` is 1, in particular it is polynomial (in contrast to `h1`). In this case `h3` would be zero.

```
> IsPolynomial(h1);
false
> IsPolynomial(h2);
true 1
> IsEqual(h2, PolyToSeries(One(Qxy)));
true
> IsZero(h3);
true
```

52.7 Modifiers

The following functions modify the representation of the power series or apply a simple automorphism.

`ScaleGenerators(s,ls)`

Let $\{\gamma_i\}_i$ be the basis (determined from the representation chosen by MAGMA) of the exponent lattice of the series s, and let $\sigma : \underline{x}^{\gamma_i} \mapsto ls[i]\underline{x}^{\gamma_i}$. Return the series $\sigma(s)$.

`ChangeRing(s,R)`

If R is a multivariate polynomial domain compatible with the approximation domain `Domain(s)`, return the same power series with new approximation domain R. This is sort of a "coercion between power series rings".

`SimplifyRep(s)`

`Factorizing` BoolElt *Default :* `true`

"Simplifies" the internal representation of a series. The result will be a series of atomic type without recursive (substitution) dependencies on other power series. The defining polynomial of the simplified series will be irreducible and therefore a minimal polynomial over `Domain(s)` (unless `Factorizing` is `false` when it will only be guaranteed to be squarefree). After the simplification, `DefiningPolynomial` returns this polynomial, which can be useful (*e.g.*, for `IsPolynomial`). However, experience shows that the resulting representation is in general neither simple nor more efficient for subsequent computations.

There is a **dangerous pitfall:**

Assume we have a series represented by a tree with nodes of type A and B. Assume further that the leaves have been constructed by `RationalPuiseux` with parameter `Gamma` set to some value. Then the intention was probably to work over the subring of a polynomial ring with restricted support. If now `SimplifyRep`, with `Factorizing` as `true`, is called, then a minimal polynomial over the whole polynomial ground ring is computed which is maybe not what one wants.

Example H52E5

We can modify `s2` by mapping generators (of Laurent polynomials) $x^{1/5}y^{-2/5} \mapsto 3x^{1/5}y^{-2/5}$ and $x^{2/5}y^{1/5} \mapsto 4x^{2/5}y^{1/5}$.

```
> Expand(ScaleGenerators(s2, [3,4]), 15);
true
64/81*x^2*y^11 - 64/3*x^5*y^5 - 16/9*x^2*y^6 + 48*x^5 + 4*x^2*y
```

One can naturally view `h1` as a series in $\mathbf{Q}(i)[[u, v]]$.

```
> Qi<i> := NumberField(R.1^2 + 1) where R is PolynomialRing(Q);
> Qiuv<u,v> := PolynomialRing(Qi, 2, "glex");
> h4 := ChangeRing(s1, Qiuv);
> Expand(h4, 4); Domain(h4);
true u^3 + 3*u^2*v + 3*u*v^2 + v^3 + u^2 + 2*u*v + v^2 + u + v
```

```
Polynomial ring of rank 2 over Qi
Graded Lexicographical Order
Variables: u, v
```

We have seen that the power series `h3` is zero, but its representation does not show this immediately. We can "explicitize" its representation.

```
> SimplifyRep(h3 : Factorizing := true);
Algebraic power series
0
> DefiningPolynomial($1);
z
```

52.8 Bibliography

[**Bec07**] Tobias Beck. Formal Desingularization of Surfaces – The Jung Method Revisited –. Technical Report 2007-31, RICAM, December 2007.
URL:http://www.ricam.oeaw.ac.at/publications/reports/.

PART VIII
MODULES

53 INTRODUCTION TO MODULES

Chapter 53
INTRODUCTION TO MODULES

53.1 Overview

This section of the Handbook describes the Magma facilities for linear algebra and module theory. Since this topic is absolutely fundamental for much of algebra, it is important that the reader understand how linear algebra is presented in Magma. The structures covered under this heading include:

(i) Vector spaces;

(ii) Inner product spaces;

(iii) Modules defined over any ring or algebra

(iv) $R[G]$-modules, where R is a ring and G is a group;

(v) Linear transformations and R-module homomorphisms.

Although vector spaces are, of course, subsumed under general modules, we present a separate treatment of them, firstly because of their importance and secondly because their theory is somewhat cleaner than that of a general module. Magma users who are unfamiliar with the language of module theory will find a self-contained treatment of the vector space machinery in Chapter 28.

In the Magma universe, rectangular matrices are regarded as forming a module (actually a bimodule). We shall regard a rectangular matrix as the concrete realization of a linear transformation or R-module homomorphism. Thus, an $m \times n$ matrix over a ring R is considered to be an element of the module $\mathrm{Hom}_R(M, N)$. Reflecting the dual nature of matrices, the $\mathrm{Hom}_R(M, N)$ operations include the standard module-theoretic operations as well as operations that interpret an element of $\mathrm{Hom}_R(M, N)$ as a homomorphism.

53.2 General Modules

A module M is always regarded as a submodule or quotient module of the free module $S^{(n)}$, for some ring or algebra S. The types of module that are definable in the system fall into three classes:

(a) **Abstract Modules**: Given a ring R, a set M and a mapping $\phi : R \times M \to M$, the pair (M, ϕ) will be referred to as an *abstract* R-module. Because of the very general nature of this construction, only the basic arithmetic operations may be applied to modules of this type.

(b) **Modules with Scalar Action**: Given a general ring R, an R-module with *scalar action* is a submodule or quotient module of the free R-module $R^{(n)}$, where the action is that of ring multiplication in R.

(c) **Modules with Matrix Action**: Let R be a PIR and suppose S is a R-algebra. Thus there exists a ring homomorphism $\phi : R \to S$, and so S is a (left) R-module with the

R-action defined by $r * s = \phi(r) * s$. Indeed, any S-module M is a (left) S-module with action defined by $r * m = \phi(r) * m$. Furthermore, if $\phi(R)$ lies in the centre of S, then S acts on M as a ring of R-module endomorphisms. Consequently, M is an S-module. We take M to be the free R-module $R^{(n)}$, and so the action of S on M is given by the action of a subring of $M_n(R)$ on M. Thus, given an R-algebra S, an S-module of the form $M = R^{(n)}$ may be specified by giving M together with a homomorphism of S into $M_n(R)$.

53.3 The Presentation of Submodules

Let N be a submodule of the module $M = R^{(m)}$. For simplicity, assume that N is free, and has dimension n where $n < m$. There are two ways in which N may be viewed:

(a) As a submodule embedded in M. Thus, though N is a module of dimension n, its elements are regarded as elements of M.

(b) As the module $R^{(n)}$ represented on a reduced basis, together with a morphism ϕ defining the inclusion of N into M.

The presentation (a) is the usual way submodules are regarded in elementary linear algebra. However, this presentation is inconvenient for more advanced applications. For example, many of the major functions available for studying an $R[G]$-module N expect that N is given relative to a reduced basis. We shall refer to a submodule presentation (a) as the *embedded presentation*, and (b) as the *reduced presentation*.

To provide the user with the maximum flexibility, Magma supports both forms of submodule presentation for the important classes of modules. Usually, a module calculation commences with the definition of one or two modules from which further modules are created by the operations of forming submodules, quotient modules and extensions. Let us call these latter modules *descendants* of the original module. Magma provides parallel creation functions which allow the user to choose the form of submodule presentation. Once that choice has been made, all descendants of the initial module(s) will follow the same presentation convention.

The module creation functions that select the embedded form are usually of the form *Qualifier*`Space`, while those that adopt the standard form are usually of the form *Qualifier*`Module`. Thus in the case of vector spaces the function `KSpace(K, n)` constructs the n-dimensional vector space over the field K, where submodules are to be presented in embedded form. On the other hand, `RModule(K, n)` constructs the n-dimensional vector space over the field K, where submodules are to be presented in reduced form.

54 FREE MODULES

Chapter 54
FREE MODULES

54.1 Introduction

54.1.1 Free Modules

This chapter describes the facilities provided for modules in the following representations:

Tuple Modules: Modules whose elements are n-tuples over a fixed ring R, i.e., modules $R^{(n)}$;

Matrix Modules: Modules whose elements are homomorphisms of modules, i.e., $Hom_R(M, N)$. The elements of these modules are $m \times n$ matrices over the ring R;

The ring R acts on the right of the module element by scalar multiplication. If R is not an Euclidean Domain then, currently, only arithmetic with vectors is supported. In particular, the ability to work with submodules and quotient modules is restricted to situations where R is either a field or Euclidean Domain.

In the first part of the chapter we describe the operations that apply to modules generally, while in the second half we describe the creation of modules $Hom_R(M, N)$ together with the operations that are specific to them. Insofar as elementary module-theoretic operations are concerned, there is no real difference between tuple modules and matrix modules except for the input and display of elements. Many special operations provided for matrices are described in the chapter on matrices.

The reader is referred to the chapter on vector spaces for descriptions of the extensive functionality provided for modules over fields.

54.1.2 Module Categories

The family of all finitely generated modules over a given ring R forms a category, while the set of all finitely generated modules forms a family of categories indexed by the ring R. In this family of categories, objects are modules and the morphisms are module homomorphisms. The category name for modules is `ModRng`. We distinguish the following subcategories of `ModRng`:

`ModTupFld` - the category of modules of n-tuples over a field;

`ModMatFld` - the category of modules of $m \times n$ matrices over a field;

`ModTupEd` - the category of modules of n-tuples over an euclidean domain.

`ModTupRng` - the category of modules of n-tuples over a ring;

`ModMatRng` - the category of modules of $m \times n$ matrices over a ring;

54.1.3 Presentation of Submodules

Let N be a free submodule of the R-module M. We have two alternative ways of presenting N. Firstly, we can present it on a set of generators that are elements of M; we call such a presentation an *embedded presentation*. Alternatively, given that N has rank r, we can present it as the module $S^{(r)}$, with appropriate action induced from the action of R on M. We call this presentation of N a *reduced presentation*.

The user can control the method of submodule presentation at the time of creation of an initial module through selection of the appropriate creation function. Thus, the function `RModule` will create a module with the convention that it and all its submodules and quotient modules will have their submodules presented in reduced form. The use of `RSpace`, on the other hand, signifies that submodules are to be presented in embedded form.

54.1.4 Notation

Throughout this chapter, R will denote a ring (possibly a field) while K will denote a field. The letters M and N will denote modules, while U and V will denote vector spaces.

54.2 Definition of a Module

54.2.1 Construction of Modules of n-tuples

`RSpace(R, n)`

`RModule(R, n)`

Given a ring R and a non-negative integer n, create the free right R-module $R^{(n)}$, consisting of all n-tuples over R. The module is created with the standard basis, $e_1, \ldots, e_n$, where e_i $(i = 1, \ldots, n)$ is the vector containing a 1 in the i-th position and zeros elsewhere.

The function `RModule` creates a module in reduced mode while `RSpace` creates a module in embedded mode.

`RSpace(R, n, F)`

Given a ring R, a non-negative integer n and a square $n \times n$ symmetric matrix F, create the free right R-module $R^{(n)}$ (in embedded form), with inner product matrix F. This is the same as `RSpace(R, n)`, except that the functions `Norm` and `InnerProduct` (see below) will be with respect to the inner product matrix F.

Example H54E1

We construct the module consisting of 6-tuples over the integers.

```
> Z := IntegerRing();
> M := RModule(Z, 6);
> M;
RModule M of dimension 6 with base ring Integer Ring
```

54.2.2 Construction of Modules of $m \times n$ Matrices

`RMatrixSpace(R, m, n)`

The module comprising all $m \times n$ matrices over the ring R.

54.2.3 Construction of a Module with Specified Basis

`RModuleWithBasis(Q)`

`RSpaceWithBasis(Q)`

`RSpaceWithBasis(a)`

Given a sequence Q (or matrix a) of k independent vectors each lying in a module M, construct the submodule of M of dimension k whose basis is Q (or the rows of a). The basis is echelonized internally but all functions which depend on the basis of the space (e.g. `Coordinates`) will use the given basis.

`RMatrixSpaceWithBasis(Q)`

The module of $m \times n$ matrices whose basis is given by the linearly independent matrices of the sequence Q.

54.3 Accessing Module Information

`M . i`

Given an R-module M and a positive integer i, return the i-th generator of M. The integer i must lie in the range $[1, r]$, where r is the number of generators for M.

`CoefficientRing(M)`

`BaseRing(M)`

`CoefficientRing(M)`

`BaseRing(M)`

`CoefficientField(M)`

`BaseField(M)`

Given an R-module M which is defined as a submodule of $S^{(n)}$, return the ring S.

`Generators(M)`

The generators for the R-module M, returned as a set.

`OverDimension(M)`

Given an R-module M which is an embedded submodule of the module $S^{(n)}$, return n.

`OverDimension(u)`

Given an element u of an embedded submodule of the module $S^{(n)}$, return n.

`Moduli(M)`

The column moduli of the module M over a euclidean domain.

`Parent(u)`

Given an element u belonging to the R-module M, return M.

`Generic(M)`

Given an R-module M which is a submodule of the module $R^{(n)}$, return the module $R^{(n)}$ as an R-module.

54.4 Standard Constructions

Given one or more existing modules, various standard constructions are available to construct new modules.

54.4.1 Changing the Coefficient Ring

`ChangeRing(M, S)`

Given a module M with base ring R, together with a ring S, construct the module N with base ring S obtained by coercing the components of elements of M into N, together with the homomorphism from M to N.

`ChangeRing(M, S, f)`

Given a module M with base ring R, together with a ring S, and a homomorphism $f : R \to S$, construct the module N with base ring S obtained by mapping the components of elements of M into N by f, together with the homomorphism from M to N.

`ChangeUniverse(~x, R)`

Change the coefficient ring of x to be R.

54.4.2 Direct Sums

`DirectSum(M, N)`

Given R-modules M and N, construct the direct sum D of M and N as an R-module. The embedding maps from M into D and from N into D respectively, and the projection maps from D onto M and from D onto N respectively are also returned.

`DirectSum(Q)`

Given a sequence Q of R-modules, construct the direct sum D of these modules. The embedding maps from each of the elements of Q into D and the projection maps from D onto each of the elements of Q are also returned.

54.5 Elements

54.6 Construction of Elements

`elt< M | a1, ..., an >`

Given a module M with base module $S^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to S, construct the element $m = (a_1, \ldots, a_n)$ of M. Note that if m is not an element of M, an error will result.

`M ! Q`

Given the module M with base module $S^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to S, construct the element $m = (a_1, \ldots, a_n)$ of M. Note that if m is not an element of M, an error will result.

`CharacteristicVector(M, S)`

Given a submodule M of the module $R^{(n)}$ together with a set S of integers lying in the interval $[1, n]$, return the characteristic number of S as a vector of R.

`Zero(M)`
`M ! 0`

The zero element for the R-module M.

`Random(M)`

Given a module M defined over a finite ring or field, return a random vector.

Example H54E2

We create the module of 4-tuples over the polynomial ring $\mathbf{Z}[x]$ and define various elements.

```
> P<x> := PolynomialRing(IntegerRing());
> M := RModule(P, 4);
> a := elt< M | 1+x, -x, 2+x, 0 >;
> a;
(x + 1    -x x + 2     0)
> b := M ! [ 1+x+x^2, 0, 1-x^7, 2*x ];
> b;
(x^2 + x + 1           0    -x^7 + 1         2*x)
> zero := M ! 0;
> zero;
(0 0 0 0)
```

54.6.1 Deconstruction of Elements

`ElementToSequence(u)`

`Eltseq(u)`

Given an element u belonging to the R-module M, return u in the form of a sequence Q of elements of R. Thus, if u is an element of $R^{(n)}$, then $Q[i] = u[i], 1 \leq i \leq n$, while if u is an element of $R^{(m \times n)}$, then $Q[(i-1)n+j] = u[i,j], 1 \leq i \leq m, 1 \leq j \leq n$.

54.6.2 Operations on Module Elements

54.6.2.1 Arithmetic

`u + v`

Sum of the elements u and v, where u and v lie in the same R-module M.

`-u`

Additive inverse of the element u.

`u - v`

Difference of the elements u and v, where u and v lie in the same R-module M.

`x * u`

Given an element x belonging to a ring R, and an element u belonging to the left R-module M, return the (left) scalar product $x * u$ as an element of M.

`u * x`

Given an element x belonging to a ring R, and an element u belonging to the right R-module M, return the (right) scalar product $u * x$ as an element of M.

`u / x`

Given a non-zero element x belonging to a field K, and an element u belonging to the right K-module M, return the scalar product $u * (1/x)$ as an element of M.

54.6.2.2 Indexing

`u[i]`

Given an element u belonging to a submodule M of the R-module $R^{(n)}$ and a positive integer i, $1 \leq i \leq n$, return the i-th component of u (as an element of the ring R).

`u[i] := x`

Given an element u belonging to a submodule M of the R-module $T = R^{(n)}$, a positive integer i, $1 \leq i \leq n$, and an element x of the ring R, redefine the i-th component of u to be x. The parent of u is changed to T (since the modified element u need not lie in M).

54.6.2.3 Normalization

`Normalize(u)`

`Normalise(u)`

The element u must belong to an R-module, where R is either a field, the ring of integers or a univariate polynomial ring over a field. Assume that the vector u is non-zero. If R is a field then `Normalize` returns $\frac{1}{a} * u$, where a is the first non-zero component of u. If R is the ring of integers, `Normalize` returns $\epsilon * u$, where ϵ is $+1$ if the first non-zero component of u is positive, and -1 otherwise. If R is the polynomial ring $K[x]$, K a field, then `Normalize` returns $\frac{1}{a} * u$, where a is the leading coefficient of the first non-zero (polynomial) component of u. If u is the zero vector, it is returned as the value of this function.

`Rotate(u, k)`

Given a vector u, return the vector obtained from u by rotating by k coordinate positions.

`Rotate(~u, k)`

Given a vector u, destructively rotate u by k coordinate positions.

Example H54E3

We illustrate the use of the arithmetic operators for module elements by applying them to elements of the module of 4-tuples over the polynomial ring $\mathbf{Z}[x]$.

```
> P<x> := PolynomialRing(IntegerRing());
> M := RModule(P, 4);
> a :=  M ! [ 1+x, -x, 2+x, 0 ];
> b := M ! [ 1+x+x^2, 0, 1-x^7, 2*x ];
> a + b;
(x^2 + 2*x + 2    -x    -x^7 + x + 3    2*x)
> -a;
(-x - 1      x -x - 2      0)
> a - b;
(       -x^2          -x x^7 + x + 1        -2*x)
> (1-x + x^2)*a;
(x^3 + 1    -x^3 + x^2 - x    x^3 + x^2 - x + 2    0)
> a*(1-x);
(    -x^2 + 1      x^2 - x -x^2 - x + 2            0)
> a[3];
x + 2
> a[3] := x - 2;
> a;
(x + 1    -x x - 2     0)
> ElementToSequence(a - b);
[
    -x^2,
```

```
    -x,
    x^7 + x - 3,
    -2*x
]
> Support(a);
{ 1, 2, 3 }
```

54.6.3 Properties of Vectors

`IsZero(u)`

Returns `true` if the element u of the R-module M is the zero element.

`Depth(v)`

The index of the first non-zero entry of the vector v (0 if none such).

`Support(u)`

A set of integers giving the positions of the non-zero components of the vector u.

`Weight(u)`

The number of non-zero components of the vector u.

54.6.4 Inner Products

`(u, v)`

`InnerProduct(u, v)`

Return the inner product of the vectors u and v with respect to the inner product defined on the space. If an inner product matrix F is given when the space is created, then this is defined to be $u \cdot F \cdot v^{tr}$. Otherwise, this is simply $u \cdot v^{tr}$.

`Norm(u)`

Return the norm product of the vector u with respect to the inner product defined on the space. If an inner product matrix F is given when the space is created, then this is defined to be $u \cdot F \cdot u^{tr}$. Otherwise, this is simply $u \cdot u^{tr}$.

54.7 Bases

The application of the functions in this section is restricted either to vector spaces or to torsion-free modules over a Euclidean Domain.

For a full description of the basis functions for a module defined over a field, the reader is referred to the chapter on vector spaces.

`Basis(M)`

The current basis for the free R-module M, R an ED, returned as a sequence of module elements.

`Rank(M)`

The rank of the free R-module M.

`Coordinates(M, u)`

Given a vector u belonging to the rank r free R-module M, R an Euclidean Domain, with basis $u_1, \ldots, u_r$, return a sequence $[a_1, \ldots, a_r]$ giving the coordinates of u relative to the M-basis: $u = a_1 * u_1 + \cdots + a_r * u_r$.

54.8 Submodules

54.8.1 Construction of Submodules

Submodules may be defined for any type of module. However, functions that depend upon membership testing are only implemented for modules over Euclidean Domains (EDs). The conventions defining the presentations of submodules are as follows:

If M has been created using the function `RSpace`, then every submodule of M is given in terms of a generating set consisting of elements of M, i.e. by means of an embedded generating set.

If M has been created using the function `RModule`, then every submodule of M is given in terms of a reduced basis.

`sub< M | L >`

Given an R-module M, construct the submodule N generated by the elements of M specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of R defining an element of M;

(b) A set or sequence whose terms are elements of M;

(c) A submodule of M;

(d) A set or sequence whose terms are submodules of M.

The generators stored for N consist of the elements specified by terms L_i together with the stored generators for submodules specified by terms of L_i. Repetitions of an element and occurrences of the zero element are removed (unless N is trivial).

The constructor returns the submodule N and the inclusion homomorphism $f : N \to M$.

Example H54E4

We construct a submodule of the 4-dimensional vector space over the field of rational function $F[x]$, where F is $\mathbf{F}_5$.

```
> P := PolynomialRing(GF(5));
> R<x> := FieldOfFractions(P);
> M := RSpace(R, 4);
> N := sub< M | [1, x, 1-x, 0], [1+2*x-x^2, 2*x, 0, 1-x^4 ] >;
> N;
Vector space of degree 4, dimension 2 over Field of Fractions in x over
Univariate Polynomial Algebra over GF(5)
Generators:
(1    x    4*x + 1    0)
(4*x^2 + 2*x + 1    2*x    0    4*x^4 + 1)
Echelonized basis:
(1  0  3/(x + 4)  (x^3 + x^2 + x + 1) / (x + 4))
(0  1  (4*x^2 + 2*x + 1) / (x^2 + 4*x)  (4*x^3 + 4*x^2 + 4*x + 4) / (x^2 + 4*x))
```

54.8.2 Operations on Submodules

54.8.3 Membership and Equality

The following operations are only available for submodules of $R^{(n)}$, $\mathrm{Hom}_R(M, N)$ and $R[G]$, where R is a Euclidean Domain. If the modules involved are $R[G]$-modules, the operators refer to the underlying R-module.

`u in M`

Returns `true` if the element u lies in the R-module M, where u and M belong to the same R-module.

`u notin M`

Returns `true` if the element u does not lie in the R-module M, where u and M belong to the same R-module.

`N subset M`

Returns `true` if the R-module N is contained in the R-module M, where M and N belong to a common R-module.

`N notsubset M`

Returns `true` if the R-module N is not contained in the R-module M, where M and N belong to a common R-module.

`M eq N`

Returns `true` if the R-modules N and M are equal, where N and M belong to a common R-module.

`M ne N`

Returns `true` if the R-modules N and M are not equal, where N and M belong to a common R-module.

54.8.4 Operations on Submodules

The following operations are only available for submodules of $R^{(n)}$, $\mathrm{Hom}_R(M, N)$ and $R[G]$, where R is a Euclidean Domain. If the modules involved are $R[G]$-modules, the operators refer to the underlying R-module.

`M + N`

Sum of the submodules M and N, where M and N belong to a a common R-module.

`M meet N`

Intersection of the submodules M and N, where M and N belong to a common R-module.

54.9 Quotient Modules

54.9.1 Construction of Quotient Modules

`quo< M | L >`

Given an R-module M, construct the quotient module $P = M/N$, where N is the submodule generated by the elements of M specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of R defining an element of M;

(b) A set or sequence whose terms are elements of M;

(c) A submodule of M;

(d) A set or sequence whose terms are submodules of M.

The generators constructed for N consist of the elements specified by terms L_i together with the stored generators for submodules specified by terms of L_i.

The constructor returns the quotient module P and the natural homomorphism $f : M \to P$.

54.10 Homomorphisms

Throughout this part of the chapter, when discussing the set of all R-homomorphisms from the R-module M into the R-module N, it will be assumed that R is a **commutative** ring. We further assume that M and N are free R-modules and that bases for these modules are present. The module $\mathrm{Hom}_R(M, N)$ will be identified with the module of $m \times n$ matrices over R. Thus, an element of $\mathrm{Hom}_R(M, N)$ is represented as a matrix relative to the bases of the generic modules corresponding to M and N. For this reason, we will refer to these modules as *matrix modules*.

We remind the reader that submodules of $\mathrm{Hom}_R(M, N)$ are always presented in embedded form. If the user wishes to have submodules presented in reduced form then he/she should use the natural isomorphism between $R^{(m\times n)}$ and $R^{(mn)}$.

It should be noted that essentially operation defined for tuple modules, their elements and submodules applies to matrix modules. Thus, all of the operations discussed earlier in this chapter apply to matrix modules.

The modules M and N may themselves be matrix modules. In this case, the resulting matrix module has either a right or left action and an element belonging to it transforms a (homomorphism) element of M into a (homomorphism) element of N.

The function `Reduce` may be used to construct for a matrix module H the matrix module H' equivalent to H whose elements are with respect to the actual bases of the domain and codomain of elements of H (not the generic bases of the domain and codomain).

54.10.1 $\mathrm{Hom}_R(M, N)$ for R-modules

`Hom(M, N)`

If M is the tuple module $R^{(m)}$ and N is the tuple module $R^{(n)}$, create the module $\mathrm{Hom}_R(M, N)$ as the (R, R)-bimodule $R^{(m\times n)}$, represented as the set of all $m \times n$ matrices over R. The module is created with the standard basis, $\{E_{ij} \mid i = 1\ldots, m, j = 1\ldots, n\}$, where E_{ij} is the matrix having a 1 in the (i, j)-th position and zeros elsewhere.

`RMatrixSpace(R, m, n)`

Given a ring R and positive integers m and n, construct $H = \mathrm{Hom}(M, N)$, where $M = R^{(m)}$ and $N = R^{(n)}$, as the free (R, R)-bimodule $R^{(m\times n)}$, consisting of all $m \times n$ matrices over R. The module is created with the standard basis, $\{E_{ij} \mid i = 1\ldots, m, j = 1\ldots, m\}$. Note that the modules M and N are created by this function and may be accessed as `Domain(H)` and `Codomain(H)`, respectively.

Example H54E5

We construct the vector spaces V and W of dimensions 3 and 4, respectively, over the field of two elements and then define M to be the module of homomorphisms from V into W.

```
> F2 := GaloisField(2);
> V  := VectorSpace(F2, 3);
> W  := VectorSpace(F2, 4);
```

```
> M  := Hom(V, W);
> M;
Full KMatrixSpace of 3 by 4 matrices over GF(2)
```

54.10.2 $\mathrm{Hom}_R(M, N)$ for Matrix Modules

`Hom(M, N, "right")`

Suppose M is a matrix module over the coefficient ring R whose elements are a by b matrices and have domain D and codomain C. Suppose also that N is a matrix module over the coefficient ring R whose elements are a by c matrices and have domain D and codomain C'. Then the homomorphism module $H = \mathrm{Hom}(M, N)$ with right multiplication action exists and consists of all b by c matrices over R which multiply an element of M on the right to yield an element of N. This function constructs H explicitly. The domain of elements of H is then M and the codomain of elements of H is N and the elements are b by c matrices over R which multiply an element of M on the right to yield an element of N. Note that if M and N are proper submodules of their respective generic modules, then H may be a proper submodule of its generic module, and the correct basis of H will be explicitly constructed.

`Hom(M, N, "left")`

Suppose M is a matrix module over the coefficient ring R whose elements are a by c matrices and have domain D and codomain C. Suppose also that N is a matrix module over the coefficient ring R whose elements are b by c and have domain D' and codomain C. Then the homomorphism module $H = \mathrm{Hom}(M, N)$ with left multiplication action exists and consists of all b by a matrices over R which multiply an element of M on the left to yield an element of N. This function constructs H explicitly. The domain of elements of H is then M and the codomain of elements of H is N and the elements are b by a matrices over R which multiply an element of M on the right to yield an element of N. Note that if M and N are proper submodules of their respective generic modules, then H may be a proper submodule of its generic module, and the correct basis of H will be explicitly constructed.

Example H54E6

We construct two homomorphism modules H_1 and H_2 over $\mathbf{Q}$ and then the homomorphism module $H = \mathrm{Hom}(H_1, H_2)$ with right matrix action.

```
> Q := RationalField();
> H1 := sub<RMatrixSpace(Q, 2, 3) | [1,2,3, 4,5,6], [0,0,1, 1,3,3]>;
> H2 := sub<RMatrixSpace(Q, 2, 4) | [6,5,7,1, 15,14,16,4], [0,0,0,0, 1,2,3,4]>;
> H := Hom(H1, H2, "right");
> H: Maximal;
KMatrixSpace of 3 by 4 matrices and dimension 1 over Rational Field
```

```
Echelonized basis:

[   1    2    3    4]
[-1/2   -1 -3/2   -2]
[   0    0    0    0]

> H1.1 * H.1;
[  0   0   0   0]
[3/2   3 9/2   6]
> H1.1 * H.1 in H2;
true
> Image(H.1): Maximal;
KMatrixSpace of 2 by 4 matrices and dimension 1 over Rational Field
Echelonized basis:

[0 0 0 0]
[1 2 3 4]

> Kernel(H.1): Maximal;
KMatrixSpace of 2 by 3 matrices and dimension 1 over Rational Field
Echelonized basis:

[ 1  2  6]
[ 7 14 15]

> H1 := sub<RMatrixSpace(Q,2,3) | [1,2,3, 4,5,6]>;
> H2 := sub<RMatrixSpace(Q,3,3) | [1,2,3, 5,7,9, 4,5,6]>;
> H := Hom(H1, H2, "left");
> H: Maximal;
KMatrixSpace of 3 by 2 matrices and dimension 1 over Rational Field
Echelonized basis:

[1 0]
[1 1]
[0 1]
> Image(H.1);
KMatrixSpace of 3 by 3 matrices and dimension 1 over Rational Field
> Kernel(H.1);
KMatrixSpace of 2 by 3 matrices and dimension 0 over Rational Field
```

54.10.3 Modules $\mathrm{Hom}_R(M, N)$ with Given Basis

`RMatrixSpaceWithBasis(Q)`

Given a sequence Q of k independent matrices each lying in a matrix space $H = \mathrm{Hom}(M, N)$, where $M = R^{(m)}$ and $N = R^{(n)}$, construct the subspace of H of dimension k whose basis is Q. The basis is echelonized internally but all functions which depend on the basis of the matrix space (e.g. `Coordinates`) will use the given basis Q.

`KMatrixSpaceWithBasis(Q)`

Given a sequence Q of k independent matrices each lying in a matrix space $H = \mathrm{Hom}(M, N)$, where $M = K^{(m)}$ and $N = K^{(n)}$, with K a field, construct the subspace of H of dimension k whose basis is Q. The basis is echelonized internally but all functions which depend on the basis of the matrix space (e.g. `Coordinates`) will use the given basis Q.

54.10.4 The Endomorphsim Ring

`EndomorphismAlgebra(M)`

If M is the free R-module $R^{(m)}$, create the matrix algebra $\mathrm{Mat}_m(R)$. The algebra is created with the standard basis, $\{E_{ij} \mid i = 1 \ldots, m, j = 1 \ldots, m\}$, where E_{ij} is the matrix having a 1 in the (i, j)-th position and zeros elsewhere.

Example H54E7

We construct the endomorphism ring of the 4-dimensional vector space over the rational field.

```
> Q  := RationalField();
> R4 := RModule(Q, 4);
> M  := EndomorphismAlgebra(R4);
> M;
Full Matrix Algebra of degree 4 over Rational Field
```

54.10.5 The Reduced Form of a Matrix Module

`Reduce(H)`

Suppose H is a matrix module whose elements have domain A and codomain B. Suppose first that A and B are tuple modules (R-spaces, R-modules, or RG-modules) and that A has degree a and dimension d while B has degree b and dimension e. (For the reduced cases (R-modules or RG-modules), a equals d and b equals e). The elements of H have the natural representation with respect to the standard embedded basis of the generic modules of A and B. Thus H has degree a by b. So one can multiply a 1 by a vector of A directly by an element h of H (in the natural matrix way) to get a 1 by b vector of B. Now suppose A and B are in the embedded form (R-space) and h is in H. Then h is an a by b matrix but there is a corresponding d by e matrix h' which gives the same transformation of h from A to B but is *with respect to the bases of A and B*. We call h' the *reduced* form of h. Also, there is the *reduced module* H' corresponding to H. This function constructs the reduced module H' corresponding to H, together with the epimorphism f from H onto H'. Note that if A and B are in reduced form, then H' is the same as H.

Suppose secondly that A and B are matrix modules themselves. Suppose $A = \mathrm{Hom}(D_1, C_1)$, $B = \mathrm{Hom}(D_1, C_2)$, and $H = \mathrm{Hom}(A, B)$ with the right multiplication action. Suppose also that A has degree r by s and dimension d while B has degree r by t, and dimension e. Then H would have degree s by t so an element h of H would be s by t and would multiply a r by s element of A on the right to yield an r by t element of B. Then the reduced matrix h' corresponding to a matrix h of H would be a d by e matrix corresponding to the bases of A and B. This function similarly constructs the reduced module H' corresponding to H, together with the epimorphism f from H onto H'. Note also that in this case the domain and codomains of H' are the generic R-spaces (tuple modules) corresponding to A (of dimension d) and B (of dimension e). Similarly, for the left multiplication action there is the corresponding reduced module constructed in the obvious way.

Note also that the kernel of the epimorphism f is the submodule of H which consists of all matrices which transform all elements of A to the zero element of B.

Example H54E8

We demonstrate the function `Reduce` for a homomorphism module from one vector space to another.

```
> V1 := sub<VectorSpace(GF(3), 3) | [1,0,1], [0,1,2]>;
> V2 := sub<VectorSpace(GF(3), 4) | [1,1,0,2], [0,0,1,2]>;
> H := Hom(V1, V2);
> H;
KMatrixSpace of 3 by 4 matrices and dimension 8 over GF(3)
> R, f := Reduce(H);
> R;
Full KMatrixSpace of 2 by 2 matrices over GF(3)
> H.1;
```

```
[1 0 0 0]
[0 1 0 2]
[0 1 0 2]
> f(H.1);
[1 0]
[0 0]
> V1.1;
(1 0 1)
> V1.1 * H.1;
(1 1 0 2)
> Coordinates(V2, V1.1 * H.1);
[ 1, 0 ]
> Coordinates(V2, V1.2 * H.1);
[ 0, 0 ]
> Kernel(f): Maximal;
KMatrixSpace of 3 by 4 matrices and dimension 4 over GF(3)
Echelonized basis:

[1 0 0 0]
[2 0 0 0]
[2 0 0 0]

[0 1 0 0]
[0 2 0 0]
[0 2 0 0]

[0 0 1 0]
[0 0 2 0]
[0 0 2 0]

[0 0 0 1]
[0 0 0 2]
[0 0 0 2]
> R.1@@f;
[1 0 0 0]
[0 1 0 2]
[0 1 0 2]
```

Example H54E9

We demonstrate the function `Reduce` for a homomorphism module from one homomorphism module to another. Note that the reduced module has the same dimension as the original module but larger degrees!

```
> V1 := VectorSpace(GF(3), 2);
> V2 := VectorSpace(GF(3), 3);
> V3 := VectorSpace(GF(3), 4);
> H1 := Hom(V1, V2);
```

```
> H2 := Hom(V1, V3);
> H := Hom(H1, H2, "right");
> H1;
Full KMatrixSpace of 2 by 3 matrices over GF(3)
> H2;
Full KMatrixSpace of 2 by 4 matrices over GF(3)
> H;
Full KMatrixSpace of 3 by 4 matrices over GF(3)
> R,f := Reduce(H);
> R;
KMatrixSpace of 6 by 8 matrices and dimension 12 over GF(3)
> X := H.1;
> X;
[1 0 0 0]
[0 0 0 0]
[0 0 0 0]
> f(X);
[1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
> Domain(X);
Full KMatrixSpace of 2 by 3 matrices over GF(3)
> Domain(f(X));
Full Vector space of degree 6 over GF(3)
> Image(X): Maximal;
KMatrixSpace of 2 by 4 matrices and dimension 2 over GF(3)
Echelonized basis:

[1 0 0 0]
[0 0 0 0]

[0 0 0 0]
[1 0 0 0]

> Image(f(X));
Vector space of degree 8, dimension 2 over GF(3)
Echelonized basis:
(1 0 0 0 0 0 0 0)
(0 0 0 0 1 0 0 0)
> Kernel(X): Maximal;
KMatrixSpace of 2 by 3 matrices and dimension 4 over GF(3)
Echelonized basis:

[0 1 0]
[0 0 0]
```

```
[0 0 1]
[0 0 0]

[0 0 0]
[0 1 0]

[0 0 0]
[0 0 1]

> Kernel(f(X)): Maximal;
Vector space of degree 6, dimension 4 over GF(3)
Echelonized basis:
(0 1 0 0 0 0)
(0 0 1 0 0 0)
(0 0 0 0 1 0)
(0 0 0 0 0 1)
```

54.10.6 Construction of a Matrix

`M ! Q`

Given the matrix bimodule M over the ring R, and the sequence $Q = [a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{m1}, \ldots, a_{mn}]$ whose terms are elements of the ring R, construct the $m \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}$$

as an element of M. In the context of the `sub` or `quo` constructors the coercion clause `M !` may be omitted.

Example H54E10

We create the 4×4 Hilbert matrix $h4$ as an element of the endomorphism ring of the 4-dimensional vector space over the rational field.

```
> Q  := RationalField();
> R4 := RModule(Q, 4);
> M  := EndomorphismAlgebra(R4);
> h4 := M ! [ 1/i :  i in [1 .. 16 ] ];
> h4;
[   1  1/2  1/3  1/4]
[ 1/5  1/6  1/7  1/8]
[ 1/9 1/10 1/11 1/12]
[1/13 1/14 1/15 1/16]
```

54.10.7 Element Operations

All operations that apply to elements of tuple modules also apply to elements of matrix modules. Here, we confine our discussion to those operations which are special to matrix modules.

Throughout this section, M is a submodule of $R^{(m)}$, N is a submodule of $R^{(n)}$ and a is a homomorphism belonging to $\mathrm{Hom}_R(M, N)$, where R is a Euclidean Domain.

See also the chapter on general matrices for many other functions applicable to matrices (e.g., `EchelonForm`).

`u * a`
`a(u)`

Given an element u belonging to the module M, return the image of u under the homomorphism a as an element of the module N.

`a * b`

Given a homomorphism a belonging to a submodule of $\mathrm{Hom}(M, N)$, and a homomorphism b belonging to a submodule of $\mathrm{Hom}(N, P)$, return the composition of the homomorphisms a and b as an element of $\mathrm{Hom}(M, P)$. Note that if $\mathrm{Hom}(M, P)$ does not already exist, it will be created.

`a ^ -1`

Given a homomorphism a belonging to a submodule of $\mathrm{Hom}(M, N)$ with M and N having the same dimension, return the inverse of a as an element of $\mathrm{Hom}(N, M)$.

`Codomain(S)`

Given a submodule S of the module $\mathrm{Hom}(M, N)$, return the module N.

`Codomain(a)`

The codomain N of the homomorphism a belonging to $\mathrm{Hom}(M, N)$.

`Cokernel(a)`

The cokernel for the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$.

`Domain(S)`

The domain M of the submodule S belonging to the module $\mathrm{Hom}(M, N)$.

`Domain(a)`

The domain M of the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$.

`Image(a)`

The image of the homomorphism a belonging to the module $H = \mathrm{Hom}(M, N)$, returned as a submodule of N. Note that if the domain and codomain of a are matrix modules themselves, the image will be with respect to the appropriate action (right or left).

`Kernel(a)`

`NullSpace(a)`

The kernel of the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$, returned as a submodule of M. Note that if the domain and codomain of a are matrix modules themselves, the kernel will be with respect to the appropriate action (right or left).

`Morphism(M, N)`

Assuming the R-module M was created as a submodule of the module N, return the matrix defining the inclusion homomorphism $\phi : M \to N$ as an element of $\mathrm{Hom}_R(M, N)$. Thus ϕ gives the correspondence between elements of M (represented with respect to the standard basis of M) and elements for N.

`Rank(a)`

The dimension of the image of the homomorphism a, i.e. the rank of a.

`IsBijective(a)`

Returns `true` if the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$ is a bijective mapping.

`IsInjective(a)`

Returns `true` if the homomorphism a belonging to the module $\mathrm{Hom}(M, N)$ is an injective mapping.

`IsSurjective(a)`

Returns `true` if the homomorphism a belonging to the module $\mathrm{Hom}_R(M, N)$ is a surjective mapping.

Example H54E11

We illustrate some of these operations in the context of the module $\mathrm{Hom}_R(M, N)$, where M and N are, respectively, the 4-dimensional and 3-dimensional vector spaces over $GF(8)$.

```
> K<w> := GaloisField(8);
> V3 := VectorSpace(K, 3);
> V4 := VectorSpace(K, 4);
> M := Hom(V4, V3);
> A := M ! [1, w, w^5, 0,  w^3, w^4, w, 1,  w^6, w^3, 1, w^4 ];
> A;
[  1   w w^5]
[  0 w^3 w^4]
[  w   1 w^6]
[w^3   1 w^4]
> Rank(A);
3
> Image(A);
```

```
Full Vector space of degree 3 over GF(2^3)
> Kernel(A);
Vector space of degree 4, dimension 1 over GF(2^3)
Echelonized basis:
(  1 w^5   1   1)
> Cokernel(A);
Vector space of degree 3, dimension 0 over GF(2^3)
```

55 MODULES OVER DEDEKIND DOMAINS

Chapter 55
MODULES OVER DEDEKIND DOMAINS

55.1 Introduction

Since the structure theory for modules over arbitrary orders (which are in general not Dedekind domains) is very unsatisfactory, modules over orders in MAGMA are always modules over some maximal order of a number field or function field, they form a magma of type `ModDed`.

Let k be a number field or function field and $\mathcal{O}_k$ its ring of integers. Since $\mathcal{O}_k$ is a Dedekind domain, every finitely generated torsion free module M over $\mathcal{O}_k$ has a representation as a direct sum

$$M = \sum_{i=1}^{m} \mathcal{A}_i \alpha_i = \{ \sum_{i=1}^{m} a_i \alpha_i \mid a_i \in \mathcal{A}_i \}$$

with (fractional) ideals $\mathcal{A}_i$ and elements $\alpha_i \in kM \cong k^r$.

A (not necessarily direct) sum $\sum_{i=1}^{m} \mathcal{A}_i \alpha_i$ will be represented as a pseudo–matrix $(\mathcal{A}|A)$ where $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_m)^t$ is a column vector of ideals and $A = (\alpha_1, \ldots, \alpha_m)^t \in k^{m \times r}$ is a matrix. The ideals $\mathcal{A}_i$ are called coefficient ideals.

This pseudo–matrix is called a pseudo–basis iff the sum is direct. A pseudo–matrix $(\mathcal{A}|A)$ is in Hermite normal form iff there are $s \leq m$, $1 \leq i_1 < i_2 < \cdots < i_s$ such that $A_{j,l} = 0$ $(1 \leq l < i_j)$, $A_{j,i_j} = 1$ and $A_{j,l}$ is reduced modulo $\mathcal{A}_j {\mathcal{A}_l}^{-1}$. For $j > s$ we have $A_{j,l} = 0$.

This normal form is unique if a suitable reduction is used.

As a consequence of this normalisation, usually $\alpha_i \notin M$. To be precise: $\alpha_i \in M$ iff $1 \in \mathcal{A}_i$.

All modules are in Hermite normal form, i.e. every module is represented by a pseudo–basis in Hermite normal form.

General (non torsion free) modules are represented as quotients of a torsion free module M and a submodule S. Elements of $Q := M/S$ are represented as elements of M, arithmetic in Q is reduced to arithmetic in M followed by a reduction modulo the pseudo–basis of S.

55.2 Creation of Modules

Modules over dedekind domains can be created from orders of number fields and function fields and combinations of ideals and vector space elements. Submodules and quotient modules by submodules can also be created.

`Module(O, n)`

Create the free module O^n where O is a dedekind domain.

`Module(O)`

Create the relative order O as a module over its coefficient ring. Also returns the map from the resulting module into O.

`Module(I)`

Create the ideal I of a relative order O as a module over the coefficient ring of the order. Also returns the map from the module into O.

`Module(S)`

Create a module from the sequence of tuples of ideals of a dedekind domain and `ModElt`s with entries in the dedekind domain or its field of fractions. The elements of the resulting module will be the sum of products of an element of an ideal and the corresponding `ModElt`. Also returns the map from the vector space into the module.

`Module(S)`

Create the module which is equal to the direct sum of the ideals in the sequence.

`Module(S)`

Create the module which is freely generated by the elements of the sequence S. The elements of the sequence must be `ModElt`s with entries in a dedekind domain or field of fractions of a dedekind domain. Also returns the map from the vector space into the module.

Example H55E1

The creation of some simple modules is shown.

```
> x := ext<Integers()|>.1;
> M := MaximalOrder(x^2 + 5);
> Module(M, 5);
Module over Maximal Equation Order with defining polynomial x^2 + 5 over Z
 generated by: (in echelon form)
Principal Ideal of M
Generator:
    M.1 * ( M.1 0 0 0 0 )
Principal Ideal of M
Generator:
    M.1 * ( 0 M.1 0 0 0 )
Principal Ideal of M
```

```
Generator:
    M.1 * ( 0 0 M.1 0 0 )
Principal Ideal of M
Generator:
    M.1 * ( 0 0 0 M.1 0 )
Principal Ideal of M
Generator:
    M.1 * ( 0 0 0 0 M.1 )
> I := 1/5*M;
> Module([I, I^3, I^8]);
Module over Maximal Equation Order with defining polynomial x^2 + 5 over Z
 generated by: (in echelon form)
Fractional Principal Ideal of M
Generator:
    1/5*M.1 * ( M.1 0 0 )
Fractional Principal Ideal of M
Generator:
    1/125*M.1 * ( 0 M.1 0 )
Fractional Principal Ideal of M
Generator:
    1/390625*M.1 * ( 0 0 M.1 )
> V := RModule(M, 3);
> Module([<I, V![0, 1, 0]>, <I^4, V![2, 3, 5]>]);
Module over Maximal Equation Order with defining polynomial x^2 + 5 over Z
Fractional Principal Ideal of M
Generator:
    1/5*M.1 car Fractional Principal Ideal of M
Generator:
    1/125*M.1
```

The same can be done using orders of function fields.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(x^2 - y);
> M := MaximalOrderFinite(F);
> Module(M, 5);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x over
Rational Field
 generated by: (in echelon form)
Ideal of M
Generator:
1 * ( 1 0 0 0 0 )
Ideal of M
Generator:
1 * ( 0 1 0 0 0 )
Ideal of M
Generator:
1 * ( 0 0 1 0 0 )
```

```
Ideal of M
Generator:
1 * ( 0 0 0 1 0 )
Ideal of M
Generator:
1 * ( 0 0 0 0 1 )
> I := 1/5*M;
> Module([I, I^3, I^8]);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x over
Rational Field
 generated by: (in echelon form)
Ideal of M
Generator:
1/5 * ( 1 0 0 )
Ideal of M
Generator:
1/125 * ( 0 1 0 )
Ideal of M
Generator:
1/390625 * ( 0 0 1 )
> V := RModule(M, 3);
> Module([<I, V![0, 1, 0]>, <I^4, V![2, 3, 5]>]);
Integral Module over Maximal Equation Order of F over Univariate Polynomial Ring
in x over Rational
Field
Ideal of M
Generator:
1/5 car Ideal of M
Generator:
1/125
```

`sub< M | m >`

`sub< M | m1, .., mn >`

Construct the submodule of the module M generated by the elements in the sequence or list of elements m. Also returns the inclusion map of the submodule into M.

`quo< M | S >`

`quo< M | m >`

`quo< M | m1, .., mn >`

Construct the quotient of the module M by the submodule S or the submodule generated by the elements of the sequence or list of elements m. Also returns the inclusion map of the quotient module into M.

Example H55E2

Use of the sub and quo constructors is illustrated below. Let M and V be as above when they were referring to number fields.

```
> Mod := Module([V|[0,1,0], [4,4,0]]);
> S1 := sub<Mod | >;
> S1;
Integral Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
(0)
> Q1 := quo<Mod | Mod>;
> Q1;
Quotient of Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
by Integral Module over Maximal Equation Order with defining polynomial
x^2 + 5 over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
> S2 := sub<Mod | Mod.2>;
> S2;
Integral Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    M.1
> Q2 := quo<Mod | Mod.2>;
> Q2;
Quotient of Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
by Integral Module over Maximal Equation Order with defining polynomial
x^2 + 5 over Z
Principal Ideal of M
Generator:
    M.1
> S3 := sub<Mod | 4*Mod.1, Mod.2>;
```

```
> S3;
Integral Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
> Q3 := quo<Mod | >;
> Q3;
Integral Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
> Q4 := quo<Mod | S1>;
> Q4;
Quotient of Module over Maximal Equation Order with defining polynomial x^2 + 5
over Z
Principal Ideal of M
Generator:
    4/1*M.1 car Principal Ideal of M
Generator:
    M.1
by Integral Module over Maximal Equation Order with defining polynomial
x^2 + 5 over Z
(0)
```

55.3 Elementary Functions

Various simple properties of a module can be retrieved using the following functions.

`BaseRing(M)`

`CoefficientRing(M)`

The dedekind domain which M is a module over.

`Degree(M)`

The dimension of the vector space the module M embeds into.

`Ngens(M)`

`NumberOfGenerators(M)`

The minimum number of vectors and ideals which generate the module M.

`M . i`

The vector of the *ith* vector and ideal pair generating the module M.

`Determinant(M)`

The determinant of the module M.

`Dimension(M)`

The dimension of the vector space spanned by the module M over its coefficient ring. This is the same as the number of generators of a pseudo basis of M.

`Contents(M)`

`UseBasis` BoolElt *Default :* `false`

The contents of the module M, ie. the gcd of the ideals obtained by multiplying the coefficient ideals by the ideal generated by the coefficients in the corresponding generators. The parameter `UseBasis` decides whether a pseudo basis or pseudo generators are used.

`Simplify(M)`

`UseBasis` BoolElt *Default :* `false`

Computes a module of contents 1 by scaling each coefficient ideal by the inverse of the contents of the module M. The parameter `UseBasis` determines if the operations are performed on the pseudo generators or the pseudo basis of M.

`EmbeddingSpace(M)`

The canonical vector space containing the module M, ie.. M tensored with the field of fractions of the coefficient ring.

Example H55E3

The use of some elementary functions on a module is shown below.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(x^2 - y);
> M := MaximalOrderFinite(F);
> Vs := RModule(M, 2);
> s := [Vs | [1, 3], [2, 3]];
> Mods := Module(s);
> CoefficientRing(Mods);
Maximal Equation Order of F over Univariate Polynomial Ring in x over Rational
Field
> Mods.1;
(1 0)
> Determinant(Mods);
Ideal of M
Generator:
```

```
-3
> Vs := RSpace(M, 2);
> s := [Vs | [1, 3], [2, 3]];
> Mods := Module(s);
> sMods := sub<Mods | Mods!Vs![1, 3]>;
> qMods := quo<Mods | sMods>;
> Degree(Mods);
2
> Ngens(Mods);
2
> Ngens(sMods);
1
> Degree(sMods);
2
> Degree(qMods);
2
> Ngens(qMods);
2
> Determinant(Mods);
Ideal of M
Basis:
[1]
>  Determinant(sMods);
>>   Determinant(sMods);
               ^
Runtime error in 'Determinant': Module must be square
> Determinant(qMods);
Ideal of M
Basis:
[1]
```

55.4 Predicates on Modules

Modules and potential elements can be tested against each other for a few properties.

`M eq N`

Return `true` if M and N are equal as modules.

`x in M`

Return `true` if x can be coerced into the module M.

`M subset N`

Return `true` if M is a submodule of N. An embedding map of M in N can be returned by `IsSubmodule(M, N)`.

55.5 Arithmetic with Modules

Some arithmetic operations can be carried out involving modules and their elements and compatible ideals to gain more modules.

```
I * M
M * I
```

The module generated by the products of the ideals of the module M with I.

```
M1 + M2
```

The union of the modules $M1$ and $M2$.

```
u * I
I * u
```

The module containing elements which are products of the dedekind module element u and an element lying in the ideal I.

Example H55E4

Some module predicates and arithmetic are shown below.

```
> P<x> := PolynomialRing(Integers());
> K :=  NumberField([x^5 + 3, x^2 + 2]);
> M := MaximalOrder(K);
> Vs := RModule(M, 2);
> s := [Vs | [1, 3], [2, 3]];
> Mods := Module(s);
> sMods := sub<Mods | Mods!Vs![1, 3]>;
> Mods eq sMods;
false
> [1, 0] in Mods;
true
> [1, 0] in sMods;
false
> sMods subset Mods;
true
> Vs := RSpace(M, 2);
> s := [Vs | [Random(M, 3), 3], [2, Random(M, 2)]];
> Mods := Module(s);
> sMods := sub<Mods | Mods!s[1]>;
> (7*M + 11*K.1*M)*sMods;
Module over Maximal Equation Order with defining polynomial x^5 + [3, 0] over
its ground order
 generated by:
Ideal of M
Two element generators:
    7/1*$.1*M.1
    11/1*$.1*M.2 * ( M.1 + (-$.1 - 2/1*$.2)*M.2 + $.2*M.3 + (2/1*$.1 +
2/1*$.2)*M.5 3/1*$.1*M.1 )
```

```
 in echelon form:
Ideal of M
Two element generators:
    21/1*$.1*M.1
    33/1*$.1*M.2 * ( 1/3*$.1*M.1 + (-1/3*$.1 - 2/3*$.2)*M.2 + 1/3*$.2*M.3 +
(2/3*$.1 + 2/3*$.2)*M.5 M.1 )
> Mods + sMods;
Module over Maximal Equation Order with defining polynomial x^5 + [3, 0] over
its ground order
 generated by: (in echelon form)
Ideal of M
Two element generators:
    148919257164/1*$.1*M.1
    (148919257048/1*$.1 + 26/1*$.2)*M.1 + (32/1*$.1 + 148919257015/1*$.2)*M.2 +
    (196/1*$.1 + 148919257117/1*$.2)*M.3 + (148919257163/1*$.1 + 76/1*$.2)*M.4 +
    (148919257077/1*$.1 + 148919257116/1*$.2)*M.5 * ( M.1 0 )
Ideal of M
Two element generators:
    3/1*$.1*M.1
    ($.1 - $.2)*M.1 + -M.3 + M.4 + ($.1 + $.2)*M.5 * ( (-19866460521/1*$.1 -
33/1*$.2)*M.1 + (1/3*$.1 + 1/3*$.2)*M.4 + (1/3*$.1 + 1/3*$.2)*M.5 M.1 )
> 4*sMods;
Module over Maximal Equation Order with defining polynomial x^5 + [3, 0] over
its ground order
 generated by:
Principal Ideal of M
Generator:
    4/1*$.1*M.1 * ( (2/1*$.1 - 2/1*$.2)*M.1 + ($.1 + 2/1*$.2)*M.2 + (2/1*$.1 +
2/1*$.2)*M.3 + (2/1*$.1 - $.2)*M.5 3/1*$.1*M.1 )
 in echelon form:
Principal Ideal of M
Generator:
    12/1*$.1*M.1 * ( (2/3*$.1 - 2/3*$.2)*M.1 + (1/3*$.1 + 2/3*$.2)*M.2 +
(2/3*$.1 + 2/3*$.2)*M.3 + (2/3*$.1 - 1/3*$.2)*M.5 M.1 )
```

55.6 Basis of a Module

The basis of a module is given by vectors. However, more complete information can be supplied which includes the ideals.

`Basis(M)`

A sequence of vectors which correspond to a pseudo basis of the module M.

`PseudoBasis(M)`

A sequence of tuples containing ideals and vectors which generate the module M. The vectors are guaranteed to be linearly independent.

`PSeudoGenerators(M)`

A sequence of tuples containing ideals and vectors which generate the module M. This will return the data used to define the module, so that in contrast to `PseudoBasis` the vectors will in general not be independent.

55.7 Other Functions on Modules

Intersections of modules can taken. Several other functions are also available.

`M1 meet M2`

Return the intersection of the modules $M1$ and $M2$.

`Dual(M)`

The module dual to M.

`ElementaryDivisors(M, N)`

The elementary divisors (ideals) of the torsion part of the quotient R-module M/N: For $N \subseteq M$ we get

$$T(M/N) \cong \oplus_{i=1}^{n} R/\mathcal{A}_i$$

The $\mathcal{A}_i$ are unique if we require $R \subseteq \mathcal{A}_1 \subseteq \cdots \subseteq \mathcal{A}_n$. The $\mathcal{A}_i$ are called the elementary divisors (or elementary ideals) of M/N. This corresponds to the Smith normal form for integral matrices.

`SteinitzClass(M)`

The Steinitz class of the module M.

`SteinitzForm(M)`

The Steinitz (almost–free) form of the module M.

Example H55E5

Some bases and other functions are demonstrated below.

```
> P<x> := PolynomialRing(Rationals());
> P<y> := PolynomialRing(P);
> F<c> := FunctionField(y^3 - x^3*y^2 + y - x^7);
> M := MaximalOrderFinite(F);
> Vs := RSpace(M, 2);
> s := [Vs | [1, Random(M, 3)], [Random(M, 3), 3]];
> Mods := Module(s);
> qMods := quo<Mods | Mods!s[2]>;
> Basis(Mods);
[
```

```
    ([ 1, 0, 0 ] [ 2*x^2 - 3*x + 1/2, -2/3*x^2 - x + 1/3, 2/3*x^2 - 2*x + 1/2
        ]),
    ([ -x^2 + x + 2/3, -1/3*x^2 - 1, -3/2*x^2 - 2/3*x + 1 ] [ 3, 0, 0 ])
]
> Basis(qMods);
[
    ([ 1, 0, 0 ] [ 2*x^2 - 3*x + 1/2, -2/3*x^2 - x + 1/3, 2/3*x^2 - 2*x + 1/2
        ]),
    ([ -x^2 + x + 2/3, -1/3*x^2 - 1, -3/2*x^2 - 2/3*x + 1 ] [ 3, 0, 0 ])
]
> PseudoBasis(Mods) eq PseudoBasis(qMods);
> Vs := RModule(M, 2);
> s := [Vs | [Random(M, 3), Random(M, 3)], [2, 3]];
> Mods := Module(s);
> sMods := sub<Mods | Mods!s[1]>;
> Mods meet sMods;
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Ideal of M
Generator:
(x + 1)*c^2 + (-3/2*x^2 - 3/2*x)*c + 1/2*x^2 + 2/3*x + 1/2
> ElementaryDivisors(Mods, sMods);
[ Ideal of M
Basis:
[1 0 0]
[0 1 0]
[0 0 1], Ideal of M
Generator:
0 ]
> Dual(Mods);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Fractional ideal of M
Generator:
(-3*x^4 + 21*x^3 + 20*x^2 + 80*x - 387)/(x^17 - 24*x^16 + 192*x^15 - 510*x^14 -
    94/3*x^13 + 304/3*x^12 + 902/3*x^11 - 3109/3*x^10 - 389/9*x^9 + 664/9*x^8 +
    1094/3*x^7 - 1540/3*x^6 - 101/9*x^5 + 128/3*x^4 + 2000/27*x^3 - 4361/9*x^2 -
    427*x + 2056)*c^2 + (-3*x^9 + 48*x^8 - 189*x^7 - 24*x^6 + 3*x^5 - 48*x^4 +
    195*x^3 - 3*x^2 - x + 24)/(x^17 - 24*x^16 + 192*x^15 - 510*x^14 - 94/3*x^13
    + 304/3*x^12 + 902/3*x^11 - 3109/3*x^10 - 389/9*x^9 + 664/9*x^8 + 1094/3*x^7
    - 1540/3*x^6 - 101/9*x^5 + 128/3*x^4 + 2000/27*x^3 - 4361/9*x^2 - 427*x +
    2056)*c + (3*x^12 - 48*x^11 + 192*x^10 + 3*x^9 - 20*x^8 - 56*x^7 + 192*x^6 +
    3*x^5 - 5*x^4 - 8*x^3 + 272/3*x^2 + 128*x - 771)/(x^17 - 24*x^16 + 192*x^15
    - 510*x^14 - 94/3*x^13 + 304/3*x^12 + 902/3*x^11 - 3109/3*x^10 - 389/9*x^9 +
    664/9*x^8 + 1094/3*x^7 - 1540/3*x^6 - 101/9*x^5 + 128/3*x^4 + 2000/27*x^3 -
    4361/9*x^2 - 427*x + 2056) car Ideal of M
Generator:
1/3
```

```
> Dual(sMods);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Fractional ideal of M
Generator:
(3/2*x^6 + 3*x^5 - 3/4*x^4 - 4*x^3 - 25/12*x^2 - 5/6*x - 1/2)/(x^17 + 3*x^16 +
    21/4*x^15 + 27/4*x^14 + 1/24*x^13 - 247/24*x^12 - 173/24*x^11 + 61/24*x^10 +
    305/72*x^9 + 17/72*x^8 - 4*x^7 - 53/12*x^6 - 1/8*x^5 + 23/6*x^4 +
    377/108*x^3 + 23/18*x^2 + 1/3*x + 1/8)*c^2 + (-5/2*x^9 - 5*x^8 - 1/4*x^7 +
    9/2*x^6 + 13/4*x^5 + 5/4*x^4 - 3/4*x^3 - 7/4*x^2 - 3/4*x)/(x^17 + 3*x^16 +
    21/4*x^15 + 27/4*x^14 + 1/24*x^13 - 247/24*x^12 - 173/24*x^11 + 61/24*x^10 +
    305/72*x^9 + 17/72*x^8 - 4*x^7 - 53/12*x^6 - 1/8*x^5 + 23/6*x^4 +
    377/108*x^3 + 23/18*x^2 + 1/3*x + 1/8)*c + (x^12 + 2*x^11 - 1/2*x^10 -
    7/2*x^9 - 8/3*x^8 - 5/12*x^7 + 11/4*x^6 + 19/4*x^5 - 1/4*x^4 - 25/6*x^3 -
    67/36*x^2 - 1/3*x - 1/4)/(x^17 + 3*x^16 + 21/4*x^15 + 27/4*x^14 + 1/24*x^13
    - 247/24*x^12 - 173/24*x^11 + 61/24*x^10 + 305/72*x^9 + 17/72*x^8 - 4*x^7 -
    53/12*x^6 - 1/8*x^5 + 23/6*x^4 + 377/108*x^3 + 23/18*x^2 + 1/3*x + 1/8)
> SteinitzClass(Mods) eq SteinitzClass(sMods);
false
> SteinitzForm(Mods);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Ideal of M
Generator:
3 car Ideal of M
Generator:
1
> SteinitzForm(sMods);
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Ideal of M
Generator:
1
```

55.8 Homomorphisms between Modules

It is possible to create a homomorphism between two modules, take the image and kernel of such and verify that these are submodules of the codomain and domain respectively. The Hom–module can also be created as a module of a dedekind domain.

`hom< M -> N | T >`

`ModuleBasis` BoolElt *Default :* `true`

Return a homomorphism from the module M into the module N as specified by T from which the images of the generators can be inferred. T may be a map between the vector spaces of same degree as M and N, a matrix over the field of fractions or a sequence of vectors. If `ModuleBasis` is `true` then the matrix will be taken to be a transformation between the modules and as such will be expected to have size `Dimension(M)*Dimension(N)` otherwise it will be interpreted as a transformation between the corresponding vector spaces and will be expected to have size `Degree(M)*Degree(N)`.

`Hom(M, N)`

The module of homomorphisms between the module M and the module N and the map from the hom–module to the collection of maps from M to N, (such that given an element of the hom–module a homomorphism from M to N is returned). The module is over the same dedekind domain as M and N.

`IsSubmodule(M, N)`

Return `true` if M is a submodule of N and the map embedding M into N.

`Morphism(M, N)`

The map giving the morphism from the module M to the module N. Either M is a submodule of N, in which case the embedding of M into N is returned, or N is a quotient module of M, in which case the natural epimorphism from M onto N is returned.

Example H55E6

This example demonstrates the use of homomorphisms between modules over dedekind domains. Let `M` and `V` be as above referring to function fields.

```
> S := [V|[0,1,0], [4,4,0]];
> Mod := Module(S);
> W := KModule(FieldOfFractions(M), 4);
> S := [W|[3, 2, 1, 0]];
> N := Module(S);
>  h := hom<Mod -> N | >;
>> h := hom<Mod -> N | >;
           ^
Runtime error in map< ... >: No images given
>  h := hom<Mod -> N | V.1, V.2, V.3>;
>> h := hom<Mod -> N | V.1, V.2, V.3>;
           ^
Runtime error in map< ... >: An image for each generator is required
> h := hom<Mod -> N | W![3, 2, 1, 0], W![3*(M!F.1 + 1), 2*(M!F.1 + 1),
> M!F.1 + 1, 0] >;
```

```
> h(Mod!(4*V.1));
( 4 )
> h(Mod!V![0, 1, 0]);
( x^2 + 1 )
> I := Image(h);
> I;
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x
over Rational Field
Ideal of M
Generator:
1
> K := Kernel(h);
> K;
Integral Module over Maximal Equation Order of F over Univariate Polynomial Ring
in x over Rational Field
Ideal of M
Generator:
1
> IsSubmodule(K, Mod);
true Mapping from: ModDed: K to ModDed: Mod
> H, m := Hom(Mod, N);
> H; m;
Module over Maximal Equation Order of F over Univariate Polynomial Ring in x over
Rational Field
 generated by: (in echelon form)
Ideal of M
Generator:
1/4 * ( 1 0 )
Ideal of M
Generator:
1 * ( 0 1 )
Mapping from: ModDed: H to Power Structure of Map given by a rule [no inverse]
> m(H![5, 20]);
Mapping from: ModDed: Mod to ModDed: N
using
[5]
[20]
```

55.9 Elements of Modules

55.9.1 Creation of Elements

`M ! v`

Coerce v into an element of M. v can be a sequence of length dimension of M, a module element or vector or an element of another module over a dedekind domain which is compatible with M.

Example H55E7

Let `Mod` and its submodules and quotient modules be as in the sub and quotient module example above.

```
> m := 4*Mod.1;
> m;
(4/1*M.1 0)
> Q1!m;
( 4/1*M.1 0 )
> Q2!m;
( 4/1*M.1 0 )
> m := Mod!m;
> Q3!m;
( 4/1*M.1 0 )
> Q4!m;
( 4/1*M.1 0 )
>  S1!m;
>> S1!m;
     ^
Runtime error in '!': Illegal coercion
LHS: ModDed
RHS: ModDedElt
> S1!Mod!V!0;
(  )
> S2!Mod!Mod.2;
( M.1 )
> S3!Mod!(4*Mod.1);
( 4/1*M.1 0 )
```

55.9.2 Arithmetic with Elements

Basic arithmetic can be performed with elements of a module over a dedekind domain.

```
x + y
```

The sum of the module elements.

```
x - y
```

The difference of the module elements.

```
u * c
c * u
```

The product of the module element u and the ring element c.

```
u / c
```

The product of u and $1/c$ if it lies in the parent module of u.

```
I * u
u * I
```

The module containing elements which are products of u and an element lying in I.

55.9.3 Other Functions on Elements

Elements of modules over a dedekind domain can be tested for equality and represented as a sequence.

```
x eq y
```

Return `true` if x and y are the same element of a module.

```
ElementToSequence(a)
Eltseq(a)
```

The module element a expressed as a sequence.

55.10 Pseudo Matrices

A pseudo matrix, ie. an object of type `PMat`, is a sequence of ideals together with a matrix. Pseudo matrices arise naturally in the (computational) theory of finitely generated torsion free modules over Dedekind domains, they are a natural extension of ordinary matrices which should be thought of as pseudo matrices where all the ideals are generated by 1. Pseudo matrices are generally used to represent the module that is generated by the rows of the matrix scaled by the elements of the corresponding ideal. Thus, if the matrix is regular, a linear combination of the rows lies in the module if and only if the coefficient of the ith row is a member of the ith ideal. The ideals are therefore called coefficient ideals.

55.10.1 Construction of a Pseudo Matrix

`PseudoMatrix(I, m)`

Construct the pseudo matrix with coefficient ideals the elements of the sequence I and matrix m.

`PseudoMatrix(m)`

Construct the pseudo matrix with trivial coefficient ideals and the matrix m.

`PseudoMatrix(M)`

`Generators` BoolElt *Default :* `false`

Construct the pseudo matrix described by the pseudo basis of the module M. If `Generators` is `true` the pseudo matrix described by the pseudo generators of M is returned.

55.10.2 Elementary Functions

`CoefficientIdeals(P)`

Return the coeffficient ideals of the pseudo matrix P.

`Matrix(P)`

Return the matrix of the pseudo matrix P.

`Order(pm)`

Return the order the pseudo matrix pm is over.

`Dimension(pm)`

The dimension of the pseudo matrix pm. This is the numer of columns of the matrix.

`Length(pm)`

The length or dimension of the pseudo matrix pm. This is the number of coefficient ideals of pm.

55.10.3 Basis of a Pseudo Matrix

`Basis(P)`

Return a list of sequences of ring elements corresponding to the entries of each row of the matrix of the pseudo matrix P.

55.10.4 Predicates

`p1 eq p2`

Return whether the pseudo matrices $p1$ and $p2$ are equal, that is, whether they have the same matrix and the same sequence of coefficient ideals.

55.10.5 Operations with Pseudo Matrices

`Transpose(P)`

Return the pseudo matrix whose coefficient ideals are the same as those of P but whose matrix is the transpose of the matrix of P. This function requires the matrix to be square.

`HermiteForm(X)`

Return the Hermite normal form H of the pseudo matrix X together with a regular transformation matrix such that the module generated by X is the same as the one generated by H and such that (for the matrix parts) $H = TX$ holds.

`VerticalJoin(X, Y)`

Return the pseudo matrix whose matrix is the vertical join of the matrices of the pseudo matrices X and Y with coefficient ideals the concatenation of those of X and Y.

`X meet Y`

Return the intersection of the pseudo matrices X and Y.

`Module(X)`

Return the module $\sum C_i * m_i$ where C_i are the coefficient ideals of the pseudo matrix X and m_i are the rows of the matrix of X.

`I * X`
`X * I`

The pseudo matrix whose coefficient ideals are those of the pseudo matrix X multiplied by I and whose matrix is the matrix of X.

56 CHAIN COMPLEXES

Chapter 56
CHAIN COMPLEXES

56.1 Complexes of Modules

Complexes of modules are a fundamental object in homological algebra. MAGMA supports the type `ModCpx` representing a complex of modules. Conceptually, a complex is an infinite sequence of modules, indexed by integers, with maps between successive modules such that the composition of any two maps is zero. Complexes are often written

$$\ldots \xrightarrow{f_{n+1}} M_n \xrightarrow{f_n} M_{n-1} \xrightarrow{f_{n-1}} M_{n-2} \xrightarrow{f_{n-2}} \ldots$$

where the map f_n has domain M_n and codomain M_{n-1}. The indices on the modules and maps decrease to the right. In practice, MAGMA requires all but a finite number of the modules and maps to be zero.

The homomorphism from M_n to M_{n-1} in the complex is the n^{th} boundary map of the complex. The homology of the complex in degree n is the quotient of the kernel of the n^{th} boundary map by the cokernel of the boundary map of degree $n-1$. MAGMA computes the homology only if both boundary maps are defined. A complex is said to be *exact* if the image of each map is equal to the kernel of the next.

Currently, there are two types of modules over which complexes are supported:

(a) Modules over a basic algebra A (see Chapter 85);

(b) Modules over a multivariate polynomial ring over a field (see Chapter 109 and in particular the function `FreeResolution`).

Most of the functions in this chapter work for either type of module but exceptions are noted.

56.1.1 Creation

`Complex(L, d)`

Given a list L of maps between successive A-modules create the corresponding complex. The last term of the complex has degree d. This function returns an error if the maps don't actually form a complex.

`Complex(f, d)`

Given a map f between A-modules M and N, form the complex consisting of the two term complex whose only map is f. The term N is in degree d.

`ZeroComplex(A, m, n)`

Given a basic algebra A and integers m and n such that $m > n$, create a complex of modules over the basic algebra A, starting with a term of degree m and ending with a term of degree n, where all the modules and maps are zero.

```
Dual(C)
```
```
Dual(C, n)
```

The dual of the complex C over a basic algebra A as a complex over the opposite algebra of A. If an integer n is supplied then the last term of the dual complex is in degree n. Otherwise, the last term of the dual complex is in degree 0.

56.1.2 Subcomplexes and Quotient Complexes

Let C be a complex of A-modules, $M_m, \ldots, M_n$. A subcomplex S of C is a complex whose terms are submodules of the terms of C and whose maps are the restrictions of the maps of C to the terms of S.

```
sub< C | Q >
```
```
sub< C | L >
```

Given a complex C and a sequence Q of submodules of the terms of C, returns the smallest subcomplex whose terms contain the modules in Q. The input can also be a list L of sequences of elements of the successive terms of C. In this case the submodules generated by the elements is computed. The function also returns the chain map giving the inclusion of the subcomplex in C.

```
RandomSubcomplex(C, Q)
```

Given a chain complex C over a basic algebra in degrees a to $a-t+1$ and a sequence $Q = [q_1, \ldots, q_t]$ of natural numbers, the function creates the minimal chain complex whose term in degree $a - i + 1$ is a submodule generated by q_i random elements of the term in degree $a - i + 1$ of C. The function also returns the chain map that is the inclusion of the subcomplex into C.

```
quo< C | D >
```
```
quo< C | S >
```
```
quo< C | L >
```

Given a complex C and a subcomplex D of C, returns the quotient complex C/D together with the natural quotient map. If given as a sequence S of submodules of the terms of C or a list L of sequences of elements of the terms of C, then the submodule generated by S or L is created and the quotient computed. The function also returns the chain map of C on the quotient.

56.1.3 Access Functions

```
Degrees(C)
```

Returns the first and last degrees of the defined terms of the complex C.

```
Algebra(C)
```

Given a complex C over a basic algebra A, this function returns the algebra A.

`BoundaryMap(C, n)`

Returns the boundary map of the complex C from the term of degree n to the term of degree $n-1$.

`BoundaryMaps(C)`

The list of boundary maps of the complex C.

`DimensionsOfHomology(C)`

Returns the list of the dimensions of the homology groups of the complex C.

`DimensionOfHomology(C, n)`

Returns the dimension of the homology group of the complex C in degree n.

`DimensionsOfTerms(C)`

Returns the list of the dimensions of the terms of the complex C.

`Term(C, n)`

The module in the complex C in degree n.

`Terms(C)`

The sequence of terms of the complex C.

56.1.4 Elementary Operations

`DirectSum(C, D)`

Returns the direct sum of the complex C and the complex D.

`Homology(C)`

`HomologyOfChainComplex(C)`

The sequence of homology groups of the complex C as a sequence of modules.

`Homology(C, n)`

Returns the homology group in degree n of the complex C, as an A-module.

`Prune(C)`

Returns the complex that consists of the terms of C with the right end term (term of lowest degree) removed.

`Prune(C,n)`

Returns the complex that consists of the terms of C with n terms removed from the right end.

`Preprune(C)`

Returns the complex that consists of the terms of C with the left end term (term of highest degree) removed.

`Preprune(C,n)`

Returns the complex that consists of the terms of C with n terms removed from the left end.

`Shift(C, n)`

Given the complex C in degrees r to s, returns the complex in shifted degrees $r+n$ to $s+n$. The integer n may be either positive or negative. The maps in the complex are all multiplied by the scalar $(-1)^n$.

`ShiftToDegreeZero(C)`

Given the complex C, returns the shift of C so that the last term, the term of lowest degree, is in degree 0.

`Splice(C, D)`

Given two complexes C and D over the same basic algebra, such that the end term of C coincides with the initial term of D, form the complex that corresponds to the concatenation of C and D (as sequences of maps) This function checks that the resulting sequence of maps forms a complex. The degrees of complex D remain unchanged while the degrees of the terms in complex C are changed to fit.

`Splice(C, D, f)`

The splice of the complex C with the complex D along the map f from the last term of C to the first term of D. The degree of the last term of the splice is the same as the degree of the last term of the complex D.

56.1.5 Extensions

The following are elementary operations related to extending complexes

`LeftExactExtension(C)`

Given a complex C of modules over a basic algebra, returns the complex of length one greater that is obtained by adjoining the inclusion map from the kernel of the boundary map in highest degree to the term of highest degree in C.

`RightExactExtension(C)`

Given a complex C of modules over a basic algebra, returns the complex of length one greater that is obtained by adjoining the quotient map to the cokernel of the boundary map of lowest degree from the term of lowest degree in C.

`ExactExtension(C)`

Returns the left and right exact extensions of the complex C.

`LeftZeroExtension(C)`

`LeftZeroExtension(C, n)`

Given a complex C of modules over a basic algebra, returns the complex of length one greater that is obtained by adjoining the zero map from the zero module to the term of highest degree in the complex. If a natural number n is included in the input then the operation is performed n times.

`RightZeroExtension(C)`

`RightZeroExtension(C, n)`

Given a complex C of modules over a basic algebra, returns the complex of length one greater that is obtained by adjoining the zero map to the zero module from the term of lowest degree in the complex. If a natural number n is included in the input then the operation is performed n times.

`ZeroExtension(C)`

Returns the left and right zero extensions of the complex C.

`EqualizeDegrees(C, D)`

`EqualizeDegrees(Q)`

Given complexes C and D over the same algebra, the function returns the complexes obtained by taking zero extensions of C and D, if necessary, so that both complexes have the same degrees. The input can also be given as a sequence Q of complexes, in which case the function returns the sequence of complexes obtained by taking zero extensions of the elements of Q, if necessary, until all of the elements of the sequence have the same degrees.

`EqualizeDegrees(C, D, n)`

The two complexes, C and D, with zero extension sufficient that the first and the shift by degree n of the second have the same degrees.

56.1.6 Predicates

The following functions return a Boolean value.

`IsExact(C)`

Returns `true` if and only if the complex C is an exact sequence, i.e. the image of each map in C is equal to the kernel of the succeeding map (with the obvious exceptions of the first and last maps of the complex). If the complex has only two terms then this is true trivially.

`IsShortExactSequence(C)`

Returns `true` if the complex C consists of a short exact sequence together with other terms that are zero. The function returns also the degrees of the complex of nonzero terms.

`IsExact(C, n)`

Returns `true` if and only if the complex C is an exact sequence in degree n, i.e. the image of the map in C into the term of degree n is equal to the kernel of the succeeding map.

`IsZeroComplex(C)`

Returns `true` if and only if the complex C is composed entirely of zero modules and maps.

`IsZeroMap(C, n)`

Returns `true` if and only if the boundary map in degree n of the complex C is the zero map.

`IsZeroTerm(C, n)`

Returns `true` if and only if the term in degree n of the complex C is the zero object.

Example H56E1

We construct the quiver algebra of a quiver with three nodes and three arrows going from node 1 to node 2, from 2 to 1 and 2 to 3. The relation is that $(ab)^3a = 0$ where a is the first arrow and b is the second. Then we construct projective and injective resolutions of the first simple module.

```
> ff := GF(8);
> FA<e1,e2,e3,a,b,c> := FreeAlgebra(ff,6);
> rrr := [a*b*a*b*a*b*a];
> D := BasicAlgebra(FA,rrr,3,[<1,2>,<2,1>,<2,3>]);
> D;
Basic algebra of dimension 23 over GF(2^3)
Number of projective modules: 3
Number of generators: 6
> DimensionsOfProjectiveModules(D);
[ 10, 12, 1 ]
> DimensionsOfInjectiveModules(D);
[ 8, 7, 8 ]
```

Here is the opposite algebra:

```
> OD := OppositeAlgebra(D);
reverse trees
> OD;
Basic algebra of dimension 23 over GF(2^3)
Number of projective modules: 3
Number of generators: 6
```

```
> s1 := SimpleModule(D,1);
> P,mu := ProjectiveResolution(s1,7);
> P;
Basic algebra complex with terms of degree 7 down to 0
Dimensions of terms: 12 12 12 12 12 12 12 10
> Q,nu := InjectiveResolution(s1,7);
> Q;
Basic algebra complex with terms of degree 0 down to -7
Dimensions of terms: 8 7 0 0 0 0 0 0
```

Note that the projective and injective resolutions are complexes with the appropriate augmentation and coaugmentation maps. First we form the two term complex whose boundary map is the composition of the augmentation and coaugmentation maps.

```
> theta := MapToMatrix(hom<Term(P,0)-> Term(Q,0)|mu*nu>);
> E := Complex(theta,0);
```

Then we splice all of this together.

```
> R := Splice(P,E);
> S := Splice(R,Q);
> S;
Basic algebra complex with terms of degree 8 down to -7
Dimensions of terms: 12 12 12 12 12 12 12 10 8 7 0 0 0 0 0 0
```

56.2 Chain Maps

A chain map from complex C to complex D is a sequence of homomorphisms between the terms of C and the terms of D such that the maps commute with the boundary homomorphisms on the two complexes. The chain map has degree zero if the map on C_n has it's image in D_n. Otherwise the degree of the chain map expresses the extent to which the degrees of the terms are raised or lowered by the chain map.

Chain maps do not have to be defined in every degree for which either its domain or codomain is defined. On the other hand it must be defined wherever possible, i.e. for any degree i for which the term of the domain is defined and for which the term of the codomain is defined in degree $i + n$ where n is the degree of the chain map.

The freedom of definition can cause problems with constructions such as the kernel or cokernel. That is, the kernel of a chain complex may not be a complex. For this reason we allow the Kernel and Cokernel functions to work only for proper chain maps where "proper" is defined as follows. Suppose C is a chain complex in degrees a to b and D is a complex in degrees u to v. A chain map in degree n from C to D is proper provided $u \geq a + n$ and $v \geq b + n$.

56.2.1 Creation

`ChainMap(Q, C, D, n)`

Returns the chain map given by the sequence of maps in Q. Each element in Q is a map from a term of the complex C to a corresponding term of the complex D. The integer n is the degree of the chain map. This means that for any i the term in degree i of C is mapped to the term of degree $n + i$ of D. There should be a map defined for any degree i in the range of degrees of C such that there is a corresponding term in degree $n + i$ for the complex D.

`ZeroChainMap(C, D)`

Given chain complexes C and D, construct the chain map from C to D all of whose terms are zero.

56.2.2 Access Functions

`Degree(f)`

Given a chain map f, returns the degree of f. Note that $f : C \to D$ has degree n if it takes the term in degree i of the complex C to the term in degree $i + n$ of the complex D.

`ModuleMap(f, n)`

Given a chain map $f : C \to D$ and an integer n, this function returns the map from the term in degree n of the complex C to the term in degree $n + Degree(f)$ of the complex D.

`Kernel(f)`

Returns the kernel complex of f and the inclusion of the kernel in the domain of f. The function is only defined for proper chain maps. If the chain map is not defined for a certain term of the domain then that term is equal to the term of the kernel complex. That is, if f is not defined on a term then the functions acts as if f were the zero map on that term.

`Cokernel(f)`

Returns the cokernel complex of f and the projection of the cokernel onto the codomain of f. The function is only defined for proper chain maps. If the chain map is not defined for a certain term of the codomain then that term is equal to the term of the cokernel complex. That is, if f is not defined on a term then the function acts as if f were the zero map on that term.

`Image(f)`

Returns the image complex of f and the inclusion of the image in the codomain of f and the projection of the domain of f on to the image. The function is only defined for proper chain maps.

56.2.3 Elementary Operations

`f + g`

The sum of two chain maps with the same domain and codomain.

`a * g`

The product of the chain map g by the scalar a in the base ring of the algebra.

`f * g`

The composition of the two chain maps f and g.

56.2.4 Predicates

The following functions return a Boolean value.

`IsSurjective(f)`

Returns `true` if the chain map f is a surjection in every degree, `false` otherwise.

`IsInjective(f)`

Returns `true` if the chain map f is an injection in every degree, `false` otherwise.

`IsZero(f)`

Returns `true` if the chain map f is zero in every degree, `false` otherwise.

`IsIsomorphism(f)`

Returns `true` if the chain map f is an isomorphism of chain complexes, `false` otherwise.

`IsShortExactSequence(f, g)`

Returns `true` if the sequence of chain complexes, $0 \to Domain(f) \to Domain(g) \to Codomain(g) \to 0$, where the internal maps are f and g, is exact.

`IsChainMap(L, C, D, n)`

Returns `true` if the list of maps L from the terms of complex C to the terms of the complex D is a chain map of degree n, i. e. it has the right length and the diagram commutes.

`IsChainMap(f)`

Returns `true` if the supposed chain map f really is a chain map, i. e. the diagrams commute.

`IsProperChainMap(f)`

Returns `true` if the chain map f is a proper chain map, a necessary condition for taking kernel and cokernel.

`HasDefinedModuleMap(C,n)`

Returns `true` if the module map in degree n of the complex C is defined.

Example H56E2

We from the basic algebra of the direct product of a cyclic group of order 3 with symmetric group on three letters over the field with three elements.

```
> FA<e1,e2,a,b> := FreeAlgebra(GF(3),4);
> MM:= [e1 +e2 - FA!1, a*b*a, b*a*b];
> BS3 := BasicAlgebra(FA, MM, 2, [<1,2>,<2,1>]);
> gg := CyclicGroup(3);
> BC3 := BasicAlgebra(gg,GF(3));
> A := TensorProduct(BS3,BC3);
> A;
Basic algebra of dimension 18 over GF(3)
Number of projective modules: 2
Number of generators: 6
```

Now we want the projective resolution of the second simple module as a complex. This we will manipulate and take a somewhat random complex.

```
> PR := ProjectiveResolution(SimpleModule(A,2),12);
> PR;
Basic algebra complex with terms of degree 12 down to 0
Dimensions of terms: 117 108 99 90 81 72 63 54 45 36 27 18 9
> PR := Prune(PR);
> PR := Prune(PR);
> PR := Prune(PR);
> PR;
Basic algebra complex with terms of degree 12 down to 3
Dimensions of terms: 117 108 99 90 81 72 63 54 45 36
> PR := Prune(PR);
> PR := Prune(PR);
> PR;
Basic algebra complex with terms of degree 12 down to 5
Dimensions of terms: 117 108 99 90 81 72 63 54
> PR := ZeroExtension(PR);
> PR;
Basic algebra complex with terms of degree 13 down to 4
Dimensions of terms: 0 117 108 99 90 81 72 63 54 0
> PR := Shift(PR,-4);
> PR;
Basic algebra complex with terms of degree 9 down to 0
Dimensions of terms: 0 117 108 99 90 81 72 63 54 0
> S := [* *];
> for i := 1 to 10 do
>    S[i] := [Random(Term(PR,10-i)),Random(Term(PR,10-i))];
> end for;
> C,mu := Subcomplex(PR,S);
> C;
Basic algebra complex with terms of degree 9 down to 0
Dimensions of terms: 0 36 67 64 62 63 58 54 51 0
```

```
> Homology(C);
[
    AModule of dimension 4 over GF(3),
    AModule of dimension 6 over GF(3),
    AModule of dimension 5 over GF(3),
    AModule of dimension 3 over GF(3),
    AModule of dimension 2 over GF(3),
    AModule of dimension 1 over GF(3),
    AModule of dimension 4 over GF(3),
    AModule of dimension 26 over GF(3)
]
[
    Mapping from: AModule of dimension 4 over GF(3) to AModule of dimension 4
    over GF(3),
    Mapping from: AModule of dimension 38 over GF(3) to AModule of dimension 6
    over GF(3),
    Mapping from: AModule of dimension 34 over GF(3) to AModule of dimension 5
    over GF(3),
    Mapping from: AModule of dimension 33 over GF(3) to AModule of dimension 3
    over GF(3),
    Mapping from: AModule of dimension 31 over GF(3) to AModule of dimension 2
    over GF(3),
    Mapping from: AModule of dimension 33 over GF(3) to AModule of dimension 1
    over GF(3),
    Mapping from: AModule of dimension 29 over GF(3) to AModule of dimension 4
    over GF(3),
    Mapping from: AModule of dimension 51 over GF(3) to AModule of dimension 26
    over GF(3)
]
>  D := Cokernel(mu);
>  D;
Basic algebra complex with terms of degree 9 down to 0
Dimensions of terms: 0 81 41 35 28 18 14 9 3 0
>  Homology(D);
[
    AModule of dimension 64 over GF(3)
    AModule of dimension 5 over GF(3),
    AModule of dimension 3 over GF(3),
    AModule of dimension 2 over GF(3),
    AModule of dimension 1 over GF(3),
    AModule of dimension 4 over GF(3),
    AModule of dimension 3 over GF(3),
    AModule of dimension 3 over GF(3)
]
[
    Mapping from: AModule of dimension 64 over GF(3) to AModule of dimension 64
    over GF(3),
    Mapping from: AModule of dimension 22 over GF(3) to AModule of dimension 5
```

```
    over GF(3),
    Mapping from: AModule of dimension 22 over GF(3) to AModule of dimension 3
    over GF(3),
    Mapping from: AModule of dimension 15 over GF(3) to AModule of dimension 2
    over GF(3),
    Mapping from: AModule of dimension 14 over GF(3) to AModule of dimension 1
    over GF(3),
    Mapping from: AModule of dimension 8 over GF(3) to AModule of dimension 4
    over GF(3),
    Mapping from: AModule of dimension 9 over GF(3) to AModule of dimension 3
    over GF(3),
    Mapping from: AModule of dimension 3 over GF(3) to AModule of dimension 3
    over GF(3)
]
```

56.2.5 Maps on Homology

`InducedMapOnHomology(f, n)`

The homomorphism induced on homology by the chain map f in degree n.

`ConnectingHomomorphism(f, g, n)`

The connecting homomorphism in degree n of the short exact sequence of chain complexes given by the chain maps f and g.

`LongExactSequenceOnHomology(f, g)`

The long exact sequence on homology for the exact sequence of complexes given by the chain maps f and g as a chain complex with the homology group in degree i for the Cokernel of the complex C appearing in degree $3i$.

Example H56E3

We create a basic algebra with two simple modules and four nonidempotent generators. The relations are given in the sequence `rrr`.

```
> ff := GF(3);
> p := Characteristic(ff);
> FA<e1, e2, y, x, a, b> := FreeAlgebra(ff,6);
> rrr := [y^p,x^p,x*y+y*x,x*a*b-a*b*x,y*a*b-a*b*y,(b*a)^2,(b*a)^2];
> A := BasicAlgebra(FA,rrr,2,[<1,1>,<1,1>,<1,2>,<2,1>]);
> A;
Basic algebra of dimension 81 over GF(3)
Number of projective modules: 2
Number of generators: 6
> DimensionsOfProjectiveModules(A);
```

```
[ 45, 36 ]
```

Now we generate the simple modules and their projective covers. We want a module PP that is the direct sum of two copies of each of the projective indecomposable module.

```
> S1 := SimpleModule(A,1);
> PP := ProjectiveModule(A,[2,2]);
```

Now we are going to create a complex all of whose terms are isomorphic to PP. To make it interesting we will insist that the first homomorphism have its image in the second radical layer of PP.

```
> J1 := JacobsonRadical(PP);
> theta1 := Morphism(J1,PP);
> J2 := JacobsonRadical(J1);
> theta2 := Morphism(J2,J1);
> theta := theta2*theta1;
> HomPJ := AHom(PP, J2);
> HomPJ;
KMatrixSpace of 162 by 150 matrices and dimension 300 over GF(3)
> gamma := Random(HomPJ)*theta;
> LL := [* gamma *];
```

So we now have the first boundary map of the complex. The other boundary maps are generated randomly.

```
> for i := 1 to 15 do
>     K, phi := Kernel(gamma);
>     HomPK := AHom(PP,K);
>     gamma := Random(HomPK)*MapToMatrix(phi);
>     LL := [* gamma *] cat LL;
> end for;
```

Now we make the list into a chain complex.

```
> C := Complex(LL,0);
> C;
Basic algebra complex with terms of degree 16 down to 0
Dimensions of terms: 162 162 162 162 162 162 162 162 162 162 162 162 162 162
162 162 162
> DimensionsOfHomology(C);
[ 30, 23, 34, 32, 35, 32, 35, 42, 30, 34, 32, 33, 29, 42, 21 ]
Now we will get a random subcomplex.
> a,b := Degrees(C);
> S, mu := RandomSubcomplex(C,[2: i in [1 .. a-b+1]]);
> S;
Basic algebra complex with terms of degree 16 down to 0
Dimensions of terms: 162 155 162 162 162 154 142 162 162 148 154 162 162 152
155 152 162
> DimensionsOfHomology(S);
```

```
[ 27, 27, 34, 32, 35, 33, 48, 42, 28, 44, 38, 33, 29, 48, 24 ]
```

Now take the quotient.

```
> Q,nu := quo<C|S>;
> Q;
Basic algebra complex with terms of degree 16 down to 0
Dimensions of terms: 0 7 0 0 0 8 20 0 0 14 8 0 0 10 7 10 0
> DimensionsOfHomology(Q);
[ 7, 0, 0, 0, 6, 18, 0, 0, 12, 6, 0, 0, 8, 5, 10 ]
```

Now we check to see if this is a short exact sequence of chain maps.

```
> IsShortExactSequence(mu,nu);
true
> lll := LongExactSequenceOnHomology(mu,nu);
Basic algebra complex with terms of degree 47 down to 3
Dimensions of terms: 27 30 7 27 23 0 34 34 0 32 32 0 35 35 6 33 32 18 48 35 0
42 42 0 28 30 12 44 34 6 38 32 0 33 33 0 29 29 8 48 42 5 24 21 10
> IsExact(lll);
true
```

PART IX
FINITE GROUPS

57 GROUPS

Chapter 57
GROUPS

57.1 Introduction

Groups arise in several different categories in MAGMA. In the case of the category of permutation groups and the category of soluble groups defined by a power-conjugate presentation, all groups in the category are finite. However, the finitely-presented group category, the polycyclic group category, the abelian group category and the matrix group category contain both finite and infinite groups. In the case of the abelian group category and the matrix group category, a large number of functions are available for finite groups only. In the near future, these functions will be extended to finite finitely-presented groups of moderate order.

In this chapter, we discuss the functions that are provided for groups collectively, noting especially those functions that are available only for finite groups. Descriptions of functions that depend upon the particular category may be found in the chapter devoted to that category.

57.1.1 The Categories of Finite Groups

At present MAGMA contains five main categories of finite groups:

(i) Permutation groups: category GrpPerm;

(ii) Finite matrix groups: category GrpMat;

(iii) Finite solvable groups given by a power-conjugate presentation: category GrpPC;

(iv) Finite abelian groups: category GrpAb;

(v) Finite polycyclic groups: category GrpGPC.

Note that the categories GrpMat, GrpAb and GrpGPC contain both finite and infinite groups; most of the operations described in this chapter apply only to finite groups belonging to these categories. In this chapter we will use the category name `GrpFin` to collectively refer to categories GrpPerm and GrpPC and the subcategories of GrpMat, GrpAb and GrpGPC consisting of finite groups. The category name `Grp` will be used when the operation does not depend upon the finiteness of the group.

57.2 Construction of Elements

57.2.1 Construction of an Element

Throughout this subsection we shall assume that the carrier set for the group G is a subset of the set S. Thus, if G is a permutation group on the set X, its carrier set will be a subset of $\mathrm{Sym}(X)$.

```
elt< G | L >
```

Given a group G whose elements are a subset of the set S, and a list L of objects $a_1, a_2, \ldots, a_n$ defining an element of S, construct this element g of S. Then, the element g will be tested for membership of G, and if g is not an element of G, the function will fail. If g does lie in G, g will be returned with G as its parent.

```
G ! Q
```

Given a group G whose elements are a subset of the set S, and a sequence $Q = [a_1, a_2, \ldots, a_n]$ defining an element of S, construct this element g of S. Then, the element g will be tested for membership of G, and if g is not an element of G, the function will fail. If g does lie in G, g will be returned with G as its parent.

```
Identity(G)
```

```
Id(G)
```

Construct the identity element in the group G.

57.2.2 Coercion

```
G ! g
```

Given a group G and an element g of H, where G and H are subgroups of some common over-group and g is contained in G, embed g in G. Thus this operator changes the parent of g into G. The coercion may fail for groups in the category GrpFP.

57.2.3 Homomorphisms

```
hom< G -> H | L >
```

Return the group homomorphism $\phi : G \to H$ defined by extending the map of the generators of G, as given by the list L on the right side of the constructor. Suppose that the generators of G are $g_1, \ldots, g_n$, and that $\phi(g_i) = h_i$ for each i. Then L must be one of the following:

(a) a list of the n 2-tuples $< g_i, h_i >$ (order not important);

(b) a list of the n arrow-pairs g_i `->` h_i (order not important);

(c) $h_1, \ldots, h_n$ (order is important).

For its computations, MAGMA often assumes that the mapping so defined is a homomorphism without attempting to verify this.

For certain categories of groups, e.g. GrpGPC, the homomorphism constructor provides some additional functionality. See the chapter on the appropriate category for further information.

`hom< G -> H | x :-> e(x) >`

Return the group homomorphism $\phi : G \to H$ defined by the rule $\phi(x) = e(x)$, where x is a general element of G and $e(x)$ is an expression in x. The symbol x may be any identifier name, and has local scope. For its computations, MAGMA assumes the expression defines a homomorphism, but does not verify this.

`IdentityHomomorphism(G)`

Return the identity homomorphism $\phi : G \to G : x \mapsto x$.

Example H57E1

Construction of an isomorphism from the cyclic group of order 15 to the abelian group isomorphic to $\mathbf{Z}/15\mathbf{Z}$, by giving the image of the generator:

```
> C15 := CyclicGroup(15);
> C15;
Permutation group C15 acting on a set of cardinality 15
Order = 15 = 3 * 5
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
> A15 := AbelianGroup([15]);
> A15;
Abelian Group isomorphic to Z/15
Defined on 1 generator
Relations:
    15*A15.1 = 0
> iso11 := hom< C15 -> A15 | C15.1 -> 11*A15.1 >;
> A15 eq iso11(C15);
true
> forall{ <c, d> : c, d in C15 | iso11(c * d) eq iso11(c) * iso11(d) };
true
```

Example H57E2

An endomorphism of the same cyclic group, defined using an expression. The image is cyclic of order 5.

```
> C15 := CyclicGroup(15);
> h := hom< C15 -> C15 | g :-> g^3 >;
> forall{ <c, d> : c, d in C15 | h(c * d) eq h(c) * h(d) };
true
> im := h(C15);
> im;
Permutation group im acting on a set of cardinality 15
Order = 5
```

```
    (1, 4, 7, 10, 13)(2, 5, 8, 11, 14)(3, 6, 9, 12, 15)
> IsCyclic(im);
true
```

57.2.4 Arithmetic with Elements

`g * h`

Product of element g and element h, where g and h belong to the same generic group U. If g and h both belong to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h belong to subgroups H and K of a subgroup G of U, then the product is returned as an element of G. Otherwise, the product is returned as an element of U. The product in abelian groups is called the sum and is written `g + h` instead.

`g ^ n`

The n-th power of the group element g, where n is a positive, negative or zero integer. In abelian groups, this is written as a scalar product `n * g` instead.

`g / h`

Product of the group element g by the inverse of the group element h, i.e., the element gh^{-1}. Here g and h must belong to the same generic group U. The rules for determining the parent group of g/h are the same as for gh. In abelian groups, this is written additively as `g - h`.

`g ^ h`

Conjugate of the group element g by the group element h, i.e., the element $h^{-1}gh$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g^h are the same as for gh. In abelian groups, this operation does not exist.

`(g, h)`

Commutator of the group elements g and h, i.e., the element $g^{-1}h^{-1}gh$. Here g and h must belong to the same generic group U. The rules for determining the parent group of (g, h) are the same as those for gh.

`(g1, ..., gr)`

Given r elements $g_1, \ldots, g_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

`g eq h`

Given elements g and h belonging to the same generic group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given elements g and h belonging to the same generic group, return `true` if g and h are distinct elements, `false` otherwise.

`IsId(g)`

`IsIdentity(g)`

Returns `true` if the group element g is the identity element.

`Order(g)`

The order of the group element g.

Example H57E3

We illustrate the arithmetic operations by applying them to some elements of Sym(9).

```
> G := Sym(9);
> x := G ! (1,2,4)(5,6,8)(3,9,7);
> y := G ! (4,5,6)(7,9,8);
> x*y;
(1, 2, 5, 4)(3, 8, 6, 7)
> x^-1;
(1, 4, 2)(3, 7, 9)(5, 8, 6)
> x^2;
(1, 4, 2)(3, 7, 9)(5, 8, 6)
> x / y;
(1, 2, 6, 9, 8, 4)(3, 7)
> x^y;
(1, 2, 5)(3, 8, 9)(4, 7, 6)
> (x, y);
(1, 7, 3, 6)(4, 5, 9, 8)
> x^y eq y^x;
false
> CycleStructure(x^2*y);
[ <6, 1>, <2, 1>, <1, 1> ]
> Degree(y);
6
> Order(x^2*y);
6
```

57.3 Construction of a General Group

57.3.1 The General Group Constructors

The chapters on the individual group categories describe several methods for constructing groups; this section indicates one approach only.

```
PermutationGroup< X | L >
```

```
PermutationGroup< n | L >
```

```
MatrixGroup< n, R | L >
```

These expressions construct, respectively: a permutation group G acting on the set X; a permutation group G acting on the set $X = \{1, \ldots, n\}$; or a matrix group G of degree n over the ring R. The generic group U of which G is a subgroup will be $\mathrm{Sym}(X)$ in the permutation case or $\mathrm{GL}(n, R)$ in the matrix case. There are two return values: G, and the inclusion homomorphism from G to U.

The generators of G are defined by the list L. Each term of L must be an object of one of the following types:

(a) Either (permutation case) a sequence of n elements of X, or (matrix case) a sequence of n^2 elements of R, defining an element of U;

(b) A set or sequence of sequences of type (a);

(c) An element of U;

(d) A set or sequence of elements of U;

(e) A subgroup of U;

(f) A set or sequence of subgroups of U.

Each element or group specified by the list must belong to the *same* generic group. The group G will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of G consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed (unless G is trivial).

The `PermutationGroup` constructor is shorthand for the two statements:

```
U := SymmetricGroup(X);
G := sub< U | L >;
```

and the `MatrixGroup` constructor is shorthand for the two statements:

```
U := GeneralLinearGroup(n, R);
G := sub< U | L >;
```

where `sub< ... >` is the subgroup constructor described in the next subsection.

```
Group< X | R >
```

```
PolycyclicGroup< X | R >
```

```
AbelianGroup< X | R >
```

These expressions construct, respectively, a finitely presented group, a finite soluble group given by a power-conjugate presentation or a polycyclic group, and an abelian group, in the categories GrpFP, GrpPC or GrpGPC, and GrpAb. Given a list X of identifier names $x_1, \ldots, x_r$, and a list of relations R over them, first construct the free group F (in GrpFP or GrpAb) on the generators $x_1, \cdots, x_r$, and then construct the quotient G of F corresponding to the normal subgroup of F defined by the relations R. There are two return values: G, and the natural homomorphism from F to G.

The relations of G are defined by the list R. Each term of R must be an object of one of the following types:

(a) A word w of F, interpreted as the relator $w =$ identity of F;

(b) A relation $w_1 = w_2$, where w_1 and w_2 are words of F;

(c) A relation list $w_1 = w_2 = \cdots = w_r$, where the w_i are words of F, interpreted as the set of relations $w_1 = w_r, \ldots, w_{r-1} = w_r$.

Within R, the identity element of F may be represented by the digit `1` for `Group` or `PolycyclicGroup`, and `0` for `AbelianGroup`.

The construct $x_1, \ldots, x_n$ defines names for the generators of G that are local to the constructor, i.e., they are used when writing down the relations to the right of the bar. However, no assignment of *values* to these identifiers is made. If the user wants to refer to the generators by these (or other) names, then the *generators assignment* construct must be used on the left hand side of an assignment statement.

The constructor `PolycyclicGroup` returns either a finite soluble group given by a power-conjugate presentation (category GrpPC) or a general polycyclic group (category GrpGPC), depending on the arguments. R must be either a valid power-conjugate presentation for a finite soluble group or a consistent polycyclic presentation. If R is a valid power-conjugate presentation for a finite soluble group, a group in the category GrpPC is returned, unless the parameter `Class` is set to `"GrpGPC"`. If the parameter `Class` is set to `"GrpGPC"` or if R is not a valid power-conjugate presentation for a finite soluble group and the parameter `Class` is not set to `"GrpPC"`, a general polycyclic group in the category GrpGPC is returned. In any case, the free group F is in the category GrpFP. If R is neither a valid power-conjugate presentation for a finite soluble group nor a consistent polycyclic presentation, or if R does not match the value of the parameter `Class`, a runtime error is caused.

For a detailed description of this constructor and in particular for a description of power-conjugate presentations and consistent polycyclic presentations, we refer to Chapter 63 and Chapter 72, respectively.

Example H57E4

(1) The permutation group of degree 8 generated by the permutations $(1,7,2,8)(3,6,4,5)$ and $(1,4,2,3)(5,7,6,8)$:

```
> G := PermutationGroup< 8 |
>     (1, 7, 2, 8)(3, 6, 4, 5), (1, 4, 2, 3)(5, 7, 6, 8) >;
> G;
Permutation group G acting on a set of cardinality 8
    (1, 7, 2, 8)(3, 6, 4, 5)
    (1, 4, 2, 3)(5, 7, 6, 8)
```

(2) A matrix group of degree 2 over $\mathbf{F}_9$:

```
> K<w> := GF(9);
> M := MatrixGroup< 2, K | [w,w,1,2*w], [0,2*w,1,1], [1,0,1,2] >;
> M;
MatrixGroup(2, GF(3^2))
Generators:
    [  w   w]
    [  1 w^5]

    [  0 w^5]
    [  1   1]

    [  1   0]
    [  1   2]
> Order(M);
5760
```

(3) The finitely presented group Q defined by the presentation

$$< s,t,u \mid t^2, u^{17}, s^2 = t^s = t, u^s = u^{16}, u^t = u >,$$

together with the natural homomorphism from the free group to Q:

```
> Q<s,t,u>, h := Group< s, t, u |
>     t^2, u^17, s^2 = t^s = t, u^s = u^16, u^t = u >;
> Q;
Finitely presented group Q on 3 generators
Relations
    t^2 = Id(Q)
    u^17 = Id(Q)
    s^2 = t
    t^s = t
    u^s = u^16
    u^t = u
> Domain(h);
Finitely presented group on 3 generators (free)
```

(4) The soluble group of order 70 defined by the presentation $< a,b,c \mid a^2 = b, b^5 = c, c^7 >$:

```
> G<a,b,c> := PolycyclicGroup< a, b, c | a^2 = b, b^5 = c, c^7 >;
```

```
> G;
GrpPC : G of order 70 = 2 * 5 * 7
PC-Relations:
a^2 = b,
b^5 = c,
c^7 = Id(G)
```

(5) A finite abelian group on 4 generators:

```
> G := AbelianGroup< h, i, j, k | 5*h, 4*i, 7*j, 2*k - h >;
> G;
Abelian Group isomorphic to Z/2 + Z/140
Defined on 4 generators
Relations:
    G.1 + 8*G.4 = 0
    4*G.2 = 0
    7*G.3 = 0
    10*G.4 = 0
> Order(G);
280
```

Example H57E5

Using the constructor `PolycyclicGroup` with different values of the parameter `Class`, we construct the dihedral group of order 10 first as a finite soluble group given by a power-conjugate presentation (GrpPC) and next as a general polycyclic group (GrpGPC). Note that the presentation $\langle a, b \,|\, a^2, b^5, b^a = b^4 \rangle$ is both a valid power-conjugate presentation and a consistent polycyclic presentation, so we have to set the parameter `Class` to `"GrpGPC"` if we want to construct a group in the category GrpGPC.

```
> G1<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 >;
> G1;
GrpPC : G1 of order 10 = 2 * 5
PC-Relations:
    a^2 = Id(G1),
    b^5 = Id(G1),
    b^a = b^4
> G2<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 : Class := "GrpGPC">;
> G2;
GrpGPC : G2 of order 10 = 2 * 5 on 2 PC-generators
PC-Relations:
    a^2 = Id(G2),
    b^5 = Id(G2),
    b^a = b^4
```

We construct the infinite dihedral group as a group in the category GrpGPC from a consistent polycyclic presentation. We do not have to use the parameter `Class` in this case.

```
> G3<a,b> := PolycyclicGroup< a,b | a^2, b^a=b^-1>;
> G3;
```

```
GrpGPC : G3 of infinite order on 2 PC-generators
PC-Relations:
    a^2 = Id(G3),
    b^a = b^-1
```

The presentation $\langle a, b \,|\, a^2, b^4, b^a = b^3 \rangle$ is not a valid power-conjugate presentation for the dihedral group of order 8, since the exponent of b is not prime. However, it is a consistent polycyclic presentation. Consequently, the constructor `PolycyclicGroup` without specifying a value for the parameter `Class` returns a group in the category GrpGPC.

```
> G4<a,b> := PolycyclicGroup< a,b | a^2, b^4, b^a=b^3 >;
> G4;
GrpGPC : G4 of order 2^3 on 2 PC-generators
PC-Relations:
    a^2 = Id(G3),
    b^4 = Id(G3),
    b^a = b^3
```

57.3.2 Construction of Subgroups

`sub< G | L >`

Given the group G, construct the subgroup H of G, generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) A MAGMA object which may be coerced into G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic group. The subgroup H will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of H consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed (unless H is trivial).

`ncl< G | L >`

Given the group G, construct the subgroup H of G that is the *normal closure* of the subgroup H generated by the elements specified by the list L, where the possibilities for L are the same as for the `sub`-constructor.

Example H57E6

Let Q be the finitely presented group in generators s, t, u constructed in an earlier example. We construct the subgroup S of Q generated by ts^2 and u^4:

```
> Q<s,t,u>, h := Group< s, t, u |
>     t^2, u^17, s^2 = t^s = t, u^s = u^16, u^t = u >;
> S := sub< Q | t*s^2, u^4 >;
> S;
Finitely presented group S on 2 generators
Generators as words
    S.1 = $.2 * $.1^2
    S.2 = $.3^4
```

57.3.3 Construction of Quotient Groups

`quo< G | L >`

Given the group G, construct the quotient group $Q = G/N$, where N is the normal closure of the subgroup of G generated by the elements specified by L. The clause L is a list of one or more items of the following types:

(a) A MAGMA object which can be coerced into G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic group. The function returns

(a) the quotient group Q, and

(b) the natural homomorphism $f : G \to Q$.

Arbitrary quotients may be readily constructed in the case of the categories GrpFP, GrpGPC, GrpPC and GrpAb. However, in the case of permutation and matrix groups, currently the quotient group is constructed via its regular representation, so that the application of this operator is restricted to the case where the index of N in G is less than 2^{30}.

The second return value is the epimorphism from G to the resulting quotient group.

`G / N`

Given a (normal) subgroup N of the group G, construct the quotient of G by N.

If G is in category GrpFP, N is not checked to be normal in G. In fact, the returned group is the quotient of G by the normal closure of N in G. For all other categories of groups, passing a subgroup which fails to be normal causes a runtime error.

If G is a permutation or matrix group, the quotient group is constructed via its regular representation, so that the application of this operator is restricted to the case where the index of N in G is at most a million. The result returned need not be regular, as an attempt is made to reduce the degree of the result.

Example H57E7

Construction of the quotient of an abelian group, with a demonstration of the use of the natural homomorphism:

```
> G<[x]>, f := AbelianGroup< h, i, j, k | 8*h, 4*i, 6*j, 2*k - h >;
> T, n := quo< G | x[1] + 2*x[2] + 24*x[3], 16*x[3] >;
> T;
Abelian Group isomorphic to Z/2 + Z/16
Defined on 2 generators
Relations:
    4*T.1 = 0
    16*T.2 = 0
> n(x);
[
    2*T.1,
    T.1 + 12*T.2,
    T.2
]
> n(sub< G | x[1] + x[2] + x[3] >);
Abelian Group isomorphic to Z/16
Defined on 1 generator in supergroup T:
    $.1 = 3*T.1 + T.2
Relations:
    16*$.1 = 0
```

57.4 Standard Groups and Extensions

57.4.1 Construction of a Standard Group

A number of functions are provided which construct various standard groups. The effect of these functions is to construct the group on some standard set of generators. The group category of the result may be specified as an argument to the function.

`AbelianGroup(C, Q)`

`AbelianGroup(Q)`

Construct the abelian group defined by the sequence $Q = [n_1, \ldots, n_r]$ of positive integers. The function constructs the direct product of cyclic groups $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_r}$. In some categories, n_i may also be 0, denoting the infinite cyclic group $\mathbf{Z}$. If the single-argument version of the function is used, the group will be constructed in the category GrpAb; otherwise, its category will be C, where C may be GrpAb, GrpFP, GrpGPC, GrpPC or GrpPerm.

`AlternatingGroup(C, n)`

`AlternatingGroup(n)`

`Alt(C, n)`

`Alt(n)`

Construct the alternating group on n letters. If the single-argument version of the function is used, the group will be constructed in the category GrpPerm; otherwise, its category will be C, where C may be GrpFP or GrpPerm.

`CyclicGroup(C, n)`

`CyclicGroup(n)`

Construct the cyclic group of order n. If the single-argument version of the function is used, the group will be constructed in the category GrpPerm; otherwise, its category will be C, where C may be GrpAb, GrpFP, GrpGPC, GrpPC or GrpPerm.

`DihedralGroup(C, n)`

`DihedralGroup(n)`

Construct the dihedral group of order $2 * n$. If the single-argument version of the function is used, the group will be constructed in the category GrpPerm; otherwise, its category will be C, where C may be GrpFP, GrpGPC, GrpPC or GrpPerm.

`DicyclicGroup(n)`

`DicyclicGroup(A, a)`

The first intrinsic constructs the dicyclic group of order $4n$. The second, when given an abelian group A and an element a of order 2, constructs the associated dicyclic group generated by A and an x with $x^2 = a$ and $a^x = a^{-1}$ for all $x \in A$.

`SymmetricGroup(C, n)`

`SymmetricGroup(n)`

`Sym(GrpFin, n)`

`Sym(n)`

Construct the symmetric group on n letters. If the single-argument version of the function is used, the group will be constructed in the category GrpPerm; otherwise, its category will be C, where C may be GrpFP or GrpPerm.

`ExtraSpecialGroup(C, p, n : parameters)`

`ExtraSpecialGroup(p, n : parameters)`

Given a prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1}. The isomorphism type of G can be selected using the parameter `Type` described below.

If the two-argument version of the function is used, the group will be constructed in the category GrpPerm; otherwise, its category will be C, where C may be GrpFP, GrpGPC, GrpPC or GrpPerm. If C is GrpFP, GrpPC or GrpPerm, the prime p must be small.

`Type` MonStgElt *Default :* "+"

Possible values for this parameter are "+" (default) and "−".

If `Type` is set to "+", the function returns for $p = 2$ the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If `Type` is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

Example H57E8

(1) The abelian group $\mathbf{Z}_6 \times \mathbf{Z}_2 \times \mathbf{Z}_7$ in the category GrpAb:

```
> A := AbelianGroup([6, 2, 7]);
> A;
Abelian Group isomorphic to Z/2 + Z/42
Defined on 3 generators
Relations:
    6*A.1 = 0
    2*A.2 = 0
    7*A.3 = 0
```

(2) The alternating group on 6 letters as a permutation group:

```
> A6 := Alt(6);
> A6;
Permutation group A6 acting on a set of cardinality 6
Order = 360 = 2^3 * 3^2 * 5
```

```
    (1, 2)(3, 4, 5, 6)
    (1, 2, 3)
```

(3) The dihedral group of order 8 as a GrpPC:

```
> D8 := DihedralGroup(GrpPC, 4);
> D8;
GrpPC : D8 of order 8 = 2^3
PC-Relations:
  D8.2^2 = D8.3,
  D8.2^D8.1 = D8.2 * D8.3
```

(4) The symmetric group on 7 letters as a finitely presented group on generators a and b:

```
> S7<a, b> := SymmetricGroup(GrpFP, 7);
> S7;
Finitely presented group S7 on 2 generators
Relations
    a^7 = Id(S7)
    b^2 = Id(S7)
    (a * b)^6 = Id(S7)
    (a^-1 * b * a * b)^3 = Id(S7)
    (b * a^-2 * b * a^2)^2 = Id(S7)
    (b * a^-3 * b * a^3)^2 = Id(S7)
```

57.4.2 Construction of Extensions

`DirectProduct(G, H)`

Given two groups G and H belonging to the category C, construct the direct product of G and H as a group in C.

`DirectProduct(Q)`

Given a sequence Q of n groups belonging to the category C, construct the direct product $Q[1] \times Q[2] \times \ldots \times Q[n]$ as a group in the category C.

`SemidirectProduct(K, H, f:` *parameters*`)`

Given two groups K and H and a homomorphism $f : H \to \mathrm{Aut}(K)$, construct the semidirect product of K and H where the elements of H act on K via the map f. Return the semidirect product, and maps embedding H and K into the semidirect product.

`MaxDeg` RngIntElt *Default :* 1000000

The maximum degree permutation representation the algorithm will attempt.

`UseRegular` BoolElt *Default :* `false`

Setting `UseRegular` to true forces the algorithm to go via the regular representations of K and H.

Example H57E9

We define G to be the symmetric group of degree 4 and H to be the dihedral group of order 8. We then form the direct product of G and H.

```
> G := SymmetricGroup(4);
> H := DihedralGroup(3);
> D := DirectProduct(G, H);
> D;
Permutation group D acting on a set of cardinality 7
    (1, 2, 3, 4)
    (1, 2)
    (5, 6, 7)
    (5, 6)
> Order(D);
144
```

57.5 Transfer Functions Between Group Categories

Since certain group computations are possible or feasible only for particular group representations, it is often useful to transfer a group from one category to another. The functions in this section take a group and return a group isomorphic to it (or isomorphic to some related group) in another category.

`pQuotient(F, p, c: parameters)`

Given a group F in category GrpFP, a prime p and a positive integer c, construct the largest p-quotient G of F having lower exponent-p class at most c (or 127, if c is given as 0) as group in the category GrpPC. The function also returns the homomorphism from F to G.

The parameters are:

`Exponent` RngIntElt *Default :* 0

If `Exponent := m`, enforce the exponent law, $x^m = 1$, on the group.

`Metabelian` BoolElt *Default :* `false`

If `Metabelian := true`, then a consistent pcp is constructed for the largest metabelian p-quotient of F having lower exponent-p class at most c.

`Print` RngIntElt *Default :* 0

This parameter controls the volume of printing. By default its value is that returned by `GetVerbose("pQuotient")`, which is 0 unless it has been changed through use of `SetVerbose`. The effect is the following:

`Print := 0` : No output.

`Print := 1` : Report order of p-quotient at each class.

`Print := 2` : Report statistics and redundancy information about tails, consistency, collection of relations and exponent enforcement components of calculation.

`Print := 3` : Report in detail on the construction of each class.
Note that the presentation displayed is a *power-commutator* presentation (since this is the version stored by the p-quotient).

`Workspace` RngIntElt *Default :* 5000000

The amount of space requested for the p-quotient computation.

`CosetAction(G, H)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced permutation group L (the image of f);

(c) (if possible) The kernel K of the action (a subgroup of G).

If G is a finitely presented group, then K may be returned undefined.

The permutation representation is obtained by using the Todd-Coxeter procedure to construct the coset table for H in G. Note that G may be an infinite group: it is only necessary that the index of H in G be finite.

`CosetImage(G, H)`

Given a subgroup H of the group G, construct the image of G given by its action on the (right) coset space of H in G, returning it as a permutation group. (This is also the second return value of `CosetAction(G, H)`.)

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of G in its action on the (right) coset space of H in G. (This is also the third return value of `CosetAction(G, H)`.) This function may fail if G is a finitely presented group; it is only available when the index of H in G is very small.

`GPCGroup(G)`

Given a soluble group G, in the category GrpPerm, GrpMat, GrpAb or GrpPC, construct a polycyclic group P isomorphic to G. Currently G must be finite, if it is in the category GrpMat. In addition to returning P, the function returns an isomorphism $\phi : G \to P$.

`PCGroup(G)`

Given a finite soluble group G, in the category GrpPerm, GrpMat, GrpAb or GrpGPC, construct a group S given by a power-conjugate presentation, which is isomorphic to G. In addition to returning S, the function returns an isomorphism $\phi : G \to S$.

FPGroup(G: *parameters*)

StrongGenerators	BoolElt	*Default :* false
Random	BoolElt	*Default :* true
Max	RngIntElt	*Default :* 100
Run	RngIntElt	*Default :* 20

Given a group G, in the category GrpPerm, GrpMat, GrpGPC or GrpPC, construct a finitely presented group F isomorphic to G, by presenting the group on its given generators. For groups in the category GrpPerm and GrpMat, the Todd-Coxeter Schreier algorithm is used to construct the presentation and a choice of a presentation on the given generators or on the strong generators is available. In addition to returning F, the function returns an isomorphism $\phi : F \to G$, such that $\phi(F.i) = G.i$ for all i.

If the parameter StrongGenerators is set to true (GrpPerm and GrpMat only), the presentation will be constructed on the strong generators of G instead of the given generators. If strong generators are not already known for G, they will be constructed; in this case, the other parameters are also meaningful. The parameter Random with its associated parameters Max and Run may be used to apply the Random Schreier algorithm to construct a probable BSGS before commencing the construction of the presentation.

Example H57E10

We construct a finitely presented group G and a subgroup H, then find the permutation representation of G given by its action on the cosets of H. Since the induced permutation group L has the same order as G, the representation is faithful, and the homomorphism $f : G \to L$ is an isomorphism.

```
> G<a, b> := Group< a, b | a^3, b^3, (b * a)^4,
>     ((b^-1)^a * b^-1)^2 * b^a * b >;
> Order(G);
168
> H := sub< G | a^2 * b^2, (a * b)^2 >;
> Index(G, H);
7
> f, L := CosetAction(G, H);
> f;
Mapping from: GrpFP: G to GrpPerm: L
> L;
Permutation group L acting on a set of cardinality 7
    (1, 2, 3)(4, 7, 5)
    (1, 3, 4)(2, 5, 6)
> Order(L);
168
```

Example H57E11

A permutation representation of $\mathrm{Sp}(2,4)$.

```
> M := SymplecticGroup(2, 4);
> #M;
60
> Ms := sub< M | M.1 * M.2 >;
> Index(M, Ms);
12
> PG := CosetImage(M, Ms);
> PG;
Permutation group PG acting on a set of cardinality 12
    (1, 2, 4)(3, 5, 7)(6, 8, 10)(9, 11, 12)
    (1, 3, 2)(4, 6, 8)(5, 7, 9)(10, 12, 11)
> #PG;
60
```

Example H57E12

A finitely presented group isomorphic to $\mathrm{PSU}(3,3)$:

```
> G := PSU(3, 3);
> F<a, b>, phi := FPGroup(G);
> F;
Finitely presented group F on 2 generators
Relations
    a^8 = Id(F)
    b^8 = Id(F)
    (b * a^-1 * b)^3 = Id(F)
    b * a^-1 * b^-1 * a^-1 * b^-1 * a^-1 * b * a^-1 * b * a^-1 = Id(F)
    b^-1 * a^-2 * b^-1 * a^-2 * b^-1 * a^-1 * b^-2 * a^-1 = Id(F)
> phi(a) eq G.1 and phi(b) eq G.2;
true
```

57.6 Basic Operations

57.6.1 Accessing Group Information

The functions in this group provide access to basic information stored for a group G.

`G . i`

The i-th defining generator for G, if $i > 0$. If $i < 0$, then the inverse of the $-i$-th defining generator is returned. `G.0` is equivalent to `Identity(G)`.

`Generators(G)`

A set containing the defining generators for G.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for G.

`Generic(G)`

Given a group G in the category GrpPerm or GrpMat, return the generic group containing G, i.e., the largest group in which G is naturally embedded. The precise definition of generic group depends upon the category to which G belongs.

`Parent(g)`

The parent group G for the group element g.

Example H57E13

The Suzuki simple group $G = \mathrm{Sz}(8)$ is constructed. Its generic group is $\mathrm{GL}(4, K)$, where K is the finite field with 8 elements. The field K is constructed first, so that its generator may be given the printname z. Then the three generators of G are printed, in the standard order of indexing.

```
> K<z> := GF(2, 3);
> G := SuzukiGroup(8);
> Generic(G);
GL(4, GF(2, 3))
> Ngens(G);
3
> for i in [1..3] do
>    print "generator", i, G.i;
>    print "order", Order(G.i), "\r";
> end for;
generator 1
[  0   0   0   1]
[  0   0   1   0]
[  0   1   0   0]
[  1   0   0   0]
order 2

generator 2
```

```
[z^2   0   0   0]
[  0 z^6   0   0]
[  0   0   z   0]
[  0   0   0 z^5]
order 7

generator 3
[  1   0   0   0]
[z^2   1   0   0]
[  0   z   1   0]
[z^5 z^3 z^2   1]
order 4
```

`Orbit(G, M, x)`

Given a finitely generated group G that acts on the parent structure of x through the map (or user defined function) M, compute the orbit of x under G. Thus, for every generator g of G, $M(g)$ must return a function that can be applied to x or any other element in the parent of x.

If the orbit is infinite, this process will eventually run out of memory.

`OrbitClosure(G, M, S)`

Given a finitely generated group G acting on the universe of S through the map or user defined function M, compute the smallest subset T containing S that is G-invariant. Thus, for every generator g of G, $M(g)$ must return a function that can be applied to an arbitrary element in the universe of S.

If the orbit closure is infinite, this process will eventually run out of memory.

57.7 Operations on the Set of Elements

57.7.1 Order and Index Functions

`Order(G)`

`#G`

The order of the group G as an integer. If the order is not currently known, it will be computed. Computing the order may fail for groups in the category GrpFP; cf. Chapter 70.

`FactoredOrder(G)`

The order of the finite group G returned as a factored integer. The factorization is returned in the form of a sequence Q which is defined as follows: If $\#G = p_1^{e_1} \dots p_n^{e_n}, \quad e_i > 0$, then Q will be the integer sequence $[\langle p_1, e_1\rangle, \dots, \langle p_n, e_n\rangle]$. If the orders of G is not known, it will be computed. Computing the order may fail for groups in the category GrpFP; cf. Chapter 70.

`Index(G, H)`

The index of the subgroup H in the group G. The index is returned as an integer. Computing the index may fail for groups in the category GrpFP; cf. Chapter 70.

`FactoredIndex(G, H)`

The index of the subgroup H in the group G. H must have finite index in G. The index is returned as a factored integer. The format is the same as for `FactoredOrder`. Computing the index may fail for groups in the category GrpFP; cf. Chapter 70.

Example H57E14

Exploration of the order and index functions for a finitely presented group and its subgroup:

```
> Q<s,t,u>, h := Group< s, t, u |
>     t^2, u^17, s^2 = t^s = t, u^s = u^16, u^t = u >;
> Order(Q);
68
> FactoredOrder(Q);
[ <2, 2>, <17, 1> ]
> S := sub< Q | t*s^2, u^4 >;
> Index(Q, S);
4
> #S;
17
```

57.7.2 Membership and Equality

`g in G`

Given a group element g and a group G, return `true` if g is an element of G, `false` otherwise.

`g notin G`

Given a group element g and a group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G`

Given a group G and a set S of group elements belonging to a group H, where G and H belong the same generic group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G`

Given a group G and a set S of group elements belonging to a group H, where G and H belong the same generic group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G`

Given groups G and H belonging to the same generic group, return `true` if H is a subgroup of G, `false` otherwise.

`H notsubset G`

Given groups G and H belonging to the same generic group, return `true` if H is not a subgroup of G, `false` otherwise.

`H eq G`

Given groups G and H belonging to the same generic group, return `true` if G and H are the same group, `false` otherwise.

`H ne G`

Given groups G and H belonging to the same generic group, return `true` if G and H are distinct groups, `false` otherwise.

57.7.3 Set Operations

`NumberingMap(G)`

Given a finite group G in the category GrpPerm, GrpMat, GrpPC or GrpAb, return a bijective mapping from the group G onto the set of integers $\{1 \ldots |G|\}$. The actual mapping depends upon the particular representation of G.

`Representative(G)`

`Rep(G)`

An element chosen from the group G.

Example H57E15

We use the function `NumberingMap` to construct the multiplication table for the dihedral group of order 12.

```
> G := DihedralGroup(6);
> f := NumberingMap(G);
> [ [ f(x*y) : y in G ] : x in G ];
[
    [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ],
    [ 2, 3, 4, 5, 6, 1, 12, 7, 8, 9, 10, 11 ],
    [ 3, 4, 5, 6, 1, 2, 11, 12, 7, 8, 9, 10 ],
    [ 4, 5, 6, 1, 2, 3, 10, 11, 12, 7, 8, 9 ],
    [ 5, 6, 1, 2, 3, 4, 9, 10, 11, 12, 7, 8 ],
    [ 6, 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 7 ],
    [ 7, 8, 9, 10, 11, 12, 1, 2, 3, 4, 5, 6 ],
    [ 8, 9, 10, 11, 12, 7, 6, 1, 2, 3, 4, 5 ],
    [ 9, 10, 11, 12, 7, 8, 5, 6, 1, 2, 3, 4 ],
    [ 10, 11, 12, 7, 8, 9, 4, 5, 6, 1, 2, 3 ],
```

```
    [ 11, 12, 7, 8, 9, 10, 3, 4, 5, 6, 1, 2 ],
    [ 12, 7, 8, 9, 10, 11, 2, 3, 4, 5, 6, 1 ]
]
```

57.7.4 Random Elements

`Random(G: parameters)`

`Short` BoolElt *Default :* `false`

A randomly chosen element for the group G. If a representation of the carrier set of G has already been created, then the element chosen will be genuinely random. If such a representation has not yet been created, then the *random* element is chosen by multiplying out a *random* word in the generators. Since it is not usually practical to choose words long enough to properly sample the elements of G, the element returned will usually be biased. The boolean-valued parameter `Short` is used in this situation to indicate that a short word will suffice. Thus, if `Random` is invoked with `Short` assigned the value `true` then the element is constructed using a short word.

Example H57E16

We illustrate the use of the function `Random` using the wreath product of the symmetric group of degree 4 and the cyclic group of order 6.

```
> G := WreathProduct(Sym(4), CyclicGroup(6));
> G;
Permutation group G acting on a set of cardinality 24
    (1, 5, 9, 13, 17, 21)(2, 6, 10, 14, 18, 22) (3, 7, 11, 15, 19, 23)
        (4, 8, 12, 16, 20, 24)
    (1, 2, 3, 4)
    (1, 2)
> Order(G);
1146617856
> Random(G);
(1, 17, 12, 4, 18, 10, 3, 20, 9, 2, 19, 11)(5, 22, 13, 6, 21, 15)
    (7, 24, 16)(8, 23, 14)
// We display the cycle structures of 10 random elements of G
> R := [ CycleStructure(Random(G)) : i in [1..10] ];
> R;
[
    [ <6, 1>, <3, 6> ],
    [ <9, 1>, <6, 2>, <3, 1> ],
    [ <9, 2>, <3, 2> ],
    [ <12, 1>, <9, 1>, <3, 1> ],
    [ <18, 1>, <6, 1> ],
    [ <18, 1>, <6, 1> ],
    [ <12, 1>, <6, 2> ],
```

```
    [ <6, 3>, <2, 3> ],
    [ <6, 1>, <4, 3>, <2, 3> ],
    [ <6, 3>, <3, 2> ]
]
```

`RandomProcess(G)`

`RandomProcessWithWords(G)`

`RandomProcessWithValues(G, Q)`

`RandomProcessWithWordsAndValues(G, Q)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 50
`WordGroup`	GrpSLP	*Default :*

Create a process to generate randomly chosen elements from the group G. The process uses a variant of the product-replacement method similar to the *Rattle* method of [LGM02]. The generating set stored in the process has N elements, where N is the maximum of the specified value for `Slots` and `Ngens`$(G)+1$. Initially they are the generators of G repeated as necessary and the accumulator is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call returns the current value of the accumulator, modifying the generating set as for product-replacement, and modifying the accumulator by multiplying by the new member of the generating set. Setting `Scramble := m` causes m such operations to be performed before the process is returned.

The functions with words and values create a process that returns extra group elements for each call. A process created with words returns, as second return value for each call to `Random(P)`, the `GrpSLPElt` describing the random element returned as a straight-line program in the group generators. The parameter `WordGroup` may be used to specify a particular group for the words to be elements of.

A process created with values takes as input a sequence of group elements Q giving the values assigned to each generator of G. The second value returned is the result of computing in parallel with these values as with the generators of G. In particular, if the elements of Q are homomorphic images of the generators of G, then the second return value from `Random(P)` will be the image of the first under this homomorphism.

A process created with words and values does all of the above, with the three return values of `Random(P)` being a random element of G, the straight-line program and the value.

The use of this function on a finitely-presented group G is not recommended. Since there is no reduction of words, the random elements generated may be extremely long.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the finite group G, construct a random element of G by forming a random product over the expanded generating set constructed when the process was created. For large permutation or matrix groups for which a BSGS is not known, this function should be used in preference to `Random(G)`.

If the process was created with words or values then there will be second and third return values as described under `RandomProcess` above.

`InitialiseProspector(G:`*parameters*`)`

`InitialiseProspector(G:`*parameters*`)`

Initialise a product-replacement prospector for the given group. This is an extension of the product-replacement algorithm that searches for an element $x \in G$ such that some predicate is true for this element. The prospector aims to find elements x so that the corresponding straight-line program for x is short. Statistical tests and various heuristics are used to achieve this.

Generally, output from product-replacement with short straight-line programs is not very random. Prospector aims to run product-replacement until the output looks random, then start a search for the element wanted. At all times, if the output starts to look non-random, or word lengths grow too far without finding an element, the prospector may return to a previous state of product replacement and try again, searching in a different direction. The statistical tests are used to make concrete the notion of "looks random". For permutation groups the test used is based on number of cycles of the element. For matrix groups the test statistic is the number of factors of the characteristic polynomial.

`Prospector(G, f:`*parameters*`)`

Run an initialised prospector for group G to find $x \in G$ such that $f(x)$ is true. The first return value gives the success or failure of the search. If this value is true, then the second and third return values are x and a straight-line program giving x in terms of the group generators. The parameter `MaxTries` may be set to limit the number of random selections made by the prospector when attempting to find x.

Example H57E17

We find a random pair of generators for the symmetric group of degree 300 and use a random process to find an element which is a 300-cycle as a straight-line program in the generators. The proportion of such elements is 1 in 300, so we expect the program to have length 600.

```
> SetSeed(1);
> S := Sym(300);
> repeat G := sub<S|Random(S),Random(S)>;
> until IsSymmetric(G);
> P := RandomProcessWithWords(G);
> repeat x,w := Random(P);
```

```
> until CycleStructure(x) eq [<300,1>];
> #w;
936
```

Note that the group S, known to be a symmetric group, has an efficient uniform random element generator available as above. The word length was somewhat longer than the expected value. Now we set up a prospector and use it to search for an element of the same cycle structure. The defining word for the new element should be shorter than the expected 600.

```
> InitialiseProspector(G);
true
> test := func<x|CycleStructure(x) eq [<300,1>]>;
> a,x,w := Prospector(G, test);
> a;
true
> #w;
206
> Evaluate(w, [G.1,G.2]) eq x;
true
```

57.7.5 Action on a Coset Space

`CosetTable(G, H)`

The (right) coset table for G over subgroup H relative to its defining generators.

`#CosetTable(G, f)`

The coset table for G corresponding to the permutation representation f of G, where f is a homomorphism of G onto a transitive permutation group.

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a group G and a subgroup H of G, this function returns

(a) An indexed set of elements T of G forming a right transversal for G over H;

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and $g \in G$, ϕ is defined by $\phi(g) = t_i$, where $g \in Ht_i$.

`CosetAction(G, H)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced group L;

(c) The kernel K of the action (a subgroup of G).

Note that G may be any type of group. If G is a finitely presented group, then K may be returned undefined.

`CosetImage(G, H)`

Given a subgroup H of the group G, construct the image L of G given by the action of G on the set of (right) cosets of H in G.

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of the action of G on the set of (right) cosets of H in G.

57.8 Standard Subgroup Constructions

Some functions described in this section may not exist or may have restrictions for some categories of groups. Details can be found in the chapters on the individual categories.

`H ^ g`

`Conjugate(H, g)`

Construct the conjugate $g^{-1}Hg$ of the group H by the element g. The group H and the element g must belong to the same generic group.

`H meet K`

Given groups H and K which belong to the same symmetric group, construct the intersection of H and K.

`CommutatorSubgroup(G, H, K)`

`CommutatorSubgroup(H, K)`

Given groups H and K, both subgroups of the group G, construct the commutator subgroup of H and K in the group G. If K is a subgroup of H, then the group G may be omitted.

`Centralizer(G, g)`

`Centraliser(G, g)`

Construct the centralizer of the element g in the group G.

`Centralizer(G, H)`
`Centraliser(G, H)`

Construct the centralizer of the group H in the group G.

`Core(G, H)`

Given a subgroup H of the group G, construct the maximal normal subgroup of G that is contained in the subgroup H.

`H ^ G`
`NormalClosure(G, H)`

Given a subgroup H of the group G, construct the normal closure of H in G.

`Normalizer(G, H)`
`Normaliser(G, H)`

Given a subgroup H of the group G, construct the normalizer of H in G.

`pCore(G, p)`

Given a group G and a prime p dividing the order of G, construct the maximal normal p-subgroup of G.

`SylowSubgroup(G, p)`
`Sylow(G, p)`

Given a group G and a prime p, construct a Sylow p-subgroup of G.

57.8.1 Abstract Group Predicates

Some functions described in this section may not exist or may have restrictions for some categories of groups. Details can be found in the chapters on the individual categories.

`IsAbelian(G)`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsCentral(G, H)`

Return `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function returns a second value if the elements are conjugate: an element k which conjugates g into h.

`IsConjugate(G, H, K)`

Given a group G and subgroups H and K belonging to G, return the value `true` if H and K are conjugate in G. The function returns a second value if the subgroups are conjugate: an element z which conjugates H into K.

`IsExtraSpecial(G)`

Given a group G is a p-group G, return `true` if G is extra-special, `false` otherwise.

`IsMaximal(G, H)`

Returns `true` if the subgroup H of the group G is a maximal subgroup of G. This function is evaluated by constructing the permutation representation of G on the cosets of H and testing this representation for primitivity. For this reason, the use of `IsMaximal` should be avoided if the index of H in G exceeds a one hundred thousand.

`IsNilpotent(G)`

Return `true` if the group G is nilpotent, `false` otherwise.

`IsNormal(G, H)`

Return `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

`IsPerfect(G)`

Return `true` if the group G is perfect, `false` otherwise.

`IsSelfNormalizing(G, H)`
`IsSelfNormalising(G, H)`

Return `true` if the subgroup H of the group G is self-normalizing in G, `false` otherwise.

`IsSimple(G)`

Return `true` if the group G is simple, `false` otherwise.

`IsSoluble(G)`
`IsSolvable(G)`

Return `true` if the group G is soluble, `false` otherwise.

`IsSpecial(G)`

Given a p-group G, return `true` if G is special, `false` otherwise.

`IsSubnormal(G, H)`

Return `true` if the subgroup H of the group G is subnormal in G, `false` otherwise.

`IsTrivial(G)`

Return `true` if G is trivial, `false` otherwise.

57.9 Characteristic Subgroups and Normal Structure

57.9.1 Characteristic Subgroups and Subgroup Series

Some functions described in this section may not exist or may have restrictions for some categories of groups. Details can be found in the chapters on the individual categories.

`Centre(G)`
`Center(G)`

Construct the centre of the group G.

`Hypercentre(G)`
`Hypercenter(G)`

Construct the hypercentre of the group G (the stationary term of the upper central series).

`DerivedLength(G)`

The derived length of G. If G is non-soluble, the function returns the number of terms in the series terminating with the soluble residual.

`DerivedSeries(G)`

The derived series of the group G. The series is returned as a sequence of subgroups.

`DerivedSubgroup(G)`
`DerivedGroup(G)`

The derived subgroup of the group G.

`FittingSubgroup(G)`

The Fitting subgroup of the group G.

`FrattiniSubgroup(G)`

Given a group G that is a p-group, return the Frattini subgroup.

`JenningsSeries(G)`

Given a p-group G, return the Jennings series for G. The series is returned as a sequence of subgroups.

`LowerCentralSeries(G)`

The lower central series of G. The series is returned as a sequence of subgroups.

`NilpotencyClass(G)`

The nilpotency class of the group G. If the group is not nilpotent, the value -1 is returned.

`H ^ G`

`NormalClosure(G, H)`

The normal closure of the subgroup H of group G.

`NormalLattice(G)`

The normal subgroups of G arranged as a lattice.

`NormalSubgroups(G)`

The normal subgroups of G.

`pCentralSeries(G, p)`

Given a soluble group G, and a prime p dividing $|G|$, return the lower p-central series for G. The series is returned as a sequence of subgroups.

`Radical(G)`

The maximal normal solvable subgroup of the group G.

`SolubleResidual(G)`

`SolvableResidual(G)`

The solvable residual of the group G.

`SubnormalSeries(G, H)`

Given a group G and a subnormal subgroup H of G, return a sequence of subgroups commencing with G and terminating with H, such that each subgroup is normal in the previous one. If H is not subnormal in G, the empty sequence is returned.

`UpperCentralSeries(G)`

The upper central series of G. The series is returned as a sequence of subgroups commencing with the trivial subgroup. Since the algorithm used requires the conjugacy classes of G, this function is much more restricted in its range of application than `DerivedSeries` and `LowerCentralSeries`.

f	Family name
1	A(d, q)
2	B(d, q)
3	C(d, q)
4	D(d, q)
5	G$(2, q)$
6	F$(4, q)$
7	E$(6, q)$
8	E$(7, q)$
9	E$(8, q)$
10	2A(d, q)
11	2B$(2, q)$
12	2D(d, q)
13	3D$(4, q)$
14	2G$(2, q)$
15	2F$(4, q)$
16	2E$(6, q)$
17	Alternating(d)
18	Sporadic group — see Table 2.
19	Cyclic(q)

Table 1: Family numbers and names

d	Group name
1	M_{11}
2	M_{12}
3	M_{22}
4	M_{23}
5	M_{24}
6	J_1
7	HS
8	J_2
9	MCL
10	SUZ
11	J_3
12	CO_1
13	CO_2
14	CO_3
15	HE
16	$M(22)$
17	$M(23)$
18	$M(24)$
19	LY
20	RU
21	ON
22	TH
23	HA
24	BM
25	M
26	J_4

Table 2: Sporadic groups

57.9.2 The Abstract Structure of a Group

`CompositionFactors(G)`

Given a finite group G in the category GrpPerm, GrpMat, GrpPC of GrpAb, return a sequence S of tuples that represent the composition factors of G, ordered according to some composition series of G. Each tuple is a triple of integers f, d, q that defines the isomorphism type of the corresponding composition factor. A triple $< f, d, q >$ describes a simple group as follows. The integer f defines the family to which the group belongs, and d and q are the parameters of the family. The length of the sequence S is the number of composition factors of G. The families are listed in Tables 1 and 2 on page 1495.

`AbelianInvariants(G)`

`Invariants(G)`

Given an abelian group G in the category GrpPerm, GrpMat, GrpPC of GrpAb, return a sequence Q containing the types of each p-primary component of G.

`AbelianBasis(G)`

Given an abelian group G in the category GrpPerm, GrpPC of GrpAb, return sequences B and I where I contains the types of each p-primary component of G and B contains corresponding elements of G which have the order given and generate G.

57.10 Conjugacy Classes of Elements

There are three aspects of the conjugacy problem for elements: determining whether two elements are conjugate in a group G, determining a set of representatives for the conjugacy classes of elements of G, and listing all the elements in a particular class of G. The algorithms used depend on the category of G. If G is in category GrpPerm or GrpMat, conjugacy is determined by means of a backtrack search over base-images. If G is in category GrpPC, testing conjugacy is performed by transforming each element into a standard representative of its conjugacy class by an orbit-stabilizer process that works down a sequence of increasing quotients of G. Conjugacy testing for a group G in category GrpGPC is only possible if G is nilpotent. In this case, an algorithm by E. Lo [Lo98] is used.

Some functions described in this section may not exist or may have restrictions for some categories of groups. Details can be found in the chapters on the individual categories.

`Class(H, x)`

`Conjugates(H, x)`

Given a group H and an element x belonging to a group K such that H and K are subgroups of the same symmetric group, this function returns the set of conjugates of x under the action of H. If $H = K$, the function returns the conjugacy class of x in H.

`ClassMap(G:` *parameters*`)`

Given a group G, construct the conjugacy classes and the class map f for G. For any element x of G, $f(x)$ will be the index of the conjugacy class of x in the sequence returned by the `Classes` function.

If G is a permutation group, the construction may be controlled using the parameters `Orbits`, `WeakLimit` and `StrongLimit`. If the parameter `Orbits` is set `true`, the classes are computed as orbits of elements under conjugation and the class map is stored as a list of images of the elements of G (a list of length $|G|$). This option gives fast evaluation of the class map but is practical only in the case of very small groups. With `Orbits := false`, `WeakLimit` and `StrongLimit` are used to control the random classes algorithm (see function `Classes`).

`ConjugacyClasses(G: parameters)`

`Classes(G: parameters)`

`WeakLimit`	RngIntElt	*Default :* 200
`StrongLimit`	RngIntElt	*Default :* 500
`Reps`	[GrpFinElt]	*Default :*
`Al`	MonStg	*Default :*

Construct a set of representatives for the conjugacy classes of G. The classes are returned as a sequence of triples containing the element order, the class length and a representative element for the class. The parameters `Reps` and `Al` enable the user to select the algorithm that is to be used when G is a permutation or matrix group.
`Reps := Q`: Create the classes of G by assuming that Q is a sequence of class representatives for G. The orders and lengths of the classes will be computed and checked for consistency.
`Al := "Action"`: Create the classes of G by computing the orbits of the set of elements of G under the action of conjugation. This option is only feasible for small groups.
`Al := "Random"`: Construct the conjugacy classes of elements for a permutation or matrix group G using an algorithm that searches for representatives of all conjugacy classes of G by examining a random selection of group elements and their powers. The behaviour of this algorithm is controlled by two associated optional parameters `WeakLimit` and `StrongLimit`, whose values are positive integers n_1 and n_2, say. Before describing the effect of these parameters, some definitions are needed: A mapping $f : G \to I$ is called a *class invariant* if $f(g) = f(g^h)$ for all $g, h \in G$. For permutation groups, the cycle structure of g is a readily computable class invariant. In matrix groups, the primary invariant factors are used where possible, or the characteristic or minimal polynomials otherwise. Two elements g and h are said to be *weakly conjugate* with respect to the class invariant f if $f(g) = f(h)$. By definition, conjugacy implies weak conjugacy, but the converse is false. The random algorithm first examines n_1 random elements and their powers, using a test for weak conjugacy. It then proceeds to examine a further n_2 random elements and their powers, using a test for ordinary conjugacy. The idea behind this strategy is that the algorithm should attempt to find as many classes as possible using the very cheap test for weak conjugacy, before employing the more expensive ordinary conjugacy test to recognize the remaining classes.
`Al := "Extend"`: Construct the conjugacy classes of G by first computing classes in a quotient G/N and then extending these classes to successively larger quotients G/H until the classes for $G/1$ are known. More precisely, a maximal series of subgroups $1 = G_0 < G_1 < \cdots < G_r = R < G$ is computed such that R is the (solvable) radical of G and G_{i+1}/G_i is elementary abelian. A representation of G/R is computed using an algorithm of Derek Holt and its classes computed and represented as elements of G. To extend to the next larger quotient, a group is computed from each class which acts on the transversal. Each distinct orbit in

that action gives rise to a new class. To compute the classes of G/R, the default algorithm (excluding the extension method) is used. The same set of parameters is passed on, so you can control limits in the random classes method if it is chosen. The limitations of the algorithm are that R may be trivial, in which case nothing is done except to call a different algorithm, or one or more of the sections may be so large as to prohibit computing the action on the transversal. This algorithm is currently only available for permutation groups.

`ClassRepresentative(G, x)`

Given a group G for which the conjugacy classes are known and an element x of G, return the designated representative for the conjugacy class of G containing x.

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function returns a second value if the elements are conjugate: an element k which conjugates g into h.

`IsConjugate(G, H, K)`

Given a group G and subgroups H and K belonging to G, return the value `true` if G and H are conjugate in G. The function returns a second value if the subgroups are conjugate: an element z which conjugates H into K.

`Exponent(G)`

The exponent of the group G.

`NumberOfClasses(G)`
`Nclasses(G)`

The number of conjugacy classes of elements for the group G.

`PowerMap(G)`

Given a group G, construct the power map for G. Suppose that the order of G is m and that G has r conjugacy classes. When the classes are determined by MAGMA, they are numbered from 1 to r. Let C be the set of class indices $\{1, \ldots, r\}$. The *power map* f for G is the mapping

$$f : C \times \mathbf{Z} \to C$$

where the value of $f(i, j)$ for $i \in C$ and $j \in \mathbf{Z}$ is the number of the class which contains x_i^j, where x_i is a representative of the i-th conjugacy class.

Example H57E18

The conjugacy classes of the Mathieu group M_{11} can be constructed as follows:

```
> M11 := sub<Sym(11) | (1,10)(2,8)(3,11)(5,7), (1,4,7,6)(2,11,10,9)>;
> Classes(M11);
Conjugacy Classes of group M11
------------------------------
[1]     Order 1       Length 1
        Rep Id(M11)

[2]     Order 2       Length 165
        Rep (3, 10)(4, 9)(5, 6)(8, 11)

[3]     Order 3       Length 440
        Rep (1, 2, 4)(3, 5, 10)(6, 8, 11)

[4]     Order 4       Length 990
        Rep (3, 6, 10, 5)(4, 8, 9, 11)

[5]     Order 5       Length 1584
        Rep (1, 3, 6, 2, 8)(4, 7, 10, 9, 11)

[6]     Order 6       Length 1320
        Rep (1, 11, 2, 6, 4, 8)(3, 10, 5)(7, 9)

[7]     Order 8       Length 990
        Rep (1, 4, 5, 6, 2, 7, 11, 10)(8, 9)

[8]     Order 8       Length 990
        Rep (1, 7, 5, 10, 2, 4, 11, 6)(8, 9)

[9]     Order 11      Length 720
        Rep (1, 11, 9, 10, 4, 3, 7, 2, 6, 5, 8)

[10]    Order 11      Length 720
        Rep (1, 9, 4, 7, 6, 8, 11, 10, 3, 2, 5)
```

57.11 Conjugacy Classes of Subgroups

MAGMA contains a new algorithm for computing representatives of the conjugacy classes of subgroups. Let R denote the maximal normal soluble subgroup of the finite group G. The algorithm first constructs representatives for the conjugacy classes of subgroups of $Q = G/R$, and then successively extends these to larger and larger quotients of G until G itself is reached. If G is soluble, then Q is trivial and so its subgroups are known. If G is non-soluble, we attempt to locate the quotient in a database of groups with trivial Fitting subgroup. This database contains all such groups of order up to 216 000, and all such which are perfect of order up to 1 000 000. If Q is found then either all its subgroups, or its maximal subgroups are read from the database. (In some cases only the maximal subgroups are stored.) If Q is not found then we attempt to find the maximal subgroups of Q using a method of Derek Holt. For this to succeed all simple factors of the socle of Q must be found in a second database which currently contains all simple groups of order less than 1.6×10^7, as well as M_{24}, HS, J_3, McL, $Sz(32)$ and $L_6(2)$. There are also special routines to handle numerous other groups. These include: A_n for $n \leq 999$, $L_2(q)$, $L_3(q)$, $L_4(q)$ and $L_5(q)$ for all q, $S_4(q)$, $U_3(q)$ and $U_4(q)$ for all q, $L_d(2)$ for $d \leq 14$, and the following groups: $L_6(3)$, $L_7(3)$, $U_6(2)$, $S_8(2)$, $S_{10}(2)$, $O_8^{\pm}(2)$, $O_{10}^{\pm}(2)$, $S_6(3)$, $O_7(3)$, $O_8^-(3)$, $G_2(4)$, $G_2(5)$, ${}^3D_4(2)$, ${}^2F_4(2)'$, Co_2, Co_3, He, Fi_{22}.

If we have only maximal subgroups of Q, and more are required, we apply the algorithm recursively to the maximal subgroups to determine all subgroups of Q. This may take some time.

57.11.1 Conjugacy Classes of Subgroups

In this section we describe the functions that allow a user to create representatives of the conjugacy classes of subgroups, possibly subject to conditions. The main function, `Subgroups`, finds representatives for conjugacy classes of subgroups subject to certain user-supplied conditions on the order. The alternative functions `ElementaryAbelianSubgroups` and `AbelianSubgroups`, `CyclicSubgroups`, `NilpotentSubgroups`, `SolubleSubgroups`, `PerfectSubgroups`, `NonsolvableSubgroups`, `SimpleSubgroups` and `RegularSubgroups` allow the user to construct particular classes of subgroups.

Most of the features described in this section are currently only available for groups in the category `GrpPerm`, `GrpMat` or `GrpPC`.

`SubgroupClasses(G: parameters)`

`Subgroups(G: parameters)`

Representatives for the conjugacy classes of subgroups for the group G. The subgroups are returned as a sequence of records where the i-th record contains:

(a) A representative subgroup H for the i-th conjugacy class (field name `subgroup`).

(b) The order of the subgroup (field name `order`).

(c) The number of subgroups in the class (field name `length`).

(d) [Optionally] A presentation for H (field name `presentation`).

`Presentation` BOOLELT *Default :* `false`

`Presentation := true`: In the case in which G is a permutation group, construct a presentation for each subgroup.

`OrderEqual` RngIntElt *Default :*

`OrderEqual := n`: Only construct subgroups having order equal to n.

`OrderDividing` RngIntElt *Default :*

`OrderDividing := n`: Only construct subgroups having order dividing n.

`IsNormal` BoolElt *Default :* `false`

`IsNormal := true`: Only construct normal subgroups.

`IsRegular` BoolElt *Default :* `false`

`IsRegular := true`: In the case in which G is a permutation group, only construct regular subgroups.

`LayerSizes` SeqEnum *Default :* see below

`LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]` is equivalent to the default. When constructing an Elementary Abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1.

`ElementaryAbelianSubgroups(G: parameters)`

Representatives for the conjugacy classes of elementary abelian subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`AbelianSubgroups(G: parameters)`

Representatives for the conjugacy classes of abelian subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`CyclicSubgroups(G: parameters)`

Representatives for the conjugacy classes of cyclic subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`NilpotentSubgroups(G: parameters)`

Representatives for the conjugacy classes of nilpotent subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`SolubleSubgroups(G: parameters)`

`SolvableSubgroups(G: parameters)`

Representatives for the conjugacy classes of solvable subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`NonsolvableSubgroups(G: parameters)`

Representatives for the conjugacy classes of nonsolvable subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`PerfectSubgroups(G: parameters)`

Representatives for the conjugacy classes of perfect subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`SimpleSubgroups(G: parameters)`

Representatives for the conjugacy classes of non-abelian simple subgroups for the group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`RegularSubgroups(G: parameters)`

Representatives for the conjugacy classes of regular subgroups for the permutation group G. The subgroups are returned as a sequence of records having the same format as `Subgroups`. The optional parameters are also the same as for `Subgroups`.

`SetVerbose("SubgroupLattice", i)`

Turn on verbose printing for the subgroup algorithm. The level i can be 2 for maximal printing or 1 for moderate printing. The algorithm works down an elementary abelian series of the group and at each level, the possible extensions of each subgroup are listed.

`Class(G, H)`

`Conjugates(G, H)`

The G-conjugacy class of subgroups containing the group H.

Example H57E19

We construct the conjugacy classes of subgroups for the dihedral group of order 12.

```
> G := DihedralGroup(6);
> S := Subgroups(G);
> S;
Conjugacy classes of subgroups
------------------------------
[ 1]    Order 1            Length 1
        Permutation group acting on a set of cardinality 6
        Order = 1
          Id($)
[ 2]    Order 2            Length 3
        Permutation group acting on a set of cardinality 6
          (2, 6)(3, 5)
[ 3]    Order 2            Length 3
        Permutation group acting on a set of cardinality 6
          (1, 4)(2, 3)(5, 6)
[ 4]    Order 2            Length 1
        Permutation group acting on a set of cardinality 6
          (1, 4)(2, 5)(3, 6)
[ 5]    Order 3            Length 1
        Permutation group acting on a set of cardinality 6
          (1, 5, 3)(2, 6, 4)
[ 6]    Order 4            Length 3
        Permutation group acting on a set of cardinality 6
          (2, 6)(3, 5)
          (1, 4)(2, 5)(3, 6)
[ 7]    Order 6            Length 1
        Permutation group acting on a set of cardinality 6
          (1, 5, 3)(2, 6, 4)
          (1, 4)(2, 5)(3, 6)
[ 8]    Order 6            Length 1
        Permutation group acting on a set of cardinality 6
          (2, 6)(3, 5)
          (1, 5, 3)(2, 6, 4)
[ 9]    Order 6            Length 1
        Permutation group acting on a set of cardinality 6
          (1, 4)(2, 3)(5, 6)
          (1, 5, 3)(2, 6, 4)
[10]    Order 12           Length 1
        Permutation group acting on a set of cardinality 6
          (2, 6)(3, 5)
          (1, 5, 3)(2, 6, 4)
          (1, 4)(2, 5)(3, 6)
> // We extract the representative subgroup for class 7
> h := S[7]`subgroup;
> h;
```

```
Permutation group h acting on a set of cardinality 6
    (1, 3, 5)(2, 4, 6)
    (1, 4)(2, 5)(3, 6)
```

57.11.2 The Poset of Subgroup Classes

In addition to finding representatives for conjugacy classes of subgroups, MAGMA allows the user to create the poset L of subgroup classes. The elements of the poset correspond to the conjugacy classes of subgroups. Two lattice elements a and b are joined by an edge if either some subgroup of the conjugacy class a is a maximal subgroup of some subgroup of conjugacy class b or vice-versa. The elements of L are called *subgroup-poset elements* and are numbered from 1 to n, where n is the cardinality of L. Various functions allow the user to identify maximal subgroups, normalizers, centralizers and other relatives in the lattice. Given an element e of L, one can easily create the subgroup H of G corresponding to e and one can also create the element of L corresponding to a subgroup of G.

The features described in this section are currently only available for groups in the category GrpPerm or GrpPC.

57.11.2.1 Creating the Poset of Subgroup Classes

`SubgroupLattice(G)`

Create the poset L of subgroup classes of G.

`Properties` BOOLELT *Default :* `false`

`Properties := true`: As the subgroup classes are put into the poset, record their abstract type, i.e., elementary abelian, abelian, nilpotent, soluble, simple or perfect.

`Centralizers` BOOLELT *Default :* `false`

`Centralizers := true`: As each subgroup class e is put into the poset, record the class in which the centralizers of the subgroups of e lie.

`Normalizers` BOOLELT *Default :* `false`

`Normalizers := true`: As each subgroup class e is put into the poset, record the class in which the normalizers of the subgroups of e lie.

Example H57E20

We create the subgroup poset for the group ASL$(2, 3)$.

```
> G := ASL(2, 3);
> L := SubgroupLattice(G : Properties := true, Normalizers:= true,
>                            Centralizers:= true);
> L;
Partially ordered set of subgroup classes
----------------------------------------

[ 1]  Order 1 Length 1 C = [20] N = [20]
```

```
      Maximal Subgroups:
---
[ 2]  Order 2 Length 9 Cyclic. C = [16] N = [16]
      Maximal Subgroups: 1
[ 3]  Order 3 Length 12 Cyclic. C = [14] N = [14]
      Maximal Subgroups: 1
[ 4]  Order 3 Length 24 Cyclic. C = [10] N = [10]
      Maximal Subgroups: 1
[ 5]  Order 3 Length 4 Cyclic. C = [15] N = [18]
      Maximal Subgroups: 1
---
[ 6]  Order 4 Length 27 Cyclic. C = [6] N = [12]
      Maximal Subgroups: 2
[ 7]  Order 6 Length 12 Soluble. C = [3] N = [14]
      Maximal Subgroups: 2 5
[ 8]  Order 6 Length 36 Cyclic. C = [8] N = [8]
      Maximal Subgroups: 2 3
[ 9]  Order 9 Length 4 Elementary Abelian. C = [9] N = [18]
      Maximal Subgroups: 3 5
[10]  Order 9 Length 8 Elementary Abelian. C = [10] N = [15]
      Maximal Subgroups: 4 5
[11]  Order 9 Length 1 Elementary Abelian. C = [11] N = [20]
      Maximal Subgroups: 5
---
[12]  Order 8 Length 9 Nilpotent. C = [2] N = [16]
      Maximal Subgroups: 6
[13]  Order 18 Length 1 Soluble. C = [1] N = [20]
      Maximal Subgroups: 7 11
[14]  Order 18 Length 12 Soluble. C = [3] N = [14]
      Maximal Subgroups: 7 8 9
[15]  Order 27 Length 4 Nilpotent. C = [5] N = [18]
      Maximal Subgroups: 9 10 11
---
[16]  Order 24 Length 9 Soluble. C = [2] N = [16]
      Maximal Subgroups: 8 12
[17]  Order 36 Length 3 Soluble. C = [1] N = [19]
      Maximal Subgroups: 6 13
[18]  Order 54 Length 4 Soluble. C = [1] N = [18]
      Maximal Subgroups: 13 14 15
---
[19]  Order 72 Length 1 Soluble. C = [1] N = [20]
      Maximal Subgroups: 12 17
---
[20]  Order 216 Length 1 Soluble. C = [1] N = [20]
      Maximal Subgroups: 16 18 19
```

57.11.2.2 Operations on Subgroup Class Posets

In the following, L is the poset of subgroup classes for a group G.

`#L`

The cardinality of L, i.e., the number of conjugacy classes of subgroups of G.

`L ! i`

Create the i-th element of the poset L. The elements of L are sorted so that classes i and j of groups whose orders o_i and o_j are the products of e_i and e_j prime numbers respectively will be ordered so that i comes before j is $e_i < e_j$ or $e_i = e_j$ and $o_i < o_j$.

`L ! H`

Create the element of the poset L corresponding to the subgroup H of the group G.

`Bottom(L)`

Create the bottom of the poset L, i.e., the element of L corresponding to the trivial subgroup of G. If the poset was created with restrictions on the type of subgroups constructed, the bottom of the poset may not be the trivial subgroup.

`Top(L)`

Create the top of the poset L, i.e., the element of L corresponding to G.

`Random(L)`

Create a random element of L.

Example H57E21

We create the subgroup lattice of $A\Gamma L(1, 8)$ and locate the Fitting subgroup in the lattice.

```
> G := AGammaL(1, 8);
> L := SubgroupLattice(G);
> L;

Subgroup Lattice
----------------

[ 1]  Order 1 Length 1
      Maximal Subgroups:
---
[ 2]  Order 2 Length 7
      Maximal Subgroups: 1
[ 3]  Order 3 Length 28
      Maximal Subgroups: 1
[ 4]  Order 7 Length 8
      Maximal Subgroups: 1
---
[ 5]  Order 4 Length 7
```

```
      Maximal Subgroups: 2
[ 6]  Order 6 Length 28
      Maximal Subgroups: 2 3
[ 7]  Order 21 Length 8
      Maximal Subgroups: 3 4
---
[ 8]  Order 8 Length 1
      Maximal Subgroups: 5
[ 9]  Order 12 Length 7
      Maximal Subgroups: 3 5
---
[10]  Order 24 Length 7
      Maximal Subgroups: 6 8 9
[11]  Order 56 Length 1
      Maximal Subgroups: 4 8
---
[12]  Order 168 Length 1
      Maximal Subgroups: 7 10 11

> F := FittingSubgroup(G);
> F;
Permutation group F acting on a set of cardinality 8
Order = 8 = 2^3
    (1, 2)(3, 6)(4, 8)(5, 7)
    (1, 6)(2, 3)(4, 7)(5, 8)
    (1, 5)(2, 7)(3, 4)(6, 8)
> L!F;
8
```

We now construct a chain from the bottom to the top of the lattice.

```
> H := Bottom(L);
> Chain := [H];
> while H ne Top(L) do
>     H := Representative(MinimalOvergroups(H));
>     Chain := Append(Chain, H);
> end while;
> Chain;
[ 1, 2, 5, 8, 10, 12 ]
```

57.11.2.3 Operations on Poset Elements

In the following, L is the poset of subgroups for a group G. Elements of L are identified with the integers $[1..\#L]$.

`IntegerRing() ! e`

The integer corresponding to poset element e.

`e eq f`

Returns `true` if and only if poset elements e and f are equal.

`e ge f`

Returns `true` if and only if poset element e contains poset element f.

`e ge f`

Returns `true` if and only if poset element e strictly contains poset element f.

`e le f`

`e subset f`

Returns `true` if and only if poset element e is contained in poset element f.

`e lt f`

Returns `true` if and only if poset element e is strictly contained in poset element f.

57.11.2.4 Class Information from a Conjugacy Class Poset

In the following, L is the poset of subgroups for a group G. Elements of L are identified with the integers $[1..\#L]$.

`Group(e)`

The subgroup of G that is the chosen class representative corresponding to the element e of the poset L.

`Centraliser(e, f)`

`Centralizer(e, f)`

Given poset elements e and f, return the poset element that corresponds to the class of subgroups that contains the centralizers of the subgroups of class f (taken in a subgroup of class e). If no subgroup of class f lies in class e, the construction fails.

`Normaliser(e, f)`

`Normalizer(e, f)`

Given poset elements e and f, return the poset element that corresponds to the class of subgroups that contain the normalizers of the subgroups of class f (taken in a subgroup of class e). If no subgroup of class f lies in class e, the construction fails.

`Length(e)`

The number of subgroups in the class corresponding to e.

`Order(e)`

The order of the subgroup of G corresponding to e.

`MaximalSubgroups(e)`

The maximal subgroups of e, returned as a set of poset elements.

`MinimalOvergroups(e)`

The minimal overgroups of e, returned as a set of poset elements.

`NumberOfInclusions(e, f)`

The number of elements of the conjugacy class of subgroups e that lie in a fixed representative of the conjugacy class of subgroups f.

57.12 Cohomology

In the following description, G is a group in the category `GrpPerm`, p is a prime number, and K is the finite field of order p. Further, F is a finitely presented group having the same number of generators as G, and is such that its relations are satisfied by the corresponding generators of G. In other words, the mapping taking the i-th generator of F to the i-th generator of G must be an epimorphism. Usually this mapping will be an isomorphism, although this is not mandatory.

`pMultiplicator(G, p)`

Given the group G and a prime p, return the invariant factors of the p-part of the Schur multiplicator of G.

`pCover(G, F, p)`

Given the group G and the finitely presented group F such that G is an epimorphic image of G in the sense described above, return a presentation for the p-cover of G, constructed as an extension of the p-multiplier by F.

`CohomologicalDimension(G, M, i)`

Given the group G, the $K[G]$-module M and an integer i (equal to 1 or 2), return the dimension of the i-th cohomology group of G acting on M.

`ExtensionProcess(G, M, F)`

Create an extension process for the group G by the module M.

`Extension(P, Q)`

`#NextExtension(P)`

Return the next extension of G as defined by the process P.
Assume that F is isomorphic to the permutation group G, and that we wish to determine presentations for one or more extensions of the K-module M by F, where K is the field of p elements. We first create an extension process using `ExtensionProcess(G, M, F)`. The possible extensions of M by G are in one-one correspondence with the elements of the second cohomology group $H^2(G, M)$ of G acting on M. Let $b_1, \ldots, b_l$ be a basis of $H^2(G, M)$. A general element of $H^2(G, M)$ therefore has the form $a_1b_1 + \cdots + a_lb_l$ and so can be defined by a sequence Q of l integers $[a_1, \ldots, a_l]$. Now, to construct the corresponding extension of M by G we call the function `Extension(P, Q)`. The required extension is returned as a finitely presented group. If all the extensions are required then they may be obtained successively by making p^l calls to the function `NextExtension`.

`SplitExtension(G, M, F)`

The split extension of the module M by the group G.

57.13 Characters and Representations

A set of functions are provided for computing with the characters and representations of a group. A full account of the character functions may be found in Chapter 91. Full details of the functions for constructing and analyzing representations may be found in Chapter 89. For the reader's convenience we include here a description of the basic functions for creating characters and representations.

Some functions described in this section may be missing or may have slightly different calling sequences for some categories of groups. For a complete description of the features available for a special category of groups, we refer to the chapter devoted to that category.

57.13.1 Character Theory

`CharacterDegrees(G)`

Given a finite pc-group G, return the sequence $[\langle d_1, c_1\rangle, langled_2, c_2\rangle, \ldots]$, where c_i is the number of irreducible characters of G having degree d_i. For details of the algorithm see Conlon [Con90b].

`CharacterTable(G)`

Construct the table of irreducible characters for the group G.

`PermutationCharacter(G)`

Given a group G represented as a permutation group, construct the character of G afforded by the defining permutation representation of G.

`PermutationCharacter(G, H)`

Given a group G and some subgroup H of G, construct the ordinary character of G afforded by the permutation representation of G given by the action of G on the coset space of the subgroup H in G.

57.13.2 Representation Theory

We describe the main functions for creating $K[G]$-modules for finite groups. The machinery for working with these modules is described in Chapter 89.

`GModule(G, S)`

Let G be a group defined on r generators and let S be a subalgebra of the matrix algebra $M_n(R)$, also defined by r non-singular matrices. It is assumed that the mapping from G to S defined by $\phi : G.i \mapsto S.i$, for $i = 1, \ldots, r$, extends to a group homomorphism. Let M be the natural module for the matrix algebra S. The function `GModule` gives M the structure of an $S[G]$-module, where the action of the i-th generator of G on M is given by the i-th generator of S.

`GModule(G, A, B)`

Given a finite group G, a normal subgroup A of G and a normal subgroup B of A such that the section A/B is elementary abelian of order p^n, create the $K[G]$-module M corresponding to the action of G on A/B, where K is the field $\mathbf{F}_p$. If B is trivial, it may be omitted. The function returns:

(a) the module M; and,

(b) the homomorphism $\phi : A/B \to M$.

`PermutationModule(G, H, R)`

Given a finite group G and a ring R, create the $R[G]$-module for G corresponding to the permutation action of G on the cosets of H.

`PermutationModule(G, R)`

Given a finite permutation group G and a ring R, create the natural permutation module for G over R.

Example H57E22

The permutation module for the group M_{10} over $GF(2)$ may be created as follows:

```
> m10 := PermutationGroup< 10 |  (1, 3, 9, 10, 2, 8, 7, 6, 4, 5),
>                                (1, 7)(2, 4, 3, 6, 8, 10, 9, 5) >;
> p := PermutationModule(m10, GF(2));
> p : Maximal;

GModule p of dimension 10 over GF(2)
Generators of acting algebra:
```

```
[0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 1 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 1]
[0 1 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 1 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0]
[1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0]
```

Example H57E23

The group G defined below is the split extension of an elementary abelian group E of order 16 by $Alt(6)$. After setting up the group, we construct the module M for G corresponding to its action on E.

```
> G := PermutationGroup< 16 |
>         (1, 15, 7, 5, 12)(2, 9, 13, 14, 8)(3, 6, 10, 11, 4),
>         (1, 4, 5)(2, 8, 10)(3, 12, 15)(6, 13, 11)(7, 9, 14),
>         (1, 16)(2, 3)(4, 5)(6, 7)(8, 9)(10, 11)(12, 13)(14, 15) >;
> CS := ChiefSeries(G);
> [ Order(H) : H in CS ];
[ 5760, 16, 1 ]
> M := GModule(G, CS[2]);
> M:Maximal;

GModule M of dimension 4 over GF(2)
Generators of acting algebra:

[0 1 0 0]
[0 1 1 0]
[0 0 1 1]
[1 0 0 1]

[0 0 1 0]
[0 0 0 1]
```

```
[1 0 1 0]
[0 1 0 1]

[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

57.14 Databases of Groups

MAGMA contains the following databases of groups:

Small Groups: Contains all groups of order up to 1000, excluding orders 512 and 768.

Perfect Groups: This database contains all perfect groups up to order 50000, and many classes of perfect groups up to order one million. Each group is defined by means of a finite presentation. Further information is also provided which allows the construction of permutation representations.

Rational Maximal Matrix Groups: Contains rational maximal finite matrix groups and their invariant forms, for small dimensions (up to 31 at V2.9 and above). Each entry can be accessed either as a matrix group or as a lattice.

Quaternionic Matrix Groups: A database of the finite absolutely irreducible subgroups of $\mathrm{GL}_n(\mathcal{D})$ where $\mathcal{D}$ is a definite quaternion algebra whose centre has degree d over $\mathbf{Q}$ and $nd \leq 10$. Each entry can be accessed either as a matrix group or as a lattice.

Transitive Permutation Groups: MAGMA has a database containing all transitive permutation groups having degree up to 22.

Primitive Permutation Groups: MAGMA has a database containing all primitive permutation groups having degree up to 50.

For a description of these databases, we refer to Chapter 66.

57.15 Bibliography

[**Con90**] S. B. Conlon. Computing modular and projective character degrees of soluble groups. *J. Symbolic Comp.*, 9:551–570, 1990.

[**LGM02**] C. R. Leedham-Green and Scott H. Murray. Variants of product replacement. *Contemp. Math.*, 298:97–104, 2002.

[**Lo98**] Eddie H. Lo. Finding intersections and normalizers in finitely generated nilpotent groups. *J. Symbolic Comput.*, 25(1):45–59, 1998.

58 PERMUTATION GROUPS

Chapter 58
PERMUTATION GROUPS

58.1 Introduction

58.1.1 Terminology

A permutation group G is a group of bijections $X \to X$, for some set X. The group G is said to *act* on X and the elements of G are called permutations (of the set X). A given permutation group G may have actions on sets other than the one on which it is defined. Thus, any set upon which G has a legitimate action will be called a G-set. The set X is called the *natural G-set* for the group G, and the action of G on X is called the *natural action* of G. Note that the group G also has a natural induced action on the G-closure of any *derived set* of X (see Section 14.8.1). MAGMA expects the G-set X to be of finite cardinality n. Usually, X will be $\{1, 2, \ldots, n\}$, but, as we shall see below, X may be a set of strings, or any other legitimate MAGMA set.

The elements of a G-set are called *points*. Let Y be a G-set for G. The (possibly empty) subset of Y whose points are fixed by every permutation of G, is called the *fixed-point set* for G, while the subset of Y consisting of points moved by some permutation of G is called the *support* of G. Similarly, for an element g of G the *fixed-point set* and the *support* of g are, respectively, the subsets of Y consisting of the points fixed and moved by g. The *degree* of G is defined to be the cardinality of the natural G-set of G; whereas the *degree* of an element g of G is defined to be the cardinality of the support of g, i.e. the number of points moved by g.

Permutation groups in MAGMAare limited to degree less than 2^{30}.

58.1.2 The Category of Permutation Groups

The family of all permutation groups of finite degree forms a category. The objects are the permutation groups and the morphisms are group homomorphisms. The MAGMA designation for this category of permutation groups is `GrpPerm`.

58.1.3 The Construction of a Permutation Group

Every permutation group acting on a set X is created as a subgroup of the symmetric group $\mathrm{Sym}(X)$. Thus, the construction of a general permutation group is a two-step process:

(i) The appropriate symmetric group, $\mathrm{Sym}(X)$, is constructed;

(ii) The required group G is then defined as a subgroup of $\mathrm{Sym}(X)$.

For convenience, a constructor `PermutationGroup< ... >`, which combines these two steps, is provided.

58.2 Creation of a Permutation Group

58.2.1 Construction of the Symmetric Group

`Sym(n)`

`SymmetricGroup(n)`

Given an integer $n \geq 1$, create the generic permutation group acting on the natural G-set $\Omega = \{1, 2, \ldots, n\}$, i.e. the symmetric group $\mathrm{Sym}(\Omega)$. Initially, only a structure table is created for $\mathrm{Sym}(n)$, so that, in particular, generators are not defined. This function is normally used to provide a context for the creation of elements and subgroups of $\mathrm{Sym}(n)$. If structural computation is attempted with the group created by `Sym(n)`, then generators will be created dynamically.

`Sym(X)`

`SymmetricGroup(X)`

Given a finite set X of cardinality $n \geq 1$, create the generic group G of permutations of X – the symmetric group $\mathrm{Sym}(X)$. Initially, only a structure table is created for $\mathrm{Sym}(X)$, so that, in particular, generators are not defined. This function is normally used to provide a context for the creation of elements and subgroups of $\mathrm{Sym}(X)$. If structural computation is attempted with the group created by `Sym(X)`, then generators will be created dynamically. Although the group G is defined on the set X, G is represented internally as a group of permutations of the set $\Omega = \{1, 2, \ldots, n\}$. Translation between X and Ω is done at input/output time. The precise representation can be found by using the `Labelling` function. If X is an indexed set then the indexing of elements of X determines the correspondence.

`StandardGroup(G)`

Return a group H isomorphic to G, but acting on the standard set $\{ 1, \ldots, n\}$. This function is useful when the natural G-set for G is not the standard set. If the natural G-set for G is the standard set, G is returned. The isomorphism from G to H is also returned.

Example H58E1

We define the symmetric group on the set of strings { "*a*" "*b*" "*c*" "*d*" }:

```
> S4 := Sym({ "a", "b", "c", "d" });
> S4;
Symmetric group S4 acting on a set of cardinality 4
Order = 24 = 2^3 * 3
> GSet(S4);
GSet{@ c, b, a, d @}
```

We define the symmetric group of degree 10 acting on the set $\{0, 1, \ldots, 9\}$.

```
> G := Sym({ 0..9 });
```

```
> G;
Symmetric group G acting on a set of cardinality 10
Order = 3628800 = 2^8 * 3^4 * 5^2 * 7
> GSet(G);
GSet{@ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 @}
```

58.2.2 Construction of a Permutation

Throughout this subsection we shall assume that the permutation group G has natural G-set X.

`elt< G | L >`

Given a permutation group G defined as acting on the set $X = \{x_1, \ldots, x_n\}$ of cardinality $n \geq 1$, and a list L of distinct elements $a_1, a_2, \ldots, a_n$ of X, construct the element g of G defined by $x_i \to a_i$, for $i = 1, \ldots, n$. Unless G is known to be the generic permutation group of degree n, the permutation will be tested for membership of G, and if g is not an element of G, the function will fail. If g does lie in G, g will have G as its parent. Since the membership test may involve constructing a base and strong generating set for G, this constructor may occasionally be very costly. Hence, a permutation g should be defined as an element of a subgroup of the generic group only when membership of G is required by subsequent operations involving g.

`G ! Q`

Given a permutation group G defined as acting on the set $X = \{x_1, \ldots, x_n\}$ of cardinality $n \geq 1$, and a sequence $Q = [a_1, a_2, \ldots, a_n]$ of distinct elements of X, construct the permutation g of X defined by $x_i \to a_i$, for $i = 1, \ldots, n$. This permutation will have G as its parent structure. As in the case of the `elt`-constructor, the operation will fail if g is not an element of G and the same observations concerning the cost of membership testing apply.

`G ! (...)(...)...(...)`

Given a permutation group G defined as acting on the set $X = \{x_1, x_2, \ldots, x_n\}$, construct the permutation g corresponding to the given product of cycles. Adjacent letters must be separated by commas. Further, *cycles of length one must be omitted*. The coercion operator ! may be omitted only within the context of the standard constructors `sub<>`, `ncl<>` and `quo<>`. Once the permutation g has been constructed, it will be tested for membership in G. If it is not a member, the construction fails.

`G ! \(...)(...)...(...)`

Given a permutation group G defined as acting on the set $X = \{1..n\}$, construct the permutation g corresponding to the given product of literal cycles of integers. Adjacent integers must be separated by commas. Once the permutation g has been constructed, it will be tested for membership in G. If it is not a member, the construction fails. This construction is strongly recommended when creating large permutations to avoid overhead in constructing unnecessarily large parse trees by MAGMA.

`G ! Q`

Given a permutation group G defined as acting on the set $X = \{1..n\}$, construct the permutation g corresponding to the given product of cycles. The indexed sets in Q must be disjoint subsets of X, which are interpreted as the disjoint cycles of the permutation being constructed. Cycles of length 1 may be omitted, but do not have to be omitted. Once the permutation g has been constructed, it will be tested for membership in G. If it is not a member, the construction fails. Note that the `Cycle` function produces results suitable for use as members of Q.

`ElementToSequence(g)`

`Eltseq(g)`

The sequence Q of images of the G-set of g. In particular, it has the property that `Parent(g)!Eltseq(g) eq g`.

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity permutation in the permutation group G.

Example H58E2

The three different constructions are illustrated by the following code, which assigns to each of the variables x, y and z the permutation (1)(2,3)(4,5,6).

```
> S6 := Sym(6);
> x := elt<S6 | 1,3,2,5,6,4>;
> x;
(2, 3)(4, 5, 6)
> y := S6![1,3,2,5,6,4];
> y;
(2, 3)(4, 5, 6)
> z := S6!(2,3)(4,5,6);
> z;
(2, 3)(4, 5, 6)
> S6!1;
Id(S6)
```

58.2.3 Construction of a General Permutation Group

`PermutationGroup< X | L >`

Suppose that the cardinality of the set X is n. Construct the permutation group G acting on the set X generated by the permutations defined by the list L. A term of the list L must be an object of one of the following types:

(a) A sequence of n elements of X defining a permutation of X (note that this is only well-defined when X is an indexed set);

(b) A set or sequence of sequences of type (a);

(c) An element of $\mathrm{Sym}(X)$;

(d) A set or sequence of elements of $\mathrm{Sym}(X)$;

(e) A subgroup of $\mathrm{Sym}(X)$;

(f) A set or sequence of subgroups of $\mathrm{Sym}(X)$.

Each element or group specified by the list must belong to the *same* generic permutation group. The group G will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of G consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list.

The `PermutationGroup` constructor is shorthand for the two statements:

```
SX := Sym(X); G := sub< SX | L >;
```

where `sub< ... >` is the subgroup constructor described in the next subsection.

`PermutationGroup< n | L >`

Construct the permutation group G acting on the set $X = \{1, 2, \ldots, n\}$ generated by the permutations defined by the list L. The possibilities for the terms of the list L are the same as for the constructor `PermutationGroup< X | L >`.

Example H58E3

The Hessian group generated by the permutations $(1,2,4)(5,6,8)(3,9,7)$ and $(4,5,6)(7,9,8)$ may be created by the statement:

```
> H := PermutationGroup< 9 | (1,2,4)(5,6,8)(3,9,7), (4,5,6)(7,9,8) >;
> H;
Permutation group H acting on a set of cardinality 9
       (1, 2, 4)(3, 9, 7)(5, 6, 8)
       (4, 5, 6)(7, 9, 8)
```

58.3 Elementary Properties of a Group

58.3.1 Accessing Group Information

The functions in this group provide access to basic information stored for a permutation group G.

`G . i`

The i-th defining generator for G. A negative subscript indicates that the inverse of the generator is to be created. The identity element of G will be created by $G.0$.

`Degree(G)`

The degree of the permutation group G.

`Generators(G)`

A set of elements of G that generate G.

`GeneratorsSequence(G)`

The sequence of elements used to define the group G. Any occurrences of the identity element or any repetitions of a generator, as removed by `Generators(G)`, are retained in this sequence. This function has the same effect as the expression `[G.i : i in [1..Ngens(G)]]` .

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for G.

`FewGenerators(G)`

A typically short sequence of random elements generating the group. Especially when groups are generated as subgroups, the result of `FewGenerators` is a much shorter sequence than returned by `GeneratorsSequence`.

`Generic(G)`

The generic group containing G, i.e. the symmetric group in which G is naturally embedded.

`Parent(g)`

The parent group G for the permutation g.

`GSet(G)`

The natural G-set for the permutation group G.

Example H58E4

Consider the group G of order 648 generated by the permutations (1,6,7)(2,5,8,3,4,9)(11,12) and (1,3)(4,9,12)(5,8,10,6,7,11).

```
> G := PermutationGroup< 12 | (1,6,7)(2,5,8,3,4,9)(11,12),
>                             (1,3)(4,9,12)(5,8,10,6,7,11) >;
> G;
Permutation group G acting on a set of cardinality 12
    (1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12)
    (1, 3)(4, 9, 12)(5, 8, 10, 6, 7, 11)
> G.1;
(1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12)
> G.1*G.2;
(1, 7, 3, 9, 2, 8)(4, 12, 5, 10, 6, 11)
> Degree(G);
12
> GSet(G);
GSet{@ 1 .. 12 @}
> Generic(G);
Symmetric group acting on a set of cardinality 12
Order = 479001600 = 2^10 * 3^5 * 5^2 * 7 * 11
> Generators(G);
{
    (1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12),
    (1, 3)(4, 9, 12)(5, 8, 10, 6, 7, 11)
}
> Ngens(G);
2
> x := G ! (1,6,7)(2,5,8,3,4,9)(11,12);
> x;
(1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12)
> Parent(x);
Permutation group G acting on a set of cardinality 12
Order = 648 = 2^3 * 3^4
    (1, 6, 7)(2, 5, 8, 3, 4, 9)(11, 12)
    (1, 3)(4, 9, 12)(5, 8, 10, 6, 7, 11)
]
```

58.3.2 Group Order

Unless the order is already known, each of the functions in this family will create a base and strong generating set for the group if one does not already exist.

`Order(G)`
`#G`

The order of the group G as an integer. If the order is not currently known, a base and strong generating set will be constructed for G.

`FactoredOrder(G)`

The order of the group G returned as a factored integer. The factorization is returned in the form of a sequence Q which is defined as follows: If $\#G = p_1^{e_1} \dots p_n^{e_n}, \quad e_i \neq 0$, then Q will be the integer sequence $[< p_1, e_1 >, \dots, < p_n, e_n >]$. If the order of G is not known, it will be computed.

58.3.3 Abstract Properties of a Group

`IsAbelian(G)`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsSpecial(G)`

Given a p-group G, return `true` if G is special, `false` otherwise.

`IsExtraSpecial(G)`

Given a group G is a p-group G, return `true` if G is extra-special, `false` otherwise.

`IsNilpotent(G)`

Returns `true` if the group G is nilpotent, `false` otherwise.

`IsSoluble(G)`
`IsSolvable(G)`

Returns `true` if the group G is soluble, `false` otherwise. Uses the algorithm of Sims [Sim90].

`IsPerfect(G)`

Returns `true` if the group G is perfect, `false` otherwise.

`IsSimple(G)`

Returns `true` if the group G is simple, `false` otherwise.

`IsWreathProduct(G)`

Returns `true` if the group G is isomorphic to a wreath product $A \wr B$, where B is transitive, and `false` otherwise. If true, then three subgroups of G, call them A, B, C, are also returned. In this case we have G isomorphic to `WreathProduct(A, CosetImage(B, C))`.

Example H58E5

We determine the orders of those subgroups of the Mathieu group M_{24} which are perfect but not simple. We use the function `PerfectSubgroups` which returns a representative from each conjugacy class of perfect subgroups.

```
> load m24;
Loading "/home/magma/libs/pergps/m24"
M24 - Mathieu group on 24 letters - degree 24
Order 244 823 040 = 2^10 * 3^3 * 5 * 7 * 11 * 23;  Base 1,2,3,4,5,6,7
Group: G
> time S := PerfectSubgroups(G);
Time: 29.460
> [ Order(H) : R in S | not IsSimple(H) where H := R`subgroup ];
[ 120, 120, 120, 180, 180, 240, 240, 336, 336, 336, 336, 504, 720, 1008, 1080,
  960, 960, 960, 1344, 1344, 1344, 1920, 2688, 2688, 2688, 2688, 2688, 2880,
  3840, 3840, 5760, 10752, 11520, 11520, 40320, 21504, 21504, 32256, 64512,
  69120, 322560 ]
```

58.4 Homomorphisms

Homomorphisms are a central concept in group theory, and MAGMA provides extensive facilities for group homomorphisms. Many useful homomorphisms are returned by constructors and intrinsic functions. Examples of these are the `quo` constructor, the `sub` constructor and intrinsic functions such as `OrbitAction`, `BlocksAction`, `FPGroup` and `RadicalQuotient`, which are described in more detail elsewhere in this chapter. In this section we describe how the user may create their own homomorphisms with domain a permutation group.

`hom< G -> H | L >`

Given the permutation group G, construct the homomorphism $f : G \to H$ given by the generator images in L. H must be a group. The clause L may be any one of the following types:

(a) A list of elements of H, giving images of the generators of G;

(b) A list of pairs, where the first in the pair is an element of G and the second its image in H;

(c) A sequence of elements of H, as in (a);

(d) A set or sequence of pairs, as in (b);

Each image element specified by the list must belong to the *same* group H. In the cases where pairs are given the given elements of G must generate G.

`Domain(f)`

The domain of the homomorphism f.

`Codomain(f)`

The codomain of the homomorphism f.

`Image(f)`

The image or range of the homomorphism f. This will be a subgroup of the codomain of f. The algorithm computes the image and kernel simultaneously (see [LGPS91]).

`Kernel(f)`

The kernel of the homomorphism f. This will be a normal subgroup of the domain of f. The algorithm computes the image and kernel simultaneously (see [LGPS91]).

`IsHomomorphism(G, H, Q)`

Return the value `true` if the sequence Q defines a homomorphism from the group G to the group H. The sequence Q must have length `Ngens(G)` and must contain elements of H. The i-th element of Q is interpreted as the image of the i-th generator of G and the function decides if these images extend to a homomorphism. If so, the homomorphism is also returned. The algorithm employed is described in [LGPS91].

Example H58E6

Consider the group G of order 648 generated by the permutations (1,6,7)(2,5,8,3,4,9)(11,12) and (1,3)(4,9,12)(5,8,10,6,7,11). We construct a permutation representation of G of degree 8 by considering the conjugation action of G on one of its elements. We then construct the preimage of a normal subgroup of the image.

```
> G := PermutationGroup< 12 | (1,6,7)(2,5,8,3,4,9)(11,12),
>                             (1,3)(4,9,12)(5,8,10,6,7,11) >;
> #G;
648
> x := G ! (1, 2, 3)(7, 8, 9)(10, 11, 12);
> x_class := {@ x ^ y : y in G @};
> #x_class;
8
> S := SymmetricGroup(8);
> images := [S![Index(x_class, x_class[i]^(G.j)):i in [1..8]] :j in [1..2]];
> f := hom< G -> S | images>;
```

The map f is the homomorphism of G onto the group induced by the action of the element x. We computer the images of some elements and then find the image and kernel of f.

```
> (G.1*G.-2) @ f;
(2, 5, 7)(3, 8, 6)
> ((G.1) @ f) * ((G.2) @ f) ^ -1;
(2, 5, 7)(3, 8, 6)
> H := Image(f);
> H;
Permutation group acting on a set of cardinality 8
Order = 24 = 2^3 * 3
  (1, 2, 3, 4, 6, 5)(7, 8)
  (1, 2, 8, 4, 6, 7)(3, 5)
> Kernel(f);
Permutation group acting on a set of cardinality 12
Order = 27 = 3^3
  (1, 2, 3)(4, 6, 5)(7, 8, 9)(10, 12, 11)
  (4, 5, 6)(7, 9, 8)
  (7, 9, 8)(10, 11, 12)
```

We now find the preimage of $O_2(H)$ as a subgroup of G.

```
> pCore(H, 2) @@ f;
Permutation group acting on a set of cardinality 12
Order = 216 = 2^3 * 3^3
  (4, 5, 6)(7, 9, 8)
  (1, 2, 3)(4, 6, 5)(7, 8, 9)(10, 12, 11)
  (1, 4, 2, 5, 3, 6)(7, 12, 9, 11, 8, 10)
  (1, 10, 3, 11, 2, 12)(4, 9, 5, 8, 6, 7)
  (2, 3)(4, 5)(8, 9)(11, 12)
  (7, 9, 8)(10, 11, 12)
```

58.5 Building Permutation Groups

Examples of permutation groups are routinely constructed by taking one or more standard groups and applying some extension procedure to construct a group having the given groups as subgroups or quotient groups. In the first subsection we describe functions which construct some well-known groups and in the following subsection we give functions for constructing direct and wreath products.

58.5.1 Some Standard Permutation Groups

A number of functions are provided which construct various standard groups. The effect of these functions is to construct the group on some standard set of generating permutations.

`AbelianGroup(GrpPerm, Q)`

Construct the abelian group defined by the sequence $Q = [n_1, \ldots, n_r]$ of positive integers. The function constructs the direct product of cyclic groups

$$Z(n_1) \times Z(n_2) \times \cdots \times Z(n_r).$$

`AlternatingGroup(GrpPerm, n)`

`AlternatingGroup(n)`

`Alt(n)`

Construct the alternating group of degree n on generators $(3, 4, \ldots, n)$ and $(1, 2, 3)$, if n is odd, or $(1, 2)(3, 4, \ldots, n)$ and $(1, 2, 3)$, if n is even.

`CyclicGroup(GrpPerm, n)`

`CyclicGroup(n)`

Construct the cyclic group of order n with generator $(1, 2, \ldots, n)$.

`DihedralGroup(GrpPerm, n)`

`DihedralGroup(n)`

Construct the dihedral group of degree n and order $2 * n$ on generators $(1, 2, \ldots, n)$ and $(1, n)(2, n-1)\cdots$.

`Sym(GrpPerm, n)`

`SymmetricGroup(GrpPerm, n)`

`Sym(n)`

`SymmetricGroup(n)`

Construct the symmetric group of degree n on generators $(1, 2, \ldots, n)$ and $(1, 2)$.

`ExtraSpecialGroup(GrpPerm, p, n : parameters)`

`ExtraSpecialGroup(p, n : parameters)`

Given a small prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1} in the category `GrpPerm`. The isomorphism type of G can be selected using the parameter `Type`.

`Type` MonStgElt *Default :* "+"

Possible values for this parameter are "+" (default) and "−".

If `Type` is set to "+", the function returns for $p = 2$ the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If `Type` is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

`YoungSubgroup(L)`

`Full` RngIntElt *Default :* `false`

Given a sequence L of positive integers, compute the Young subgroup parameterized by L, i.e., the direct product of the symmetric groups on L_i points. If the optional parameter `Full` is given, construct the group as a subgroup of the symmetric group on `Full` elements.

Example H58E7

(1) The abelian group $Z_2 \times Z_2 \times Z_4$:

```
> A := AbelianGroup(GrpPerm, [2, 2, 4] );
> A;
Permutation group A acting on a set of cardinality 8
Order = 16 = 2^4
    (1, 2)
    (3, 4)
    (5, 6, 7, 8)
```

(2) The alternating group of degree 12:

```
> A12 := AlternatingGroup(GrpPerm, 12);
> A12;
Permutation group A12 acting on a set of cardinality 12
Order = 239500800 = 2^9 * 3^5 * 5^2 * 7 * 11
    (1, 2)(3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
    (1, 2, 3)
```

(3) The cyclic group Z_{24}:

```
> Z24 := CyclicGroup(GrpPerm, 24);
> Z24;
Permutation group Z24 on a set of cardinality 24
```

```
Order = 24 = 2^3 * 3
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
        16, 17, 18, 19, 20, 21, 22, 23, 24)
```

(4) The dihedral group of order 24:

```
> D12 := DihedralGroup(GrpPerm, 12);
> D12;
Permutation group D12 acting on a set of cardinality 12
Order = 24 = 2^3 * 3
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
    (1, 12)(2, 11)(3, 10)(4, 9)(5, 8)(6, 7)
```

(5) The symmetric group of degree 8:

```
> S8 := SymmetricGroup(GrpPerm, 8);
> S8;
Symmetric group S8 acting on a set of cardinality 8
Order = 40320 = 2^7 * 3^2 * 5 * 7
```

58.5.2 Direct Products and Wreath Products

`DirectProduct(G, H)`

Given two permutation groups G and H, construct the direct product D of G and H as an intransitive group having degree equal to the sum of the degrees of G and H. In addition, the sequences I of inclusions and P of projections are returned, satisfying $I[i] : K_i \to D(K_i)$ and $P[i] : D \to K_i$ (where $K_1 = G, K_2 = H$ and $D(K)$ is the group K represented naturally as a subgroup of D).

`DirectProduct(Q)`

Given a sequence Q of n permutation groups, construct the direct product $Q[1] \times Q[2] \times \ldots \times Q[n]$ as an intransitive group of degree equal to the sum of the degrees of the groups $Q[i]$, $(i = 1, \ldots, n)$. In addition, the sequences I of inclusion and P of projections are returned, satisfying $I[i] : Q[i] \to D(Q[i])$ and $P[i] : D \to Q[i]$ (where $D(K)$ is the group K represented naturally as a subgroup of D).

`PrimitiveWreathProduct(G, H)`

Given permutation groups G and H, construct the wreath product $G \wr H$ of G and H, where $G \wr H$ has product action.

`PrimitiveWreathProduct(Q)`

Given a sequence Q of n permutation groups, construct the iterated wreath product $T = (\ldots (Q[1] \wr Q[2]) \wr \ldots \wr Q[n])$, where T has product action.

`WreathProduct(G, H)`

Given permutation groups G and H, construct the wreath product $W = G \wr H$ of G and H, where $G \wr H$ has imprimitive action. The function also returns the sequence of `Degree(`H`)` inclusions of G into W, the inclusion of H into W and the projection of W onto H.

`WreathProduct(Q)`

Given a sequence Q of n permutation groups, construct the iterated wreath product $W = (\ldots(Q[1] \wr Q[2]) \wr \ldots \wr Q[n])$, where W has imprimitive action.

`WreathProduct(B)`

Given a block system B of some permutation group G, compute the wreath-product corresponding to B.

`WreathProduct(G, B)`

Compute the smallest wreath product W to the block system B of G such that $G \subseteq W$. Also return the complement as a subgroup of W. The third parameter is a subgroup which is isomorphic to the action within a block.

Example H58E8

We define G to be the symmetric group of degree 4 and H to be the dihedral group of order 8. We then proceed to form the direct, primitive-wreath and wreath products of G and H.

```
> G := SymmetricGroup(GrpPerm, 4);
> H := DihedralGroup(GrpPerm, 3);
> D := DirectProduct(G, H);
> D;
Permutation group D acting on a set of cardinality 7
Order = 144 = 2^4 * 3^2
    (1, 2, 3, 4)
    (1, 2)
    (5, 6, 7)
    (5, 6)
> T := PrimitiveWreathProduct(G, H);
> T;
Permutation group T acting on a set of cardinality 64
Order = 82944 = 2^10 * 3^4
    (2, 5, 17)(3, 9, 33)(4, 13, 49)(6, 21, 18)(7, 25, 34)(8, 29, 50)
       (10, 37, 19) (11, 41, 35)(12, 45, 51) (14, 53, 20)(15, 57, 36)
       (16, 61, 52)(23, 26, 38) (24, 30, 54)(27, 42, 39)(28, 46, 55)
       (31, 58, 40) (32, 62, 56)(44, 47, 59)(48, 63, 60)
    (2, 5)(3, 9)(4, 13)(7, 10)(8, 14)(12, 15)(18, 21)(19 , 25)(20, 29)
       (23, 26)(24, 30)(28, 31)(34, 37)(35 , 41)(36, 45)(39, 42)(40, 46)
       (44, 47)(50, 53)(51 , 57)(52, 61)(55, 58)(56, 62)(60, 63)
    (1, 2, 3, 4)(5, 6, 7, 8)(9, 10, 11, 12)(13, 14, 15, 16)(17, 18, 19, 20)
       (21, 22, 23, 24)(25, 26, 27, 28)(29, 30, 31, 32)(33, 34, 35, 36)
```

```
       (37, 38, 39, 40)(41, 42, 43, 44)(45, 46, 47, 48)(49, 50, 51, 52)
       (53, 54, 55, 56)(57, 58, 59, 60)(61, 62, 63, 64)
    (1, 2)(5, 6)(9, 10)(13, 14)(17, 18)(21, 22)(25, 26)( 29, 30)(33, 34)
       (37, 38)(41, 42)(45, 46)(49, 50)( 53, 54)(57, 58)(61, 62)
> W := WreathProduct(G, H);
> W;
Permutation group W acting on a set of cardinality 12
Order = 82944 = 2^10 * 3^4
    (1, 5, 9)(2, 6, 10)(3, 7, 11)(4, 8, 12)
    (1, 5)(2, 6)(3, 7)(4, 8)
    (1, 2, 3, 4)
    (1, 2)
```

58.6 Permutations

58.6.1 Coercion

`G ! g`

Given a subgroup G of $\mathrm{Sym}(X)$ and a permutation g belonging to $\mathrm{Sym}(X)$ that is contained in G, embed g in G. Thus, this operator changes the parent of g to be G.

`G !! H`

Given a group H whose natural G-set X is a subset of the natural G-set Y for the group G, embed H as a subgroup of G. The operator fails if the image of H in $\mathrm{Sym}(Y)$ is not a subgroup of G.

58.6.2 Arithmetic with Permutations

`g * h`

Product of permutation g and permutation h, where g and h belong to the same generic group U. If g and h both belong to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h belong to subgroups H and K of a subgroup G of U, then the product is returned as an element of G. Otherwise, the product is returned as an element of U.

`g ^ n`

The n-th power of the permutation g, where n is a positive, negative or zero integer.

`g / h`

Product of the permutation g by the inverse of the permutation h, i.e. the element $g * h^{-1}$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g/h are the same as for $g * h$.

`g ^ h`

Conjugate of the permutation g by the permutation h, i.e. the element $h^{-1} * g * h$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g^h are the same as for $g * h$.

`(g, h)`

Commutator of the permutations g and h, i.e. the element $g^{-1} * h^{-1} * g * h$. Here g and h must belong to the same generic group U. The rules for determining the parent group of (g, h) are the same as those for $g * h$.

`(g`$_1$`, ..., g`$_r$`)`

Given r permutations $g_1, \ldots, g_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

58.6.3 Properties of Permutations

`CycleStructure(g)`

Given a permutation g belonging to a group of degree n, return the partition of n corresponding to the cycles of g. This partition is returned in the form of a sequence Q of pairs, where the terms of Q correspond to the distinct cycle lengths of g. The value of the term $Q[i]$ is a tuple $< l_i, n_i >$ belonging to $\mathbf{Z} \times \mathbf{Z}$. Here l_i is the length of a cycle of g and n_i is the number of cycles of length l_i.

`Degree(g)`

Given a permutation g, return the degree of g, i.e. the number of points moved by g.

`IsEven(g)`

Returns `true` if the permutation g is an even permutation, `false` otherwise.

`Sign(g)`

Return 1 if the permutation g is even, return -1 if g is odd.

`Order(g)`

Order of the permutation g.

58.6.4 Predicates for Permutations

`g eq h`

Given permutations g and h belonging to the same generic group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given permutations g and h belonging to the same generic group, return `true` if g and h are distinct elements, `false` otherwise.

`IsId(g)`

`IsIdentity(g)`

Returns `true` if the permutation g is the identity permutation.

Example H58E9

We illustrate the permutation operations by applying them to some elements of Sym(9).

```
> G := Sym(9);
> x := G ! (1,2,4)(5,6,8)(3,9,7);
> y := G ! (4,5,6)(7,9,8);
> x*y;
(1, 2, 5, 4)(3, 8, 6, 7)
> x^-1;
(1, 4, 2)(3, 7, 9)(5, 8, 6)
> x^2;
(1, 4, 2)(3, 7, 9)(5, 8, 6)
> x / y;
(1, 2, 6, 9, 8, 4)(3, 7)
> x^y;
(1, 2, 5)(3, 8, 9)(4, 7, 6)
> (x, y);
(1, 7, 3, 6)(4, 5, 9, 8)
> x^y eq y^x;
false
> CycleStructure(x^2*y);
[ <6, 1>, <2, 1>, <1, 1> ]
> Degree(y);
6
> Order(x^2*y);
6
```

58.6.5 Set Operations

The creation of a base and strong generating set (BSGS) for a permutation group G provides us with a very compact representation of the set of elements of G. A particular BSGS imposes an order on the elements of G (lexicographic ordering of base images). It thus makes sense to talk about the 'number' of a group element relative to a particular BSGS.

`G * H`

Given permutation groups G and H, where G and H both belong to the same generic group, form the set product $\{g * h | g \in G, h \in H\}$ as a set of group elements.

`ElementSet(G, H)`

Given a group G and a subgroup H of G, return the elements of H in the form of a set of elements of G. This function is only applicable to very small groups.

`NumberingMap(G)`

A bijective mapping from the group G onto the set of integers $\{1 \ldots |G|\}$. The actual mapping depends upon the base and strong generating set chosen for G.

`RandomProcess(G)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 20

Create a process to generate randomly chosen elements from the finite group G. The process is based on the product-replacement algorithm of [CLGM+95], modified by the use of an accumulator. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G)+1$. Initially, these are just the generators of G. As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses one of the elements in the slots and multiplies it into the accumulator. The element in that slot is replaced by the product of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting `Scramble := m` causes m such operations to be performed before the process is returned.

`Random(G: parameters)`

`Short`	BoolElt	*Default :* `false`

A randomly chosen element for the group G. If a BSGS is known for G, then the distribution will be uniform over G. If no BSGS is known, then the *random* element is chosen by multiplying out a *random* word in the generators. Since it is usually not practical to choose words long enough to properly sample the elements of G, the element returned will usually be biased. The boolean-valued parameter `Short` is used in this situation to indicate that a short word will suffice. Thus, if `Random` is invoked with `Short` assigned the value `true` then the element is constructed using a short word.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the finite group G, construct a random element of G by forming a random product over the expanded generating set constructed when the process was created. For large degree groups, or groups for which a BSGS is not known, this function should be used in preference to `Random(G)`.

`Representative(G)`

`Rep(G)`

An element chosen from the permutation group G.

Example H58E10

We use the function `NumberingMap` to construct the multiplication table for the dihedral group of order 12.

```
> G := DihedralGroup(GrpPerm, 6);
> f := NumberingMap(G);
> [ [ f(x*y) : y in G ] : x in G ];
[
    [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ],
    [ 2, 3, 4, 5, 6, 1, 12, 7, 8, 9, 10, 11 ],
    [ 3, 4, 5, 6, 1, 2, 11, 12, 7, 8, 9, 10 ],
    [ 4, 5, 6, 1, 2, 3, 10, 11, 12, 7, 8, 9 ],
    [ 5, 6, 1, 2, 3, 4, 9, 10, 11, 12, 7, 8 ],
    [ 6, 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 7 ],
    [ 7, 8, 9, 10, 11, 12, 1, 2, 3, 4, 5, 6 ],
    [ 8, 9, 10, 11, 12, 7, 6, 1, 2, 3, 4, 5 ],
    [ 9, 10, 11, 12, 7, 8, 5, 6, 1, 2, 3, 4 ],
    [ 10, 11, 12, 7, 8, 9, 4, 5, 6, 1, 2, 3 ],
    [ 11, 12, 7, 8, 9, 10, 3, 4, 5, 6, 1, 2 ],
    [ 12, 7, 8, 9, 10, 11, 2, 3, 4, 5, 6, 1 ]
]
```

Example H58E11

We illustrate the use of the function `Random` using the wreath product of the symmetric group of degree 4 and the cyclic group of order 6.

```
> G := WreathProduct( Sym(4), CyclicGroup(GrpPerm, 6));
> G;
Permutation group G acting on a set of cardinality 24
    (1, 5, 9, 13, 17, 21)(2, 6, 10, 14, 18, 22) (3, 7, 11, 15, 19, 23)
        (4, 8, 12, 16, 20, 24)
    (1, 2, 3, 4)
    (1, 2)
> Order(G);
```

```
1146617856
> Random(G);
(1, 17, 12, 4, 18, 10, 3, 20, 9, 2, 19, 11)(5, 22, 13, 6, 21, 15)
    (7, 24, 16)(8, 23, 14)
```

We display the cycle structures of 10 random elements of G.

```
> R := [ CycleStructure(Random(G)) : i in [1..10] ];
> R;
[
    [ <6, 1>, <3, 6> ],
    [ <9, 1>, <6, 2>, <3, 1> ],
    [ <9, 2>, <3, 2> ],
    [ <12, 1>, <9, 1>, <3, 1> ],
    [ <18, 1>, <6, 1> ],
    [ <18, 1>, <6, 1> ],
    [ <12, 1>, <6, 2> ],
    [ <6, 3>, <2, 3> ],
    [ <6, 1>, <4, 3>, <2, 3> ],
    [ <6, 3>, <3, 2> ]
]
```

58.7 Conjugacy

`Class(H, x)`

`Conjugates(H, x)`

Given a group H and an element x belonging to a group K such that H and K are subgroups of the same symmetric group, this function returns the set of conjugates of x under the action of H. If $H = K$, the function returns the conjugacy class of x in H.

`ConjugacyClasses(G: parameters)`

`Classes(G: parameters)`

Construct a set of representatives for the conjugacy classes of G. The classes are returned as a sequence of triples containing the element order, the class length and a representative element for the class. The parameters `Reps` and `Al` enable the user to select the algorithm that is to be used.

`Reps` [GrpPermElt] *Default :*

`Reps :=` Q: Create the classes of G by using the random algorithm but using the group elements in Q as the "random" elements tried. The element orders and lengths of the classes will be computed and checked for consistency.

`Reps` [<GrpPermElt, RngIntElt>]

`Reps :=` Q Create the classes of G assuming that the first elements of the tuples in Q form a complete set of conjugacy class representatives and the corresponding integer is the size of the conjugacy class. The only check performed is that the class sizes sum to the group order.

`Al`	MonStgElt	*Default :* "*Default*"
`WeakLimit`	RngIntElt	*Default :* 500
`StrongLimit`	RngIntElt	*Default :* 5000

`Al := "Action"`: Create the classes of G by computing the orbits of the set of elements of G under the action of conjugation. This option is only feasible for small groups.
`Al := "Random"`: Construct the conjugacy classes of elements for a permutation group G using an algorithm that searches for representatives of all conjugacy classes of G by examining a random selection of group elements and their powers. The behaviour of this algorithm is controlled by two associated optional parameters `WeakLimit` and `StrongLimit`, whose values are positive integers n_1 and n_2, say. Before describing the effect of these parameters, some definitions are needed: A mapping $f : G \to I$ is called a class invariant if $f(g) = f(g^h)$ for all $g, h \in G$. For permutation groups, the cycle structure of g is a readily computable class invariant. Two elements g and h are said to be *weakly conjugate* with respect to the class invariant f if $f(g) = f(h)$. By definition, conjugacy implies weak conjugacy, but the converse is false. The random algorithm first examines n_1 random elements and their powers, using a test for weak conjugacy. It then proceeds to examine a further n_2 random elements and their powers, using a test for ordinary conjugacy. The idea behind this strategy is that the algorithm should attempt to find as many classes as possible using the very cheap test for weak conjugacy, before employing the more expensive ordinary conjugacy test to recognize the remaining classes.
`Al := "Inductive"`: Use G. Butler's inductive method to compute the classes. See Butler [But94] for a description of the algorithm. The action and random algorithms are used by this algorithm to find the classes of any small groups it is called upon to deal with.
`Al := "Extend"`: Construct the conjugacy classes of G by first computing classes in a quotient G/N and then extending these classes to successively larger quotients G/H until the classes for $G/1$ are known. More precisely, a series of subgroups $1 = G_0 < G_1 < \cdots < G_r = R < G$ is computed such that R is the (solvable) radical of G and G_{i+1}/G_i is elementary abelian. The radical quotient G/R is computed and its classes and centralizers of their representatives found and pulled back to G. The parameters `TFAl` and `ASAl` control the algorithm used to compute the classes of G/R.

To extend from G/G_{i+1} to the next larger quotient G/G_i, an affine action of each centralizer on a quotient of the elementary abelian layer G_{i+1}/G_i is computed. Each distinct orbit in that action gives rise to a new class of the larger quotient (see Mecky and Neubuser [MN89]).

`Al := "Default"`: First some special cases are checked for: If `IsAltsym(G)` then the classes of G are computed from the partitions of `Degree(G)`. If G is solvable, an isomorphic representation of G as a pc-group is computed and the classes computed in that representation. In general, the action algorithm will be used if $|G| \leq 5000$, otherwise the extension algorithm will be applied.

`TFAl` MonStgElt *Default :* "*Default*"

This parameter controls the algorithm used to compute the classes of a group with trivial Fitting subgroup, such as the group G/R in the description of the `"Extend"` method. The possible settings are the same as for `Al`, except that `"Extend"` is not a valid choice. The `"Action"`, `"Random"` and `"Inductive` settings behave as described above. The default algorithm is Derek Holt's generalisation of the Cannon/Souvignier fusion method to all classes of the group. The original fusion algorithm used fusion only on classes within a direct product normal subgroup, see [CS97]. Holt has extended the use of fusion to all conjugacy classes, avoiding the random part of the Cannon/Souvignier method. This algorithm must use another algorithm to find the classes of almost simple groups arising from the socle of the TF-group. The algorithm used for this is controlled by the parameter `ASAl`.

`ASAl` MonStgElt *Default :* "*Default*"

This parameter controls the algorithm used to compute the classes of an almost simple group from within the default TF-group algorithm. The possibilities for this parameter are as for `TFAl`. The default algorithm first determines if `Altsym(G)` is true. If so, classes are deduced from the partitions of `Degree(G)`. Next, if the order of G is ≤ 5000 then the action algorithm is used. If the socle of G has the correct order to be $PSL(2, q)$, for some q, then the random algorithm is used on G. Otherwise the inductive algorithm is used.

`Centralisers` BoolElt *Default :* `false`

A flag to force the storing of the centralisers of the class representatives with the class table. This does not apply to the action algorithm. In the case of the extension algorithm, this will do extra work to lift the centralisers through the final elementary abelian layer.

`PowerMap` BoolElt *Default :* `false`

A flag to force the storing of whatever power map information is produced by the classes algorithm used. In the case of the extension algorithm, this flag forces the computation of the full power map en-route, and may take considerably longer than not doing so. However, it is overall faster to set this flag true when the `PowerMap` is going to be computed anyway.

`ClassRepresentative(G, x)`

Given a group G for which the conjugacy classes are known and an element x of G, return the designated representative for the conjugacy class of G containing x.

`ClassCentraliser(G, i)`

The centraliser of the representative element stored for conjugacy class number i in group G. The group computed is stored with the class table for reference by future calls to this function.

`ClassMap(G:` *parameters*`)`

Given a group G, construct the conjugacy classes and the class map f for G. For any element x of G, $f(x)$ will be the index of the conjugacy class of x in the sequence returned by the `Classes` function. If the parameter `Orbits` is set `true`, the classes are computed as orbits of elements under conjugation and the class map is stored as a list of images of the elements of G (a list of length $|G|$). This option gives fast evaluation of the class map but is practical only in the case of very small groups. With `Orbits := false`, `WeakLimit` and `StrongLimit` are used to control the random classes algorithm (see function `Classes`).

`IsConjugate(G, g, h:` *parameters*`)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function returns a second value if the elements are conjugate: an element k which conjugates g into h. The method used is the backtrack search of Leon [Leo97]. This search may be speeded considerably by knowledge of (subgroups of) the centralizers of g and h in G. The parameters relate to these subgroups.

`Centralizer`	MonStgElt	*Default :* "*Default*"
`LeftSubgroup`	GrpPerm	*Default :* $\langle g \rangle$
`RightSubgroup`	GrpPerm	*Default :* $\langle h \rangle$

The `LeftSubgroup` and `RightSubgroup` parameters enable the user to supply known subgroups of the centralizers of g and h respectively to the algorithm. By default, the cyclic subgroups generated by g and h are the known subgroups. The `Centralizer` parameter controls whether the algorithm starts by computing one or both centralizers in full before the conjugacy test. The `"Default"` behaviour is to compute the left centralizer, i.e. $C_G(g)$, unless either a left or right subgroup is specified, in which case no centralizer calculation is done. Other possible values are the four possibilities `"Left"` which forces computation of $C_G(g)$, `"Right"` which forces $C_G(h)$ to be computed, `"Both"` which computes both centralizers, and `"None"` which will compute no centralizers.

`IsConjugate(G, H, K:` *parameters*`)`

Given a group G and subgroups H and K of G, return the value `true` if H and K are conjugate in G. The function returns a second value if the subgroups are conjugate: an element z which conjugates H into K. The method used is the backtrack search of Leon [Leo97].

`Compute`	MonStgElt	*Default :* "*Default*"

This parameter may be set to any of "Both", "Default", "Left", "None", or "Right". This controls which normalisers are computed before starting the conjugacy test. The default strategy is currently "Left", which computes the normalizer of H in G before starting the conjugacy test between H and K. The greater the difference between H and K and their normalizers in G, the more helpful to the search it is to compute their normalizers.

`LeftSubgroup`	GrpPerm	*Default :* H
`RightSubgroup`	GrpPerm	*Default :* K

Instead of having the `IsConjugate` function compute the normalizers of H and K, the user may supply any known subgroup of G normalizing H (as `LeftSubgroup`) or normalizing K (as `RightSubgroup`) to be used as the normalizing groups to aid the search.

`Exponent(G)`

The exponent of the group G. This is computed as the product of the exponents of the Sylow subgroups fo G.

`NumberOfClasses(G)`

`Nclasses(G)`

The number of conjugacy classes of elements for the group G.

`PowerMap(G)`

Given a group G, construct the power map for G. Suppose that the order of G is m and that G has r conjugacy classes. When the classes are determined by Magma, they are numbered from 1 to r. Let C be the set of class indices $\{1, \ldots, r\}$ and let Z be the set of integers. The *power map* f for G is the mapping

$$f : C \times Z \to C$$

where the value of $f(i, j)$ for $i \in C$ and $j \in Z$ is the number of the class which contains x_i^j, where x_i is a representative of the i-th conjugacy class.

`AssertAttribute(G, "Classes", Q)`

Assert the class representatives of G. The action taken is identical to using the `ConjugacyClasses` function described above, with the parameter `Reps` set to Q. Thus Q may be a sequence of group elements or a sequence of tuples giving class representatives and class lengths.

Example H58E12

The conjugacy classes of the Mathieu group M_{11} can be constructed as follows:

```
> SetSeed(2);
> M11 := sub<Sym(11) | (1,10)(2,8)(3,11)(5,7), (1,4,7,6)(2,11,10,9)>;
> Classes(M11);
Conjugacy Classes of group M11
------------------------------
[1]     Order 1       Length 1
        Rep Id(M11)

[2]     Order 2       Length 165
        Rep (3, 10)(4, 9)(5, 6)(8, 11)

[3]     Order 3       Length 440
        Rep (1, 2, 4)(3, 5, 10)(6, 8, 11)

[4]     Order 4       Length 990
        Rep (3, 6, 10, 5)(4, 8, 9, 11)

[5]     Order 5       Length 1584
        Rep (1, 3, 6, 2, 8)(4, 7, 10, 9, 11)

[6]     Order 6       Length 1320
        Rep (1, 11, 2, 6, 4, 8)(3, 10, 5)(7, 9)

[7]     Order 8       Length 990
        Rep (1, 4, 5, 6, 2, 7, 11, 10)(8, 9)

[8]     Order 8       Length 990
        Rep (1, 7, 5, 10, 2, 4, 11, 6)(8, 9)

[9]     Order 11      Length 720
        Rep (1, 11, 9, 10, 4, 3, 7, 2, 6, 5, 8)

[10]    Order 11      Length 720
        Rep (1, 9, 4, 7, 6, 8, 11, 10, 3, 2, 5)
```

Example H58E13

The default values for the random class algorithm are adequate for a large variety of groups. We look at what happens when we vary the parameters in the case of the Higman-Sims simple group represented on 100 letters. In this case the default strategy reduces to a random search. The first choice of parameters does not look at enough random elements. Increasing the limit on the number of random elements examined will ensure the algorithm succeeds.

```
> G := sub<Sym(100) |
```

```
>    (2,8,13,17,20,22,7)(3,9,14,18,21,6,12)(4,10,15,19,5,11,16)
>        (24,77,99,72,64,82,40)(25,92,49,88,28,65,90)(26,41,70,98,91,38,75)
>        (27,55,43,78,86,87,45)(29,69,59,79,76,35,67)(30,39,42,81,36,57,89)
>        (31,93,62,44,73,71,50)(32,53,85,60,51,96,83)(33,37,58,46,84,100,56)
>        (34,94,80,61,97,48,68)(47,95,66,74,52,54,63),
>    (1,35)(3,81)(4,92)(6,60)(7,59)(8,46)(9,70)(10,91)(11,18)(12,66)(13,55)
>        (14,85)(15,90)(17,53)(19,45)(20,68)(21,69)(23,84)(24,34)(25,31)(26,32)
>        (37,39)(38,42)(40,41)(43,44)(49,64)(50,63)(51,52)(54,95)(56,96)(57,100)
>        (58,97)(61,62)(65,82)(67,83)(71,98)(72,99)(74,77)(76,78)(87,89) >;
>  K := Classes(G:WeakLimit := 20, StrongLimit := 50);
Runtime error in 'Classes': Random classes algorithm failed
> K := Classes(G: WeakLimit := 20, StrongLimit := 100);
> NumberOfClasses(G);
24
```

As the group has only 24 classes, the first random search could have succeeded by looking at 50 elements. On this occasion it did not, but looking at 100 elements did succeed.
We print the order, length and cycle structure for each conjugacy class.

```
> [ < k[1], k[2], CycleStructure(k[3]) > : k in K ];
[
       <1, 1, [ <1, 100> ]>,
       <2, 5775, [ <2, 40>, <1, 20> ]>,
       <2, 15400, [ <2, 50> ]>,
       <3, 123200, [ <3, 30>, <1, 10> ]>,
       <4, 11550, [ <4, 20>, <2, 10> ]>,
       <4, 173250, [ <4, 20>, <2, 6>, <1, 8> ]>,
       <4, 693000, [ <4, 20>, <2, 8>, <1, 4> ]>,
       <5, 88704, [ <5, 20> ]>,
       <5, 147840, [ <5, 20> ]>,
       <5, 1774080, [ <5, 19>, <1, 5> ]>,
       <6, 1232000, [ <6, 15>, <2, 5> ]>,
       <6, 1848000, [ <6, 12>, <3, 6>, <2, 4>, <1, 2> ]>,
       <7, 6336000, [ <7, 14>, <1, 2> ]>,
       <8, 2772000, [ <8, 10>, <4, 4>, <2, 2> ]>,
       <8, 2772000, [ <8, 10>, <4, 3>, <2, 3>, <1, 2> ]>,
       <8, 2772000, [ <8, 10>, <4, 4>, <2, 2> ]>,
       <10, 2217600, [ <10, 8>, <5, 4> ]>,
       <10, 2217600, [ <10, 10> ]>,
       <11, 4032000, [ <11, 9>, <1, 1> ]>,
       <11, 4032000, [ <11, 9>, <1, 1> ]>,
       <12, 3696000, [ <12, 6>, <6, 3>, <4, 2>, <2, 1> ]>,
       <15, 2956800, [ <15, 6>, <5, 2> ]>,
       <20, 2217600, [ <20, 4>, <10, 2> ]>,
       <20, 2217600, [ <20, 4>, <10, 2> ]>
]
```

We construct the power map and tabulate the second, third and fifth powers of each class.

```
> p := PowerMap(G);
```

```
> [ < i, p(i, 2), p(i, 3), p(i, 5) > : i in [1 .. #K] ];
[
    <1, 1, 1, 1>,
    <2, 1, 2, 2>,
    <3, 1, 3, 3>,
    <4, 4, 1, 4>,
    <5, 2, 5, 5>,
    <6, 2, 6, 6>,
    <7, 2, 7, 7>,
    <8, 8, 8, 1>,
    <9, 9, 9, 1>,
    <10, 10, 10, 1>,
    <11, 4, 3, 11>,
    <12, 4, 2, 12>,
    <13, 13, 13, 13>,
    <14, 7, 14, 14>,
    <15, 6, 15, 15>,
    <16, 7, 16, 16>,
    <17, 8, 17, 2>,
    <18, 9, 18, 3>,
    <19, 20, 19, 19>,
    <20, 19, 20, 20>,
    <21, 12, 5, 21>,
    <22, 22, 9, 4>,
    <23, 17, 23, 5>,
    <24, 17, 24, 5>
]
```

58.8 Subgroups

58.8.1 Construction of a Subgroup

`sub< G | L >`

Given the permutation group G, construct the subgroup H of G, generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) A sequence of n integers defining a permutation of G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic permutation group. The subgroup H will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of H consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list.

`ncl< G | L >`

Given the permutation group G, construct the subgroup H of G that is the *normal closure* of the subgroup H generated by the elements specified by the list L (see [BC82]), where the possibilities for L are the same as for the `sub`-constructor.

Example H58E14

The group PGL$(2,7)$ in its natural action on projective points is generated by the set of permutations $\{(1,2,3,4,5,6,7),(2,4,3,7,5,6),(1,8)(2,7)(3,4)(5,6)\}$. Using the above syntax, the group may be defined in any of the following ways:
(a) By means of a list of generating permutations written as products of cycles:

```
> PGL27 := sub< Sym(8) | (1,2,3,4,5,6,7), (2,4,3,7,5,6), (1,8)(2,7)(3,4)(5,6)>;
> PGL27;
Permutation group PGL27 acting on a set of cardinality 8
    (1, 2, 3, 4, 5, 6, 7)
    (2, 4, 3, 7, 5, 6)
    (1, 8)(2, 7)(3, 4)(5, 6)
```

(b) By means of a list of integer sequences representing generators:

```
> PGL27 := sub< Sym(8) |
>               [2,3,4,5,6,7,1,8], [1,4,7,3,6,2,5,8], [8,7,4,3,6,5,2,1] >;
```

(c) In terms of preassigned elements of the symmetric group of degree 8:

```
> S8 := Sym(8);
> a := S8!(1,2,3,4,5,6,7);
> b := S8!(2,4,3,7,5,6);
> c := S8!(1,8)(2,7)(3,4)(5,6);
> PGL27 := sub<S8 | a, b, c>;
```

(d) By means of a set of generators:

```
> S8 := Sym(8);
> gens := { S8 | (1,2,3,4,5,6,7), (2,4,3,7,5,6), (1,8)(2,7)(3,4)(5,6) };
> PGL27 := sub<S8 | gens>;
```

(e) By means of a sequence of generators:

```
> S8 := Sym(8);
> gens := [ S8 | (1,2,3,4,5,6,7), (2,4,3,7,5,6), (1,8)(2,7)(3,4)(5,6) ];
> PGL27 := sub<S8 | gens>;
```

Example H58E15

A representation H of a 2-generator transitive group G in its action on unordered pairs is constructed as follows:

```
> G := AlternatingGroup(7);
> deg1 := Degree(G);
> pairs := [ { i, j } : j in [i+1..deg1], i in [1..deg1-1] ];
> deg2 := #pairs;
> h1 := [ Position(pairs, pairs[i] ^ G.1): i in [1..deg2] ];
> h2 := [ Position(pairs, pairs[i] ^ G.2): i in [1..deg2] ];
> H := sub<Sym(deg2) | h1, h2>;
> H;
Permutation group H acting on a set of cardinality 21
 (2,3,4,5,6)(7,8,9,10,11)(12,16,19,21,15)(13,17,20,14,18),
 (1,7,2)(3,8,12)(4,9,13)(5,10,14)(6,11,15)
```

Example H58E16

We illustrate the `ncl`-constructor by using it to construct the derived subgroup of the Hessian group H. We exploit the fact that the derived subgroup may be obtained as the normal closure of the subgroup generated by the commutators of the generators of H.

```
> H := PermutationGroup< 9 | (1,2,4)(5,6,8)(3,9,7), (4,5,6)(7,9,8) >;
> Order(H);
216
> D := ncl< H | (H.1, H.2) >;
> D;
Permutation group D acting on a set of cardinality 9
Order = 72 = 2^3 * 3^2
    (1, 7, 3, 6)(4, 5, 9, 8)
    (2, 9, 3, 5)(4, 6, 7, 8)
    (2, 6, 3, 8)(4, 5, 7, 9)
```

58.8.2 Membership and Equality

`g in G`

Given a permutation g and a permutation group G, return `true` if g is an element of G, `false` otherwise.

`g notin G`

Given a permutation g and a permutation group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G`

Given a permutation group G and a set S of permutations belonging to a group H, where G and H belong the same generic group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G`

Given a permutation group G and a set S of permutations belonging to a group H, where G and H belong the same generic group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G`

Given permutation groups G and H belonging to the same generic group, return `true` if H is a subgroup of G, `false` otherwise.

`H notsubset G`

Given permutation groups G and H belonging to the same generic group, return `true` if H is not a subgroup of G, `false` otherwise.

`H eq G`

Given permutation groups G and H belonging to the same generic group, return `true` if G and H are the same group, `false` otherwise.

`H ne G`

Given permutation groups G and H belonging to the same generic group, return `true` if G and H are distinct groups, `false` otherwise.

58.8.3 Elementary Properties of a Subgroup

`Index(G, H)`

The index of the subgroup H in the group G. The index is returned as an integer. If the orders of G and H are not known, they will be computed.

`FactoredIndex(G, H)`

The index of the subgroup H in the group G. The index is returned as a factored integer. The format is the same as for `FactoredOrder`. If the orders of G and H are not known, they will be computed.

`IsCentral(G, H)`

Returns `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

`IsNormal(G, H)`

Returns `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

`IsSelfNormalizing(G, H)`

`IsSelfNormalising(G, H)`

Returns `true` if the subgroup H of the group G is self-normalizing in G, `false` otherwise.

`IsSubnormal(G, H)`

Returns `true` if the subgroup H of the group G is subnormal in G, `false` otherwise.

58.8.4 Standard Subgroups

Unless the order is already known, each of the functions in this family will create a base and strong generating set for the group if one does not already exist.

`H ^ g`

`Conjugate(H, g)`

Construct the conjugate $g^{-1} * H * g$ of the permutation group H by the permutation g. The group H and the element g must belong to the same symmetric group.

`H meet K`

Given groups H and K which belong to the same symmetric group, construct the intersection of H and K. The intersection is found using the backtrack search of J. Leon [Leo97].

`IntersectionWithNormalSubgroup(G, N)`

`Check` BoolElt *Default :* `true`

Given groups G and N which belong to the same symmetric group and so that G normalises N, construct the intersection of G and N. The algorithm used is that of Cooperman, Finkelstein and Luks [CFL89], which uses a permutation representation of double the degree of G and N. Setting `Check` to `false` suppresses checking that G normalises N.

`CommutatorSubgroup(G, H, K)`

`CommutatorSubgroup(H, K)`

Given groups H and K, both subgroups of the group G, construct the commutator subgroup of H and K in the group G. If K is a subgroup of H, then the group G may be omitted. The algorithm used is described in [BC82].

`Centralizer(G, g: parameters)`

`Centraliser(G, g: parameters)`

Construct the centralizer of the permutation g in the group G; g and G must belong to a common symmetric group. A backtrack search through G as described in [Leo97] is employed.

`Subgroup`	GrpPerm	*Default :*

The parameter `Subgroup` may be used to supply a known subgroup of the centralizer. This may speed the search.

`Centralizer(G, H)`

`Centraliser(G, H)`

Construct the centralizer of the group H in the group G; G and H must belong to a common symmetric group. A backtrack search through G as described in [Leo97] is employed.

`CentralizerOfNormalSubgroup(G, H)`

Given G and H, belonging to a common symmetric group, with the restriction that H is a normal subgroup of G, construct the centralizer of H in G. A polynomial-time reduction algorithm described in Beals [Bea93] is used.

`SectionCentraliser(G, H, K)`

`SectionCentralizer(G, H, K)`

Return the full preimage in G of the centralizer in G/K of H/K. H and K must be normal subgroups of G with K contained in H. An algorithm of Luks [Luk93] is employed which involves computing the core of a subgroup in a group having twice the degree of G.

`Core(G, H)`

Given a subgroup H of the permutation group G, construct the maximal normal subgroup of G that is contained in the subgroup H. The algorithm employs repeated conjugation and intersection using the backtrack search of Leon [Leo97].

`H ^ G`

`NormalClosure(G, H)`

Given a subgroup H of the permutation group G, construct the normal closure of H in G.

`Normalizer(G, H: parameters)`

`Normaliser(G, H: parameters)`

`Subgroup`	GrpPerm	*Default :* H
`Bound`	RngIntElt	*Default :*

Given a subgroup H of the group G, construct the normalizer of H in G. A backtrack search as described in Leon [Leo97] is employed.

The parameter `Subgroup` may be used to pass the search a known subgroup of the normalizer. The default value of the starting subgroup is H. If `Bound` is set, the search will be terminated once the normalizing group found has order at least equal to `Bound`. If this does not happen, the search will complete as normal.

`SymmetricNormalizer(G)`

`Subgroup`	GrpPerm	*Default :* H
`Bound`	RngIntElt	*Default :*

`SymmetricNormaliser(G)`

Given a permutation group G acting on the set X, return the normalizer of G in the symmetric group on X. The parameters are as for `Normalizer` above.

`SylowSubgroup(G, p)`

`Sylow(G, p)`

Given a group G and a prime p, construct a Sylow p-subgroup of G. The algorithm used is that of Cannon, Cox and Holt [CCH97].

Example H58E17

We illustrate the use of these functions by applying them to a group of degree 30.

```
> M := PermutationGroup< 30 |
>         (1,2,3)(4,14,8)(5,15,9)(6,13,7)(25,27,26),
>         (4,20,13)(5,21,14)(6,19,15)(16,17,18)(27,28,29),
>         (1, 15)(2, 13)(3, 14)(4, 22)(5, 23)(6, 24)(7, 18)(8, 16)
>           (9, 17)(10, 21)(11, 19)(12, 20)(25, 29)(26, 27)(28, 30) >;
> FactoredOrder(M);
[ <2, 8>, <3, 10>, <5, 1> ]
> S2 := SylowSubgroup(M, 2);
> S2;
Permutation group S2 acting on a set of cardinality 30
Order = 256 = 2^8
    (1, 10)(2, 11)(3, 12)(4, 8)(5, 9)(6, 7)(13, 19)(14, 20)(15, 21)
        (16, 22)(17, 23)(18, 24)
    (1, 24)(2, 22)(3, 23)(4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)
        (10, 18)(11, 16)(12, 17)
    (4, 8)(5, 9)(6, 7)(13, 19)(14, 20)(15, 21)
    (4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)(25, 26)(29, 30)
```

```
    (1, 4)(2, 5)(3, 6)(7, 12)(8, 10)(9, 11)(13, 23)(14, 24)(15, 22)
        (16, 21)(17, 19)(18, 20)(25, 26)
    (27, 28)(29, 30)
    (27, 29)(28, 30)
    (25, 26)(29, 30)
```

We try to find a second Sylow subgroup $S2a$ that has trivial intersection with $S2$.

```
> b := exists(t){ x : x in M | Order(S2 meet S2^x) eq 1 };
> b;
true
> S2a := S2^t;
> N := Normalizer(M, S2);
> N;
Permutation group N acting on a set of cardinality 30
Order = 768 = 2^8 * 3
    (4, 8)(5, 9)(6, 7)(13, 19)(14, 20)(15, 21)
    (4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)
    (1, 10)(2, 11)(3, 12)(4, 8)(5, 9)(6, 7)(13, 19)(14, 20)(15, 21)
        (16, 22)(17, 23)(18, 24)
    (1, 24)(2, 22)(3, 23)(4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)
        (10, 18)(11, 16)(12, 17)
    (1, 22, 12)(2, 23, 10)(3, 24, 11)(4, 21, 13)(5, 19, 14)(6, 20, 15)
        (7, 8, 9)(16, 17, 18)
    (27, 29)(28, 30)
    (1, 14, 24, 4)(2, 15, 22, 5)(3, 13, 23, 6)(7, 12, 19, 17)(8, 10, 20, 18)
        (9, 11, 21, 16)(29, 30)
    (4, 14)(5, 15)(6, 13)(7, 19)(8, 20)(9, 21)(25, 26)(29, 30)
    (27, 28)(29, 30)
```

Thus the Sylow 2-subgroup is normalized by an element of order 3.

58.8.5 Maximal Subgroups

`IsMaximal(G, H: parameters)`

`Al` MonStgElt *Default :* "*Subgroups*"

Returns `true` if the subgroup H of the group G is a maximal subgroup of G. The algorithm used depends on the value of the parameter `Al`. The default value `Subgroups` computes the maximal subgroups of G if the index of H in G is over 1000 and the maximal subgroups are computable.The subgroup H is then tested for conjugacy with each class found. In the other cases, or when the `Al` parameter is set to `CosetImage`, the function is evaluated by first calling `IsProbablyMaximal` and if that returns `true` then constructing the permutation representation of G on the cosets of H and testing this representation for primitivity.

IsProbablyMaximal(G, H: *parameters*)

Tries RngIntElt *Default :* 20

Given a group G and a subgroup H of G, this function performs a probabilistic test for the maximality of H in G. The test involves adjoining random elements of G to H and determining if the result G. If not, then `false` is returned, otherwise `true` s returned. The number of random elements used is controlled by the parameter `Tries`, which is set to 20 by default.

MaximalSubgroups(G: *parameters*)

Construct the sequence of maximal subgroup classes of G. This is equivalent to the command `Subgroups(G: Al := "Maximal")`. The same parameters as for `Subgroups` are available to limit the search. The algorithm is described in [CH04].

Example H58E18

The `Subgroups` family of commands can deal with fairly large groups. We look at the maximal subgroups of the group of the $4 \times 4 \times 4$ Rubik's cube. This group has order about 1.7×10^{55}.

```
> load rubik444;
Loading "/home/magma/libs/pergps/rubik444"
The automorphism group of the 4 x 4 x 4 Rubik cube.
The group is represented as a permutation group of degree 72.
Its order is
2^50 * 3^29 * 5^9 * 7^7 * 11^4 * 13^2 * 17^2 * 19^2 * 23^2.
Group: G
> time max := MaximalSubgroups(G);
Time: 100.559
> #max;
46
> [Index(G, x`subgroup) : x in max];
[ 51090942171709440000, 51090942171709440000, 9161680528000,
9161680528000, 4509264634875, 4509264634875, 316234143225,
316234143225, 96197645544, 96197645544, 1577585295,
1577585295, 2496144, 2496144, 1961256, 1961256, 1307504,
1307504, 346104, 346104, 42504, 42504, 2187, 1352078,
1352078, 735471, 735471, 134596, 134596, 10626, 10626, 2024,
2024, 120, 276, 276, 56, 24, 24, 105, 28, 8, 35, 2, 2, 2 ]
```

58.8.6 Conjugacy Classes of Subgroups

`SubgroupClasses(G: parameters)`

`Subgroups(G: parameters)`

Representatives for the conjugacy classes of subgroups for the group G. The subgroups are returned as a sequence of records where the i-th record contains:

(a) A representative subgroup H for the i-th conjugacy class (field name `subgroup`).

(b) The order of the subgroup (field name `order`).

(c) The number of subgroups in the class (field name `length`).

(d) [Optional] A presentation for H (field name `presentation`).

`Al` MonStgElt *Default :* "*All*"

`Al := "All"`: Construct all subgroups of G.
`Al := "Maximal"`: Only construct maximal subgroups of G. This option reduces the number of intersections with any elementary abelian layer that need be considered and eliminates the need to recursively apply the algorithm.
`Al := "Normal"`: Only construct normal subgroups of G. This option does not use database lookup to find the normal subgroups of the radical quotient of G and also reduces the number of intersections with any layer that need be considered.

`LayerSizes` SeqEnum *Default :* See below

`LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]` is equivalent to the default. When constructing an Elementary Abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1.

`Series` SeqEnum *Default :* See below

Use the given elementary abelian series rather than constructing the default series. The first subgroup in the series must be the solvable radical of G. The subgroups must form a descending chain of normal subgroups of G, such that each quotient is elementary abelian. The last subgroup in the series must be either elementary abelian or trivial.

`Presentation` BoolElt *Default :* `false`

`Presentation := true`: Construct a presentation for each subgroup.

`OrderEqual` RngIntElt *Default :*

`OrderEqual := n`: Only construct subgroups having order equal to n.

`OrderDividing` RngIntElt *Default :*

`OrderDividing := n`: Only construct subgroups having order dividing n.

`OrderMultipleOf` RngIntElt *Default :*

`OrderMultipleOf := n`: Only construct subgroups having order a multiple of n.

`IndexLimit` RngIntElt *Default :*

`IndexLimit := n`: Only construct subgroups having index in G less than or equal to n.

`IsElementaryAbelian` BoolElt *Default :* `false`

`IsElementaryAbelian := true`: Only construct elementary abelian subgroups of G.

`IsCyclic` BoolElt *Default :* `false`

`IsCyclic := true`: Only construct cyclic subgroups of G.

`IsAbelian` BoolElt *Default :* `false`

`IsAbelian := true`: Only construct abelian subgroups of G.

`IsNilpotent` BoolElt *Default :* `false`

`IsNilpotent := true`: Only construct nilpotent subgroups of G.

`IsSolvable` BoolElt *Default :* `false`

`IsSolvable := true`: Only construct solvable subgroups of G.

`IsNotSolvable` BoolElt *Default :* `false`

`IsNotSolvable := true`: Only construct insolvable subgroups of G.

`IsPerfect` BoolElt *Default :* `false`

`IsPerfect := true`: Only construct perfect subgroups of G.

`IsRegular` BoolElt *Default :* `false`

`IsRegular := true`: Only construct regular subgroups of G.

`IsTransitive` BoolElt *Default :* `false`

`IsTransitive := true`: Only construct transitive subgroups of G.

The Algorithm: (See Cannon, Cox and Holt [CCH01]) This command proceeds by first constructing an elementary abelian series for G together with G's radical quotient Q. We first attempt to locate the quotient in a database of groups with trivial Fitting subgroup. This database contains all such groups of order up to 216 000, and all such which are perfect of order up to 1 000 000. If Q is found then either all its subgroups, or its maximal subgroups are read from the database. (In some cases only the maximal subgroups are stored.) If Q is not found then we attempt to find the maximal subgroups of Q using a method of Derek Holt. For this to succeed all simple factors of the socle of Q must be found in a second database which currently contains all simple groups of order less than 1.6×10^7, as well as M_{24}, HS, J_3, McL, $Sz(32)$ and $L_6(2)$. There are also special routines to handle numerous other groups. These include: A_n for $n \leq 999$, $L_2(q)$, $L_3(q)$, $L_4(q)$, $L_5(q)$, $L_6(q)$ and $L_7(q)$ for all q, $U_3(q)$ for q prime and $q = 8, 9, 16, 25$, $U_4(q)$ for $q = 4, 5, 7$, $S_4(q)$ for all odd q and even $q \leq 16$, $L_d(2)$ for $d \leq 14$, and the following groups: $U_6(2)$, $S_8(2)$, $S_{10}(2)$, $O_8^{\pm}(2)$, $O_{10}^{\pm}(2)$, $S_6(3)$, $O_7(3)$, $O_8^{-}(3)$, $G_2(4)$, $G_2(5)$, ${}^3D_4(2)$, ${}^2F_4(2)'$, Co_2, Co_3, He, Fi_{22}.

If we have only maximal subgroups of Q, and more are required, we apply the algorithm recursively to the maximal subgroups to determine all subgroups of Q. This may take some time.

The subgroups are then extended to the whole group by stepwise extension through each layer of the elementary abelian series. For each layer this involves determining all possible intersections of a subgroup with this layer and all extensions with this intersection.

The limitations are that the simple factors of the socle of Q must be in the database, which is limited as above. Further, it may take some time to construct all subgroups from the maximal subgroups first found, and, if there is a large elementary abelian layer, there will be many possible intersections, which could also make the algorithm prohibitively slow.

There are numerous parameters for this function which allow the user to place restrictions on which subgroup classes are constructed. Using these restrictions may help overcome the problems noted above.

`SubgroupsLift(G, A, B, Q`: *parameters*`)`

This function isolates one step of the extension process used by the `Subgroups` family of functions. Q is a sequence of records such as returned by `Subgroups`(G). A and B are normal subgroups of G with A/B elementary abelian. The records in Q are interpreted as subgroups of G/A, which are lifted to all possible corresponding subgroups of G/B, subject to the parameters given.

`LowIndexSubgroups(G, n`: *parameters*`)`

`LowIndexSubgroups(G, t`: *parameters*`)`

Returns a sequence of subgroups of G, each with index at most n. The sequence will contain one representative from each conjugacy class of G-subgroups satisfying the index constraint. The algorithm used is described in Cannon, Holt, Slattery & Steel [CHSS03].

The previous version of the algorithm is available by setting the parameter `Algorithm` to the string `"Subgroups"`. In this case the group G is subject to the same restrictions as the group input to the `Subgroups` function above.

In the second form t should be a pair of integers $\langle a, b\rangle$, and subgroups with index in the interval $[a, b]$ will be returned.

Other parameters are `Presentation` which may be set `true` to return a second sequence of presentations of the groups found, and `Print` which may be set to a positive integer to turn on diagnostic printing of the progress of the algorithms.

Example H58E19

With the `Subgroups` family of commands we can get the entire collection of (classes of) subgroups of a group G. We look at the double cover of M_{12}.

```
> load m12cover;
Loading "/home/magma/libs/pergps/m12cover"
```

```
The two-fold cover of the Mathieu group M12 on 24 letters.
Order is 190080 = 2^7 * 3^3 * 5 * 11.
Group: G
> time s := SubgroupClasses(G);
Time: 4.469
> #s;
293
```

This may be too many. The parameters allow us to restrict attention to a subset of the subgroups. We specify that the function is to return only the elementary abelian 2-subgroups of G.

```
> se := SubgroupClasses(G : IsElementaryAbelian := true,
>               OrderMultipleOf := 2);
> #se;
14
> se : Minimal;
Conjugacy classes of subgroups
------------------------------

[ 1]    Order 2            Length 1
        GrpPerm: $, Degree 24, Order 2
[ 2]    Order 2            Length 495
        GrpPerm: $, Degree 24, Order 2
[ 3]    Order 2            Length 495
        GrpPerm: $, Degree 24, Order 2
[ 4]    Order 4            Length 495
        GrpPerm: $, Degree 24, Order 2^2
[ 5]    Order 4            Length 495
        GrpPerm: $, Degree 24, Order 2^2
[ 6]    Order 4            Length 1485
        GrpPerm: $, Degree 24, Order 2^2
[ 7]    Order 4            Length 1980
        GrpPerm: $, Degree 24, Order 2^2
[ 8]    Order 4            Length 5940
        GrpPerm: $, Degree 24, Order 2^2
[ 9]    Order 8            Length 495
        GrpPerm: $, Degree 24, Order 2^3
[10]    Order 8            Length 495
        GrpPerm: $, Degree 24, Order 2^3
[11]    Order 8            Length 1485
        GrpPerm: $, Degree 24, Order 2^3
[12]    Order 8            Length 1980
        GrpPerm: $, Degree 24, Order 2^3
[13]    Order 8            Length 1980
        GrpPerm: $, Degree 24, Order 2^3
[14]    Order 16           Length 495
        GrpPerm: $, Degree 24, Order 2^4
```

Example H58E20

Using the `SubgroupLattice` function we obtain a representative subgroup for each conjugacy class together with the inclusion relations between subgroups.
WARNING: *Computing the inclusions is very time consuming and should only be performed for small groups.*

```
> G := PSL(2,9);
> time L := SubgroupLattice(G);
Time: 0.200
> L;
Partially ordered set of subgroup classes
-----------------------------------------
[ 1]  Order 1    Length 1   Maximal Subgroups:
---
[ 2]  Order 2    Length 45  Maximal Subgroups: 1
[ 3]  Order 3    Length 20  Maximal Subgroups: 1
[ 4]  Order 3    Length 20  Maximal Subgroups: 1
[ 5]  Order 5    Length 36  Maximal Subgroups: 1
---
[ 6]  Order 4    Length 15  Maximal Subgroups: 2
[ 7]  Order 4    Length 15  Maximal Subgroups: 2
[ 8]  Order 4    Length 45  Maximal Subgroups: 2
[ 9]  Order 6    Length 60  Maximal Subgroups: 2 3
[10]  Order 6    Length 60  Maximal Subgroups: 2 4
[11]  Order 9    Length 10  Maximal Subgroups: 3 4
[12]  Order 10   Length 36  Maximal Subgroups: 2 5
---
[13]  Order 8    Length 45  Maximal Subgroups: 6 7 8
[14]  Order 12   Length 15  Maximal Subgroups: 4 6
[15]  Order 12   Length 15  Maximal Subgroups: 3 7
[16]  Order 18   Length 10  Maximal Subgroups: 9 10 11
---
[17]  Order 24   Length 15  Maximal Subgroups: 10 13 14
[18]  Order 24   Length 15  Maximal Subgroups: 9 13 15
[19]  Order 36   Length 10  Maximal Subgroups: 8 16
[20]  Order 60   Length 6   Maximal Subgroups: 9 12 15
[21]  Order 60   Length 6   Maximal Subgroups: 10 12 14
---
[22]  Order 360  Length 1   Maximal Subgroups: 17 18 19 20 21
> NumberOfInclusions(L!5, L!20);
6
> L[5];
Permutation group acting on a set of cardinality 10
Order = 5
  (1, 8, 9, 3, 4)(2, 7, 5, 10, 6)
```

The order and class length of each class of subgroups is listed, along with the information about where to find the maximal subgroups of a member of this class. Further information about

inclusions is available from the lattice. We see that 6 members of class 5 are contained in any fixed member of class 20.

58.8.7 Classes of Subgroups Satisfying a Condition

`NormalSubgroups(G: parameters)`

Construct the sequence of normal subgroup classes of G. This is equivalent to `Subgroups(G: Al := "Normal")`. The same parameters as for `Subgroups` are available to limit the search.

`ElementaryAbelianSubgroups(G: parameters)`

Construct the sequence of elementary abelian subgroups of G. This is equivalent to `Subgroups(G: IsElementaryAbelian := true)`. The same parameters as for `Subgroups` are available to limit the search.

`CyclicSubgroups(G: parameters)`

Construct the sequence of cyclic subgroups of G. This is equivalent to `Subgroups(G: IsCyclic := true)`. The same parameters as for `Subgroups` are available to limit the search.

`AbelianSubgroups(G: parameters)`

Construct the sequence of abelian subgroups of G. Equivalent to `Subgroups(G: IsAbelian := true)`. The same parameters as for `Subgroups` are available to limit the search.

`NilpotentSubgroups(G: parameters)`

Construct the sequence of nilpotent subgroups of G. This is equivalent to `Subgroups(G: IsNilpotent := true)`. The same parameters as for `Subgroups` are available to limit the search.

`SolvableSubgroups(G: parameters)`

Construct the sequence of solvable subgroups of G. This is equivalent to `Subgroups(G: IsSolvable := true)`. The same parameters as for `Subgroups` are available to limit the search.

`PerfectSubgroups(G: parameters)`

Construct the sequence of perfect subgroups of G. Equivalent to `Subgroups(G: IsNotSolvable := true)`. The same parameters as for `Subgroups` are available to limit the search.

`NonsolvableSubgroups(G: parameters)`

Construct the sequence of insolvable subgroups of G. This is equivalent to `Subgroups(G: IsNotSolvable := true)`. The same parameters as for `Subgroups` are available to limit the search.

`SimpleSubgroups(G: parameters)`

Construct the sequence of non-abelian simple subgroup classes of G. This is equivalent to `Subgroups(G: Al := "Simple")`. The same parameters as for `Subgroups` are available to limit the search.

58.9 Quotient Groups

58.9.1 Construction of Quotient Groups

`quo< G | L >`

Given the permutation group G, construct the quotient group $Q = G/N$, where N is the normal closure of the subgroup of G generated by the elements specified by L. The clause L is a list of one or more items of the following types:

(a) A sequence of n integers defining a permutation of G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic permutation group. The function returns

(a) the quotient group Q, and

(b) the natural homomorphism $f : G \to Q$.

Currently, the quotient group is constructed via the regular representation of the quotient, so the application of this operator is restricted to the case where the index of N in G is small. The representation of the quotient group that is returned is the result of a degree reduction applied to the regular representation, so need not be regular. The generators of the quotient are images of the generators of G.

The second return value is the epimorphism from G to the resulting quotient group.

`G / N`

Given a normal subgroup N of the permutation group G, construct the quotient of G by N. Currently, the quotient group is constructed via the regular representation of the quotient, so the application of this operator is restricted to the case where the index of N in G is small. The representation of the quotient group that is returned is the result of a degree reduction applied to the regular representation, so need not be regular. The generators of the quotient are images of the generators of G.

Example H58E21

The quotient of Sym(4) by the Klein 4-group is constructed by the following statement:

```
> Q, f := quo< Sym(4) | (1,2)(3,4), (1,3)(2,4) >;
> Q;
Permutation group Q acting on a set of cardinality 3
Order = 6 = 2 * 3
    (2, 3)
    (1, 2)
```

58.9.2 Abelian, Nilpotent and Soluble Quotients

A number of standard quotients may be constructed. The method first constructs a presentation for the permutation group and then applies the appropriate fp-group algorithm.

`AbelianQuotient(G)`

The maximal abelian quotient G/G' of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism $\pi : G \to G/G'$ is returned as second value.

`ElementaryAbelianQuotient(G, p)`

The maximal p-elementary abelian quotient Q of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism $\pi : G \to Q$ is returned as second value.

`pQuotient(G, p, c)`

Given a permutation group G, a prime p and a positive integer c, construct a pc-presentation for the largest p-quotient P of G having lower exponent-p class at most c. If c is given as 0, then the limit 127 is placed on the class.

The function also returns the natural homomorphism π from G to P, a sequence S describing the definitions of the pc-generators of P and a flag indicating whether P is the maximal p-quotient of G.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $P.k$ of P as follows.

- If $S[k] = [0, r]$, then $P.k$ is defined via the image of $G.r$ under π.
- If $S[k] = [r, 0]$, then $P.k$ is defined via the power relation for $P.r$.
- If $S[k] = [r, s]$, then $P.k$ is defined via the conjugate relation involving $P.r^{P.s}$.

`NilpotentQuotient(G, c)`

This function returns the class c nilpotent quotient of G, together with the epimorphism π from G onto this quotient.

`SolvableQuotient(G)`

`SolubleQuotient(G)`

The function returns the largest soluble quotient S of the permutation group G together with the epimorphism $\pi : G \to S$.

Example H58E22

The soluble quotient of the wreath product of Sym(6) with the dihedral group of order 12 is easily constructed:

```
> G := WreathProduct( Sym(6), DihedralGroup(6));
> #G;
1671768834048000000
> SQ, phi := SolubleQuotient(G);
SQ;
GrpPC : SQ of order 768 = 2^8 * 3
PC-Relations:
    SQ.1^2 = SQ.5,
    SQ.2^2 = Id(SQ),
    SQ.3^2 = Id(SQ),
    SQ.4^2 = Id(SQ),
    SQ.5^3 = Id(SQ),
    SQ.6^2 = Id(SQ),
    SQ.7^2 = Id(SQ),
    SQ.8^2 = Id(SQ),
    SQ.9^2 = Id(SQ),
    SQ.2^SQ.1 = SQ.2 * SQ.5,
    SQ.3^SQ.1 = SQ.3 * SQ.4 * SQ.6 * SQ.8,
    SQ.4^SQ.1 = SQ.4 * SQ.9,
    SQ.4^SQ.2 = SQ.4 * SQ.6 * SQ.7 * SQ.8,
    SQ.5^SQ.2 = SQ.5^2,
    SQ.5^SQ.3 = SQ.5 * SQ.7,
    SQ.5^SQ.4 = SQ.5 * SQ.6 * SQ.8,
    SQ.6^SQ.1 = SQ.6 * SQ.8,
    SQ.6^SQ.2 = SQ.7 * SQ.8,
    SQ.6^SQ.5 = SQ.6 * SQ.7 * SQ.8 * SQ.9,
    SQ.7^SQ.1 = SQ.8,
    SQ.7^SQ.2 = SQ.9,
    SQ.7^SQ.5 = SQ.7 * SQ.9,
    SQ.8^SQ.1 = SQ.7 * SQ.9,
    SQ.8^SQ.2 = SQ.6 * SQ.9,
    SQ.8^SQ.5 = SQ.6 * SQ.9,
    SQ.9^SQ.1 = SQ.6 * SQ.8 * SQ.9,
    SQ.9^SQ.2 = SQ.7,
    SQ.9^SQ.5 = SQ.7
```

58.10 Permutation Group Actions

58.10.1 G-Sets

Let G be a group. A *G-set* is a pair (Y, f), where Y is a set and $f : Y \times G \to Y$ is a mapping such that

$$(a) \qquad f(f(y, g), h) = f(y, gh), \quad \text{for all } g, h \in G$$

and

$$(b) \qquad f(y, 1) = y, \quad \text{for all } y \in Y.$$

The mapping f defines the *action* of G on the set Y.

If G is defined as a permutation group acting on the set X and Y is another G-set then there is a homomorphism of G^X into G^Y.

We distinguish three types of G-set for a permutation group G. The set on which G is defined will be referred to as the *natural G-set* and the action of G on X as the *natural action* of G.

Let A be a set. A *derived set* of A is defined (recursively) as follows:

(i) A subset of A is a derived set;

(ii) A set of k-subsets of A is a derived set;

(iii) A set of k-sequences of A is a derived set;

(iv) A set of ordered partitions of A is a derived set;

(v) A subset of a cartesian product of derived sets of A is a derived set.

The natural action of G on X induces a natural action on the G-closure Y of any derived set of X. Such a set Y is also a G-set. For example, a subset of X is a G-set for G if and only if it is a union of orbits for G.

Finally, a *general G-set* is an arbitrary set Y with an action f satisfying the conditions (a) and (b).

The notion of a G-set enables the user to work with several different actions of G. Rather than having to always work with the image of G with respect to an action on a set Y, the user may specify the required operation in terms of G.

58.10.2 Creating a G-Set

`GSetFromIndexed(G, Y)`

Given a group G and an indexed set Y with the same cardinality as the natural G-set, return a G-set corresponding to the natural bijection between the labelling L (= Labelling(G)) of G and Y. Explicitly, the bijection is

$$\phi : L \to Y : l \mapsto Y[\text{Position}(L, l)].$$

Then the returned G-set is the set Y endowed with the action

$$f : Y \times G \to Y : (y, p) \mapsto \phi(p(\phi^{-1}(y))).$$

`GSet(G, X, Y)`

`GSet(G, Y)`

Return the smallest derived G-set containing Y as a subset under the action of G on X. If X is omitted, then the natural action will be assumed. In practice, the set Y is expanded until for each element y of the expanded Y, the image of y under each generator of G under the action described by X is also in Y. The action of G on Y is then the action induced by the action of G on X.

`GSet(G)`

Given a permutation group G, return the G-set corresponding to the natural action of G.

`GSet(G, Y, f)`

Construct the smallest G-set containing Y as a subset with the given action f. The map f must satisfy the requirements of a G-set action. In particular, the domain of f must be a superset of $Y \times G$, the codomain a superset of Y and the two conditions listed at the beginning of this section must be met.

`Action(Y)`

The map giving the action of the group on the G-set Y.

`Group(Y)`

The group associated with the G-set Y.

`Labelling(G)`

Given a permutation group G of degree n, return an indexed set giving the internal mapping of the natural G-set of G onto the set $\{1, \ldots, n\}$, where n is the degree of G.

`Degree(g, Y)`

`Degree(g)`

Given an element g of a permutation group G and a G-set Y, return the cardinality of the subset of Y consisting of points that are moved by g. If Y is omitted, the natural G-set X is assumed.

`Degree(G, Y)`

`Degree(G)`

Given a G-set Y, return the cardinality of Y. If Y is omitted, the natural G-set X is assumed.

`Support(g, Y)`

`Support(g)`

Given an element g of a permutation group G and a G-set Y, return the subset of Y consisting of points that are moved by g. If Y is omitted, the natural G-set X is assumed.

`Support(G, Y)`

`Support(G)`

Given a permutation group G and a G-set Y, return the subset of Y consisting of points that are moved by at least one element of G. If Y is omitted the natural G-set for G is assumed.

Example H58E23

We construct a G-set with a user defined action. Our example will take a group G and a normal subgroup N of index 4. The G-set will be the irreducible characters of N, with the usual G action obtained from permuting the elements of N by conjugation. As this is not a derived set we will define the action via a map.

```
> G := PGammaL(2, 9);
> N := PSL(2, 9);
> CT := CharacterTable(N);
> X := SequenceToSet(CT);
> XxG := CartesianProduct(X, G);
> f := map< XxG -> X | x :-> x[1]^x[2] >;
> Y := GSet(G, X, f);
```

This defines our G-set `Y`. The inertia group of a character is its stabilizer in this action. Let us compute an inertia group.

```
> chi := CT[2];
> I := Stabilizer(G, Y, chi);
> Index(G, I);
2
> [#o : o in Orbits(G, Y)];
[ 1, 1, 1, 2, 2 ]
```

We find that two of the characters have inertia groups of index 2 in G, while three are G-invariant.

58.10.3 Images, Orbits and Stabilizers

`x ^ g`

Given a permutation group G with natural G-set X and an object x which is an element of some derived G-set of X, find the image of x under G.

`Image(g, Y, y)`

`Image(g, y)`

Given a permutation g belonging to a group G, a G-set Y, and an element y of Y, find the image of y under g. If y is an element of some derived G-set of G, the set Y may be omitted.

`Fix(g, Y)`

`Fix(g)`

Given a permutation g belonging to a group G and a G-set Y, construct the fixed-point set of g in its action on Y. In the case in which Y is the natural G-set, Y may be omitted. The fixed-point set is returned as a subset of points of Y.

`Fix(G, Y)`

`Fix(G)`

The fixed-point set of the permutation group G in its action on the G-set Y (or the natural G-set for G if Y is omitted).

`x ^ G`

Given a permutation group G with natural G-set X and an element x belonging to some derived G-set of X, construct the orbit of x under G. The orbit is returned as a G-set.

`Cycle(e, x)`

Let e be an element of a permutation group defined as acting on a set containing x. Returns the set of images of x under repeated application of e as an indexed set with x the first element. This gives the cycle containing x in the disjoint cycle representation of e.

`CycleDecomposition(e)`

Let e be an element of a permutation group defined as acting on a set X. Returns a sequence of indexed sets partitioning X, each of which is a cycle of e. This gives the full disjoint cycle representation of e.

`Orbit(G, Y, y)`

`Orbit(G, y)`

Given a permutation group G, a G-set Y, and an element y belonging to Y, construct the orbit of y under G. The orbit is returned as a G-set. If y is an element of some derived G-set of G, the set Y may be omitted.

`Orbits(G, Y)`

`Orbits(G)`

Given a permutation group G and a G-set Y, construct the orbits of G on Y. If the set Y is omitted, the orbits of G on its natural G-set are constructed. The orbits are returned as a sequence of G-sets.

`OrbitRepresentatives(G)`

Given a permutation group G, construct the orbits of G on its natural G-set. The orbit descriptions are returned as a sequence of tuples $\langle l, r \rangle$ giving the length l and a representative r of each orbit of G on its support.

This function stores *only* the orbit representatives and so is more space-efficient than `Orbits`. However, it should be used only if the user wants to determine just the orbit representatives; queries about the orbits containing other elements of the support will cause further computation.

`OrbitClosure(G, Y, S)`

`OrbitClosure(G, S)`

Given a subset S of the G-set Y, construct the smallest G-invariant subset of Y that contains S. If Y is the natural G-set for G it may be omitted.

`IsConjugate(G, Y, y, z)`

`IsConjugate(G, y, z)`

Given elements y and z belonging either to a G-set Y or to a (restricted) derived set of Y, return the value `true` if there exists an element $g \in G$ such that $y^g = z$. Otherwise, return `false`. If such an element exists, then it is returned as the second value of the function. If y and z belong to the natural G-set, then Y may be omitted. Currently, y and z are restricted to being elements, sets of elements, multisets of elements, sequences of elements, ordered partitions, or unordered partitions of Y.

`Stabilizer(G, Y, y)`

`Stabiliser(G, Y, y)`

`Stabilizer(G, y)`

`Stabiliser(G, y)`

Given a permutation group G and a G-set Y, and an object y which is either an element, a sequence of elements, a set of elements, an ordered partition or a tuple over the G-set Y, find the stabilizer of y in G. The stabilizer is returned as a subgroup of G. If Y is the natural G-set, it may be omitted.

`IsPrimitive(G, Y)`

`IsPrimitive(G)`

Returns `true` if G acts primitively on the G-set Y. If Y is the natural G-set, the set Y may be omitted.

`IsTransitive(G, Y)`

`IsTransitive(G)`

Returns `true` if G acts transitively on the G-set Y. If Y is the natural G-set, the set Y may be omitted.

`IsTransitive(G, Y, k)`

`IsTransitive(G, k)`

Returns `true` if G acts k-transitively on the G-set Y. If Y is the natural G-set, the set Y may be omitted.

`IsSharplyTransitive(G, Y, k)`

`IsSharplyTransitive(G, k)`

Returns `true` if G acts sharply k-transitively on the G-set Y. If Y is the natural G-set, the set Y may be omitted.

`Transitivity(G, Y)`

`Transitivity(G)`

The degree of transitivity of G acting on the G-set Y. The set Y may be omitted if it is the same as the natural G-set.

`IsRegular(G, Y)`

`IsRegular(G)`

Returns `true` if G acts regularly on the G-set Y. If Y is the natural G-set, the set Y may be omitted. The algorithm used is that of Sims, see [CB92].

`IsSemiregular(G, Y)`

`IsSemiregular(G)`

Returns `true` if G acts semiregularly on the G-set Y. If Y is the natural G-set, the set Y may be omitted. The algorithm used is a variation of Sims' regularity test, see [CB92].

`IsSemiregular(G, Y, S)`

`IsSemiregular(G, S)`

Given a permutation group G, a G-set Y for G, and a union of orbits S for G in its action on Y, return `true` if G acts semiregularly on S. If Y is the natural G-set, then Y may be omitted.

`IsFrobenius(G)`

Returns `true` if the permutation group G is a Frobenius group with respect to its natural action, `false` otherwise. (A group G defined as acting on X is Frobenius if it acts transitively but non-regularly on X and if the pointwise stabilizer of any two distinct points of X is the trivial group.)

Example H58E24

We apply some of these functions to the Mathieu group M_{24}, taking as generators the following three permutations:

$$(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,24),$$

$$(2,16,9,6,8)(3,12,13,18,4)(7,17,10,11,22)(14,19,21,20,15),$$

$$(1,22)(2,11)(3,15)(4,17)(5,9)(6,19)(7,13)(8,20)(10,16)(12,21)(14,18)(23,24).$$

```
> M24 := sub< Sym(24) |
>  (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,24),
>  (2,16,9,6,8)(3,12,13,18,4)(7,17,10,11,22)(14,19,21,20,15),
>  (1,22)(2,11)(3,15)(4,17)(5,9)(6,19)(7,13)(8,20)(10,16)(12,21)(14,18)(23,24)>;
> M24;
Permutation group M24 acting on a set of cardinality 24
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
       21, 22, 24)
    (2, 16, 9, 6, 8)(3, 12, 13, 18, 4)(7, 17, 10, 11, 22)(14, 19, 21, 20, 15)
    (1, 22)(2, 11)(3, 15)(4, 17)(5, 9)(6, 19)(7, 13)(8, 20)(10, 16)(12, 21)
       (14, 18)(23, 24)
```

Choosing a random element x of M_{24}, we use it to compute some images.

```
> x := Random(M24);
> 1^x;
7
> [1,2,3,4]^x;
```

```
[ 7, 9, 8, 17 ]
> { 1,2,3,4}^x;
{ 17, 7, 8, 9 }
```

We find the stabilizer of the point 1, which is the group M_{23}.

```
> S1 := Stabilizer(M24, 1);
> S1;
Permutation group S1 acting on a set of cardinality 24
Order = 10200960 = 2^7 * 3^2 * 5 * 7 * 11 * 23
    (2, 16, 9, 6, 8)(3, 12, 13, 18, 4)(7, 17, 10, 11, 22)(14, 19, 21, 20, 15)
    (7, 17, 22)(8, 11, 13)(9, 14, 12)(10, 20, 19)(15, 23, 18)(16, 21, 24)
    (3, 6, 18)(5, 16, 14)(7, 21, 22)(8, 19, 17)(9, 20, 24)(11, 12, 13)
    (6, 18, 15)(7, 19, 16)(8, 13, 11)(9, 10, 22)(12, 21, 20)(14, 17, 24)
    (4, 12, 6, 19)(5, 22, 24, 8)(7, 17, 20, 14)(9, 15, 13, 18)(10, 21)(11, 16)
    (6, 22, 7)(8, 13, 11)(9, 20, 16)(10, 18, 21)(12, 15, 19)(14, 24, 23)
    (5, 12, 21)(6, 15, 18)(7, 22, 8)(9, 16, 17)(10, 14, 13)(11, 24, 19)
```

We next compute the stabilizer of the sequence $[1, 2, 3, 4, 5]$.

```
> SQ := Stabilizer(M24, [1,2,3,4,5]);
> SQ;
Permutation group SQ acting on a set of cardinality 24
Order = 48 = 2^4 * 3
    (6, 18, 15)(7, 19, 16)(8, 13, 11)(9, 10, 22)(12, 21, 20)(14, 17, 24)
    (7, 17, 22)(8, 11, 13)(9, 14, 12)(10, 20, 19)(15, 23, 18)(16, 21, 24)
    (6, 22, 7)(8, 13, 11)(9, 20, 16)(10, 18, 21)(12, 15, 19)(14, 24, 23)
> Orbits(SQ);
[
    GSet{@ 1 @},
    GSet{@ 2 @},
    GSet{@ 3 @},
    GSet{@ 4 @},
    GSet{@ 5 @},
    GSet{@ 6,18, 22, 15, 21, 9, 7, 23, 19, 20, 24, 10,
    14, 17, 16 12 @},
    GSet{@ 8, 13, 11 @}
]
```

The five fixed points together with the orbit of length 3 form a block of a $5-(24, 8, 1)$ design. By computing the orbit of this block under M_{24}, we obtain all the blocks of the design.

```
> B := { 1,2,3,4,5,8,11,13};
> D := B^M24;
> #D;
759
```

Finally, we check that the set stabilizer of the block $\{ 1, 2, 3, 4, 5, 8, 11, 13\}$ has index 759 in M_{24}.

```
> Index(M24, Stabilizer(M24, { 1,2,3,4,5,8,11,13}));
759
```

58.10.4 Action on a G-Space

`Action(G, Y)`

Given a permutation group G defined to be acting on X and a set Y, construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. The function returns:

(a) The natural homomorphism $\phi : G \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of G).

`ActionImage(G, Y)`

Given a permutation group G defined to be acting on X and a set Y, construct the permutation group L giving the action of G on the set Y.

`ActionKernel(G, Y)`

Construct the kernel of the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the G-set Y.

`IsFaithful(G, Y)`

Returns `true` if the action of G on the G-set Y is faithful.

Example H58E25

We take the group PSL(3, 4) acting on projective points and construct its representation on flags (point-line pairs). In order to construct the flags, we need to find a line. If H is the stabilizer of a point α in PSL(3, 4) in its action on projective points, then a line consists of α together with the points in any non-trivial orbit of $O_2(G)$.

```
> G := ProjectiveSpecialLinearGroup(3, 4);
> O2 := pCore( Stabilizer(G, 1), 2 );
> O2;
Permutation group O2 acting on a set of cardinality 21
Order = 16 = 2^4
       (3, 4)(5, 7)(9, 16)(10, 17)(11, 15)(13, 18)(14, 19)(20, 21)
       (3, 20)(4, 21)(5, 15)(7, 11)(9, 10)(13, 19)(14, 18)(16, 17)
       (2, 8)(5, 15)(6, 12)(7, 11)(9, 17)(10, 16)(13, 18)(14, 19)
       (2, 12)(5, 11)(6, 8)(7, 15)(9, 16)(10, 17)(13, 19)(14, 18)
> flag := < 1, Orbit(O2, 2) >;
> flag;
<1, GSet{@ 2, 6, 8, 12 @}>
> flags := GSet(G, Orbit(G, flag));
> #flags;
105
> GOnFlags := ActionImage(G, flags);
> GOnFlags;
Permutation group GOnFlags acting on a set of
```

```
cardinality 105
Order = 20160 = 2^6 * 3^2 * 5 * 7
> Stabilizer(GOnFlags, Rep(flags));
Permutation group acting on a set of cardinality 105
Order = 192 = 2^6 * 3
```

58.10.5 Action on Orbits

The operations described here are concerned with the class of G-sets consisting of G-invariant subsets of the natural G-set. Because of the special nature of such G-sets, more efficient algorithms are available for computing with homomorphisms of G induced by the action of G on such a G-set. See Butler [But85] for more details.

`OrbitAction(G, T)`

The homomorphism $f : G \to L$ induced by the action of G on the G-invariant subset T of X (a union of orbits).

`OrbitImage(G, T)`

The group L defined by the action of G on the G-invariant subset T of X (a union of orbits).

`OrbitKernel(G, T)`

The kernel of the homomorphism $f : G \to L$, where the group L gives the action of G on the G-invariant subset T of X (a union of orbits).

`IsOrbit(G, S)`

Returns `true` if the subset S of Support(G) is invariant under G.

Example H58E26

We study an intransitive group of degree 36 generated by the permutations

$$(3, 17, 26)(4, 16, 25)(5, 18, 27)(8, 15, 24),$$
$$(1, 32, 10)(2, 31, 11)(3, 35, 12)(6, 30, 15),$$
$$(12, 33, 24)(13, 29, 20)(14, 28, 19)(17, 30, 21),$$
$$(6, 26, 33)(7, 22, 34)(8, 21, 35)(9, 23, 36).$$

```
> G := PermutationGroup< 36 | (3, 17, 26)(4, 16, 25)(5, 18, 27)(8, 15, 24),
>                             (1, 32, 10)(2, 31, 11)(3, 35, 12)(6, 30, 15),
>                             (12, 33, 24)(13, 29, 20)(14, 28, 19)(17, 30, 21),
>                             (6, 26, 33)(7, 22, 34)(8, 21, 35)(9, 23, 36) >;
> IsTransitive(G);
false
> Orbit(G, 1);
```

```
GSet{@ 1, 32, 10 @}
> O := Orbits(G);
> O;
[
    GSet{@ 1, 32, 10 @},
    GSet{@ 2, 31, 11 @},
    GSet{@ 4, 16, 25 @},
    GSet{@ 5, 18, 27 @},
    GSet{@ 7, 22, 34 @},
    GSet{@ 9, 23, 36 @},
    GSet{@ 13, 29, 20 @},
    GSet{@ 14, 28, 19 @}
    GSet{@ 3, 17, 35, 26, 30, 12, 8, 33, 15, 21, 24, 6 @},
]
> Order(G);
933120
```

We see that the group is intransitive having eight orbits of size 3 and one orbit of size 12. We consider the action of G on the orbit of size 12.

```
> f := OrbitAction(G, O[9]);
> f;
Mapping from: GrpPerm: G to GrpPerm: $
> Im := Image(f);
> Im;
Permutation group acting on a set of cardinality 12
Order = 11520 = 2^8 * 3^2 * 5
    (1, 6, 9)(3, 5, 8)
    (4, 11, 8)(6, 10, 7)
    (6, 8)(7, 11)
    (2, 8, 10)(4, 12, 6)
    (3, 10, 8)(4, 6, 9)
> Ker := Kernel(f);
> Ker;
Permutation group acting on a set of cardinality 36
Order = 81 = 3^4
    (4, 16, 25)(5, 18, 27)
    (7, 22, 34)(9, 23, 36)
    (13, 29, 20)(14, 28, 19)
    (1, 32, 10)(2, 31, 11)(4, 25, 16)(5, 27, 18)
> IsElementaryAbelian(Ker);
true
```

Thus G has an elementary abelian normal subgroup of order 81 which is the kernel of the restriction of G to the orbit of size 12.

58.10.6 Action on a G-invariant Partition

This section describes the functions supplied by MAGMA for computing with block systems for a permutation group.

`IsBlock(G, S)`

Given a transitive permutation group G with natural G-set X, and a subset S of X, return `true` if S is a block for G in its action on X.

`IsPrimitive(G)`

Returns `true` if the transitive permutation group G is primitive.

`MaximalPartition(G)`

Construct a G-invariant partition P for the transitive permutation group G with natural G-set X. The partition P is maximal in the sense that there is no G-invariant partition P' of X such that some block of P' properly contains a block of the partition P. The block system is returned as a partition of X. If G is primitive, the partition with one block is returned.

`MinimalPartition(G: parameters)`

Construct a non-trivial G-invariant partition P of the natural G-set X of the transitive permutation group G. The partition P is minimal in the sense that there is no G-invariant partition P' of X such that some block of P' is properly contained in some block of the partition P. The block system is returned as a partition of X. If G is primitive, or if no partition satisfying the side-conditions (see below) is found, then the empty set is returned. The algorithm used is based on Schönert & Seress [SS94].

`Block := S` { ELT } *Default :* []

If S is non-empty, then the partition P must possess a block B such that S is a subset of B. In this case the algorithm used is that of Atkinson, [Atk75].

`MinimalPartitions(G: parameters)`

Construct all non-trivial minimal G-invariant partitions of the natural G-set X of the transitive permutation group G. A partition P is minimal in the sense that there is no G-invariant partition P' of X such that some block of P' is properly contained in some block of the partition P.

The minimal block systems are returned as a sequence of sets of sets. If G is primitive, the function returns the empty sequence.

The algorithm used is based on Schönert & Seress [SS94].

`Limit := n` RNGINTELT *Default :* ∞

The function will return after creating at most n block systems. This option is useful in situations where, say, two distinct minimal blocks systems are required for a reduction algorithm.

`AllPartitions(G)`

Construct all non-trivial G-invariant partitions of the natural G-set X of the transitive permutation group G. The structure returned is a set containing one block from each such partition. The block chosen is the block containing the first element of `Labelling(G)`.

`BlocksAction(G, P)`
`BlocksAction(G, P)`
`BlocksAction(G, P)`
`BlocksAction(G, P)`

Given a transitive permutation group G with natural G-set X and a G-invariant partition P of X, construct the group L induced by the action of G on the blocks of P. In the second form, P is specified by giving a single block of the partition. The function returns

(a) The natural homomorphism $f : G \to L$;

(b) The induced group;

(c) The kernel of the action (a subgroup of G).

The relationship between the supports of G and L is given by the returned mapping, which may also be used as a map from `Labelling(G)` to `Labelling(L)`. In the forward direction this takes each element in the support of G to its block number in the support of L, while in the reverse direction this takes a block number to a representative of the block.

`BlocksImage(G, P)`
`BlocksImage(G, P)`
`BlocksImage(G, P)`
`BlocksImage(G, P)`

Given a transitive permutation group G with natural G-set X and a G-invariant partition P of X, construct the group induced by the action of G on the blocks of P. In the second form, P is specified by giving a single block of the partition.

`BlocksKernel(G, P)`
`BlocksKernel(G, P)`
`BlocksKernel(G, P)`
`BlocksKernel(G, P)`

Given a transitive permutation group G with natural G-set X and a G-invariant partition P of X, construct the kernel of the action of G on the blocks of P. In the second form, P is specified by giving a single block of the partition.

Example H58E27

An imprimitive group of degree 100 constructed by Capel has several different block systems.

```
> G := sub< Sym(100) |
>   (1,21,41,61,81)(2,82,62,42,22)(3,23,43,63,83)(4,84,64,44,24)
>     (5,25,45,65,85)(6,86,66,46,26)(7,27,47,67,87)(8,88,68,48,28)
>     (9,29,49,69,89)(10,90,70,50,30)(11,31,51,71,91)(12,92,72,52,32)
>     (13,33,53,73,93)(14,94,74,54,34)(15,35,55,75,95)(16,96,76,56,36)
>     (17,37,57,77,97)(18,98,78,58,38)(19,39,59,79,99)(20,100,80,60,40),
>   (1,4,6,7,10)(2,3,5,8,9)(11,19,17,15,14)(12,20,18,16,13)(21,24,26,27,30)
>      (22,23,25,28,29)(31,39,37,35,34)(32,40,38,36,33)(41,44,46,47,50)
>      (42,43,45,48,49)(51,59,57,55,54)(52,60,58,56,53)(61,64,66,67,70)
>      (62,63,65,68,69)(71,79,77,75,74)(72,80,78,76,73)(81,84,86,87,90)
>      (82,83,85,88,89)(91,99,97,95,94)(92,100,98,96,93),
>   (1,11,2,12)(3,13,4,14)(5,16,6,15)(7,17,8,18)(9,20,10,19)(21,31,22,32)
>      (23,33,24,34)(25,36,26,35)(27,37,28,38)(29,40,30,39)(41,51,42,52)
>      (43,53,44,54)(45,56,46,55)(47,57,48,58)(49,60,50,59)(61,71,62,72)
>      (63,73,64,74)(65,76,66,75)(67,77,68,78)(69,80,70,79)(81,91,82,92)
>      (83,93,84,94)(85,96,86,95)(87,97,88,98)(89,100,90,99) >;
> MaxPart := MaximalPartition(G);
> #MaxPart;
2
> MaxPart;
GSet{@
    { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 31, 32, 33, 34, 35,
    36, 37, 38, 39, 40, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 71,
    72, 73, 74, 75, 76, 77, 78, 79, 80, 91, 92, 93, 94, 95, 96, 97,
    98, 99, 100 },
    { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 21, 22, 23, 24, 25, 26, 27, 28,
    29, 30, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 61, 62, 63, 64,
    65, 66, 67, 68, 69, 70, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90 }
@}
> MinPart := MinimalPartition(G);
> #MinPart;
50
```

We see that the group has a (maximal) system of imprimitivity consisting of 2 blocks of size 50 and a (minimal) system of imprimitivity consisting of 50 blocks of size 2.

```
> Parts := MinimalPartitions(G);
> [ #p : p in Parts ];
[ 50, 50, 50, 50, 20, 50 ]
```

Thus the group has six distinct minimal G-invariant partitions. Of these five have 50 blocks of size two while the remaining one has 20 blocks of size 5. We examine the action of G on one of the partitions into 50 blocks of size 2.

```
> f, Im, Ker := BlocksAction(G, Parts[1]);
> f;
Mapping from: GrpPerm: G to GrpPerm: Im
```

```
> Im;
Permutation group Im acting on a set of cardinality 50
Order = 7812500 = 2^2 * 5^9
    (1, 11, 31, 32, 12)(2, 13, 33, 34, 14)(3, 15, 35, 36, 16)
         (4, 17, 37, 38, 18) (5, 19, 39, 40, 20)(6, 21, 41, 42, 22)
         (7, 23, 43, 44, 24)(8, 25, 45, 46, 26)(9, 27, 47, 48, 28)
         (10, 29, 49, 50, 30)
    (1, 2, 3, 4, 5)(6, 10, 9, 8, 7)(11, 14, 16, 17, 20)(12, 13, 15, 18, 19)
         (21, 29, 27, 25, 24)(22, 30, 28, 26, 23)(31, 34, 36, 37, 40)
         (32, 33, 35, 38, 39)(41, 49, 47, 45, 44)(42, 50, 48, 46, 43)
    (1, 6)(2, 7)(3, 8)(4, 9)(5, 10)(11, 21, 12, 22)(13, 23, 14, 24)
        (15, 26, 16, 25)(17, 27, 18, 28)(19, 30, 20, 29)(31, 41, 32, 42)
        (33, 43, 34, 44)(35, 46, 36, 45)(37, 47, 38, 48)(39, 50, 40, 49)
> Ker;
Permutation group Ker acting on a set of cardinality 100
Order = 1
```

Thus, G acts faithfully on this block system.

Example H58E28

When analyzing a permutation group, it is sometimes necessary to reduce it to its primitive components. This can be done by using the constituent homomorphism and blocks homomorphism functions. We illustrate their use by analyzing the group of Rubik's cube, represented as a permutation group on 48 letters:

```
> G := sub<Sym(48) |
>     (1,3,8,6)(2,5,7,4)(9,48,15,12)(10,47,16,13)(11,46,17,14),
>     (6,15,35,26)(7,22,34,19)(8,30,33,11)(12,14,29,27)(13,21,28,20),
>     (1,12,33,41)(4,20,36,44)(6,27,38,46)(9,11,26,24)(10,19,25,18),
>     (1,24,40,17)(2,18,39,23)(3,9,38,32)(41,43,48,46)(42,45,47,44),
>     (3,43,35,14)(5,45,37,21)(8,48,40,29)(15,17,32,30)(16,23,31,22),
>     (24,27,30,43)(25,28,31,42)(26,29,32,41)(33,35,40,38)(34,37,39,36) >;
> O1 := Orbits(G);
> O1;
[
    GSet{@ 1, 3, 6, 8, 9, 11, 12, 14, 15, 17, 24, 26, 27, 29,
     30, 32, 33, 35, 38, 40, 41, 43, 46, 48 @},
    GSet{@ 2, 4, 5, 7, 10, 13, 16, 18, 19, 20, 21, 22, 23, 25, 28,
     31, 34, 36, 37, 39, 42, 44, 45, 47 @}
]
```

Thus, G has two orbits, each of size 24. We consider the restriction of the action of G to the first of these orbits.

```
> f1, Im1, Ker1 := OrbitAction(G, O1[1]);
> FactoredOrder(Im1);
[ <2, 7>, <3, 9>, <5, 1>, <7, 1> ]
> IsPrimitive(Im1);
false
```

```
> P1 := MinimalPartition(Im1);
> #P1;
8
> f2, Im2, Ker2 := BlocksAction(Im1, P1);
> FactoredOrder(Im2);
[ <2, 7>, <3, 2>, <5, 1>, <7, 1> ]
> IsPrimitive(Im2);
true
> IsSymmetric(Im2);
true
> FactoredOrder(Ker2);
[ <3, 7> ]
> IsElementaryAbelian(Ker2);
true
```

Hence the group obtained by restricting G to its first orbit is isomorphic to $Sym(8)$ acting on an elementary abelian normal subgroup of order 3^7. We next investigate the kernel $Ker1$ of the restriction of G to the first orbit of length 24. We know that $Ker1$ must fix all the points in the first orbit of G so we first take its restriction to the second orbit.

```
> f3, Im3, Ker3 := OrbitAction(Ker1, O1[2]);
> IsTransitive(Im3);
true
> FactoredOrder(Im3);
[ <2, 20>, <3, 5>, <5, 2>, <7, 1>, <11, 1> ]
> FactoredOrder(Ker3);
[]
> IsPrimitive(Im3);
false
```

The kernel acts transitively and faithfully on the second orbit. As it is imprimitive, we look at its action on a system of imprimitivity.

```
> P := MinimalPartition(Im3);
> f4, Im4, Ker4 := BlocksAction(Im3, P);
> Im4;
Permutation group Im4 acting on a set of cardinality 12
Order = 239500800 = 2^9 * 3^5 * 5^2 * 7 * 11
    (10, 12, 11)
    (9, 12, 11)
    (8, 12, 9)
    (7, 9)(8, 12)
    (6, 9, 10)
    (5, 6, 9)
    (4, 6, 9)
    (3, 6, 9)
    (2, 6, 9, 5)(4, 10)
    (1, 9, 6, 5)(4, 10)
> IsPrimitive(Im4);
true
```

```
> IsAlternating(Im4);
true
> FactoredOrder(Ker4);
[ <2, 11> ]
> IsElementaryAbelian(Ker4);
true
```

The kernel of the restriction of G to the first orbit is isomorphic to $Alt(12)$ acting on an elementary abelian group of order 2^{11}. So we now know the composition factors of G together with an indication of how they fit together.

58.10.7 Action on a Coset Space

`CosetAction(G, H: parameters)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced group L;

(c) The kernel K of the action (a subgroup of G).

Note that G may be any type of group. If G is a finitely presented group, then K may be returned undefined.

`Al` MonStgElt *Default :* "*Default*"

`Al := "Wang"`: Construct the coset action using Wang da Fang's algorithm which builds the action up the stabilizer chains of G and H, using a sequence of induction and block image operations. This algorithm is particularly efficient when H has fixed points that are not fixed points of G, and is the default choice when H is trivial.

`Al := "Canonical"`: Compute the cosets using an orbit algorithm, which describes each coset found by computing a canonical element of that coset. The canonical element is one with the minimal base image in the group G. The algorithm is due to Richardson, [Ric73].

`Al := "Default"`: Choose one of the above to use.

`CosetImage(G, H: parameters)`

Given a subgroup H of the group G, construct the image L of G given by the action of G on the set of (right) cosets of H in G. L is returned as a permutation group. Possible parameters are as for the previous function.

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of the action of G on the set of (right) cosets of H in G.

58.10.8 Reduced Permutation Actions

If a permutation group is intransitive or imprimitive, then orbit actions and blocks actions provide natural permutation representations of lower degree.

`TransitiveQuotient(G)`

Returns the transitive constituent of G acting on its longest orbit, together with the action homomorphism and the kernel of the action. If G is transitive then the return values are G, the identity map on G, and the trivial subgroup of G.

`PrimitiveQuotient(G)`

For a transitive group G, returns the blocks image of G acting on a maximal block system, together with the action homomorphism and the kernel of the action. If G is primitive then the return values are G, the identity map on G, and the trivial subgroup of G.

`DegreeReduction(G)`

Use a combination of orbit images and blocks images to attempt to find a faithful permutation representation of G with lower degree than G. The second return value is the isomorphism from G to the representation found. If no lower degree faithful representation is found then G and the identity map on G is returned.

58.10.9 The Jellyfish Algorithm

The Jellyfish reduction algorithm was introduced in [LNPS06]. See this article for an explanation of the name "Jellyfish". It attempts to find faithful low degree permutation representations for a family of large-base primitive permutation groups. We now define the target family as given in [LNPS06]. Consider the group $W = S_n \wr S_r$, as a permutation group in its primitive action on the set of r-tuples of k-subsets of the chosen n-set (see `PrimitiveWreathProduct`). Let M be the socle of W, $M = A_n^r$. Let G be any subgroup of W, with $M \leq G$, such that the conjugation action of G on the r copies of A_n in M is transitive. A group T is in the target family of the algorithm if T is permutation isomorphic to some such G, having $n > 2rk^2$, and $rk > 1$. The degree of such a T is $\binom{n}{k}^r$, and any such T is primitive. The image sought by the algorithm has degree nk. The utility of the algorithm is that any primitive group not in the target family is either alternating, symmetric, or has a short base.

The algorithm is one-sided Monte-Carlo in that, if it reports success, then it has found a faithful representation of the group. There is a small probability that the algorithm will find no faithful representation, even when the group given is in the target family.

Note that the Jellyfish algorithm implemented in Magma may succeed even when the input group is not in the target family. In all cases, success of the algorithm guarantees a faithful representation of the group.

The Magma implementation offers functions for testing the group for applicability of the Jellyfish algorithm. If successful, this test constructs data structures as in the cited article for quick evaluation of the homomorphism to the low-degree representation found and stores these with the group. There are also functions to compute the image and

preimage of elements under the representation map. The preimage function is an addition to the algorithms given in [LNPS06], using an extension of their data structure. The preimage algorithm is nearly linear in the degree of the large degree primitive group.

`JellyfishConstruction(G: parameters)`

`Limit` RngIntElt *Default :*

Attempt to construct a set of jellyfish for G. If unsuccessful, return false. Otherwise, construct data structures corresponding to T1 and T5 of [LNPS06], attach them to G, and return `true`. The parameter `Limit` controls how many attempts are made to find the orbits of the point stabilizer of G, which is the initial step in constructing a jellyfish. This construction phase terminates after a sequence of `Limit` random elements of G fails to change the orbits found. If the orbits found so far are not the orbits of a point stabilizer in G, the algorithm may fail. The same limit is used in the next phase, constructing the first jellyfish. The current default is the maximum of 15 and $2\lfloor \log_2 d \rfloor$, where d is the degree of G.

`JellyfishImage(G)`

If the `JellyfishConstruction` function applied to G has returned `true`, return the faithful image of G found by the jellyfish algorithm. Otherwise an error results. If the `JellyfishConstruction` has not yet been applied to G, then it is applied first with default parameters. A failure here results in an error.

`JellyfishImage(G, x)`

If the `JellyfishConstruction` function applied to G has returned `true`, return the image of x as a permutation of the jellyfish. The function will attempt this for any x in the same symmetric group as G. The algorithm may fail when $x \notin G$, in which case an error results, and this proves that x is not in G. It is possible for this map to succeed when x is not in G. In recognition of this, the parent of the result will be a symmetric group. If the `JellyfishConstruction` has not yet been applied to G, then it is applied first with default parameters. A failure here results in an error.

`JellyfishPreimage(G, x)`

If the `JellyfishConstruction` function applied to G has returned `true`, return the preimage of x as a permutation in the symmetric group of G. The element x is assumed to be a permutation of the jellyfish for G. The function will attempt this for any x in the same symmetric group as the `JellyfishImage` of G. The algorithm may fail when x is not in the image group, in which case an error results. It is possible for this map to succeed when x is not in the image group. In recognition of this, the parent of the result will be a symmetric group. If the `JellyfishConstruction` has not yet been applied to G, then it is applied first with default parameters. A failure here results in an error.

58.11 Normal and Subnormal Subgroups

58.11.1 Characteristic Subgroups and Normal Series

`DerivedSeries(G)`

The derived series of the group G. The series is returned as a sequence of subgroups. The algorithm used is described in [BC82].

`CompositionSeries(G)`

A composition series of the group G, ie. a descending chain of normal subgroups, such that each quotient is a simple group. The series is returned as a sequence of subgroups.

`CommutatorSubgroup(G)`
`DerivedSubgroup(G)`
`DerivedGroup(G)`

The derived subgroup of the group G.

`SolubleResidual(G)`
`SolvableResidual(G)`

The solvable residual (the last term of the derived series) of the group G.

`DerivedLength(G)`

The derived length of G. If G is non-soluble, the function returns the number of terms in the series terminating with the soluble residual.

`LowerCentralSeries(G)`

The lower central series of G. The series is returned as a sequence of subgroups, the first of which is the group G. The algorithm used is described in [BC82].

`NilpotencyClass(G)`

The nilpotency class of the group G. If the group is not nilpotent, the value -1 is returned.

`UpperCentralSeries(G)`

The upper central series of G. The series is returned as a sequence of subgroups commencing with the trivial subgroup. The algorithm used is to compute the centre of G and then section centralisers (see [Luk93]) up the chain. This requires computing cores of subgroups, so this function is more restricted in its range of application than `DerivedSeries` and `LowerCentralSeries`.

`Centre(G)`
`Center(G)`

Construct the centre of the group G. The centre is found by applying the function `CentralizerOfNormalSubgroup` to G in G.

`Hypercentre(G)`
`Hypercenter(G)`

Construct the hypercentre of the group G (the stationary term of the upper central series).

`pCore(G, p)`

Given a group G and a prime p, construct the maximal normal p-subgroup of G. The algorithm employed is described in Unger [Ung06b].

`pCoreQuotient(G, p)`

Given a group G and a prime p, construct the quotient of G by `K := pCore(G, p)`. The return values are the quotient, Q, represented as a permutation group of the same degree as G, the quotient map from G onto Q, and K.

`FittingSubgroup(G)`

The Fitting subgroup of the group G. It is computed as the product of the p-cores of the radical of G.

`FrattiniSubgroup(G)`

Given a group G, return the Frattini subgroup. For p-groups this is computed as the derived group with pth powers of the generators added. Solvable groups are converted to their `GrpPC` representation and the problem solved there. Non-solvable groups are treated by finding their maximal subgroups and forming the intersection, so are subject to the same restrictions as the `MaximalSubgroups` command.

`JenningsSeries(G)`

Given a p-group G, return the Jennings series for G. The series is returned as a sequence of subgroups.

`pCentralSeries(G, p)`

Given a soluble group G, and a prime p dividing $|G|$, return the lower p-central series for G. The series is returned as a sequence of subgroups.

`SubnormalSeries(G, H)`

Given a group G and a subnormal subgroup H of G, return a sequence of subgroups commencing with G and terminating with H, such that each subgroup is normal in the previous one. If H is not subnormal in G, the empty sequence is returned.

Example H58E29

We compute the various series in the wreath product of the symmetric group of degree 4 with the dihedral group of order 8 (a soluble group).

```
> G := WreathProduct(Sym(4), DihedralGroup(4));
> G;
Permutation group G acting on a set of cardinality 16
    (1, 5, 9, 13)(2, 6, 10, 14)(3, 7, 11, 15)(4, 8, 12, 16)
    (1, 13)(2, 14)(3, 15)(4, 16)(5, 9)(6, 10)(7, 11)(8, 12)
    (1, 2, 3, 4)
    (1, 2)
> [ FactoredOrder(H) : H in DerivedSeries(G) ];
[
    [ <2, 15>, <3, 4> ],
    [ <2, 12>, <3, 4> ],
    [ <2, 9>, <3, 4> ],
    [ <2, 8>, <3, 4> ],
    [ <2, 8> ],
    []
]
> DerivedLength(G);
5
> [ FactoredOrder(H) : H in LowerCentralSeries(G) ];
[
    [ <2, 15>, <3, 4> ],
    [ <2, 12>, <3, 4> ],
    [ <2, 10>, <3, 4> ],
    [ <2, 9>, <3, 4> ],
    [ <2, 8>, <3, 4> ]
]
> NilpotencyClass(G);
-1
> Centre(G);
Permutation group acting on a set of cardinality 16
Order = 1
    Id($)
> pCentralSeries(G, 2);
[
    [ <2, 15>, <3, 4> ],
    [ <2, 12>, <3, 4> ],
    [ <2, 10>, <3, 4> ],
    [ <2, 9>, <3, 4> ],
    [ <2, 8>, <3, 4> ]
]
> [ FactoredOrder(H) : H in pCentralSeries(G, 3) ];
[
    [ <2, 15>, <3, 4> ]
```

```
]
```

58.11.2 Maximal and Minimal Normal Subgroups

`MaximalNormalSubgroup(G)`

A maximal normal subgroup of G. The trivial subgroup is returned if G is simple. The algorithm takes homomorphic reductions to a primitive group and then uses O'Nan-Scott type considerations to get its result.

`MinimalNormalSubgroups(G)`

The minimal normal subgroups of G. These are obtained by first computing the socle of G and then splitting off the normal factors.

58.11.3 Lattice of Normal Subgroups

`NormalSubgroups(G)`

The normal subgroups of G. These are determined by the method of Cannon and Souvignier [CS].

`NormalLattice(G)`

The normal subgroup lattice of G. The subgroups are first found using the same algorithm as the function `NormalSubgroups` and then inclusions are determined.

Example H58E30

We determine all normal subgroups of the wreath product of $Sym(8)$ and the dihedral group of order 8.

```
> G := WreathProduct(Sym(8), DihedralGroup(4));
> Order(G);
21143266346926080000
> time N := NormalSubgroups(G);
Time: 1.050
> #N;
29
> [ < Order(H`subgroup), FactoredOrder(H`subgroup) > : H in N ];
[
    <1, []>,
    <165181768335360000, [ <2, 24>, <3, 8>, <5, 4>, <7, 4> ]>,
    <330363536670720000, [ <2, 25>, <3, 8>, <5, 4>, <7, 4> ]>,
    <660727073341440000, [ <2, 26>, <3, 8>, <5, 4>, <7, 4> ]>,
    <1321454146682880000, [ <2, 27>, <3, 8>, <5, 4>, <7, 4> ]>,
    <1321454146682880000, [ <2, 27>, <3, 8>, <5, 4>, <7, 4> ]>,
    <1321454146682880000, [ <2, 27>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
```

```
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <2642908293365760000, [ <2, 28>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <5285816586731520000, [ <2, 29>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <10571633173463040000, [ <2, 30>, <3, 8>, <5, 4>, <7, 4> ]>,
    <21143266346926080000, [ <2, 31>, <3, 8>, <5, 4>, <7, 4> ]>
]
```

58.11.4 Composition and Chief Series

`ChiefFactors(G)`

Given a group G, return a sequence of the isomorphism types $< f, d, q, m >$ of the chief factors. An isomorphism type in a chief factor should be understood as the direct product of m copies of the simple group described by $< f, d, q >$ (see `CompositionFactors` below). For the algorithm, see Unger [Ung].

`ChiefSeries(G)`

Given a group G, return the chief series of G and a sequence of the corresponding isomorphism types $< f, d, q, m >$ of the chief factors. An isomorphism type in a chief factor should be understood as the direct product of m copies of the simple group described by $< f, d, q >$ (see `CompositionFactors` below). The series will be organised to include the soluble radical of G, and, if G is insoluble, the socle of the quotient of G by the soluble radical.

f	Family name
1	A(d, q)
2	B(d, q)
3	C(d, q)
4	D(d, q)
5	G$(2, q)$
6	F$(4, q)$
7	E$(6, q)$
8	E$(7, q)$
9	E$(8, q)$
10	2A(d, q)
11	2B$(2, q)$
12	2D(d, q)
13	3D$(4, q)$
14	2G$(2, q)$
15	2F$(4, q)$
16	2E$(6, q)$
17	Alternating(d)
18	Sporadic group — see Table 2.
19	Cyclic(q)

Table 1: Family numbers and names

d	Group name
1	M_{11}
2	M_{12}
3	M_{22}
4	M_{23}
5	M_{24}
6	J_1
7	HS
8	J_2
9	MCL
10	SUZ
11	J_3
12	CO_1
13	CO_2
14	CO_3
15	HE
16	$M(22)$
17	$M(23)$
18	$M(24)$
19	LY
20	RU
21	ON
22	TH
23	HA
24	BM
25	M
26	J_4

Table 2: Sporadic groups

`CompositionFactors(G)`

Given a permutation group G, return a sequence S of tuples that represent the composition factors of G, ordered according to some composition series of G. Each tuple is a triple of integers f, d, q that defines the isomorphism type of the corresponding composition factor. A triple $\langle f, d, q\rangle$ describes a simple group as follows. The integer f defines the family to which the group belongs, and d and q are the parameters of the family. The length of the sequence S is the number of composition factors of G. The algorithm used is the "tabular" algorithm of Kantor [Kan91], extended to be valid for groups of degree $\leq 10^7$. The families are listed in Tables 1 and 2 on page 1590.

Example H58E31

We illustrate the function `CompositionFactors` by applying it to the group associated with Rubik's cube.

```
> G := sub<Sym(48) |
>     (1,3,8,6)(2,5,7,4)(9,48,15,12)(10,47,16,13)(11,46,17,14),
>     (6,15,35,26)(7,22,34,19)(8,30,33,11)(12,14,29,27)(13,21,28,20),
>     (1,12,33,41)(4,20,36,44)(6,27,38,46)(9,11,26,24)(10,19,25,18),
>     (1,24,40,17)(2,18,39,23)(3,9,38,32)(41,43,48,46)(42,45,47,44),
>     (3,43,35,14)(5,45,37,21)(8,48,40,29)(15,17,32,30)(16,23,31,22),
>     (24,27,30,43)(25,28,31,42)(26,29,32,41)(33,35,40,38)(34,37,39,36)
>         >;
> FactoredOrder(G);
[ <2, 27>, <3, 14>, <5, 3>, <7, 2>, <11, 1> ]
> CompositionFactors(G);
    G
    |  Cyclic(2)
    *
    |  Alternating(12)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Alternating(8)
    *
    |  Cyclic(3)
    *
    |  Cyclic(3)
    *
```

```
|  Cyclic(3)
*
|  Cyclic(3)
*
|  Cyclic(3)
*
|  Cyclic(3)
*
|  Cyclic(3)
1
```

58.11.5 The Socle

`Socle(G)`

The socle of the group G. This is computed using the algorithms described in Cannon and Holt [CH97].

`SocleFactor(G)`

A simple factor of the socle of the group G.

`SocleFactors(G)`

The simple factors of the socle of the group G. The index of each factor in the sequence corresponds to the points of the image group of `SocleAction` and `SocleImage`.

`SocleSeries(G)`

A chain of subgroups

$$S_1, S_1 \times S_2, \ldots, S_1 \times \ldots \times S_r,$$

where $S_1, ..., S_r$ are the simple factors of the socle of the primitive group G.

`EARNS(G)`

The elementary abelian regular normal subgroup (EARNS) of the primitive group G. If G does not have an EARNS, then the trivial subgroup is returned. The algorithm used is that of Neumann [Neu86].

`IsAffine(G)`

Decide if the permutation group G is of primitive affine type. If so, the elementary abelian regular normal subgroup of G is returned as second return value. If the group G is either intransitive or transitive and imprimitive or primitive and not of affine type, then the result will be false (only). This function combines `IsTransitive`, `IsPrimitive` and `EARNS`.

`AffineAction(G)`

Given a primitive group G which has a non-trivial elementary abelian regular normal subgroup A, construct the representation of G given by the action of G on elements of the elementary abelian group A. The image is realised as a point-stabilizer in G and the kernel of the action is A. As with the other action functions, `AffineAction` returns the homomorphism, the image and the kernel of the action.

`AffineImage(G)`

Given a primitive group G which has an elementary abelian regular normal subgroup A, construct the permutation group that results from the action of G on elements of the elementary abelian group A. This image is realised as a point-stabilizer in G.

`AffineKernel(G)`

Given a primitive group G which has a non-trivial elementary abelian regular normal subgroup A, construct the kernel of the action of G on elements of the elementary abelian group A. This kernel equals A.

`SocleAction(G)`

Given a non-trivial permutation group G which has trivial Fitting subgroup, construct the permutation representation of G given by the action of G on the simple factors of N. Note that a primitive group has a perfect socle if and only if it has no elementary abelian regular normal subgroup. As with the other action functions, `SocleAction` returns the homomorphism, the image and the kernel of the action. The socle factor corresponding to point i in the support of the image group is the ith element in the sequence `SocleFactors(G)`.

`SocleImage(G)`

Given a non-trivial permutation group G which has trivial Fitting subgroup, construct the permutation group L induced by the action of G on the simple factors of N.

`SocleKernel(G)`

Given a non-trivial permutation group G which has trivial Fitting subgroup, construct the kernel of the action of G on the simple factors of N.

`SocleQuotient(G)`

Given a permutation group G which has trivial Fitting subgroup, construct a permutation representation of G/N. If U_i denote the simple factors of N, then the degree of the result is bounded by $\sum_i |Out(U_i)|$ (see Cannon and Souvignier [CS]). Note that a primitive group has a perfect socle if and only if it has no elementary abelian regular normal subgroup. `SocleQuotient` returns G/N, the quotient homomorphism and the kernel of the map (which is the socle of G).

`RefineSection(G, M, N)`

Given M, N normal subgroups of G with $N < M$, return a sequence of G-normal subgroups $L_1, \ldots, L_r$ with $N = L_0$, $L_i < L_{i+1}$ and $L_r = M$ such that each of the quotients L_{i+1}/L_i is either elementary abelian or a direct product of non-abelian simple groups.

Example H58E32

We examine the normal structure of a primitive group, the primitive-wreath product of Sym(5) and Sym(3) (with product action).

```
> G := PrimitiveWreathProduct(Sym(5), Sym(3));
> FactoredOrder(G);
[ <2, 10>, <3, 4>, <5, 3> ]
> E := EARNS(G);
> E;
Permutation group E acting on a set of cardinality 125
Order = 1
> DerivedSeries(G);
[
    Permutation group G acting on a set of cardinality 125
    Order = 10368000 = 2^10 * 3^4 * 5^3
    Permutation group acting on a set of cardinality 125
    Order = 2592000 = 2^8 * 3^4 * 5^3,
    Permutation group acting on a set of cardinality 125
    Order = 864000 = 2^8 * 3^3 * 5^3,
    Permutation group S acting on a set of cardinality 125
    Order = 216000 = 2^6 * 3^3 * 5^3
]
> S := Socle(G);
> S;
Permutation group S acting on a set of cardinality 125
Order = 216000 = 2^6 * 3^3 * 5^3
> Q := SocleFactors(G);
> Q;
[
    Permutation group acting on a set of cardinality 125
    Order = 60 = 2^2 * 3 * 5,
    Permutation group acting on a set of cardinality 125
    Order = 60 = 2^2 * 3 * 5,
    Permutation group acting on a set of cardinality 125
    Order = 60 = 2^2 * 3 * 5
]
> R := SocleSeries(G);
> R;
[
    Permutation group acting on a set of cardinality 125
    Order = 60 = 2^2 * 3 * 5,
```

```
    Permutation group acting on a set of cardinality 125
    Order = 3600 = 2^4 * 3^2 * 5^2,
    Permutation group acting on a set of cardinality 125
    Order = 216000 = 2^6 * 3^3 * 5^3
]
> SQ := SocleQuotient(G);
> SQ;
Permutation group SQ acting on a set of cardinality 6
Order = 48 = 2^4 * 3
    (1, 2, 3)(4, 5, 6)
    (2, 3)(4, 5)
    Id($)
    (2, 4)
```

58.11.6 The Soluble Radical and its Quotient

Very efficient algorithms have been developed for computing invariants such as subgroups, normal subgroups and conjugacy classes of elements for soluble groups defined by means of polycyclic presentations. Almost all such algorithms employ a top-down *Lifting Strategy*. Let P be a quotient-invariant property for a soluble group. In general, an algorithm that constructs the set of elements or subgroups $X_P(G)$ satisfying property P for the group G, proceeds as follows: Let G be a non-simple soluble group and let N be a normal subgroup of G. The set $X_P(G/N)$ is constructed and its elements are lifted back into G, thereby yielding $X_P(G)$. This process is usually iterated with successive normal subgroups N being chosen as the terms of some descending normal series (e.g., an elementary abelian series).

In generalizing this approach to permutation groups, our approach has been to construct the soluble radical R of G, use special methods to solve the problem for the quotient G/R, and then proceed (as in the case of a soluble group) to lift the solution down the successive terms of an elementary abelian series for G using the Lifting Strategy. Derek Holt has shown that the quotient group G/R has a faithful permutation representation of degree no greater than that of G.

The functions in this section enable the user to construct the radical, its quotient and an elementary abelian series.

`Radical(G)`
`SolubleRadical(G)`
`SolvableRadical(G)`

Given a group G, return the maximal normal solvable subgroup of G. The algorithm used is described in Unger [Ung06b].

`RadicalQuotient(G)`

Given a group G, compute a representation of the quotient G/R where R is the (solvable) radical of G. The resulting representation has the same degree as G. Both the permutation group Q isomorphic to G/R and a homomorphism $\phi : G \to Q$ are returned. The algorithm proceeds by repeatedly applying `AbelianNormalQuotient` up the terms of the derived series of the radical. The third return value is R, the radical of G and the kernel of the homomorphism.

`ElementaryAbelianSeries(G: parameters)`

`ElementaryAbelianSeries(G, N: parameters)`

`LayerSizes` SeqEnum[RngIntElt] *Default :* []

An elementary abelian series is a chain of normal subgroups $R = N_1 > N_2 > ... > N_r = 1$ with the property that the quotient of each pair of successive terms in the series is elementary abelian and that there is no group $R < H < G$ such that H/R is elementary abelian and H normal in G. The top of the series R is called the solvable radical and is the maximal normal solvable subgroup of G.

In the second form N must be a normal subgroup of G and the returned series has the form $R = N_1 > N_2 > ... > N_r = N$, so is an elementary abelian series for G/N.

The parameter `LayerSizes` controls possible refinement of the series. The default is no refinement. As an example, take `LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]`. When constructing an elementary abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1. Setting `LayerSizes` to `[2, 1]` will attempt to split all layers, resulting in a portion of a chief series for G.

`ElementaryAbelianSeriesCanonical(G)`

Gives a similar result to using `ElementaryAbelianSeries`, except the series returned depends only on the isomorphism type of the solvable radical, and consists of characteristic subgroups of G. This function may be slower than `ElementaryAbelianSeries`.

Example H58E33

We illustrate these functions by considering the group of degree 16 generated by the following permutations:

$$(1, 8, 11, 3, 6, 14, 15, 10)(2, 7, 12, 4, 5, 13, 16, 9),$$
$$(1, 2)(3, 16, 9, 14, 8, 12)(4, 15, 10, 13, 7, 11),$$
$$(1, 13, 12, 16)(2, 14, 11, 15)(7, 9)(8, 10),$$

```
> G := PermutationGroup< 16 |
>         (1, 8, 11, 3, 6, 14, 15, 10)(2, 7, 12, 4, 5, 13, 16, 9),
>         (1, 2)(3, 16, 9, 14, 8, 12)(4, 15, 10, 13, 7, 11),
```

```
>         (1, 13, 12, 16)(2, 14, 11, 15)(7, 9)(8, 10) >;
> Radical(G);
Permutation group acting on a set of cardinality 16
Order = 256 = 2^8
   (3, 4)(5, 6)(7, 8)(13, 14)(15, 16)
   (3, 4)(7, 8)(9, 10)(11, 12)
   (7, 8)(13, 14)
   (1, 2)(7, 8)(9, 10)(11, 12)(13, 14)(15, 16)
   (9, 10)
   (15, 16)
   (11, 12)(15, 16)
   (13, 14)(15, 16)
> RadicalQuotient(G);
Permutation group acting on a set of cardinality 16
Order = 40320 = 2^7 * 3^2 * 5 * 7
   (1, 7, 11, 3, 5, 13, 15, 9)(2, 8, 12, 4, 6, 14, 16, 10)
   (3, 15, 9, 13, 7, 11)(4, 16, 10, 14, 8, 12)
   (1, 13, 11, 15)(2, 14, 12, 16)(7, 9)(8, 10)
Mapping from: GrpPerm: g to GrpPerm: $, Degree 16
> ElementaryAbelianSeries(G);
[
   Permutation group acting on a set of cardinality 16
   Order = 256 = 2^8
       (3, 4)(5, 6)(7, 8)(13, 14)(15, 16)
       (3, 4)(7, 8)(9, 10)(11, 12)
       (7, 8)(13, 14)
       (1, 2)(7, 8)(9, 10)(11, 12)(13, 14)(15, 16)
       (9, 10)
       (15, 16)
       (11, 12)(15, 16)
       (13, 14)(15, 16),
   Permutation group acting on a set of cardinality 16
   Order = 1
]
```

58.11.7 Complements and Supplements

`Complements(G, M)`

Given a group G and a normal subgroup M, this function returns a sequence containing one representative from each conjugacy class of complements of M in G.

`Complements(G, M, N)`

Given a group G, a normal subgroup M of G and a normal subgroup N of G, that is strictly contained in M, the function returns a sequence comprising representatives for the conjugacy classes of complements of M/N in G/N, as subgroups of G.

`HasComplement(G, M)`

The group M must be a normal subgroup of G. Returns whether M has a complement in G and, if so, one such complement.

`Supplements(G, M)`

Given a group G and a soluble normal subgroup M of G, the function returns a sequence containing one representative from each conjugacy class of minimal supplements for M in G.

`Supplements(G, M, N)`

Given a group G, a normal subgroup M of G and a normal subgroup N of G such that (a), N is strictly contained in M, and (b), M/N is soluble, the function returns a sequence comprising representatives for the conjugacy classes of minimal supplements of M/N in G/N, as subgroups of G.

`HasSupplement(G, M)`

The group M must be a soluble normal subgroup of G. Returns whether M has a proper supplement in G and, if so, one such supplement.

Example H58E34

We illustrate these functions by considering a normal subgroup H of the group G of degree 16 generated by the following permutations:

$$(1,3,2,4)(5,16,6,13)(7,14,8,15)(9,12,11,10),$$
$$(1,16,9)(2,15,12)(3,14,11)(4,13,10)(6,8,7).$$

```
> G := PermutationGroup< 16 |
>         (1, 3, 2, 4)(5, 16, 6, 13)(7, 14, 8, 15)(9, 12, 11, 10),
>         (1, 16, 9)(2, 15, 12)(3, 14, 11)(4, 13, 10)(6, 8, 7) >;
Permutation group G acting on a set of cardinality 16
Order = 165888 = 2^11 * 3^4
    (1, 3, 2, 4)(5, 16, 6, 13)(7, 14, 8, 15)(9, 12, 11, 10)
    (1, 16, 9)(2, 15, 12)(3, 14, 11)(4, 13, 10)(6, 8, 7)
> H := ncl< G | (6, 7, 8)(14, 16, 15) >;
> H;
Permutation group H acting on a set of cardinality 16
Order = 6912 = 2^8 * 3^3
   (6, 7, 8)(14, 16, 15)
   (6, 7, 8)(13, 14, 15)
   (6, 7, 8)(9, 12, 11)
   (5, 8, 7)(13, 14, 15)
   (6, 7, 8)(10, 11, 12)
   (1, 2, 3)(6, 7, 8)
   (2, 4, 3)(6, 7, 8)
```

```
> C := Complements(G, H);
> C;
[
   Permutation group acting on a set of cardinality 16
   Order = 24 = 2^3 * 3
       (3, 4)(5, 14)(6, 15)(7, 16)(8, 13)(10, 12)
       (2, 4)(6, 7)(9, 14)(10, 15)(11, 13)(12, 16)
       (1, 14)(2, 15)(3, 16)(4, 13)(7, 8)(10, 11)
       (1, 14, 9)(2, 13, 10)(3, 16, 12)(4, 15, 11)(6, 8, 7)
]
```

So the normal subgroup has one conjugacy class of complements. We check that the representative subgroup is indeed a complement for H.

```
> K := C[1];
> IsTrivial(K meet H );
true
> #K * #H eq #G;
true
```

58.11.8 Abelian Normal Subgroups

`AbelianNormalSubgroup(G)`

An abelian normal subgroup of G. If none exists, the trivial subgroup is returned.

`AbelianNormalQuotient(G, H)`

A quotient of G by an abelian normal subgroup that contains the abelian normal subgroup H. The quotient is represented as a permutation group of the same degree as G. The other values returned are the quotient epimorphism and its kernel K. The kernel K will contain H, $\#K$ and $\#H$ will have the same prime divisors, and if H is elementary abelian then so is K.

`SolubleNormalQuotient(G, H)`

A quotient of G by a soluble normal subgroup that contains the soluble normal subgroup H. The quotient is represented as a permutation group of the same degree as G. The other values returned are the quotient epimorphism and its kernel K. As with `AbelianNormalQuotient`, K will contain H, and $\#K$ and $\#H$ will have the same prime divisors.

`ElementaryAbelianNormalSubgroup(G)`

An elementary abelian normal subgroup of G. If none exists, the trivial subgroup is returned. The group returned is the last non-trivial group in an elementary abelian series for the radical of G.

`pElementaryAbelianNormalSubgroup(G, p)`

An elementary abelian normal p-subgroup of G. If none exists, the trivial subgroup is returned. The group returned is the last non-trivial group in an elementary abelian series for the p-core of G.

`MEANS(G)`

A minimal elementary abelian normal subgroup of G.

`MEANS(G, N)`

A minimal elementary abelian normal subgroup of G that lies in the elementary abelian normal subgroup N of G.

58.12 Cosets and Transversals

58.12.1 Cosets

`H * g`

Right coset of the subgroup H of the group G, where g is an element of G.

`DoubleCoset(G, H, g, K)`

The double coset $H * g * K$ of the subgroups H and K of the group G, where g is an element of G.

`DoubleCosetRepresentatives(G, H, K)`

Given a group G and two subgroups H and K of G, return a sequence S containing representatives of the H-K-double cosets in G. The first element of S is guaranteed to be the identity element of G. The second return sequence gives the sizes of the corresponding double cosets. The algorithm used refines double cosets down a chain of subgroups from G to one of H or K.

`ProcessLadder(L, G, U)`

Verbose `DoubleCosets` *Maximum :* 3

Given permutation groups $U < G$ and a sequence of permutation groups L such that $L_1 = G$, compute data for computations with the L_n-U-double cosets in G. The algorithm relies on the indices $(L_i : L_{i+1})$ (for $L_i < L_{i+1}$) or $(L_{i+1} : L_i)$ otherwise to be small. In contrast to the method used by `DoubleCosetRepresentatives`, the sequence used in the computation is a ladder, not necessarily a descending chain. For details see [Sch90].

`GetRep(p, R)`

For R being the result of a call to `ProcessLadder` and a permutation $p \in G$, return the canonical double coset representative for p.

`DeleteData(R)`

Deletes the data computed using `ProcessLadder`.

`YoungSubgroupLadder(L)`

`Full` RngIntElt *Default :* `false`

Computes a ladder from the full symmetric group down to the Young subgroup parametrised by the sequence L suitable for double coset enumeration using `ProcessLadder`. The optional parameter `Full` can be used if the Young subgroup should be considered as a subgroup of the symmetric group on `Full` points rather than on `&*L`.

`StabilizerLadder(G, d)`

Given a subgroup G of the symmetric group of degree n and a monomial in n indeterminates, compute a ladder down from the full symmetric group to the stabilizer of the monomial, suitable for processing with `ProcessLadder`.

`x in C`

Returns `true` if element g of group G lies in the coset C.

`x notin C`

Returns `true` if element g of group G does not lie in the coset C.

`C1 eq C2`

Returns `true` if the coset C_1 is equal to the coset C_2.

`C1 ne C2`

Returns `true` if the coset C_1 is not equal to the coset C_2.

`#C`

The cardinality of the coset C.

`CosetTable(G, H)`

The (right) coset table for G over subgroup H relative to its defining generators.

`#CosetTable(G, f)`

The coset table for G corresponding to the permutation representation f of G, where f is a homomorphism of G onto a transitive permutation group.

58.12.2 Transversals

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a permutation group G and a subgroup H of G, this function returns

(a) An indexed set of elements T of G forming a right transversal for G over H; and

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and $g \in G$, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

`TransversalProcess(G, H)`

Given a permutation group G and H, a subgroup of G, create a process to run through a left transversal for H in G. The method used is a backtrack search for a canonical coset representative. `TransversalProcess` can be used when the index of H in G is too large to allow a full transversal to be created.

`TransversalProcessRemaining(P)`

The number of coset representatives the process has yet to produce. Initially this will be the index of the subgroup in the group.

`TransversalProcessNext(P)`

Advance the process to the next coset representative and return that representative. This may only be used when `TransversalProcessRemaining(P)` is positive. The first call to `TransversalProcessNext` will always give the identity element.

`ShortCosets(p, H, G)`

Computes a set of representatives for the transversal of G modulo H of all cosets that contain p. This computation does not do a full transversal of G modulo H and may therefore be used even if the index of $(G : H)$ is very large.

58.13 Presentations

In this section we describe how to compute a presentation in terms of generators and relations for a permutation group and also how to obtain a representation of a permutation as word in the defining generators.

58.13.1 Generators and Relations

`FPGroup(G)`

Construct a presentation for the permutation group G on the set of defining generators and return the presentation in the form of a finitely presented group F that is isomorphic to G. The presentation is obtained by first computing the regular representation of G and then using the Todd-Coxeter Schreier algorithm to construct a presentation on the strong generators. In this situation the strong generators are identical to the defining generators.

A group homomorphism $\phi : F \to G$, defining G as a permutation representation of F, is also returned.

`FPQuotient(G, N)`

Given a normal subgroup N of G, compute an fp-group representation F of the quotient G/N and the homomorphism $\phi : G \to F$.

`FPGroupStrong(G: parameters)`

`Random`	BoolElt	*Default :* `true`
`Run`	RngIntElt	*Default :* 20

Construct a presentation for the permutation group G on a set of strong generators and return the presentation in the form of a finitely presented group F that is isomorphic to G. In Magma, a combination of the Schreier Todd-Coxeter Sims algorithm and the Brownie-Cannon-Sims verification procedure is used to construct the presentation. See Leon [Leo80] and Gebhardt [Geb00] for more details of the individual algorithms.

If strong generators are not already known for G, they will be constructed. If strong generators have to be constructed, the parameters `Random` and `Run` may be used to control the application of the random schreier algorithm to construct a probable BSGS before commencing the construction of the presentation. If `Random` is set to false then no randomising is performed, and the algorithm becomes the straight STCS algorithm. In the case in which strong generators are already known for G, the presentation will be on these strong generators.

The presentation will have the property that it includes a presentation for each group in the stabilizer chain of the BSGS.

The group isomorphism $\phi : F \to G$, defining G as a permutation representation of F, is also returned.

58.13.2 Permutations as Words

Consider a permutation group G defined on d generators. The *word group* of G is a free group W of rank d. Then we regard G as a homomorphic image of F with associated homomorphism $\phi : W \to G$. All operations involving words in the generators of G will be performed in W.

`WordGroup(G)`

Given a permutation group G defined on d generators, return (a) a free group W on d generators represented as a group whose elements are defined by straight-line programs (SLP group), and (b) the homomorphism ϕ from W to G such that $W.i \to G.i$, for $i = 1, \ldots, d$. The group W associated with G by this function will be referred to as the *word group* for G.

`InverseWordMap(G)`

Given a permutation group G and its associated word group W with canonical homomorphism $\phi : W \to G$, construct the inverse mapping ρ. Thus, given a permutation g of G, $g@\rho$ returns an element in the preimage of g under ϕ. If the word group W does not already exist, it will be created.

`ActingWord(G, x, y)`

Given points x and y belonging to the same G-orbit of the natural G-set X, return a word w in the word group W of G such that $x^{\phi(w)} = y$. Here ϕ is the canonical homomorphism from W to G.

58.14 Automorphism Groups

The automorphism group of a permutation group may be computed in MAGMA, subject to the same restrictions on the group as when computing maximal subgroups. (That is, the non-abelian composition factors of the group must appear in a certain database.) The methods used are those described in Cannon and Holt [CH03]. Isomorphism of permutation groups may also be determined using the same methods.

`AutomorphismGroup(G: parameters)`

Compute the full automorphism group of the permutation group G.

`IsIsomorphic(G, H: parameters)`

Test whether or not the two permutation groups G and H are isomorphic as abstract groups. If so, both the result `true` and an isomorphism from G to H is returned. If not, the result `false` is returned.

Example H58E35

We take some groups of order 120 and test for isomorphism.

```
> G1 := PermutationGroup<20 |
>  [ 2, 5, 9, 11, 12, 3, 17, 13, 18, 16, 7, 15, 10, 8, 1,
>   14, 20, 19, 6, 4 ],
>  [ 3, 6, 1, 10, 14, 2, 18, 17, 15, 4, 16, 13, 12, 5, 9,
>   11, 8, 7, 20, 19 ] >;
> #G1;
120
> G2 := PermutationGroup<24 |
```

```
>   [ 2, 4, 6, 5, 1, 7, 8, 3, 13, 15, 14, 16, 11, 9, 12, 10,
>   19, 20, 18, 17, 24, 21, 22, 23 ],
>   [ 3, 1, 2, 7, 4, 9, 5, 11, 10, 6, 12, 8, 16, 15, 18, 17,
>   13, 14, 21, 23, 22, 19, 24, 20 ],
>   [ 4, 5, 7, 1, 2, 8, 3, 6, 11, 12, 9, 10, 14, 13, 16, 15,
>   18, 17, 20, 19, 23, 24, 21, 22 ] >;
> #G2;
120
> IsIsomorphic(G1, G2);
false
> flag, isom := IsIsomorphic(G1, Sym(5));
> flag;
true
> (G1.1)@ isom;
(1, 3, 5, 4, 2)
```

The reader is invited to check that `G2` is perfect while `G1` is not, so the `false` result for their isomorphism is correct. What is the automorphism group of `G2`?

```
> A := AutomorphismGroup(G2);
> #A;
120
> #Centre(G2);
2
> OuterFPGroup(A);
Finitely presented group on 1 generator
Relations
  $.1^2 = Id($)
> A.1;
Automorphism of GrpPerm: G2, Degree 24, Order 2^3 * 3 * 5
which maps:
  (1, 2, 4, 5)(3, 6, 7, 8)(9, 13, 11, 14)(10, 15, 12,
    16)(17, 19, 18, 20)(21, 24, 23, 22) |--> (1, 16, 4,
    15)(2, 21, 5, 23)(3, 20, 7, 19)(6, 11, 8, 9)(10, 24, 12,
    22)(13, 17, 14, 18)
  (1, 3, 2)(4, 7, 5)(6, 9, 10)(8, 11, 12)(13, 16, 17)(14,
    15, 18)(19, 21, 22)(20, 23, 24) |--> (1, 11, 16)(2, 18,
    19)(3, 23, 6)(4, 9, 15)(5, 17, 20)(7, 21, 8)(10, 22,
    13)(12, 24, 14)
  (1, 4)(2, 5)(3, 7)(6, 8)(9, 11)(10, 12)(13, 14)(15,
    16)(17, 18)(19, 20)(21, 23)(22, 24) |--> (1, 4)(2, 5)(3,
    7)(6, 8)(9, 11)(10, 12)(13, 14)(15, 16)(17, 18)(19,
    20)(21, 23)(22, 24)
> IsInnerAutomorphism(A.4);
false
```

So the outer automorphism group of `G2` has order 2, and `A.4` gives this automorphism.

58.15 Cohomology

In the following description, G is a finite permutation group, p is a prime number, and K is the finite field of order p. Further, F is a finitely presented group having the same number of generators as G, and is such that its relations are satisfied by the corresponding generators of G. In other words, the mapping taking the i-th generator of F to the i-th generator of G must be an epimorphism. Usually this mapping will be an isomorphism, although this is not mandatory. The algorithms used are those of Holt, see [Hol84], [Hol85a] and [Hol85b].

`pMultiplicator(G, p)`

Given the group G and a prime p, return the invariant factors of the p-part of the Schur multiplicator of G.

`pCover(G, F, p)`

Given the group G and the finitely presented group F such that G is an epimorphic image of G in the sense described above, return a presentation for the p-cover of G, constructed as an extension of the p-multiplier by F.

`CohomologicalDimension(G, M, i)`

Given the group G, the $K[G]$-module M and an integer i (equal to 1 or 2), return the dimension of the i-th cohomology group of G acting on M.

`ExtensionProcess(G, M, F)`

Create an extension process for the group G by the module M.

`Extension(P, Q)`

`#NextExtension(P)`

Return the next extension of G as defined by the process P.

Assume that F is isomorphic to the permutation group G, and that we wish to determine presentations for one or more extensions of the K-module M by F, where K is the field of p elements. We first create an extension process using `ExtensionProcess(G, M, F)`. The possible extensions of M by G are in one-one correspondence with the elements of the second cohomology group $H^2(G, M)$ of G acting on M. Let $b_1, \ldots, b_l$ be a basis of $H^2(G, M)$. A general element of $H^2(G, M)$ therefore has the form $a_1 b_1 + \cdots + a_l b_l$ and so can be defined by a sequence Q of l integers $[a_1, \ldots, a_l]$. Now, to construct the corresponding extension of M by G we call the function `Extension(P, Q)`. The required extension is returned as a finitely presented group. If all the extensions are required then they may be obtained successively by making p^l calls to the function `NextExtension`.

`SplitExtension(G, M, F)`

The split extension of the module M by the group G.

Example H58E36

We construct a presentation for A_6 over its Schur multiplicator. First we find the size of the multiplicator by applying the `pMultiplicator` function to each relevant prime.

```
> G := Alt(6);
> &cat [pMultiplicator(G, p[1]): p in FactoredOrder(G)];
[ 2, 3, 1 ]
```

The multiplicator has order $2 \times 3 = 6$. We next construct the two-fold cover of A_6. We use the `FPGroup` function to get a presentation for A_6.

```
> F := FPGroup(G);
> F2 := pCover(G, F, 2);
```

Now we construct a three-fold cover of the two-fold cover to get the extension we are after. First we need a permutation representation of `F2`, the two-fold covering group.

```
> G2 := DegreeReduction(CosetImage(F2, sub<F2|>));
> Degree(G2);
144
> #G2;
720
> F6 := pCover(G2, F2, 3);
> F6;
Finitely presented group F6 on 4 generators
Relations
  F6.4^3 = Id(F6)
  (F6.1, F6.4) = Id(F6)
  (F6.2, F6.4) = Id(F6)
  (F6.3, F6.4) = Id(F6)
  F6.3^2 = Id(F6)
  (F6.1, F6.3) = Id(F6)
  (F6.2, F6.3) = Id(F6)
  F6.1^4 * F6.3 = Id(F6)
  F6.2^3 * F6.3 = Id(F6)
  F6.1^-1 * F6.2^-1 * F6.1 * F6.2 * F6.1^-1 * F6.2^-1 *
  F6.1 * F6.2 * F6.3 * F6.4 = Id(F6)
  F6.1^-1 * F6.2 * F6.1^2 * F6.2 * F6.1^2 * F6.2 *
  F6.1^-2 * F6.2 * F6.1^-1 * F6.3 * F6.4^-1 = Id(F6)
  F6.2 * F6.1 * F6.2 * F6.1 * F6.2 * F6.1 * F6.2 *
  F6.1 * F6.2 * F6.1 * F6.3 * F6.4 = Id(F6)
> AbelianQuotientInvariants(F6);
[]
```

The group `F6` is the six-fold cover of A_6. We easily see from the presentation that the 3rd and 4th generators generate a central cyclic subgroup of order 6. The sequence of invariants for the maximal abelian quotient of `F6` is empty, so `F6` is perfect.

Example H58E37

We construct an extension of A_5. This time the normal subgroup will be elementary abelian of order 2^5, with the action of A_5 being the natural permutation action.

```
> G := Alt(5);
> M := PermutationModule(G, GF(2));
> CohomologicalDimension(G, M, 2);
1
```

The dimension of the 2nd cohomology group is 1 over $\mathbf{F}_2$, so there are two possible extensions. We will construct them both.

```
> F := FPGroup(G);
> P := ExtensionProcess(G, M, F);
> E0 := Extension(P, [0]);
> E1 := Extension(P, [1]);
> AbelianQuotientInvariants(E0);
[ 2 ]
> AbelianQuotientInvariants(E1);
[]
```

The split extension, `E0`, is not perfect, but the non-split extension, `E1`, is a perfect group.

58.16 Representation Theory

A set of functions are provided for computing with the characters of a group. Full details of these functions may be found in Chapter 91. For convenience we include here two of the more useful character functions.

Also, functions are provided for computing with the modular representations of a group. Full details of these functions may be found in Chapter 89. For the reader's convenience we include here the functions which may be used to define a $R[G]$-module for a permutation group.

`CharacterTable(G: parameters)`

Construct the table of ordinary irreducible characters for the group G.

`Al` MonStgElt *Default :* "*Default*"

This parameter controls the algorithm used. The string `"DS"` forces use of the Dixon-Schneider algorithm. The string `"IR"` forces the use of Unger's induction/reduction algorithm [Ung06]. The `"Default"` algorithm is to use Dixon-Schneider for groups of order ≤ 5000 and Unger's algorithm for larger groups. This may change in future.

`DSSizeLimit` RngIntElt *Default :* 0

When the default algorithm is selected, a positive value n for `DSSizeLimit` means that before using Unger's algorithm, the full character space is split by some passes of Dixon-Schneider, restricted to using class matrices corresponding to conjugacy classes with size at most n.

`PermutationCharacter(G)`

Given a group G represented as a permutation group, construct the character of G afforded by the defining permutation representation of G.

`PermutationCharacter(G, H)`

Given a group G and some subgroup H of G, construct the ordinary character of G afforded by the permutation representation of G given by the action of G on the coset space of the subgroup H in G.

`GModule(G, S)`

Let G be a group defined on r generators and let S be a subalgebra of the matrix algebra $M_n(R)$, also defined by r non-singular matrices. It is assumed that the mapping from G to S defined by $\phi(G.i) \to S.i$, for $i = 1, \ldots, r$, is a group homomorphism. Let M be the natural module for the matrix algebra S. The function `GModule` gives M the structure of an $S[G]$-module, where the action of the i-th generator of G on M is given by the i-th generator of S.

`GModule(G, A, B)`

Given a finite group G, a normal subgroup A of G and a normal subgroup B of A such that the section A/B is elementary abelian of order p^n, create the $K[G]$-module M corresponding to the action of G on A/B, where K is the field $\mathbf{F}_p$. If B is trivial, it may be omitted. The function returns

(a) the module M; and

(b) the homomorphism $\phi : A/B \to M$.

`PermutationModule(G, H, R)`

Given a finite group G and a ring R, create the $R[G]$-module for G corresponding to the permutation action of G on the cosets of H.

`PermutationModule(G, R)`

Given a finite permutation group G and a ring R, create the natural permutation module for G over R.

Example H58E38

We refine an elementary abelian normal subgroup of a permutation group to a sequence of normal subgroups.

```
> G := PermutationGroup<24 |
>  [ 3, 4, 1, 2,23,24, 7, 8, 9,10,12,11,14,13,16,15,18,17,22,21,
>    20,19, 5, 6 ],
>  [ 7, 8,11,12,13,14,22,21,20,19,15,16,17,18, 6, 5, 4, 3, 1, 2,23,
>    24, 9,10 ] >;
> N := sub<G |
>  [ 24, 23, 6, 5, 4, 3, 10, 9, 8, 7, 14, 13, 12, 11, 18, 17, 16, 15, 22, 21,
>    20, 19, 2, 1 ],
```

```
>  [ 23, 24, 5, 6, 3, 4, 8, 7, 10, 9, 12, 11, 14, 13, 15, 16, 17, 18, 19, 20,
>    21, 22, 1, 2 ],
>  [ 2, 1, 4, 3, 6, 5, 7, 8, 9, 10, 11, 12, 13, 14, 17, 18, 15, 16, 21, 22, 19,
>    20, 24, 23 ]>;
> #N;
8
> IsNormal(G, N);
true
> IsElementaryAbelian(N);
true
> M, f := GModule(G, N);
> SM := Submodules(M);
> #SM;
4
> refined := [ x @@ f : x in SM ];
> forall{x : x in refined | IsNormal(G, x) };
true;
> [ #x : x in refined];
[ 1, 2, 4, 8 ]
```

The original elementary abelian normal subgroup of order 8 is the top of a chain of normal subgroups of length 3.

58.17 Identification

58.17.1 Identification as an Abstract Group

`NameSimple(G)`

Given a simple group G, determine the isomorphism type of G. The type is returned in the form of a triple of three integers f, d and q, where the interpretation of these integers is that given in the description of the function `CompositionFactors`.

58.17.2 Identification as a Permutation Group

The first functions described in this subsection detect whether or not a permutation group is alternating or symmetric in its natural representation. They are based on the algorithm 'Detect Alternating' outlined in [CB92].

`IsAlternating(G)`

Returns `true` if the permutation group G defined as acting on X is the alternating group $\mathrm{Alt}(X)$.

`IsSymmetric(G)`

Returns `true` if the permutation group G defined as acting on X is the symmetric group $\mathrm{Sym}(X)$.

`IsAltsym(G)`

Returns `true` if the permutation group G defined as acting on X contains the alternating group $\mathrm{Alt}(X)$.

`TwoTransitiveGroupIdentification(G)`

Given a 2-transitive group G, return a tuple giving the abstract isomorphism type of the group. This is an implementation of the method of Cameron and Cannon [CC91].

`RecogniseAlternatingOrSymmetric(G, n)`

Constructive recognition of the group G, which will succeed with probability $\geq 1-e^5$ if G is isomorphic to either the alternating or symmetric group of degree $n > 11$. The method is that of Beals *et al* [BLGN$^+$03], implemented by Colva Roney-Dougal.

The return values start with a flag indicating success or failure. If the algorithm was successful, then there are three more return values: a flag which is true when G is symmetric and false when alternating, and two programs. The first program takes an element x of an overgroup of G and produces a boolean to indicate whether $x \in G$ and a permutation representing x in the natural action of S_n (if such a permutation exists). The second taking a permutation to the corresponding element of G. The programs define mutually inverse group isomorphisms, implemented as MAGMA functions.

`IsEven(G)`

Given a permutation group G check if G is even, ie. contained in the alternating group.

Example H58E39

We give an example of `RecogniseAlternatingOrSymmetric` in use.

```
> SetSeed(1);
> a:= AlternatingGroup(13);
> h:= Stabiliser(a, {1,2});
> k:= CosetImage(a, h);
> Degree(k);
78
> worked, is_sym, bb_to_perm, perm_to_bb:=
> RecogniseAlternatingOrSymmetric(k, 13);
> worked;
true
> is_sym;
false
> x:= Sym(78)!(1, 35, 16, 28, 14, 26, 69, 5, 74)(2, 54,
> 67, 18, 51, 63, 6, 50, 77)(3, 33, 78, 12, 34, 29, 19, 15, 73)
> (4, 52, 61, 24, 49, 60, 68, 38, 64)(7, 20, 71, 17,
> 32, 11, 72, 8, 36)(9, 76, 47, 31, 56, 62, 13, 53, 59)
```

```
> (10, 70, 57, 23, 37, 22, 21, 27, 25)(30, 45, 46, 43, 42,
> 44, 40, 41, 75)(39, 55, 65)(48, 66, 58);
> x in k;
true;
> in_k, perm_image:= bb_to_perm(x);
> in_k;
true
> perm_image;
(1, 2, 3)(4, 7, 12, 6, 10, 11, 13, 9, 8)
> perm_to_bb(perm_image) eq x;
true
```

`RecogniseSymmetric(G, n: parameters)`

`maxtries`	RngIntElt	*Default :* $100n + 5000$
`Extension`	BoolElt	*Default :* `false`

The group G should be known to be isomorphic to the symmetric group S_n for some $n \geq 8$. The Bratus-Pak algorithm [BP00] (implemented by Derek Holt) is used to define an isomorphism between G and S_n. If successful, return `true`, homomorphism from G to S_n, homomorphism from S_n to G, the map from G to its word group and the map from the word group to G.

If the optional parameter `Extension` is set, then the group G should be known to be isomorphic either to S_n or to a perfect central extension $2.S_n$. In that case, the first two maps returned will be a homomorphism from G to S_n and a map from S_n to G that induces a homomorphism onto $G/Z(G)$. The sixth value returned will be `true`, if $G \cong 2.S_n$ and `false`, if $G \cong 2.A_n$.

If unsuccessful, `false` is returned. This will always occur if the input group is not isomorphic to S_n (or $2.S_n$ when `Extension` is set) with $n \geq 8$, and may occur occasionally even when G is isomorphic to S_n. The optional parameter `maxtries` (default $100n + 5000$) can be used to control the number of random elements chosen before giving up.

`SymmetricElementToWord (G, g)`

If g is an element of G which has been constructively recognised to be isomorphic to S_n (or $2.S_n$), then return `true` and element of word group for G which evaluates to g. Otherwise return `false`. This facilitates membership testing in G.

`RecogniseAlternating(G, n: parameters)`

`maxtries`	RngIntElt	*Default :* $100n + 5000$
`Extension`	BoolElt	*Default :* `false`

The group G should be known to be isomorphic to the alternating group A_n for some $n \geq 9$. The Bratus-Pak algorithm [BP00] (implemented by Derek Holt) is used to define an isomorphism between G and A_n. If successful, return `true`, homomorphism from G to A_n, homomorphism from A_n to G, the map from G to its word group and the map from the word group to G.

If the optional parameter `Extension` is set, then the group G should be known to be isomorphic either to A_n or to a perfect central extension $2.A_n$. In that case, the first two maps returned will be a homomorphism from G to A_n and a map from A_n to G that induces a homomorphism onto $G/Z(G)$. The sixth value returned will be `true`, if $G \cong 2.A_n$ and `false`, if $G \cong 2.A_n$.

If unsuccessful, `false` is returned. This will always occur if the input group is not isomorphic to A_n (or $2.A_n$ when `Extension` is set) with $n \geq 9$, and may occur occasionally even when G is isomorphic to A_n. The optional parameter `maxtries` (default $100n + 5000$) can be used to control the number of random elements chosen before giving up.

`AlternatingElementToWord (G, g)`

If g is an element of G which has been constructively recognised to be isomorphic to A_n (or $2.A_n$), then return `true` and element of word group for G which evaluates to g. Otherwise return `false`. This facilitates membership testing in G.

`GuessAltsymDegree(G: parameters)`

`maxtries`	RngIntElt	*Default :* 5000
`Extension`	BoolElt	*Default :* `false`

The group G should be believed to be isomorphic to S_n or A_n for some $n > 6$, or to $2.S_n$ or $2.A_n$ if the optional parameter `Extension` is set. This function attempts to determine n and whether G is symmetric or alternating. It does this by sampling orders of elements. It returns either `false`, if it is unable to make a decision after sampling `maxtries` elements (default 5000), or `true`, *type* and n, where *type* is "Symmetric" or "Alternating", and n is the degree. If G is not isomorphic to S_n or A_n (or $2.S_n$ or $2.A_n$ when `Extension` is set) for $n > 6$, then the output is meaningless - there is no guarantee that `false` will be returned. There is also a small probability of a wrong result or `false` being returned even when G is S_n or A_n with $n > 6$. This function was written by Derek Holt.

Example H58E40

For a group G which is believed to be isomorphic to S_n or A_n for some unknown value of $n > 6$, the function `GuessAltsymDegree` can be used to try to guess n, and then `RecogniseSymmetric` or `RecogniseAlternating` can be used to confirm the guess.

```
> SetSeed(1);
```

```
> G:= sub< GL(10,5) |
> PermutationMatrix(GF(5),Sym(10)![2,3,4,5,6,7,8,9,1,10]),
> PermutationMatrix(GF(5),Sym(10)![1,3,4,5,6,7,8,9,10,2]) >;
> GuessAltsymDegree(G);
true Alternating 10
> flag, m1, m2, m3, m4  := RecogniseAlternating(G,10);
> flag;
true
> x:=Random(G); Order(x);
8
> m1(x);
(1, 2, 4, 9, 10, 8, 6, 3)(5, 7)
> m2(m1(x)) eq x;
true
> m4(m3(x)) eq x;
true
> flag, w := AlternatingElementToWord(G,x);
> flag;
true
> m4(w) eq x;
true
> y := Random(Generic(G));
> flag, w := AlternatingElementToWord(G,y);
> flag;
false
> flag, m1, m2, m3, m4 := RecogniseAlternating(G,11);
> flag;
false
> flag, m1, m2, m3, m4 := RecogniseSymmetric(G,10);
> flag;
false
```

The nature of the `GuessAltsymDegree` function is that it assumes that its input is either an alternating or symmetric group and then tries to guess which one and the degree. As such, it is almost always correct when the input is an alternating or symmetric group, but will often return a bad guess when the input group is not of this form, as in the following example.

```
> GuessAltsymDegree(Sym(50));
true Symmetric 50
> GuessAltsymDegree(Alt(73));
true Alternating 73
> GuessAltsymDegree(PSL(5,5));
true Alternating 82
```

58.18 Base and Strong Generating Set

The key concept for representing a permutation group is that of a *base and strong generating set* (BSGS). Given a BSGS for a group, its order may be deduced immediately. Brownie, Cannon and Sims (1991) showed that it is practical, in some cases at least, to construct a BSGS for short-base groups having degree up to ten million.

The great majority of functions for computing with permutation groups require a BSGS to be present. If one is not known, MAGMA will attempt to automatically compute one. For large degree groups, the computation of a BSGS may be expensive and in such cases the user may achieve better performance through directly invoking a function which creates a BSGS. For example, if the group order is known in advance, it may be supplied to MAGMA and then a random method for computing a BSGS is applied which will use the group order as a termination condition.

In the first part of this section we present the elementary functions that use a BSGS, while towards the end we describe firstly, functions which allow the user to select and control the algorithm employed, and secondly, functions which provide access to the BSGS data structures. The material specific to BSGS should be omitted on a first reading.

58.18.1 Construction of a Base and Strong Generating Set

Computing structural information for a permutation group G requires, in most cases, a representation of the set of elements of G. MAGMA represents this set by means of a *base and strong generating set*, or *BSGS*, for G. Suppose the group G acts on the set Ω. A *base* B for G is a sequence of distinct points from Ω with the property that the identity is the only element of G that fixes B pointwise. A base B of length n determines a sequence of subgroups $G^{(i)}$, $1 \leq i \leq n+1$, where $G^{(i)}$ is the stabilizer of the first $i-1$ points of B. (In particular, $G^{(1)} = G$ and $G^{(n+1)}$ is trivial.) Given a base B for G, a subset S of G is said to be a *strong generating set* for G (with respect to B) if $G^{(i)} = \langle S \cap G^{(i)} \rangle$, for $i = 1, \ldots, n$.

`BSGS(G)`

The general procedure for constructing a BSGS. This version uses the default algorithm choices.

`SimsSchreier(G: parameters)`

`SV` BOOLELT *Default :* `true`

Construct a base and strong generating set for the group G using the standard Schreier-Sims algorithm. If the parameter `SV` is set `true` (default) the transversals are stored in the form of Schreier vectors. If `SV` is set `false`, then the transversals are stored both as lists of permutations and as Schreier vectors. If the base attribute has been previously defined for G, a variant of the Sims-Schreier algorithm will be employed, in which permutation multiplications are replaced by base image calculations wherever possible.

`RandomSchreier(G: parameters)`

`Max`	RngIntElt	*Default :* 100
`Run`	RngIntElt	*Default :* 20

Construct a probable base and strong generating set for the group G. The strong generators are constructed from a set of randomly chosen elements of G. The expectation is that, if sufficient random elements are taken, then, upon termination, the algorithm will have produced a BSGS for G. If the attribute Order is defined for G, the random Schreier will continue until a BSGS defining a group of the indicated order is obtained. In such circumstances this method is the fastest method of constructing a base and strong generating set for G. This is particularly so for groups of large degree. If nothing is known about G, the random Schreier algorithm provides a cheap way of obtaining lower bounds on the group's order and, in the case of a permutation group, on its degree of transitivity. This parameter has two associated parameters, `Max` and `Run`, which take positive integer values. The parameter `Max` specifies the number of random elements to be used (default 100). If the value of `Run` is n_2, then the algorithm terminates after n_2 consecutive random elements are found to lie in the set defined by the current BSGS (default 20). The two limits are independent of one another. It should be emphasized that unpredictable results may arise if the programmer uses the base and strong generators produced by this algorithm when, in fact, it does not constitute a BSGS for G.

`ToddCoxeterSchreier(G: parameters)`

Construct a BSGS for G using the Todd-Coxeter Schreier algorithm.

`SolubleSchreier(G: parameters)`

`SolvableSchreier(G: parameters)`

`Depth`	RngIntElt	*Default : See below*

Construct a base and strong generating set for the soluble permutation group G using the algorithm of Sims [Sim90]. The algorithm proceeds by recursively constructing the terms of the derived series. If G is not soluble then the algorithm will not terminate. In order to avoid non-termination, a limit on the number of terms in the normal subgroup chain constructed must be prescribed. The user may set this limit as the value of the parameter `Depth`. The default value, $\lceil 1.6 \ log_2 \ Degree(G) \rceil$, is based on an upper limit (due to Dixon) on the length of the derived series of a soluble permutation group. This algorithm is often significantly faster than the general Schreier-Sims algorithm.

`Verify(G: parameters)`

`Levels`	RngIntElt	*Default :* 0
`OrbitLimit`	RngIntElt	*Default :* 4,000

Given a permutation group G for which a probable BSGS is stored, verify the correctness of the BSGS. If it is not complete, proceed to complete it. The two parameters

`Levels` and `OrbitLimit` define how many levels the Todd-Coxeter-Schreier-Sims verifies before switching to the Brownie-Cannon-Sims algorithm. If `Levels` is set to n non-zero then n levels are verified by the TCSS algorithm before switching. If `Levels` is zero, the switch-over point is determined by the value of the parameter `OrbitLimit`. All levels with basic orbit length at most `OrbitLimit` are verified using TCSS. When a level is encountered with orbit length greater than this, a decision based on expected amount of work to do for this level by each algorithm determines what strategy is used for this level. Once one level uses the BCS method, all levels from then on will use it.

Example H58E41

The Higman-Sims simple group represented on 100 letters is generated by two permutations. To create a base and strong generating set for it using the Todd-Coxeter-Schreier algorithm, we can use the `ToddCoxeterSchreier` procedure as follows:

```
> G := sub<Sym(100) |
>    (2,8,13,17,20,22,7)(3,9,14,18,21,6,12)(4,10,15,19,5,11,16)
>        (24,77,99,72,64,82,40)(25,92,49,88,28,65,90)(26,41,70,98,91,38,75)
>        (27,55,43,78,86,87,45)(29,69,59,79,76,35,67)(30,39,42,81,36,57,89)
>        (31,93,62,44,73,71,50)(32,53,85,60,51,96,83)(33,37,58,46,84,100,56)
>        (34,94,80,61,97,48,68)(47,95,66,74,52,54,63),
>    (1,35)(3,81)(4,92)(6,60)(7,59)(8,46)(9,70)(10,91)(11,18)(12,66)(13,55)
>        (14,85)(15,90)(17,53)(19,45)(20,68)(21,69)(23,84)(24,34)(25,31)(26,32)
>        (37,39)(38,42)(40,41)(43,44)(49,64)(50,63)(51,52)(54,95)(56,96)(57,100)
>        (58,97)(61,62)(65,82)(67,83)(71,98)(72,99)(74,77)(76,78)(87,89) >;
> ToddCoxeterSchreier(G);
> Order(G);
44352000
```

Example H58E42

The simple group of Rudvalis has a permutation representation of degree 4060. A generating set for the Rudvalis group, Ru, may be found in the standard MAGMA database `pergps`, where it is called `ru`. We use the random Schreier algorithm followed by the `Verify` procedure to produce a base and strong generating set. We increase the limits for `RandomSchreier` to increase the probability that a complete base and strong generating set is found. This is done as follows:

```
> load "ru";
> RandomSchreier(G : Max := 50, Run := 20);
> Order(G);
145926144000
> Verify(G);
> Order(G);
145926144000
> Base(G);
[ 1, 2, 3, 4 ]
> BasicOrbitLengths(G);
```

```
[ 4060, 2304, 780, 20 ]
```

58.18.2 Defining Values for Attributes

If the order of a permutation group is known in advance, the construction of a base and strong generating set can be greatly speeded up by taking advantage of this knowledge. The `AssertAttribute` constructor may be used to communicate this and other useful information to Magma.

```
AssertAttribute(G, "Order", n)
```

Define the order attribute for the permutation group G.

```
AssertAttribute(G, "Order", Q)
```

Define the (factored) order of the permutation group G to be Q.

```
#AssertAttribute(G, "BSGS", S)
```

Define the base and strong generating set structure S to be the BSGS for G.

Example H58E43

The ability to set the order provides a short cut when constructing a BSGS. If the order attribute is set and the random Schreier-Sims algorithm applied, it will run until a BSGS for a group of the designated order has been constructed. We illustrate this in the case of the wreath product, with product action, of Sym(42) with Alt(8).

```
> G := WreathProduct(Sym(42), Alt(8));
> AssertAttribute(G, "Order", Factorial(42)^8 * (Factorial(8) div 2));
> RandomSchreier(G);
> Order(G);
306137301672361016520312745630712212453532972657840432\
7
542803407318869103099925605266692478702295013089037189\
1
052508966519418763874734093840686118134015088994465475\
2
702520725584513027443455822163386921007988258197506974\
2
167205546625091429100257027549968576864624041105576609\
8
237073911069065187521567666323553412613827078144015568\
3
990651535360000000000000000000000000000000000000000000\
0
00000000000000000000000000
```

58.18.3 Accessing the Base and Strong Generating Set

`Base(G)`

A base for the permutation group G. The base is returned as a sequence of points of its natural G-set. If a base is not known, one will be constructed.

`BasePoint(G, i)`

The i-th base point for the permutation group G. A base and strong generating set must be known for G.

`BasicOrbit(G, i)`

The basic orbit at level i as defined by the current base for the permutation group G. This function assumes that a BSGS is known for G.

`BasicOrbits(G)`

The basic orbits as defined by the current base for the permutation group G. This function assumes that a BSGS is known for G. The orbits are returned as a sequence of indexed sets.

`BasicOrbitLength(G, i)`

The length of the basic orbit at level i as defined by the current base for the permutation group G. This function assumes that a BSGS is known for G.

`BasicOrbitLengths(G)`

The lengths of the basic orbits as defined by the current base for the permutation group G. This function assumes that a BSGS is known for G. The lengths are returned as a sequence of integers.

`BasicStabilizer(G, i)`

`BasicStabiliser(G, i)`

Given a permutation group G for which a base and strong generating set are known, and an integer i, where $1 \leq i \leq k$ with k the length of the base, return the subgroup of G which fixes the first $i-1$ points of the base.

`BasicStabilizerChain(G)`

`BasicStabiliserChain(G)`

Given a permutation group G, return the stabilizer chain defined by the base as a sequence of subgroups of G. If a BSGS is not already known for G, it will be created.

`IsMemberBasicOrbit(G, i, a)`

Returns `true` if the point a of Ω lies in the basic orbit at level i. This function assumes that a BSGS is known for G.

`NumberOfStrongGenerators(G)`

`Nsgens(G)`

The number of elements in the current strong generating set for G.

`NumberOfStrongGenerators(G, i)`

`Nsgens(G, i)`

The number of elements in the strong generating set for the i-th term of the stabilizer chain for G.

`SchreierVectors(G)`

The Schreier vectors corresponding to the current BSGS for the permutation group G. The vectors are returned as a sequence of integer sequences.

`SchreierVector(G, i)`

The Schreier vector corresponding the i-th term of the stabilizer chain defined by the current BSGS for the permutation group G. The vector is returned as a sequence of integers.

`StrongGenerators(G)`

A set of strong generators for the permutation group G. If they are not currently available, they will be computed.

`StrongGenerators(G, i)`

A set of strong generators for the i-th term in the stabilizer chain for the permutation group G. A BSGS must be known for G.

58.18.4 Working with a Base and Strong Generating Set

`BaseImage(x)`

Given a permutation x belonging to the group G, for which a base and strong generating set is known, form the base image of x.

`Permutation(G, Q)`

Given a permutation group G acting on the set Ω, for which a base and strong generating set are known, and a sequence Q of distinct points of Ω defining an element x of G, return x as a permutation.

`SVPermutation(G, i, a)`

The permutation of G defined by the Schreier vector at level i, which takes the point a of Ω to the base point at level i. This function assumes that a BSGS is known for G.

`SVWord(G, i, a)`

An element in the word group of G defined by the Schreier vector at level i, which takes the point a of Ω to the base point at level i. This function assumes that a BSGS is known for G.

`Strip(H, x)`

Given an element x of a permutation group G, and given a group H for which a base and strong generating set is known, returns:

(a) the value of `x in H`

(b) The residual permutation y resulting from the stripping of x with respect to the BSGS for H; and

(c) The first level i such that y is not contained in $H^{(i)}$.

`WordStrip(H, x)`

Given an element x of a permutation group G, and given a group H for which a base and strong generating set is known, returns:

(a) the value of `x in H`

(b) the residual word w (an element in the word group of G) resulting from the stripping of x with respect to the BSGS for H,

(c) The first level i such that y is not contained in $H^{(i)}$.

`BaseImageWordStrip(H, x)`

Given an element x of a permutation group G, and given a group H for which a base and strong generating set is known, returns:

(a) Whether the base image strip succeeded at all levels. Note that a true value here does *not*, on its own, imply $x \in H$.

(b) the residual word w (an element in the word group of G) resulting from the stripping of x with respect to the BSGS for H,

(c) The first level i such that the strip could not continue.

`WordInStrongGenerators(H, x)`

Given an element x of a permutation group H for which a base and strong generating set is known, returns a word in the strong generators of H which represents x. This function uses base images to determine the word for x, so giving $x \notin H$ will have unpredictable results. This function returns the inverse of the second return value of `BaseImageWordStrip`, when the latter is successful.

58.18.5 Modifying a Base and Strong Generating Set

`ChangeBase(~G, Q)`

Given a group H with a base and strong generating set, change the base of G, so that the points in the sequence Q form an initial segment of the new base.

`AddNormalizingGenerator(~H, x)`

Given a group H with a base B and strong generating set X, and an element x that normalizes H belonging to a group that contains H, extend the existing BSGS for H so that they form a BSGS for the subgroup $< H, x >$.

`ReduceGenerators(~G)`

Given a group G with a base and strong generating set, remove redundant strong generators.

58.19 Permutation Representations of Linear Groups

Each of the functions in this family returns two values:

(a) A permutation group G corresponding to the action of a designated matrix group M on a vector space V; and

(b) An indexed set of affine or projective points on which M acts, such that the indexing gives the correspondence between this set and the G-set of M.

Furthermore, most of the function in this family are parameterized by two objects: the *degree* and the *coefficient field* of the matrix group. These can be supplied in one of the following three forms:

(i) Integers n and q corresponding to the degree and the field $\mathbf{F}_q$ of M ($\mathbf{F}_{q^2}$ in the case of the unitary groups).

(ii) An integer n and a finite field K corresponding to the degree and the coefficient field of M.

(iii) A vector space $V = K^n$ on which M naturally acts.

The Suzuki group, however, is only parametrised by the field, as the degree is always four. As such, it can be described by the integer q, the field $K = \mathbf{F}_q$, or the vector space K^4.

`AffineGeneralLinearGroup(`*arguments*`)`

`AGL(`*arguments*`)`

Construct the affine general linear group $G = \mathrm{AGL}(n, q)$, i.e., the group corresponding to the action of $\mathrm{GL}(n, q)$ on the affine points of the n-dimensional vector space V over $K = \mathbf{F}_q$. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the affine points and the G-set of G.

`AffineSpecialLinearGroup(`*arguments*`)`

`ASL(`*arguments*`)`

Construct the affine special linear group $G = \mathrm{ASL}(n, q)$, i.e., the group corresponding to the action of $\mathrm{SL}(n, q)$ on the affine points of the n-dimensional vector space V over $K = \mathbf{F}_q$. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the affine points and the G-set of G.

`AffineGammaLinearGroup(`*arguments*`)`

`AGammaL(`*arguments*`)`

Construct the affine gamma linear group $G = \mathrm{A\Gamma L}(n, q)$, i.e., the group corresponding to the action of $\Gamma\mathrm{L}(n, q)$ (the automorphism group of $\mathrm{GL}(n, q)$) on the affine points of the n-dimensional vector space V over $K = \mathbf{F}_q$. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`AffineSigmaLinearGroup(`*arguments*`)`

`ASigmaL(`*arguments*`)`

Construct the affine sigma linear group $G = \mathrm{A\Sigma L}(n, q)$, i.e., the group corresponding to the action of $\Sigma\mathrm{L}(n, q)$ (the automorphism group of $\mathrm{SL}(n, q)$) on the affine points of the n-dimensional vector space V over $K = \mathbf{F}_q$. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveGeneralLinearGroup(`*arguments*`)`

`PGL(`*arguments*`)`

Construct the projective general linear group $G = \mathrm{PGL}(n, q)$, i.e., the group corresponding to the action of $\mathrm{GL}(n, q)$ on the projective points of the n-dimensional vector space V over $K = \mathbf{F}_q$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialLinearGroup(`*arguments*`)`

`PSL(`*arguments*`)`

Construct the projective special linear group $G = \mathrm{PSL}(n, q)$, i.e., the group corresponding to the action of $\mathrm{SL}(n, q)$ on the projective points of the n-dimensional vector space V over $K = \mathbf{F}_q$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveGammaLinearGroup(`*arguments*`)`

`PGammaL(`*arguments*`)`

Construct an automorphism group $G = \mathrm{P\Gamma L}(n, q)$ of the projective general linear group $B = \mathrm{PGL}(n, q)$, by adding the field automorphisms of $\mathbf{F}_q$ to B. The permutation action corresponds to the natural action on 1-dimensional subspaces of the n-dimensional vector space V over the field $K = \mathbf{F}_q$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveSigmaLinearGroup(`*arguments*`)`

`PSigmaL(`*arguments*`)`

Construct an automorphism group $G = \mathrm{P\Sigma L}(n, q)$ of the projective special linear group $B = \mathrm{PSL}(n, q)$, by adding the field automorphisms of $\mathbf{F}_q$ to B. The permutation action corresponds to the natural action on 1-dimensional subspaces of the n-dimensional vector space V over the field $K = \mathbf{F}_q$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveGeneralUnitaryGroup(`*arguments*`)`

`PGU(`*arguments*`)`

Construct the projective general unitary group $G = \mathrm{PGU}(n, q)$ corresponding to the n-dimensional vector space V over the field $K = \mathbf{F}_{q^2}$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialUnitaryGroup(`*arguments*`)`

`PSU(`*arguments*`)`

Construct the projective special unitary group $G = \mathrm{PSU}(n, q)$ corresponding to the n-dimensional vector space V over the field $K = \mathbf{F}_{q^2}$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of V, giving the correspondence between these vectors and the G-set of G.

`ProjectiveGammaUnitaryGroup(`*arguments*`)`

`PGammaU(`*arguments*`)`

Construct an automorphism group $G = \mathrm{P\Gamma U}(n, q)$ of the projective general unitary group $B = \mathrm{PGU}(n, q)$, by adding the field automorphisms of $\mathbf{F}_{q^2}$ to B. The permutation action corresponds to the natural action on 1-dimensional subspaces of the n-dimensional vector space V over the field $K = \mathbf{F}_{q^2}$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveSigmaUnitaryGroup(`*arguments*`)`

`PSigmaU(`*arguments*`)`

Construct the automorphism group $G = \mathrm{P\Sigma U}(n, q)$ of the projective special unitary group $B = \mathrm{PSU}(n, q)$, by adding the field automorphisms of $\mathbf{F}_{q^2}$ to B. The permutation action corresponds to the natural action on 1-dimensional subspaces of the n-dimensional vector space V over the field $K = \mathbf{F}_{q^2}$, where $n \geq 2$ and q is a prime power. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveSymplecticGroup(`*arguments*`)`

`PSp(`*arguments*`)`

Construct the projective symplectic group $G = \mathrm{PSp}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 4. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSigmaSymplecticGroup(`*arguments*`)`

`PSigmaSp(`*arguments*`)`

Construct the group $G = \mathrm{P\Sigma Sp}(n, q)$ of the projective symplectic group $\mathrm{PSp}(n, q)$ extended by field automorphisms of $K = \mathbf{F}_q$, where V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 4. The function returns:

(a) The group G;

(b) An indexed set giving the correspondence between the points and the G-set of G.

`ProjectiveGeneralOrthogonalGroup(`*arguments*`)`

`PGO(`*arguments*`)`

Construct the projective general orthogonal group $G = \mathrm{PGO}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an odd integer greater than or equal to 3. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveGeneralOrthogonalGroupPlus(`*arguments*`)`

`PGOPlus(`*arguments*`)`

Construct the projective general orthogonal group $G = \mathrm{PGO}^+(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveGeneralOrthogonalGroupMinus(`*arguments*`)`

`PGOMinus(`*arguments*`)`

Construct the projective general orthogonal group $G = \mathrm{PGO}^-(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialOrthogonalGroup(`*arguments*`)`

`PSO(`*arguments*`)`

Construct the projective special orthogonal group $G = \mathrm{PSO}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an odd integer greater than or equal to 3. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialOrthogonalGroupPlus(`*arguments*`)`

`PSOPlus(`*arguments*`)`

Construct the projective special orthogonal group $G = \mathrm{PSO}^+(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSpecialOrthogonalGroupMinus(`*arguments*`)`

`PSOMinus(`*arguments*`)`

Construct the projective general orthogonal group $G = \mathrm{PSO}^-(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveOmega(`*arguments*`)`

`POmega(`*arguments*`)`

Construct the projective orthogonal group $G = \mathrm{P\Omega}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an odd integer greater than or equal to 3. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveOmegaPlus(`*arguments*`)`

`POmegaPlus(`*arguments*`)`

Construct the projective orthogonal group $G = \mathrm{P\Omega}(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveOmegaMinus(`*arguments*`)`

`POmegaMinus(`*arguments*`)`

Construct the projective orthogonal group $G = \mathrm{P}\Omega(n, q)$, where $K = \mathbf{F}_q$, V is an n-dimensional vector space over K, and n is an even integer greater than or equal to 2. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`ProjectiveSuzukiGroup(`*arguments*`)`

`PSz(`*arguments*`)`

Construct the permutation representation $G = \mathrm{PSz}(q)$ of the Suzuki simple group $\mathrm{Sz}(q)$, given by its action on projective points, where q is of the form 2^{2n+1}. If K is given, its cardinality is q. If V is given, it must be 4-dimensional, and over K. The function returns:

(a) The group G;

(b) An indexed set of the generators of the 1-dimensional subspaces of $K^{(n)}$, giving the correspondence between these vectors and the G-set of G.

`AffineGroup(M)`

Given a matrix group of degree d over a finite field F, construct the semidirect product $V : M$, where $V = F^d$ is the natural M-module. The result, G, is a standard permutation group of degree $|V| = |F|^d$, where the second return value gives the correspondence between the elements of V and the standard G-set.

58.20 Permutation Group Databases

Magma includes databases that contain all transitive permutation groups of degree up to 32 and all primitive permutation groups of degree up to 4095. Descriptions of these databases may be found in Chapter 66.

58.21 Ordered Partition Stacks

Ordered partition stacks have been implemented with their own type and Magma intrinsics. They implement the data structure described very briefly in section 2 of Jeff Leon's 1997 paper [Leo97]. They can be used as an aid to implementing various backtrack searches in the Magma language.

The domain set of the partitions is always $\{1..n\}$, where n is called the degree of the stack. The basic "push" operation for these stacks involves refining an ordered partition, and the precise definition of refinement used in the Magma implementation is Definition 2 of [Leo97]. This differs from the definition in Chapter 9 of [Ser03], for instance.

The word "ordered" refers to the cells of the partition being in a fixed order. The order of points in a cell is not significant, and may vary as the data structure is manipulated.

58.21.1 Construction of Ordered Partition Stacks

`OrderedPartitionStack(n)`

Create a data structure representing a complete ordered partition stack of degree n. Initially the stack has one partition on it, which is the partition having a single block.

`OrderedPartitionStackZero(n, h)`

Create a data structure representing a zero-based ordered partition stack of degree n with height limited to h. Initially the stack has one partition on it, which is the partition having a single block, and has height 0.

58.21.2 Properties of Ordered Partition Stacks

`Degree(P)`

The degree of the ordered partition stack P.

`Height(P)`

The height of the ordered partition stack P. For a complete stack, this equals the number of cells of the finest partition on the stack.

`NumberOfCells(P, h)`

The number of cells in the partition on stack P at height h. If h is omitted it is taken to be the height of P, so giving the number of cells in the finest partition on the stack P.

`CellNumber(P, h, x)`

The number of the cell of the partition at height h in P that contains the element x. If h is omitted it is taken to be the height of P.

`CellSize(P, h, i)`

The size of cell i of the partition at height h in P. If h is omitted it is taken to be the height of P.

`Cell(P, h, i)`

The contents of cell i of the partition at height h in P as a sequence of integers. If h is omitted it is taken to be the height of P. Note that the order of the points in the returned sequence may vary, as the order of points in a cell of an ordered partition is not fixed.

`Random(P, i)`

A random element of cell i of the finest partition on P.

`Representative(P, i)`

`Rep(P, i)`

An element of cell i of the finest partition on P.

`ParentCell(P, i)`

The number of the cell that was split to first create cell number i.

58.21.3 Operations on Ordered Partition Stacks

Here are listed the basic operations provided for pushing a finer partition onto an ordered partition stack, called splitting, and for popping ordered partitions off the stack.

If the top partition on the stack has k cells, and one of these cells is split to form a finer partition, then the new cell will have number $k + 1$, and the residue of the split cell will have the same number as the cell that was split. This agrees with the definition of refinement given in [Leo97], Definition 2, but disagrees with [Ser03], Chapter 9.2, and [McK81].

`SplitCell(P, i, x)`

`SplitCell(P, i, Q)`

Attempt to refine the top partition on stack P by splitting cell i. The new cell created will be $\{x\}$ if x is in cell i, or will be the intersection of Q with cell i (in the second form), if this is not empty and not all of cell i. The new partition, if any, is pushed onto the stack. The return value is true when P is changed, false otherwise. This implements the operation in Definition 6 of [Leo97].

`SplitAllByValues(P, V)`

Refine the top partition on stack P by splitting all possible cells using the values in V. This implements the operation given in Definition 15 of [Leo97]. Cells are split in increasing order of cell number, and the resulting new cells are in the curious order given in the cited definition.

The first return value is true when P is changed, false otherwise. The second is a hash value based on which cells are split, the values from V used in the split, and the sizes of the resulting cells. It is suitable for use as an indicator function, as defined in 2-16 of [McK81].

`SplitCellsByValues(P, C, V)`

`SplitCellsByValues(P, i, V)`

Refine the top partition on stack P by splitting all cells given in C, or cell i, using the values in V. Splitting and return values are as for `SplitAllByValues`, with an important difference: cells will be split in the order given in C, and, if some cell in C does not split, the operation will be terminated there, and false returned.

`Pop(P)`

`Pop(P, h)`

Reduce the height of the partition stack P to height h, or by one if h is not given. The method used is the "retract" algorithm of [Leo97], Fig. 7.

`Advance(X, L, P, h)`

This implements the "advance" algorithm of [Leo97], Fig. 7: X is a zero-based stack of degree d say, L is a sequence of length n taking values in $\{1..d\}$, representing an unordered partition of $\{1..n\}$ into d blocks, P is a complete stack of degree n, and h is a positive integer which is at most the height of P. This is a fundamental operation in Leon's unordered partition stabilizer algorithm.

Example H58E44

We set up an ordered partition stack of degree 12, and try out a few basic operations on it. The printing of a stack shows the top partition of the stack.

```
> P := OrderedPartitionStack(12);
> P;
Partn stack, degree 12, height 1
[ 1 2 3 4 5 6 7 8 9 10 11 12]
> SplitCell(P, 1, 4);
true
> P;
Partn stack, degree 12, height 2
```

```
[ 1 2 3 12 5 6 7 8 9 10 11 | 4]
```

Note that the order of the points in cell 1 is not significant. Now we will split on the values in a vector V.

```
> V := [i mod 5 + 1: i in [0..11]];
> V;
[ 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2 ]
> SplitAllByValues(P, V);
true 119375796
> P;
Partn stack, degree 12, height 6
[ 1 6 11 | 4 | 10 5 | 9 | 8 3 | 12 2 7]
```

Only cell 1 has been split. The points corresponding to the minimum value in V remain in cell 1. The new cells are cells 2 to 6. They correspond to the higher values in V, in descending order. Now pop the stack back to height 4 and try the effect of a different split by values.

```
> Pop(P, 4);
> P;
Partn stack, degree 12, height 4
[ 1 6 11 12 2 7 8 3 | 4 | 10 5 | 9]
> V := [i mod 4 + 1: i in [0..11]];
> V;
[ 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4 ]
> SplitAllByValues(P, V);
true 985543242
> P;
Partn stack, degree 12, height 8
[ 1 | 4 | 5 | 9 | 8 12 | 11 7 3 | 6 2 | 10]
> Pop(P, 3);
> P;
Partn stack, degree 12, height 3
[ 1 6 2 11 7 3 8 12 9 | 4 | 5 10]
```

58.22 Bibliography

[**Atk75**] M.D. Atkinson. An algorithm for finding the blocks of a permutation group. *Math. Comp.*, 29:911–913, 1975.

[**BC82**] G. Butler and J.J. Cannon. Computing with permutation and matrix groups I: Normal closure, commutator subgroups, series. *Math. Comp.*, 39:671–680, 1982.

[**Bea93**] G. Beals. *Algorithms for finite groups.* PhD thesis, University of Chicago, 1993.

[**BLGN$^+$03**] R. Beals, C. R. Leedham-Green, A. C. Niemeyer, C. E. Praeger, and A. Seress. A black-box algorithm for recognising finite symmetric and alternating groups, I. *Trans. Amer. Math. Soc.*, 2003. To appear.

[**BP00**] Sergey Bratus and Igor Pak. Fast constructive recognition of a black box group isomorphic to S_n or A_n using Goldbach's conjecture. *J. Symbolic Comp.*, 29:33–57, 2000.

[**But85**] Gregory Butler. Effective computation with group homomorphisms. *J. Symbolic Comp.*, 1:143–157, 1985.

[**But94**] Greg Butler. An inductive schema for computing conjugacy classes in permutation groups. *Mathematics of Computation*, 62(205):363–383, 1994.

[**CB92**] J.J. Cannon and W. Bosma. Structural computation in finite permutation groups. *CWI Quarterly*, 5(2):127–160, 1992.

[**CC91**] P.J. Cameron and J.J. Cannon. Recognizing doubly transitive groups. *J. Symb. Comp.*, 12(4/5):459–474, 1991.

[**CCH97**] J.J. Cannon, B. Cox, and D.F. Holt. Computing Sylow subgroups in permutation groups. *J. Symb. Comp.*, 24(3/4):303–316, 1997.

[**CCH01**] J.J. Cannon, B. Cox, and D.F. Holt. Computing the subgroups of a permutation group. *J. Symb. Comp.*, 31:149–161, 2001.

[**CFL89**] G. Cooperman, L. Finkelstein, and E.M. Luks. Reduction of group constructions to point stabilizers. In *Proc. of International Symposium on Symbolic and Algebraic Computation ISSAC '89*, pages 351–356. ACM, 1989.

[**CH97**] J.J. Cannon and D.F. Holt. Computing chief series, composition series and socles in large permutation groups. *J. Symb. Comp.*, 24(3/4):285–301, 1997.

[**CH03**] J.J. Cannon and D.F. Holt. Automorphism group computation and isomorphism testing in finite groups. *J. Symbolic Comp.*, 35(3):241–267, 2003.

[**CH04**] J.J. Cannon and D.F. Holt. Computing maximal subgroups of finite groups. *J. Symbolic Comp.*, 37(5):589–609, 2004.

[**CHSS03**] J.J. Cannon, D.F. Holt, M. Slattery, and A.K. Steel. Computing subgroups of low index in a finite group. 2003.

[**CLGM$^+$95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

[**CS**] J.J. Cannon and B. Souvignier. On the computation of normal subgroups in permutation groups. to appear, International Journal of Algebra and Computation.

[**CS97**] J.J. Cannon and B. Souvignier. On the computation of conjugacy classes in permutation groups. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 392–399. Association for Computing Machinery, 1997. Maui, July 21–23, 1997.

[**Geb00**] Volker Gebhardt. Constructing a short defining set of relations for a finite group. *J. Algebra*, 233:526–542, 2000.

[**Hol84**] D.F. Holt. The calculation of the Schur multiplier of a permutation group. In *Computational group theory (Durham, 1982)*, pages 307–319. Academic Press, London, 1984.

[**Hol85a**] D.F. Holt. A computer program for the calculation of a covering group of a finite group. *J. Pure Appl. Algebra*, 35(3):287–295, 1985.

[**Hol85b**] D.F. Holt. The mechanical computation of first and second cohomology groups. *J. Symbolic Comp.*, 1(4):351–361, 1985.

[**Kan91**] William M. Kantor. Finding composition factors of permutation groups of degree $n \leq 10^6$. *J. Symbolic Comp.*, 12(4/5):517–526, 1991.

[**Leo80**] Jeffrey S. Leon. On an algorithm for finding a base and a strong generating set for a group given by generating permutations. *Math. Comp.*, 35(151):941–974, 1980.

[**Leo97**] Jeffrey S. Leon. Partitions, refinements, and permutation group computation. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *Dimacs series in Discrete Mathematics and Computer Science*, pages 123–158, Providence R.I., 1997. Amer. Math. Soc.

[**LGPS91**] C.R. Leedham-Green, C.E. Praeger, and L.H. Soicher. Computing with group homomorphisms. *J. Symbolic Comp.*, 12(4/5):527–532, 1991.

[**LNPS06**] M. Law, A.C. Niemeyer, C.E. Praeger, and A. Seress. A reduction algorithm for for large-base primitive permutation groups. *LMS J. Comput. Math.*, 9:159173, 2006.

[**Luk93**] E.M. Luks. Permutation groups and polynomial-time computation. In *Groups and computation (New Brunswick, NJ, 1991)*, volume 11 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 139–175. Amer. Math. Soc., 1993.

[**McK81**] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[**MN89**] M. Mecky and J. Neubüser. Some remarks on the computation of conjugacy classes of soluble groups. *Bull. Austral, Math. Soc.*, 40(2):281–292, 1989.

[**Neu86**] P.M. Neumann. Some algorithms for computing with finite permutation groups. In C.M. Campbell E.F. Robertson, editor, *Groups - St. Andrews 1985*, number 121 in London Math. Soc. Lecture Notes Series, 1986.

[**Ric73**] J.S. Richardson. Group: a computer system for group-theoretic calculations. Master's thesis, Department of Pure Mathematics, University of Sydney, September 1973.

[**Sch90**] Bernd Schmalz. Verwendung von Untergruppenleitern zur Bestimmung von Doppelnebenklassen. *Bayreuther Mathematische Schriften*, 31:109–143, 1990.

[**Ser03**] Ákos Seress. *Permutation Group Algorithms*, volume 152 of *Cambridge Tracts in Mathematics*. Cambridge University Press, Cambridge, 2003.

[**Sim90**] Charles C. Sims. Computing the order of a solvable permutation group. *J. Symb. Comp.*, 9(5/6):699–705, 1990.

[**SS94**] M. Schönert and A. Seress. Finding blocks of imprimitivity in small-base groups in nearly linear time. In *Proc. 1994 ACM-SIGSAM Inter. Symp. on Symbolic and Algebraic Comp.*, pages 154–157, 1994.

[**Ung**] W.R. Unger. Computing chief series of a large permutation group. In preparation.

[**Ung06a**] W.R. Unger. Computing the character table of a finite group. *J. Symbolic Comp.*, 41(8):847–862, 2006.

[**Ung06b**] W.R. Unger. Computing the solvable radical of a permutation group. *J. Algebra*, 300(1):305–315, 2006.

59 MATRIX GROUPS OVER GENERAL RINGS

Chapter 59
MATRIX GROUPS OVER GENERAL RINGS

59.1 Introduction

59.1.1 Introduction to Matrix Groups

A matrix group G may be defined over any ring R for which MAGMA has a method for computing the inverse of a matrix. However, the availability of machinery for determining structural information is dependent upon the properties of the base ring R.

We distinguish several different cases.

(i) If the ring R is a finite field then the group must be finite. If the group has moderate degree and it is possible to find a low dimensional subspace of the natural vector space for G whose orbit under G has length bounded by a million or so, then it is possible to construct a stabilizer chain representation for the group similar to that used for permutation groups (the *BSGS representation* [But76]). In order to increase the chances of finding a short orbit the Murray-O'Brien [MO95] strategy for selecting base points is used. The availability of a BSGS representation allows the structure of the group to be investigated in detail.

(ii) If the coefficient ring R is a finite field but the degree and size of the group are such that it is not possible to construct a useful BSGS representation then the group may be investigated using techniques based on a theorem of Aschbacher that classifies the maximal subgroups of $\mathrm{GL}(n, q)$. This approach is under intensive development by Leedham-Green, O'Brien and others. Code implementing some parts is documented in Chapter 65.

(iii) If the ring R is the Euclidean Ring $\mathbf{Z}/m\mathbf{Z}$, then the group must be finite. If the group has moderate degree and it is possible to find a vector in the natural R-module for G whose orbit has length bounded by a million or so, then again it is possible to compute structural information using the BSGS representation.

(iv) If the ring R has infinite cardinality and satisfies certain properties, MAGMA can sometimes determine whether the group is finite or infinite. In particular, this can be done when R is the ring of integers, the field of rational numbers, or an algebraic number field (including cyclotomic and quadratic fields). If G is infinite, and has not been created as a Lie group, then MAGMA currently provides little beyond basic arithmetic on elements.

(v) If the ring R has infinite cardinality but the group G is finite and R is either a field or an Euclidean Domain then it may be possible to construct a *BSGS representation* as above and thereby undertake structural computation.

(vi) If an (infinite) matrix group can be created as a Lie group then machinery based on Lie Theory may be used to analyse the group. The facilities for Lie groups are described in Chapter 103.

Matrix groups over rings of infinite cardinality may be created regardless as to whether they are finite or not. If the coefficient ring R is either the ring of integers, the rational field, a quadratic field, a cyclotomic field, or a number field a matrix group may then be tested for finiteness by use of the function `IsFinite`. However, most functions that determine structural properties of a group apply only to finite groups.

59.1.2 The Support

Matrix groups may be defined over any ring for which MAGMA has a method for computing matrix inverses. However, the structure algorithms assume that the group is finite and is defined over either a field, an Euclidean Domain or the Euclidean Ring $\mathbf{Z}/m\mathbf{Z}$.

59.1.3 The Category of Matrix Groups

The family of matrix groups over a particular ring R forms a category where the objects are the matrix groups and the morphisms are group homomorphisms. The collection of all matrix groups forms a family of categories indexed by the category of rings. The MAGMA designation for this family of categories of matrix groups is `GrpMat`.

59.1.4 The Construction of a Matrix Group

A group of $n \times n$ matrices defined over the ring R is created as a subgroup of the general linear group $\mathrm{GL}(n, R)$. Thus the construction of a general matrix group is a two step process:

(i) The appropriate general linear group, $\mathrm{GL}(n, R)$, is constructed;

(ii) The required group G is then defined as a subgroup of $\mathrm{GL}(n, R)$.

For convenience, a constructor `MatrixGroup< ... >`, which combines these two steps, is provided.

59.2 Creation of a Matrix Group

59.2.1 Construction of the General Linear Group

`GeneralLinearGroup(n, R)`

`GL(n, R)`

Given an integer $n \geq 1$ and a ring R, create the generic matrix group, i.e. the general linear group $\mathrm{GL}(n, R)$. Initially, only a structure table is created for $\mathrm{GL}(n, R)$, so that, in particular, generators are not defined. This function is normally used to provide a context for the creation of elements and subgroups of $\mathrm{GL}(n, R)$. If structural computation is attempted with the group created by `GeneralLinearGroup(n, R)`, then generators will be created where possible. At present, this is only permitted in the cases in which R is a finite field.

Example H59E1

We define the general linear group $\mathrm{GL}(3, K)$, where K is the finite field $\mathbf{F}_4$.

```
> K<w> := FiniteField(4);
> GL34 := GeneralLinearGroup(3, K);
> GL34;
GL(3, GF(2, 2))
```

59.2.2 Construction of a Matrix Group Element

Throughout this subsection we shall assume that the matrix group G is defined over the ring R.

`elt< G | L >`

Given a matrix group G defined as a subgroup of $\mathrm{GL}(n, R)$, and the list L of expressions a_{ij} $(1 \leq i, j \leq n)$, defining elements of the ring R, construct the $n \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

Unless G is known to be the generic matrix group of degree n, the matrix will be tested for membership of G, and if g is not an element of G, the function will fail. If g does lie in G, g will have G as its parent. Since the membership test may involve constructing a base and strong generating set for G, this constructor may occasionally be very costly. Hence a matrix g should be defined as an element of a subgroup of the generic group only when membership of G is required by subsequent operations involving g.

`G ! Q`

Given the sequence Q of expressions a_{ij} $(1 \leq i, j \leq n)$, defining elements of the ring R, construct the $n \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

This matrix will have G as its parent structure. As in the case of the `elt`-constructor, the operation will fail if g is not an element of G, and the same observations concerning the cost of membership testing apply.

`ElementToSequence(g)`

`Eltseq(g)`

Given an $n \times n$ matrix $g = (a_{ij}), 1 \leq i, j \leq n$, where a_{ij} is an element of the ring R, construct the sequence

$$[a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots a_{n1}, \ldots, a_{nn}]$$

of n^2 elements of the ring R.

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity matrix in the matrix group G.

Example H59E2

The different constructions are illustrated by the following code, which assigns to each of the variables x and y an element of $\mathrm{GL}(3, 4)$.

```
> K<w> := FiniteField(4);
> GL34 := GeneralLinearGroup(3, K);
> x := elt<GL34 | w,0,1, 0,1,0, 1,0,1 >;
> x;
[w 0 1]
[0 1 0]
[1 0 1]
> y := GL34 ! [w,0,1, 0,1,0, 1,0,1];
> y;
[w 0 1]
[0 1 0]
[1 0 1]
> GL34 ! 1;
[1 0 0]
[0 1 0]
[0 0 1]
```

59.2.3 Construction of a General Matrix Group

`MatrixGroup< n, R | L >`

Construct the matrix group G of degree n over the ring R generated by the matrices defined by the list L. A term of the list L must be an object of one of the following types:

(a) A sequence of n^2 elements of R defining a matrix of $\mathrm{GL}(n, R)$;

(b) A set or sequence of sequences of type (a);

(c) An element of $\mathrm{GL}(n, R)$;

(d) A set or sequence of elements of $\mathrm{GL}(n, R)$;

(e) A subgroup of $\mathrm{GL}(n, R)$;

(f) A set or sequence of subgroups of $\mathrm{GL}(n, R)$.

Each element or group specified by the list must belong to the *same* generic matrix group. The group G will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of G consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed.

The `MatrixGroup` constructor is shorthand for the two statements:

```
GL := GeneralLinearGroup(n, R);
G := sub< GL | L >;
```

where `sub< ... >` is the subgroup constructor described in the next subsection.

Example H59E3

We use the `MatrixGroup` constructor to define a small subgroup of $\mathrm{GL}(3, 4)$.

```
> K<w> := FiniteField(4);
> H := MatrixGroup< 3, K | [1,w,0, 0,1,0, 1,w^2,1], [w,0,0, 0,1,0, 0,0,w] >;
> H;
MatrixGroup(3, GF(2, 2))
Generators:
[  1   w   0]
[  0   1   0]
[  1 w^2   1]

[w 0 0]
[0 1 0]
[0 0 w]
> Order(H);
96
```

Example H59E4

We present a function which will construct the Sylow p-subgroup of $\mathrm{GL}(n, K)$, where K is a finite field of characteristic p.

```
> GLSyl := function(n, K)
>    R := MatrixRing(K, n);
>    e := func< i, j | MatrixUnit(R, i, j) >;
>    return MatrixGroup< n, K | { R!1 + a*e(i,j) : a in K, j in [i+1],
>          i in [1 .. n - 1] | a ne 0 } >;
> end function;
> T := GLSyl(3, GF(8));
> FactoredOrder(T);
[ <2, 9> ]
> FactoredOrder(GL(3, GF(8)));
[ <2, 9>, <3, 2>, <7, 3>, <73, 1> ]
```

59.2.4 Changing Rings

`ChangeRing(G, S)`

Given a matrix group G with base ring R, construct a new matrix group H with base ring S derived from G by coercing entries of the generators of G from R into S.

`ChangeRing(G, S, f)`

Given a matrix group G with base ring R, construct a new matrix group H with base ring S derived from G by applying f to the entries of the generators of G.

`RestrictField(G, S)`

Given a matrix group G with base ring K, a finite field, and S a subfield of K, construct the matrix group H with base ring S obtained by restricting the scalars of the components of elements of G into S, together with the restriction map from G onto H.

`ExtendField(G, L)`

Given a matrix group G with base ring K, a finite field, and L an extension of K, construct the matrix group H with base ring L obtained by lifting the components of elements of G into L, together with the inclusion homomorphism from G into H.

59.2.5 Coercion between Matrix Structures

A square non-singular matrix may be defined as an element of any of the following structures:

- A subring of the complete matrix ring $M_n(R)$;
- A subgroup of the general linear group $\mathrm{GL}(n, R)$;
- A submodule of the matrix module $M^{(m \times n)}(R)$.

The coercion operator may be used to transfer matrices between any two of these three structures.

`R ! g`

Transfer the matrix g from a group into a matrix ring R.

`G ! r`

Transfer the matrix r from a ring into a matrix group G.

`M ! g`

Transfer the matrix g from a group into a matrix module M.

`G ! m`

Transfer the matrix m from a module into a matrix group G.

59.2.6 Accessing Associated Structures

The functions in this group provide access to basic information stored for a matrix group G.

`G . i`

The i-th defining generator for the matrix group G. A negative subscript indicates that the inverse of the generator is to be created. $G.0$ is `Identity(G)`.

`Degree(G)`

The degree of the matrix group G.

`Generators(G)`

A set containing the defining generators for the matrix group G.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for the matrix group G.

`CoefficientRing(G)`

`BaseRing(G)`

The coefficient ring for the matrix group G.

`RSpace(G)`

Given a matrix group G of degree n defined over a ring R, return the space $R^{(n)}$, where the action is multiplication by elements of R, i.e. scalar action.

`VectorSpace(G)`

Given a matrix group G of degree n defined over a field K, return the space $K^{(n)}$, where the action is multiplication by elements of K, i.e. scalar action.

`GModule(G)`

The natural $R[G]$-module for the matrix group G.

`Generic(G)`

The generic group containing the matrix group G, i.e. the general linear group in which G is naturally embedded.

`Parent(G)`

The power structure for the group G (the set consisting of all matrix groups).

59.3 Homomorphisms

Homomorphisms are an important part of group theory, and MAGMA supports homomorphisms between groups. Many useful homomorphisms are returned by constructors and intrinsic functions. Examples of these are the `quo` constructor, the `sub` constructor and intrinsic functions such as `OrbitAction` and `FPGroup`, which are described in more detail elsewhere in this chapter. In this section we describe how the user may create their own homomorphisms with domain a matrix group.

`hom< G - >`

Given the matrix group G, construct the homomorphism $f : G \to H$ given by the generator images in L. H must be a group. The clause L may be any one of the following types:

(a) A list of elements of H, giving images of the generators of G;

(b) A list of pairs, where the first in the pair is an element of G and the second its image in H, where pairs may be given in either of the (equivalent) forms `<g,h>` or `g -> h`;

(c) A sequence of elements of H, as in (a);

(d) A set or sequence of pairs, as in (b);

Each image element specified by the list must belong to the *same* group H. In the cases where pairs are given the given elements of G must generate G.

`Domain(f)`

The domain of the homomorphism f.

`Codomain(f)`

The codomain of the homomorphism f.

`Image(f)`

The image or range of the homomorphism f. This will be a subgroup of the codomain of f. The algorithm computes the image and kernel simultaneously (see [LGPS91]).

`Kernel(f)`

The kernel of the homomorphism f. This will be a normal subgroup of the domain of f. The algorithm computes the image and kernel simultaneously (see [LGPS91]).

`IsHomomorphism(G, H, Q)`

Return the value `true` if the sequence Q defines a homomorphism from the group G to the group H. The sequence Q must have length `Ngens(G)` and must contain elements of H. The i-th element of Q is interpreted as the image of the i-th generator of G and the function decides if these images extend to a homomorphism. If so, the homomorphism is also returned.

Example H59E5

We construct the usual degree 2 matrix representation of the dihedral group of order 20, and a homomorphism from it to the symmetric group of degree 5.

```
> K<z> := CyclotomicField(20);
> zz := RootOfUnity(10, K);
> i := RootOfUnity(4, K);
> cos := (zz+ComplexConjugate(zz))/2;
> sin := (zz-ComplexConjugate(zz))/(2*i);
> gl := GeneralLinearGroup(2, K);
> M := sub< gl | [cos, sin, -sin, cos], [-1,0,0,1]>;
> #M;
20
> S := SymmetricGroup(5);
> f := hom<M->S |[S|(1,2,3,4,5), (1,5)(2,4)]>;
> Codomain(f);
Symmetric group S acting on a set of cardinality 5
Order = 120 = 2^3 * 3 * 5
> Image(f);
Permutation group acting on a set of cardinality 5
Order = 10 = 2 * 5
  (1, 2, 3, 4, 5)
  (1, 5)(2, 4)
> Kernel(f);
MatrixGroup(2, K) of order 2
Generators:
  [-1  0]
```

```
[ 0 -1]
```

59.3.1 Construction of Extensions

`DirectProduct(G, H)`

Given two matrix groups G and H of degrees m and n respectively, construct the direct product of G and H as a matrix group of degree $m + n$.

`DirectProduct(Q)`

Given a sequence Q of n matrix groups, construct the direct product $Q[1] \times Q[2] \times \ldots \times Q[n]$ as a matrix group of degree equal to the sum of the degrees of the groups $Q[i]$, $(i = 1, \ldots, n)$.

`SemiLinearGroup(G, S)`

Given a matrix group G over the finite field K and a subfield S of K, construct the semilinear extension of G over the subfield S.

`TensorWreathProduct(G, H)`

Given a matrix group G and a permutation group H, construct action of the wreath product on the tensor power of G by H, which is the (image of) the wreath product in its action on the tensor power (of the space that G acts on). The degree of the new group is d^k where d is the degree of G and k is the degree of H.

`WreathProduct(G, H)`

Given a matrix group G and a permutation group H, construct the wreath product $G \wr H$ of G and H.

Example H59E6

We define G to be $\mathrm{SU}(3, 4)$ and H to be the symmetric group of order 6. We then proceed to form the direct product of G with itself and the tensor and wreath products of G and H.

```
> K<w> := FiniteField(4);
> G := SpecialUnitaryGroup(3, K);
> D := DirectProduct(G, G);
> D;
MatrixGroup(6, GF(2, 2))
Generators:
[  1   w   w   0   0   0]
[  0   1 w^2   0   0   0]
[  0   0   1   0   0   0]
[  0   0   0   1   0   0]
[  0   0   0   0   1   0]
[  0   0   0   0   0   1]
```

```
[w 1 1 0 0 0]
[1 1 0 0 0 0]
[1 0 0 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]

[  1   0   0   0   0   0]
[  0   1   0   0   0   0]
[  0   0   1   0   0   0]
[  0   0   0   1   w   w]
[  0   0   0   0   1 w^2]
[  0   0   0   0   0   1]

[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 w 1 1]
[0 0 0 1 1 0]
[0 0 0 1 0 0]
> Order(D);
46656
> H := SymmetricGroup(3);
> E := WreathProduct(G, H);
> Degree(E);
9
> Order(E);
60466176
> F := TensorWreathProduct(G, H);
> Degree(F);
27
> Order(F);
6718464
```

59.4 Operations on Matrices

59.4.1 Arithmetic with Matrices

`g * h`

The product of matrix g and matrix h, where g and h belong to the same generic group U. If g and h both belong to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h belong to subgroups H and K of a subgroup G of U then the product is returned as an element of G. Otherwise, the product is returned as an element of U.

`g ^ n`

The n-th power of the matrix g, where n is a positive or negative integer.

`g / h`

The product of the matrix g by the inverse of the matrix h, i.e. the element $g * h^{-1}$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g/h are the same as for $g * h$.

`g ^ h`

The conjugate of the matrix g by the matrix h, i.e. the element $h^{-1} * g * h$. Here g and h must belong to the same generic group U. The rules for determining the parent group of g^h are the same as for $g * h$.

`(g, h)`

The commutator of the matrices g and h, i.e. the element $g^{-1} * h^{-1} * g * h$. Here g and h must belong to the same generic group U. The rules for determining the parent group of (g, h) are the same as those for $g * h$.

`(g1, ..., gr)`

Given r matrices $g_1, \ldots, g_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

Example H59E7

These operations will be illustrated using the group GL$(3, 4)$.

```
> K<w> := FiniteField(4);
> GL34 := GeneralLinearGroup(3, K);
> x := GL34 ! [1,w,0, 0,w,1, w^2,0,1];
> y := GL34 ! [1,0,0, 1,w,0, 1,1,w];
> x;
[  1   w   0]
[  0   w   1]
[w^2   0   1]
> y;
[1 0 0]
[1 w 0]
```

```
[1 1 w]
> x*y;
[w^2 w^2   0]
[w^2   w   w]
[  w   1   w]
> x^10;
[  w   w   1]
[  w   1   1]
[  w w^2   w]
> x^-1;
[w^2 w^2 w^2]
[  1   w   w]
[  w   w w^2]
> x^y;
[w^2 w^2   0]
[  0 w^2   1]
[w^2 w^2   w]
> x/y;
[  0   1   0]
[  0 w^2 w^2]
[  w   w w^2]
> (x, y);
[  0   w   w]
[  w w^2   1]
[w^2   w w^2]
> (x,y,y);
[w^2   w w^2]
[w^2   w   0]
[w^2   1   w]
```

Arithmetic with group elements is not limited to elements of finite groups. We illustrate with a group of degree 3 over a function field.

```
> P<a,b,c,m,x,y,z> := FunctionField(RationalField(), 7);
> S := MatrixGroup< 3, P | [1,a,b,0,1,c,0,0,1],
>                          [1,0,m,0,1,0,0,0,1],
>                          [1,x,y,0,1,z,0,0,1] >;
>
> t := S.1 * S.2;
> t;
[    1     a b + m]
[    0     1     c]
[    0     0     1]
> t^-1;
[          1          -a a*c - b - m]
[          0           1          -c]
[          0           0           1]
> Determinant(t);
1
```

```
> t^2;
[              1              2*a a*c + 2*b + 2*m]
[              0                1             2*c]
[              0                0               1]
```

59.4.2 Predicates for Matrices

`g eq h`

Given matrices g and h belonging to the same generic group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given matrices g and h belonging to the same generic group, return `true` if g and h are distinct elements, `false` otherwise.

`IsIdentity(g)`

`IsId(g)`

Returns `true` if the matrix g is the identity matrix.

`IsScalar(g)`

Returns `true` if the matrix g is a scalar matrix.

59.4.3 Matrix Invariants

All of the functions for computing invariants of a square matrix apply to the elements of a matrix group. Here only operations of interest in the context of group elements are described. The reader is referred to chapter 26 for a complete list of functions applicable to matrices.

`Degree(g)`

The degree of the matrix g, i.e. the number of rows/columns of g.

`HasFiniteOrder(g)`

Returns `true` iff the matrix g has finite order. The second return value is the order if it is finite. The function rigorously proves its result (i.e., the result is not probable). Let R be the ring over which g is defined, and let the degree of the group in which g lies be n. If R is finite, then the first return value is trivially `true`.

If R is the integer ring then the function works as follows. Suppose first that g has finite order o. By a theorem of Minkowski (see Theorem 1.4 [KP02]), for any odd prime p, the reduction mod p of g has order o. Let $f(x) \in R[x]$ be the minimal polynomial of g. The matrix subalgebra generated by g is isomorphic to the quotient ring $R[x]/\langle f(x)\rangle$, so the order o of g equals the order of x mod $f(x)$.

For arbitrary g, the algorithm computes the order, $\bar{o}$, of the reduction of g modulo a small odd prime. If $\bar{o}$ is a possible order of an integer matrix of g's dimensions (see Theorem 2.7 *op. cit.*) then this is repeated with a larger prime. If this gives a different order, or the first attempt gave an impossible order, then g has infinite order. We now compute $x^{\bar{o}}$ mod $f(x)$. If this is 1, then $\bar{o}$ is the order of g, otherwise g has infinite order.

If R is the rational field then a necessary condition for g to have finite order is that $f(x)$ has integer coefficients, thus the above algorithm applies in this case.

If R is an algebraic number field of degree d over $\mathbf{Q}$ (including cyclotomic and quadratic fields), then the standard companion matrix blowup is applied to g to obtain a $(nd) \times (nd)$ matrix over $\mathbf{Q}$, and the above algorithm is then applied to this matrix.

`Order(g)`

`Proof` BoolElt *Default :* `true`

Given an element g of finite order belonging to a matrix group, this function returns the order of g. If g has infinite order, a runtime error results. In the case of a matrix group over a finite field, the algorithm described in [CLG97a] is used. In all other cases, simple powering of g is used.

The parameter `Proof` is associated with the case when the coefficient ring for g is a finite field. In that case, if `Proof` is set to `false`, then difficult integer factorizations will not attempted. In this situation two values are returned of which the first is a multiple n of the order of g. and the second value indicates whether n is known to be the exact order of g.

`FactoredOrder(g)`

`Proof` BoolElt *Default :* `true`

Given an element g of finite order belonging to a matrix group, this function returns the order of g as a factored integer. If g has infinite order, a runtime error results. If g has infinite order, the function generates a runtime error. In the case of a matrix group over a finite field, the algorithm described in [CLG97a] is used. In all other cases, simple powering of g is used. In that case it is more efficient to use this

function rather than factorizing the integer returned by `Order(g)`. If g has infinite order, an error ensues.

If the parameter `Proof` is `false`, then difficult integer factorizations are not attempted and the first return value F may contain composite numbers (so that the factorization expands to a multiple of the order of g); in any case the second return value indicates whether F is known to be the exact factored order of g.

`ProjectiveOrder(g)`

`Proof` BoolElt *Default :* `true`

The projective order n of the matrix g, and a scalar s such that $g^n = sI$. The projective order of g is the smallest n such that g^n is a scalar matrix (not just the identity matrix), and it always divides the true order of A. The parameter `Proof` is as for `Order`.

`FactoredProjectiveOrder(A)`

`Proof` BoolElt *Default :* `true`

Given a square invertible matrix A over a finite field K, return the projective order n of A in factored form and a scalar $s \in K$ such that $A^n = sI$. The parameter `Proof` is as for `FactoredOrder`.

`CentralOrder(g : parameters)`

`CentralOrder(g)`

`Proof` BoolElt *Default :* `true`

Return the smallest n such that g^n is central in its parent group. If g is a matrix and the optional parameter `Proof` is `false`, then accept a multiple of this value; the second value returned is `true` if the answer is exact.

`Determinant(g)`

The determinant of the matrix g.

`Trace(g)`

The trace of the matrix g.

`CharacteristicPolynomial(g: parameters)`

`Al` MonStgElt *Default :* "*Modular*"

`Proof` BoolElt *Default :* `true`

Given a matrix g belonging to a subgroup of $\mathrm{GL}(n, R)$, where R is a field or Euclidean Domain, return the characteristic polynomial of g as an element of the univariate polynomial ring over R. For details on the parameters, see the function `CharacteristicPolynomial` in the chapter on matrices.

`MinimalPolynomial(g)`

Given a matrix g belonging to a subgroup of $\mathrm{GL}(n, R)$, where R is a field or Z, return the minimal polynomial of g as an element of the univariate polynomial ring over R.

Example H59E8

We illustrate the matrix operations by applying them to some elements of $\mathrm{GL}(3, 4)$.

```
> K<w> := FiniteField(4);
> GL34 := GeneralLinearGroup(3, K);
> x := GL34 ! [w,0,1, 0,1,0, 1,0,1];
> x;
[w 0 1]
[0 1 0]
[1 0 1]
> Degree(x);
3
> Determinant(x);
w^2
> Trace(x);
w
> Order(x);
15
> m<t> := MinimalPolynomial(x);
> m;
t^3 + w*t^2 + w^2
> Factorization(m);
[
    <t + 1, 1>,
    <t^2 + w^2*t + w^2, 1>
]
> c<t> := CharacteristicPolynomial(x);
> c;
t^3 + w*t^2 + w^2
```

59.5 Global Properties

Unless otherwise noted, the functions in this section assume that a BSGS-representation for the group can be constructed.

59.5.1 Group Order

Unless the order is already known, each of the functions in this family will create a base and strong generating set for the group if one does not already exist.

`IsFinite(G)`

Given a matrix group G, return whether G is finite together with the order of G if G is finite. The function rigorously proves its result (i.e., the result is not probable). Let R be the ring over which G is defined, and let the degree of G be n. If R is finite, then the first return value is trivially `true`.

If R is the integer ring or rational field, then the function works as follows. The function successively generates random elements of G and tests whether each element has infinite order via the function `HasFiniteOrder`; if so, then the non-finiteness of G is proven. Otherwise, at regular intervals, the function attempts to construct a positive definite form fixed by G (see the function `PositiveDefiniteForm` in the chapter on matrix groups over $\mathbf{Q}$ and $\mathbf{Z}$), using a finite number of steps; if one is successively constructed, then the finiteness of G is proven. The number of steps attempted for the positive definite form constructed is increased as the algorithm progresses; if G is finite, such a form must exist and will be found when enough steps are tried, while if G is infinite, an element of infinite order is found very quickly in practice.

If R is an algebraic number field of degree d over $\mathbf{Q}$ (including cyclotomic and quadratic fields), then the standard companion matrix blowup is applied to the generators of G to obtain an isomorphic matrix group of (nd) over $\mathbf{Q}$, and the above algorithm is then applied to this matrix group.

`Order(G)`

`#G`

The order of the group G as an integer. If the order is not currently known, a base and strong generating set will be constructed for G. If G has infinite order, an error ensues.

`FactoredOrder(G)`

The order of the group G returned as a factored integer. The format is the same as for `FactoredIndex`. If the order of G is not known, it will be computed. If G has infinite order, an error ensues.

Example H59E9

```
> G := MatrixGroup<2,Integers()|[1,1,0,1],[0,1,-1,0]>;
> IsFinite(G);
false
> G24, e := ChangeRing(G, Integers(24));
> Order(G24);
9216
```

```
> G.-1*G.2;
[ 1  1]
[-1  0]
> (G.-1*G.2) @ e;
[ 1  1]
[23  0]
> (G24.2^2) @@ e;
[23  0]
[ 0 23]
```

59.5.2 Membership and Equality

`g in G`

Given a matrix g and a matrix group G, return `true` if g is an element of G, `false` otherwise.

`g notin G`

Given a matrix g and a matrix group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G`

Given a matrix group G and a set S of matrices belonging to a group H, where G and H belong to the same generic group, return `true` if S is a subset of G, `false` otherwise.

`H subset G`

Given matrix groups G and H belonging to the same generic group, return `true` if H is a subgroup of G, `false` otherwise.

`S notsubset G`

Given a matrix group G and a set S of matrices belonging to a group H, where G and H belong to the same generic group, return `true` if S is not a subset of G, `false` otherwise.

`H notsubset G`

Given matrix groups G and H belonging to the same generic group, return `true` if H is not a subgroup of G, `false` otherwise.

`H eq G`

Given matrix groups G and H belonging to the same generic group, return `true` if G and H are the same group, `false` otherwise.

`H ne G`

Given matrix groups G and H belonging to the same generic group, return `true` if G and H are distinct groups, `false` otherwise.

59.5.3 Set Operations

The creation of a base and strong generating set for a matrix group G provides us with a very compact representation of the set of elements of G. A particular BSGS imposes an order on the elements of G (lexicographic ordering of base images). It thus makes sense to talk about the 'number' of a group element relative to a particular BSGS.

`NumberingMap(G)`

A bijective mapping from the group G onto the set of integers $\{1 \dots |G|\}$. The actual mapping depends upon the base and strong generating set chosen for G.

`RandomProcess(G)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 20

Create a process to generate randomly chosen elements from the finite group G. The process is based on the product-replacement algorithm of [CLGM$^+$95], modified by the use of an accumulator. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G)+1$. Initially, these are just the generators of G. As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses one of the elements in the slots and multiplies it into the accumulator. The element in that slot is replaced by the product of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting `Scramble := m` causes m such operations to be performed before the process is returned.

`Random(G: parameters)`

`Short`	BoolElt	*Default :* `false`

A randomly chosen element for the group G. If a BSGS is known for G, then the element chosen will be genuinely random. If no BSGS is known, then the *random* element is chosen by multiplying out a *random* word in the generators. Since it is not usually practical to choose words long enough to properly sample the elements of G, the element returned will usually be biased. The boolean-valued parameter `Short` is used in this situation to indicate that a short word will suffice. Thus, if `Random` is invoked with `Short` assigned the value `true` then the element is constructed using a short word.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the finite group G, construct a random element of G by forming a random product over the expanded generating set constructed when the process was created. For large degree groups, or groups for which a BSGS is not known, this function should be used in preference to `Random(G)`.

Example H59E10

We use the random function to sample the orders of elements in the group $\mathrm{GL}(20, 16)$.

```
> G := GeneralLinearGroup(20, GF(16));
> RP := RandomProcess(G);
> [ FactoredOrder(Random(RP)) : i in [1..20] ];
[
    [ <3, 1>, <5, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>,
    <61681, 1> ],
    [ <3, 1>, <5, 1>, <17, 1>, <23, 1>, <89, 1>, <257, 1>, <397, 1>, <683, 1>,
    <2113, 1> ],
    [ <3, 1>, <5, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>,
    <61681, 1> ],
    [ <3, 1>, <31, 1>, <8191, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>,
    <61681, 1> ],
    [ <3, 3>, <5, 1>, <7, 1>, <13, 1>, <17, 1>, <19, 1>, <29, 1>, <37, 1>,
    <43, 1>, <73, 1>, <109, 1>, <113, 1>, <127, 1>, <257, 1> ],
    [ <5, 1> ],
    [ <3, 1>, <5, 1> ],
    [ <3, 1>, <5, 2>, <11, 1>, <17, 1>, <31, 1>, <41, 1>, <53, 1>, <157, 1>,
    <1613, 1>, <2731, 1>, <8191, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <13, 1>, <17, 1>, <97, 1>, <241, 1>, <257, 1>,
    <673, 1> ],
    [ <3, 1>, <5, 1>, <17, 1>, <29, 1>, <43, 1>, <113, 1>, <127, 1>, <257, 1>,
    <65537, 1> ],
    [ <3, 1>, <5, 2>, <11, 1>, <29, 1>, <31, 1>, <41, 1>, <43, 1>, <113, 1>,
    <127, 1> ],
    [ <3, 1>, <5, 2>, <11, 1>, <17, 1>, <31, 1>, <41, 1>, <53, 1>, <157, 1>,
    <1613, 1>, <2731, 1>, <8191, 1> ],
    [ <3, 2>, <5, 2>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>, <61, 1>,
    <151, 1>, <257, 1>, <331, 1>, <1321, 1> ],
    [ <3, 1>, <5, 1>, <11, 1>, <31, 1>, <41, 1>, <257, 1>, <61681, 1>,
    <4278255361, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <17, 1>, <31, 1>, <41, 1>,
    <61681, 1> ],
    [ <3, 1>, <5, 1>, <17, 1>, <23, 1>, <89, 1>, <257, 1>, <397, 1>, <683, 1>,
    <2113, 1> ], [ <3, 2>, <5, 1>, <7, 1>, <11, 1>, <13, 1>, <23, 1>, <31, 1>,
    <41, 1>, <89, 1>, <397, 1>, <683, 1>, <2113, 1> ]
]
```

59.6 Abstract Group Predicates

`IsAbelian(G)`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsNilpotent(G)`

Returns `true` if the group G is nilpotent, `false` otherwise.

`IsSoluble(G)`
`IsSolvable(G)`

Returns `true` if the group G is soluble, `false` otherwise.

`IsPerfect(G)`

Returns `true` if the group G is perfect, `false` otherwise.

`IsSimple(G)`

Returns `true` if the group G is simple, `false` otherwise.

Example H59E11

We illustrate the functions of the last two section by applying them to a group of degree 6 over the field $\mathbf{F}_9$.

```
> F9<w> := GF(9);
> y := w^6;  z := w^2;
> J2A2 := MatrixGroup< 6, F9 | [y, 1-y, z,0,0,0, 1-y ,z, -1,0,0,0, z, -1,1+y,
>                               0,0,0,0,0,0, z, 1+y, y, 0,0,0,1+y, y, -1, 0,
>                               0,0, y ,-1,1-y],
>                              [1+y, z, y, 0,0,0, z, 1+y, z, 0,0,0, y, z, 1+y,
>                               0,0,0, z, 0,0,1-y, y, z, 0, z, 0, y, 1-y, y,
>                               0,0, z, z, y, 1-y],
>                              [0,0,0,y, 0,0, 0,0,0,0,y, 0, 0,0,0,0,0,y,
>                               y, 0,0,0,0,0, 0,y, 0,0,0,0, 0,0,y, 0,0,0] >;
> J2A2;
MatrixGroup(6, GF(3, 2))
Generators:
[w^6 w^3 w^2   0   0   0]
[w^3 w^2   2   0   0   0]
[w^2   2   w   0   0   0]
[  0   0   0 w^2   w w^6]
[  0   0   0   w w^6   2]
```

```
[  0   0   0 w^6   2 w^3]

[  w w^2 w^6   0   0   0]
[w^2   w w^2   0   0   0]
[w^6 w^2   w   0   0   0]
[w^2   0   0 w^3 w^6 w^2]
[  0 w^2   0 w^6 w^3 w^6]
[  0   0 w^2 w^2 w^6 w^3]

[  0   0   0 w^6   0   0]
[  0   0   0   0 w^6   0]
[  0   0   0   0   0 w^6]
[w^6   0   0   0   0   0]
[  0 w^6   0   0   0   0]
[  0   0 w^6   0   0   0]
> Order(J2A2);
1209600
> FactoredOrder(J2A2);
[ <2, 8>, <3, 3>, <5, 2>, <7, 1> ]
> IsSoluble(J2A2);
false
> IsPerfect(J2A2);
true
> IsSimple(J2A2);
false
```

Thus the group is non-soluble and perfect but it is not a simple group. We examine its Sylow2-subgroup.

```
> S2 := SylowSubgroup(J2A2, 2);
> IsAbelian(S2);
false
> IsNilpotent(S2);
true
> IsSpecial(S2);
false
```

59.7 Conjugacy

`Class(H, x)`

`Conjugates(H, x)`

Given a group H and an element x belonging to a group K such that H and K are subgroups of the same general linear group, this function returns the set of conjugates of x under the action of H. If $H = K$, the function returns the conjugacy class of x in H.

`ClassMap(G)`

Given a group G, construct the conjugacy classes and the class map f for G. For any element x of G, $f(x)$ will be the conjugacy class representative chosen by the `Classes` function.

`ConjugacyClasses(G: parameters)`

`Classes(G: parameters)`

`WeakLimit`	RngIntElt	*Default :* 500
`StrongLimit`	RngIntElt	*Default :* 5000
`Al`	MonStgElt	*Default :*

Construct a set of representatives for the conjugacy classes of the matrix group G. The classes are returned as a sequence of triples containing the element order, the class length and a representative element for the class. The parameter `Al` enables the user to select the algorithm that is to be used.

`Al := "Action"`: Create the classes of G by computing the orbits of the set of elements of G under the action of conjugation. This option is only feasible for small groups.

`Al := "Random"`: Construct the conjugacy classes of elements for a matrix group G using an algorithm that searches for representatives of all conjugacy of G by examining a random selection of group elements and their powers. The behaviour of this algorithm is controlled by two associated optional parameters `WeakLimit` and `StrongLimit`, whose values are positive integers n_1 and n_2, say. Before describing the effect of these parameters, some definitions are needed: A mapping $f : G \to I$ is called a class invariant if $f(g) = f(g^h)$ for all $g, h \in G$. In matrix groups, the primary invariant factors are used where possible, or the characteristic or minimal polynomials otherwise. Two matrices g and h are said to be *weakly conjugate* with respect to the class invariant f if $f(g) = f(h)$. By definition, conjugacy implies weak conjugacy, but the converse is false. The random algorithm first examines n_1 random elements and their powers, using a test for weak conjugacy. It then proceeds to examine a further n_2 random elements and their powers, using a test for ordinary conjugacy. The idea behind this strategy is that the algorithm should attempt to find as many classes as possible using the very cheap test for weak conjugacy, before employing the more expensive ordinary conjugacy test to recognize the remaining classes.

`Al := "Extend"`: Construct the conjugacy classes of G by first computing classes in a quotient G/N and then extending these classes to successively larger quotients G/H until the classes for $G/1$ are known. More precisely, a series of subgroups $1 = G_0 < G_1 < \cdots < G_r = R < G$ is computed such that R is the (solvable) radical of G and G_{i+1}/G_i is elementary abelian. The radical quotient G/R is computed and its classes and centralizers of their representatives found using the permutation group algorithm, and pulled back to G. The parameters `TFAl` and `ASAl` control the algorithm used to compute the classes of G/R. See the GrpPerm chapter for more information on these parameters.

To extend from G/G_{i+1} to the next larger quotient G/G_i, an affine action of each centralizer on a quotient of the elementary abelian layer G_{i+1}/G_i is computed. Each distinct orbit in that action gives rise to a new class of the larger quotient (see Mecky and Neubuser [MN89]).

`Al := "Lifting"`: Construct a permutation representation for G, compute the classes of the representation, and lift them back to G through the kernel of the representation. Successful when the kernel is small. Currently uses the permutation action of G on its first basic orbit as the permutation representation.

`Al := "Classic"`: Construct the conjugacy classes by enumeration of class invariants. This algorithm is only available for classical groups. It has only been implemented for groups containing the special linear group and for the conformal unitary group.

Default: The classic algorithm will be used if G is recognised to contain the special linear group (using IsLinearGroup), or if G is known to be conformal unitary group in the standard representation (that is, if G was constructed by ConformalUnitaryGroup). The action algorithm will be used if $|G| \leq 2000$. If G is soluble then classes are computed in a PC-representation of G. When $|G| > 2000$ and the base ring of G is a finite field then the Extension algorithm is used. Otherwise the Lifting algorithm is used, unless the kernel size exceeds 10000. If there is a big kernel and the base ring of the group can be embedded in a field then the extension algorithm is used. Otherwise the random algorithm will be applied with the limits given by the parameters `WeakLimit` and `StrongLimit`. If that fails to compute all the classes and $|G| \leq 100000$, then the action algorithm will be used.

`ClassRepresentative(G, x)`

Given a group G for which the conjugacy classes are known and an element x of G, return the designated representative for the conjugacy class of G containing x.

`ClassCentraliser(G, i)`

The centraliser of the representative element stored for conjugacy class number i in group G. The group computed is stored with the class table for reference by future calls to this function.

`ClassInvariants(G, g)`

`ClassInvariants(G, i)`

The invariants for the conjugacy class of g in G or the conjugacy class number i in G. The type of invariants may vary depending on the group. This is only available for groups, for which the *classic* algorithm for computing conjugacy classes is available.

`ClassRepresentativeFromInvariants(G, p, h, t)`

Given a group G, for which the *classic* algorithm for computing conjugacy classes is available, and the class invariants p, h and t, return the standard class representative for the conjugacy class in G with the given invariants.

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function returns a second value in the event that the elements are conjugate: an element k which conjugates g into h.

`IsConjugate(G, H, K)`

Given a group G and subgroups H and K belonging to G, return the value `true` if H and K are conjugate in G. The function returns a second value in the event that the subgroups are conjugate: an element z which conjugates H into K.

`IsGLConjugate(H, K)`

Given H and K, both subgroups of the same general linear group $G = GL_n(q)$, return the value `true` if H and K are conjugate in G. The function returns a second value in the event that the subgroups are conjugate: an element z which conjugates H into K. The algorithm is described in Roney-Dougal [RD04].

`Exponent(G)`

The exponent of the group G.

`NumberOfClasses(G)`

`Nclasses(G)`

The number of conjugacy classes of elements for the group G.

`PowerMap(G)`

Given a group G, construct the power map for G. Suppose that the order of G is m and that G has r conjugacy classes. When the classes are determined by MAGMA, they are numbered from 1 to r. Let C be the set of class indices $\{1, \ldots, r\}$ and let P be the set of integers $\{1, \ldots, m\}$. The *power map* f for G is the mapping,

$$f : C \times P \to C$$

where the value of $f(i, j)$ for $i \in C$ and $j \in P$ is the number of the class which contains x_i^j, where x_i is a representative of the i-th conjugacy class.

`AssertAttribute(G, "Classes", Q)`

Given a group G, and a sequence Q of k distinct elements of G, one from each conjugacy class, use Q to define the classes attribute of G. The sequence Q may be either a sequence of elements of G or, preferably, a sequence of pairs `<GrpMatElt, RngIntElt>` giving class representatives and their class length. In this latter case, no backtrack searches are performed.

Example H59E12

We take a group from the database of rational matrix groups and compute its conjugacy classes. The group has degree 12 and is written over the integers.

```
> DB := RationalMatrixGroupDatabase();
> G := Group(DB, 12, 3);
> FactoredOrder(G);
[ <2, 17>, <3, 8>, <5, 2> ]
> CompositionFactors(G);
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  C(2, 3)               = S(4, 3)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  C(2, 3)               = S(4, 3)
    *
    |  Cyclic(2)
    1
```

The conjugacy classes of `G` are computed as follows:

```
> time cl := Classes(G);
Time: 18.580
> #cl;
1325
```

The group has 1325 conjugacy classes of elements.

59.8 Subgroups

59.8.1 Construction of Subgroups

`sub< G | L >`

Given the matrix group G, construct the subgroup H of G generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) A sequence of n integers defining a matrix of G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic matrix group. The subgroup H will be constructed as a subgroup of some group which contains each of the elements and groups specified in the list.

The generators of H consist of the elements specified by the terms of the list L together with the stored generators for groups specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed.

`ncl< G | L >`

Given the matrix group G, construct the subgroup H of G that is the *normal closure* of the subgroup H generated by the elements specified by the list L, where the possibilities for L are the same as for the `sub`-constructor.

Example H59E13

We define $O^-(4,2)$ as a subgroup of $\mathrm{GL}(4,2)$. Recall that $O^-(4,2)$ is isomorphic to S_5. We then locate a subset of its generators that lie within the subgroup isomorphic to A_5.

```
> GL42 := GeneralLinearGroup(4, GF(2));
> Ominus42 := sub< GL42 | [1,0,0,0, 1,1,0,1, 1,0,1,0, 0,0,0,1 ],
>                         [0,1,0,0, 1,0,0,0, 0,0,1,0, 0,0,0,1 ],
>                         [0,1,0,0, 1,0,0,0, 0,0,1,0, 0,0,1,1 ] >;
> Order(Ominus42);
120
> H := sub< Ominus42 | $.1, $.3 >;
print Order(H);
10
> N := ncl< Ominus42 | $.1, $.3 >;
> Order(N);
60
```

59.8.2 Elementary Properties of Subgroups

`Index(G, H)`

The index of the subgroup H in the group G. The index is returned as an integer. If the orders of G and H are not known, they will be computed.

`FactoredIndex(G, H)`

The index of the subgroup H in the group G. The index is returned as a factored integer. The factorization is returned in the form of a sequence Q which is defined as follows: If $[G : H] = p_1^{e_1} \dots p_n^{e_n}, \quad e_i \neq 0$, then Q will be the integer sequence $[< p_1, e_1 >, \dots, < p_n, e_n >]$. If the orders of G and H are not known, they will be computed.

`IsCentral(G, H)`

Returns `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

`IsMaximal(G, H)`

Returns `true` if the subgroup H of the group G is a maximal subgroup of G. This function is evaluated by constructing the permutation representation of G on the cosets of H and testing this representation for primitivity. For this reason, the use of `IsMaximal` should be avoided if the index of H in G exceeds a few thousand.

`IsNormal(G, H)`

Returns `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

`IsSubnormal(G, H)`

Returns `true` if the subgroup H of the group G is subnormal in G, `false` otherwise.

59.8.3 Standard Subgroups

`H ^ g`

`Conjugate(H, g)`

Construct the conjugate $g^{-1} * H * g$ of the matrix group H by the matrix g. The group H and the element g must belong to a common matrix group.

`H meet K`

Given groups H and K which belong to the same matrix group, construct the intersection of H and K.

`CommutatorSubgroup(G, H, K)`

`CommutatorSubgroup(H, K)`

Given subgroups H and K of the group G, construct the commutator subgroup of H and K as a subgroup of G. If K is a subgroup of H, then G may be omitted.

`Centralizer(G, g)`

Construct the centralizer of the matrix g in the group G; g and G must belong to a common matrix group.

`Centralizer(G, H)`

Construct the centralizer of the group H in the group G; G and H must belong to a common matrix group.

`Core(G, H)`

Given a subgroup H of the matrix group G, construct the maximal normal subgroup of G that is contained in the subgroup H.

`H ^ G`

`NormalClosure(G, H)`

Given a subgroup H of the matrix group G, construct the normal closure of H in G.

`Normalizer(G, H)`

Given a subgroup H of the group G, construct the normalizer of H in G.

`SylowSubgroup(G, p)`

`Sylow(G, p)`

Given a group G and a prime p, construct the Sylow p-subgroup of G.

`pCore(G, p)`

Given a group G and a prime p dividing the order of G, construct the maximal normal p-subgroup of G.

59.8.4 Low Index Subgroups

`LowIndexSubgroups(G, R : parameters)`

`LowIndexSubgroups(G, R: parameters)`

Given a matrix group G, and an expression R defining a positive integer range (see below), determine the conjugacy classes of subgroups of G whose indices lie in the range specified by R. The subgroups are returned as a sequence of subgroups of G. The argument R is one of the following:

(a) An integer n representing the range $[1, n]$;

(b) A tuple $< a, b >$ representing the range $[a, b]$.

The subgroups are constructed using an algorithm due to Leedham-Green & O'Brien [LGO02]. In practice, the algorithm is most useful for small values of n, say up to 8.

The algorithm proceeds by iteratively constructing better approximations to finite presentations for G/K, where K is the intersection of kernels of all homomorphisms from G into S_n, and applying `LowIndexSubgroups` to the resulting finitely-presented group. The output information displayed for various values of the `Print` parameter about the number and existence of putative subgroups of index at most n refers to the current finite presentation only, may change as this presentation is further refined, and need not be reflected in the final answer.

`Limit` RngIntElt *Default :* ∞

Terminate after finding n conjugacy classes of subgroups satisfying the designated conditions.

`Print` RngIntElt *Default :* 0

The `Print` parameter takes values from 0 to 3. The information displayed

Example H59E14

```
> G := GL (4, 5);
> L := LowIndexSubgroups (G, 4);
> #L;
3
> L[3];
MatrixGroup(4, GF(5))
Generators:
    [4 0 0 4]
    [1 0 0 0]
    [0 4 0 0]
    [0 0 4 0]

    [4 0 0 3]
    [3 0 0 0]
    [0 4 0 0]
```

```
[0 0 4 0]

[4 0 0 1]
[4 0 0 0]
[0 4 0 0]
[0 0 4 0]

[4 0 0 2]
[2 0 0 0]
[0 4 0 0]
[0 0 4 0]
```

59.8.5 Conjugacy Classes of Subgroups

`SubgroupClasses(G: parameters)`

`Subgroups(G: parameters)`

Representatives for the conjugacy classes of subgroups for the group G. The subgroups are returned as a sequence of records where the i-th record contains:

(a) A representative subgroup H for the i-th conjugacy class (field name `subgroup`).

(b) The order of the subgroup (field name `order`).

(c) The number of subgroups in the class (field name `length`).

(d) [Optional] A presentation for H (field name `presentation`).

`Al` MonStgElt *Default :* "*All*"

`Al := "All"`: Construct all subgroups of G.
`Al := "Maximal"`: Only construct maximal subgroups of G. This option reduces the number of intersections with any elementary abelian layer that need be considered and eliminates the need to recursively apply the algorithm.
`Al := "Normal"`: Only construct normal subgroups of G. This option does not use database lookup to find the normal subgroups of the radical quotient of G and also reduces the number of intersections with any layer that need be considered.

`LayerSizes` SeqEnum *Default :* See below

`LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]` is equivalent to the default. When constructing an Elementary Abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1.

`Series` SeqEnum *Default :* See below

Use the given elementary abelian series rather than constructing the default series. The first subgroup in the series must be the solvable radical of G. The subgroups must form a descending chain of normal subgroups of G, such that each quotient

is elementary abelian. The last subgroup in the series must be either elementary abelian or trivial.

`Presentation` BoolElt *Default :* `false`

`Presentation := true`: Construct a presentation for each subgroup.

`OrderEqual` RngIntElt *Default :*

`OrderEqual := n`: Only construct subgroups having order equal to n.

`OrderDividing` RngIntElt *Default :*

`OrderDividing := n`: Only construct subgroups having order dividing n.

`OrderMultipleOf` RngIntElt *Default :*

`OrderMultipleOf := n`: Only construct subgroups having order a multiple of n.

`IndexLimit` RngIntElt *Default :*

`IndexLimit := n`: Only construct subgroups having index in G less than or equal to n.

`IsElementaryAbelian` BoolElt *Default :* `false`

`IsElementaryAbelian := true`: Only construct elementary abelian subgroups of G.

`IsCyclic` BoolElt *Default :* `false`

`IsCyclic := true`: Only construct cyclic subgroups of G.

`IsAbelian` BoolElt *Default :* `false`

`IsAbelian := true`: Only construct abelian subgroups of G.

`IsNilpotent` BoolElt *Default :* `false`

`IsNilpotent := true`: Only construct nilpotent subgroups of G.

`IsSolvable` BoolElt *Default :* `false`

`IsSolvable := true`: Only construct solvable subgroups of G.

`IsNotSolvable` BoolElt *Default :* `false`

`IsNotSolvable := true`: Only construct insolvable subgroups of G.

`IsPerfect` BoolElt *Default :* `false`

`IsPerfect := true`: Only construct perfect subgroups of G.

The Algorithm: (See Cannon, Cox and Holt [CCH01]) This command proceeds by first constructing an elementary abelian series for G together with G's radical quotient Q as a permutation group. (Thus this function is limited to matrix groups over fields, where the group has a BSGS.) The required subgroups of Q are then found as for permutation groups. We first attempt to locate the quotient in a database of groups with trivial Fitting subgroup. This database contains all such groups of order up to 216 000, and all such which are perfect of order up to 1 000 000. If Q is found then either all its subgroups, or its maximal subgroups are read from the database. (In some cases only the maximal subgroups are stored.) If Q is

not found then we attempt to find the maximal subgroups of Q using a method of Derek Holt. For this to succeed all simple factors of the socle of Q must be found in a second database which currently contains all simple groups of order less than 1.6×10^7, as well as M_{24}, HS, J_3, McL, $Sz(32)$ and $L_6(2)$. There are also special routines to handle numerous other groups. These include: A_n for $n \leq 999$, $L_2(q)$, $L_3(q)$, $L_4(q)$ and $L_5(q)$ for all q, $U_3(q)$ for q prime and $q = 8, 9, 16, 25$, $U_4(q)$ for $q = 4, 5, 7$, $S_4(q)$ for all odd q and even $q \leq 16$, $L_d(2)$ for $d \leq 14$, and the following groups: $L_6(3)$, $L_7(3)$, $U_6(2)$, $S_8(2)$, $S_{10}(2)$, $O_8^{\pm}(2)$, $O_{10}^{\pm}(2)$, $S_6(3)$, $O_7(3)$, $O_8^-(3)$, $G_2(4)$, $G_2(5)$, ${}^3D_4(2)$, ${}^2F_4(2)'$, Co_2, Co_3, He, Fi_{22}.

If we have only maximal subgroups of Q, and more are required, we apply the algorithm recursively to the maximal subgroups to determine all subgroups of Q. This may take some time.

The subgroups of Q are then pulled back to G and extended to the whole group by stepwise extension through each layer of the elementary abelian series. For each layer this involves determining all possible intersections of a subgroup with this layer and all extensions with this intersection.

The limitations are that the simple factors of the socle of Q must be in the list above. Further, it may take some time to construct all subgroups from the maximal subgroups first found, and, if there is a large elementary abelian layer, there will be many possible intersections, which could also make the algorithm prohibitively slow.

There are numerous parameters for this function which allow the user to place restrictions on which subgroup classes are constructed. Using these restrictions may help overcome the problems noted above.

`MaximalSubgroups(G:` *parameters*`)`

Construct the sequence of maximal subgroup classes of the matrix group G. This is equivalent to the command `Subgroups(G: Al := "Maximal")`. The same parameters as for `Subgroups` are available to limit the search.

`SubgroupsLift(G, A, B, Q:` *parameters*`)`

This function isolates one step of the extension process used by the `Subgroups` family of functions. Q is a sequence of records such as returned by `Subgroups(G)`. A and B are normal subgroups of G with A/B elementary abelian. The records in Q are interpreted as subgroups of G/A, which are lifted to all possible corresponding subgroups of G/B, subject to the parameters given.

59.9 Quotient Groups

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.9.1 Construction of Quotient Groups

`quo< G | L >`

Given the matrix group G, construct the quotient group $Q = G/N$, where N is the normal closure of the subgroup of G generated by the elements specified by L. The clause L is a list of one or more items of the following types:

(a) A sequence of n integers defining a matrix of G;

(b) A set or sequence of sequences of type (a);

(c) An element of G;

(d) A set or sequence of elements of G;

(e) A subgroup of G;

(f) A set or sequence of subgroups of G.

Each element or group specified by the list must belong to the *same* generic matrix group. The function returns

(a) the quotient group Q, and

(b) the natural homomorphism $f : G \to Q$.

Currently in MAGMA, the quotient group is constructed via the regular representation of the quotient, so that the application of this operator is restricted to the case where the index of N in G is small. The representation of the quotient that is returned is the result of applying degree reduction to the regular representation, so need not be regular.

The generators of the quotient group correspond to the generators of G.

`G / N`

Given a normal subgroup N of the matrix group G, construct the quotient of G by N. Currently in MAGMA, the quotient group is constructed via the regular representation of the quotient, so the application of this operator is restricted to the case where the index of N in G is small. The representation of the quotient that is returned is the result of applying degree reduction to the regular representation, so need not be regular.

Example H59E15

We determine the structure of a quotient in a soluble subgroup of $\mathrm{GL}(3, 5)$.

```
> G := MatrixGroup< 3, GF(5) | [0,1,0, 1,0,0, 0,0,1], [0,1,0, 0,0,1, 1,0,0 ],
>                                    [2,0,0, 0,1,0, 0,0,1] >;
> Order(G);
384
> Q, f := quo< G | G.2 >;
> Q;
Permutation group Q of degree 8
    (1, 2)(3, 4)(5, 6)(7, 8)
```

```
    Id(Q)
    (1, 3, 5, 7)(2, 4, 6, 8)
> IsAbelian(Q);
true
> AbelianInvariants(Q);
[ 4, 2 ]
```

59.9.2 Abelian, Nilpotent and Soluble Quotients

A number of standard quotients may be constructed. The method first constructs a presentation for the matrix group and then applies the appropriate fp-group algorithm.

`AbelianQuotient(G)`

The maximal abelian quotient G/G' of the group G as `GrpAb` (cf. chapter 69). The natural epimorphism $\pi : G \to G/G'$ is returned as second value.

`ElementaryAbelianQuotient(G, p)`

The maximal p-elementary abelian quotient Q of the group G as `GrpAb` (cf. chapter 69). The natural epimorphism $\pi : G \to Q$ is returned as second value.

`pQuotient(G, p, c)`

Given a matrix group G, a prime p and a positive integer c, construct a pc-presentation for the largest p-quotient P of G having lower exponent-p class at most c. If c is given as 0, then the limit 127 is placed on the class.

The function also returns the natural homomorphism π from G to P, a sequence S describing the definitions of the pc-generators of P and a flag indicating whether P is the maximal p-quotient of G.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $P.k$ of P as follows.

- If $S[k] = [0, r]$, then $P.k$ is defined via the image of $G.r$ under π.
- If $S[k] = [r, 0]$, then $P.k$ is defined via the power relation for $P.r$.
- If $S[k] = [r, s]$, then $P.k$ is defined via the conjugate relation involving $P.r^{P.s}$.

`NilpotentQuotient(G, c)`

This function returns the class c nilpotent quotient of the matrix group G, together with the epimorphism π from G onto this quotient.

`SolvableQuotient(G)`

`SolubleQuotient(G)`

The function returns the largest soluble quotient S of the matrix group G together with the epimorphism $\pi : G \to S$.

PCGroup(G)

For a solvable group G, the function returns an isomorphic group of type GrpPC together with an isomorphism from G to the new group. If G is not solvable, then the call to PCGroup will result in an error.

Example H59E16

We take a degree 10 matrix group over the integers and compute its maximal abelian and soluble quotients. The epimorphisms supplied by these two functions may be used to pass between the group and its quotients.

```
> DB := RationalMatrixGroupDatabase();
> G := Group(DB, 10, 2);
> G : Minimal;
MatrixGroup(10, Integer Ring) of order 4147200
> A := AbelianQuotient(G); A;
Abelian Group isomorphic to Z/2 + Z/2 + Z/2
Defined on 3 generators
Relations:
  2*A.1 = 0
  2*A.2 = 0
  2*A.3 = 0
> S, f := SolubleQuotient(G); S;
GrpPC : S of order 32 = 2^5
PC-Relations:
  S.2^2 = S.4,
  S.2^S.1 = S.2 * S.4,
  S.3^S.2 = S.3 * S.4 * S.5
> G.1 @ f;
S.1 * S.4 * S.5
> S.5 @@ f in DerivedGroup(G);
true
```

59.10 Matrix Group Actions

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.10.1 Orbits and Stabilizers

Let G be a matrix group and let M be its natural module. Now G has an action on the elements and submodules of M. A derived G-set for G consists of the closure under the natural action of G of one of the following:

- A set of vectors of M;
- A set of k element subsets of vectors of M;
- A set of k element sequences of vectors of M;
- A set of submodules of M, each of which has fixed dimension k;
- A cartesian product of G-sets.

`u * g`

Given an element g belonging to the matrix group G with natural module M and an element u of this module, return the vector $u * g$.

`y ^ g`

Given an element g belonging to the matrix group G with natural module M and an object y which is an element of some derived G-set of M, find the image of y under g.

`y ^ G`

`Orbit(G, y)`

Given a matrix group G with natural module M and an object y which is either a vector of M, a submodule of M, or a tuple whose components are either vectors or submodules, find the orbit of y under G.

`OrbitBounded(G, y, b)`

Given a matrix group G with natural module M and an object y which is either a vector of M, a submodule of M, or a tuple whose components are either vectors or submodules, return `true` if the orbit of y under G has length less than or equal to b. Otherwise the function returns `false`. If it returns `true`, then the orbit of y is returned as the second value.

`Orbits(G)`

Given a matrix group G with natural R-module M, construct the orbits of G on the vectors of M. The orbits are returned as a sequence of sets.

`LineOrbits(G)`

Given a matrix group G with natural R-module M, construct the orbits of G on the rank-1 submodules of M. The orbits are returned as a sequence of sets.

`OrbitClosure(G, S)`

Given a matrix group G with natural module M and a set S of vectors or subspaces of M, return the union of orbits of the elements of S under the natural action of G on M.

`Stabilizer(G, y)`

Given a matrix group G with natural module M and an object y which is either a vector of M, a submodule of M, or a tuple whose components are either vectors or submodules, determine the stabilizer of y in G.

Example H59E17

We continue with the group $J2A2$ introduced above.

```
> V := RSpace(G);
> u := V![1,0,0,0,0,0];
> U := sub< V | u >;
> x := < u, U >;
> W := sub< V | u, u*G.1 >;
> u^G.1;
(w^6 w^3 w^2   0   0   0)
> U^G.1;
Vector space of degree 6, dimension 1 over GF(3, 2)
Echelonized basis:
(  1 w^5   2   0   0   0)
> W^G.1;
Vector space of degree 6, dimension 2 over GF(3, 2)
Echelonized basis:
(  1 w^5   0   0   0   0)
(  0   0   1   0   0   0)
> x^G.1;
<(w^6 w^3 w^2   0   0   0), Vector space of degree 6,
dimension 1 over GF(3, 2)
Echelonized basis:
(  1 w^5   2   0   0   0)>
> H := sub< G | G.1, G.2 >;
> #Orbit(H, u);
252
> #Orbit(H, U);
63
> #Orbit(G, U);
3150
> Stabilizer(G, U);
MatrixGroup(6, GF(3^2)) of order 384 = 2^7 * 3
Generators:
[  2   0   0   0   0   0]
[w^3   w   w   0   2 w^2]
[w^5 w^7 w^7   0   1 w^2]
```

```
[  0   0   1   2   1   0]
[w^7 w^5   0   0   0 w^6]
[  w w^3   0   0   0 w^6]

[w^2   0   0   0   0   0]
[w^5 w^5 w^5   0   w   0]
[w^7 w^3 w^3   0   0 w^7]
[w^2 w^3   w w^6   w w^3]
[w^3   1 w^6   0   w w^7]
[  w w^6   2   0   w w^7]

[w^6   0   0   0   0   0]
[  0   2   0   0   0   0]
[  0   0 w^6   0   0   0]
[w^2 w^7 w^6 w^2   0   0]
[  w   0   w   0   2   0]
[w^6 w^7 w^2   0   0 w^2]

[  2   0   0   0   0   0]
[  0   2   0   0   0   0]
[  0   0   2   0   0   0]
[  0   0   0   2   0   0]
[  0   0   0   0   2   0]
[  0   0   0   0   0   2]
> #Orbit(H, x);
252
> #Orbit(H, W);
28
```

59.10.2 Orbit and Stabilizer Functions for Large Groups

In this section we describe a number of constructions for orbits and stabilizers which in certain circumstances may be applicable to much larger groups than the functions described above.

`OrbitsOfSpaces(G, k)`

Determine representatives and lengths for the orbits of all k-dimensional subspaces of the natural vector space under action of a matrix group defined over a prime field; return a sequence of tuples each containing an orbit length and representative. This function is very space-efficient and hence has a significantly larger range than the general-purpose `Orbits`; however, only representatives and lengths are stored. Theoretical details of the algorithm used may be found in O'Brien [O'B90].

`NumberOfFixedSpaces(x, s)`

`NumberOfFixedSpaces(x, s)`

Return number of subspaces of dimension s fixed by matrix x.

Example H59E18

```
> G := GL (4, 5);
> H := ExteriorSquare (G);
> H;
MatrixGroup(6, GF(5))
Generators:
    [2 0 0 0 0 0]
    [0 2 0 0 0 0]
    [0 0 1 0 0 0]
    [0 0 0 2 0 0]
    [0 0 0 0 1 0]
    [0 0 0 0 0 1]

    [0 0 0 1 0 0]
    [1 0 0 0 1 0]
    [1 0 0 0 0 0]
    [0 1 0 0 0 1]
    [0 1 0 0 0 0]
    [0 0 1 0 0 0]
> O := OrbitsOfSpaces (H, 2);
```

We see that there are four orbits:

```
> O;
[
    <
        4836,
        Vector space of degree 6, dimension 2 over GF(5)
        Generators:
        (1 0 0 0 0 0)
        (0 1 0 0 0 0)
        Echelonized basis:
        (1 0 0 0 0 0)
        (0 1 0 0 0 0)
    >,
    <
        96720,
        Vector space of degree 6, dimension 2 over GF(5)
        Generators:
        (1 0 1 1 0 0)
        (0 1 0 0 0 0)
        Echelonized basis:
        (1 0 1 1 0 0)
        (0 1 0 0 0 0)
    >,
    <
        251875,
        Vector space of degree 6, dimension 2 over GF(5)
```

```
        Generators:
        (1 0 0 0 1 0)
        (0 1 0 0 0 0)
        Echelonized basis:
        (1 0 0 0 1 0)
        (0 1 0 0 0 0)
    >,
    <
        155000,
        Vector space of degree 6, dimension 2 over GF(5)
        Generators:
        (1 0 1 1 1 0)
        (0 1 1 1 0 0)
        Echelonized basis:
        (1 0 1 1 1 0)
        (0 1 1 1 0 0)
    >
]
```

We compute the number of spaces of dimension 2 fixed by $H.1$ and the number of spaces of dimension 3 fixed by $H.2$.

```
> NumberOfFixedSpaces(H.1, 2);
1023
> NumberOfFixedSpaces(H.2, 3);
2
```

`EstimateOrbit(G, v: parameters)`

`EstimateOrbit(G, U: parameters)`

`MaxSize`	RngIntElt	*Default :* 10000
`NumberCoincidences`	RngIntElt	*Default :* 15

Estimate the size of the orbit of the vector v or subspace U of natural vector space under the action of matrix group G by constructing at most `MaxSize` random elements of the orbit and counting at most `NumberCoincidences` coincidences. The function returns a lower bound, upper bound, and estimate of size; if insufficient coincidences are found to estimate the orbit size, the function returns 0. Theoretical details of the algorithm used may be found in Eick, Leedham-Green and O'Brien [ELGO02].

`ApproximateStabiliser(G, A, U: parameters)`

`ImageGenerators`	SeqEnum	*Default :* []
`MaxSize`	RngIntElt	*Default :* 10000
`NumberCoincidences`	RngIntElt	*Default :* 15
`OrderCheck`	BoolElt	*Default :* `false`

A is image of representation of G and A acts on U, a subspace or vector. Approximate the stabiliser of U under A. We assume either a $1-1$ correspondence between generators of G and those of A, or between generators of G and those elements of A supplied as `ImageGenerators`. Elements of G whose images in A fix U are obtained by constructing at most `MaxSize` elements of the orbit of U under A or until we find `Numbercoincidences` repetitions in this orbit; if `OrderCheck` is `true`, report the order of the subgroup S of A which is found. Return preimage of S in G and S, together with a lower bound, upper bound, and estimate of the size of orbit of U. If insufficient coincidences are found to estimate the orbit size, the function returns these last values as 0.

Example H59E19

```
> G := GL (4, 5);
> A := ExteriorSquare (G);
> V := VectorSpace (GF (5), 6);
> U := sub < V | [Random (V): i in [1..2]]>;
> U;
Vector space of degree 6, dimension 2 over GF(5)
Generators:
(4 3 2 1 0 2)
(3 2 2 4 4 1)
Echelonized basis:
(1 0 2 0 2 4)
(0 1 3 2 4 2)
> EstimateOrbit (A, U);
209316 594421 324272
> H, B, lb, ub, estimate := ApproximateStabiliser (G, A, U);
> #H, #B;
460800 230400
```

`StabiliserOfSpaces(Q)`

Determine the subgroup of $GL(d, F)$, for F a finite field, which stabilises the sequence Q of subspaces of the natural vector space. The function also returns generators for the largest unipotent subgroup of the stabiliser. For a description of this algorithm, see Schwingel [Sch00]; this implementation was prepared by Eamonn O'Brien.

Example H59E20

```
> V := VectorSpace(GF (3), 4);
> Spaces := [sub< V | [1,1,0,2]>, sub < V | [ 1, 0, 2, 0 ], [ 0, 1, 0, 0]>];
> S, P := StabiliserOfSpaces(Spaces);
> #S;
5184
> P;
[
    [1 1 0 0]
    [0 1 0 0]
    [0 1 1 0]
    [0 1 0 1],

    [2 0 2 0]
    [0 1 0 0]
    [1 0 0 0]
    [1 0 2 1],

    [1 0 1 2]
    [0 1 0 0]
    [0 0 2 2]
    [0 0 1 0]
]
```

Thus, the unipotent subgroup generated by P has order 3^3.

`IsUnipotent(G)`

If G is a p-subgroup of $GL(d, F)$, where F is a finite field of characteristic p, then return `true`, else return `false`.

`UnipotentStabiliser(G, U:` *parameters*`)`

Given a unipotent subgroup G of $GL(d, F)$, for F a finite field, U a subspace of the natural vector space, determine the stabiliser in G of U. The function returns the stabiliser in G of U, the *canonical* element C of the orbit of U under G, an element x of G such that $U^x = C$, and an SLP for x as an element of `WordGroup`(G). This function does *not* compute the orbit of U under G, but instead constructs the canonical element of the orbit. Hence it can be used to decide whether or not two subspaces belong to the same orbit. For a description of this algorithm, see [Sch00]; this implementation was prepared by Elliot Costi.

Example H59E21

```
> V := VectorSpace(GF (3), 4);
> G := sub< GL (4, 3) |
>     [ 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1 ],
>     [ 2, 0, 2, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 2, 1 ],
>     [ 1, 0, 1, 2, 0, 1, 0, 0, 0, 0, 2, 2, 0, 0, 1, 0 ] >;
> U := sub < V | [ 1, 2, 0, 1 ],[ 2, 2, 1, 0 ]>;
> S, C, x, w := UnipotentStabiliser(G, U);
> S;
MatrixGroup(4, GF(3))
Generators:
    [2 1 2 0]
    [0 1 0 0]
    [1 1 0 0]
    [1 1 2 1]
> #S;
3
> Index(G, S);
9
```

So the stabiliser of U has order 3 and U lies in an orbit of size 9. We print the canonical element of the orbit of U under G. The element x maps U to C and w evaluates to x.

```
> C;
Vector space of degree 4, dimension 2 over GF(3)
Echelonized basis:
(1 0 0 0)
(0 1 2 0)
>  U^x;
Vector space of degree 4, dimension 2 over GF(3)
Echelonized basis:
(1 0 0 0)
(0 1 2 0)
> W, phi := WordGroup (G);
>  phi (w);
[1 0 2 1]
[0 1 0 0]
[0 0 0 1]
[0 0 2 2]
```

59.10.3 Action on Orbits

`OrbitAction(G, T)`

Given a matrix group G with natural module M, and a set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T and construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. The function returns:

(a) The natural homomorphism $\phi : G \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of G).

`OrbitActionBounded(G, T, b)`

Given a matrix group G with natural module M, and a set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T. If the cardinality of Y does not exceed b, then construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. In this case the function returns:

(a) The boolean value `true`.

(b) The natural homomorphism $\phi : G \to L$;

(c) The induced group L;

(d) The kernel of the action (a subgroup of G). If the cardinality of Y exceeds b, simply return `false`. (The action of G on Y is not constructed in this case).

`OrbitImage(G, T)`

Given a matrix group G with natural module M, and a set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T and return the permutation group L giving the action of G on Y.

`OrbitImageBounded(G, T, b)`

Given a matrix group G with natural module M, and set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T. If the cardinality of Y does not exceed b, return `true` together with the permutation group L giving the action of G on Y. If the cardinality of Y does exceed b, the action is not constructed and the single value `false` is returned.

`OrbitKernel(G, T)`

Given a matrix group G with natural module M, and a set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T and return the the kernel of the action of G on Y.

`OrbitKernelBounded(G, T, b)`

Given a matrix group G with natural module M, and set T consisting of either (a) elements of M, (b) submodules of M or (c) tuples, form the G-closure Y of T. If the cardinality of Y does not exceed b, return the boolean value `true` together with the kernel of the action of G on Y. If the cardinality of Y does exceed b, the kernel is not constructed and the single value `false` is returned.

Example H59E22

We look for a small G-set for the group $J2A2$ (defined above) by examining eigenspaces of its generators. Having found a reasonably sized set, we then construct a permutation representation for G on this set.

```
> [ Factorization(CharacteristicPolynomial(G.i)) : i in [1..3] ];
[
    [
          <x^3 + w^5*x^2 + w^3*x + 2, 1>,
          <x^3 + w^7*x^2 + w*x + 2, 1>
    ],
    [
          <x + 2, 6>
    ],
    [
         <x + w^2, 3>,
         <x + w^6, 3>
    ]
]
> y := Eigenspace(G.2, -2);
> y;
Vector space of degree 6, dimension 3 over GF(3, 2)
Echelonized basis:
(1 0 0 1 2 1)
(0 1 0 2 1 2)
(0 0 1 1 2 1)
> #Orbit(G, y);
280
> P := OrbitImage(G, y);
> P;
Permutation group P of degree 280
> Order(P);
604800
> CompositionFactors(P);
    G
    |  J2
    1
```

Thus, our group has the simple group J_2 of Janko as a composition factor.

```
> Order(G);
```

```
1209600
```

Hence the kernel of this action has order 2.

59.10.4 Action on a Coset Space

`CosetAction(G, H)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced permutation group L;

(c) The kernel K of the action (a subgroup of G).

`CosetImage(G, H)`

Given a subgroup H of the group G, construct the image L of G given by the action of G on the set of (right) cosets of H in G. L is returned as a permutation group.

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of the action of G on the set of (right) cosets of H in G.

Example H59E23

We construct $G = \mathrm{SL}(3, 3)$, a subgroup H of G, and the permutation representation of G given by its action on the cosets of H.

```
> G := MatrixGroup< 3, GF(3) | [0,2,0, 1,1,0, 0,0,1], [0,1,0, 0,0,1, 1,0,0] >;
> Order(G);
5616
> H := sub< G | G.1^2, G.2 >;
> Order(H);
216
> P := CosetImage(G, H);
> P;
Permutation group P of degree 26
    (1, 2)(3, 4, 6, 5, 7, 9)(8, 11)(10, 13, 15, 20, 18, 17)
      (12, 16, 21, 14, 19, 24)(23, 26)
    (2, 3, 5)(4, 6, 8)(7, 10, 14)(9, 12, 17)(11, 15, 20)(13, 18, 23)
      (16, 22, 21)(19, 25, 24)
```

59.10.5 Action on the Natural G-Module

A set of functions is provided for working with the action of G on the natural G-module M, for a matrix group G defined over a finite field. Many of these functions are similar to those presented in the general module chapter.

`GModule(G)`

The natural $R[G]$-module M for the matrix group G.

`IsIrreducible(G)`

Given a matrix group G, return `true` iff G acts irreducibly on its natural module M. If G acts reducibly on M, a proper submodule S of M is also returned.

`SubmoduleAction(G, S)`

Given a matrix group G and a submodule S of the natural module M of G, return the action homomorphism f of G on S, together with the image of f.

`SubmoduleImage(G, S)`

Given a matrix group G and a submodule S of the natural module M of G, return the image of the action homomorphism of G on S.

`QuotientModuleAction(G, S)`

Given a matrix group G and a submodule S of the natural module M of G, return the quotient action homomorphism f of G on S, together with the image of f.

`QuotientModuleImage(G, S)`

Given a matrix group G and a submodule S of the natural module M of G, return the quotient image of the action homomorphism of G on S.

`IsAbsolutelyIrreducible(G)`

Given a matrix group G, return `true` if and only if G acts absolutely irreducibly on its natural module M. In addition, if G is absolutely irreducible, the function returns the (matrix algebra) generator of the endomorphism algebra E of M (which is always a field), and the dimension of E.

`AbsoluteRepresentation(G)`

Given an irreducible matrix group G, return the isomorphic reduced-degree absolute representation A of G, which is over the absolute field of the natural module M of G and is absolutely irreducible, together with the corresponding isomorphism.

`MinimalField(G)`

Given a matrix group G defined over a finite field K, return the minimal subfield of K over which G can be realised.

59.11 Normal and Subnormal Subgroups

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.11.1 Characteristic Subgroups and Subgroup Series

`Centre(G)`
`Center(G)`

Construct the centre of the group G.

`DerivedLength(G)`

The derived length of the matrix group G. If G is non-soluble, the function returns the number of terms in the series terminating with the soluble residual.

`DerivedSeries(G)`

The derived series of the group G. The series is returned as a sequence of subgroups.

`CommutatorSubgroup(G)`
`DerivedSubgroup(G)`
`DerivedGroup(G)`

The derived subgroup of the group G.

`#FittingSubgroup(G)`

The Fitting subgroup of the group G.

`LowerCentralSeries(G)`

The lower central series of the matrix group G. The series is returned as a sequence of subgroups.

`NilpotencyClass(G)`

The nilpotency class of the group G.

`H ^ G`
`NormalClosure(G, H)`

The normal closure of the subgroup H of group G.

`SolubleResidual(G)`
`SolvableResidual(G)`

The solvable residual of the group G.

`SubnormalSeries(G, H)`

Given a group G and a subnormal subgroup H of G, return a sequence of subgroups commencing with G and terminating with H, such that each subgroup is normal in the previous one. If H is not subnormal in G, the empty sequence is returned.

`UpperCentralSeries(G)`

The upper central series of the matrix group G. The series is returned as a sequence of subgroups. As the algorithm used requires the conjugacy classes of G, this function is much more restricted in its range of application than `DerivedSeries` and `LowerCentralSeries`.

Example H59E24

We demonstrate some of the series functions by applying them to a soluble subgroup of $\mathrm{GL}(3,5)$.

```
> G := MatrixGroup< 3, GF(5) | [0,1,0, 1,0,0, 0,0,1],
>                              [0,1,0, 0,0,1, 1,0,0],
>                              [2,0,0, 0,1,0, 0,0,1] >;
> Order(G);
384
> DerivedGroup(G);
MatrixGroup(3, GF(5, 1))
Generators:
[0 0 1]
[1 0 0]
[0 1 0]

[2 0 0]
[0 3 0]
[0 0 1]
> D := DerivedSeries(G);
> [ Order(d) : d in D ];
[ 384, 48, 16, 1 ]
> L := LowerCentralSeries(G);
> [ Order(l) : l in L ];
[ 384, 48 ]
> K := sub< G | [ 2,0,0, 0,3,0, 0,0,2 ] >;
> S := SubnormalSeries(G, K);
> [ Order(s) : s in S ];
[ 384, 16, 4 ]
```

59.11.2 The Soluble Radical and its Quotient

The functions in this section enable the user to construct the radical, its quotient and an elementary abelian series. They are currently restricted to matrix groups where a base and strong generating set can be constructed and the base ring is either a field or can be embedded into a field.

`Radical(G)`
`SolubleRadical(G)`
`SolvableRadical(G)`

Given a group G, return the maximal normal solvable subgroup of G. The algorithm is to compute the radical quotient map, and then compute its kernel. The algorithm used is described in Unger [Ung06b].

`RadicalQuotient(G)`

Given a group G, compute a permutation representation of the quotient G/R where R is the (solvable) radical of G. Both the permutation group Q isomorphic to G/R and a homomorphism $\phi : G \to Q$ are returned. The third return value is R, the radical of G and the kernel of the homomorphism. The algorithm used is described in Unger [Ung06b].

`ElementaryAbelianSeries(G: parameters)`

`LayerSizes` SeqEnum[RngIntElt] *Default :* []

An elementary abelian series is a chain of normal subgroups $R = N_1 > N_2 > ... > N_r = 1$ with the property that the quotient of each pair of successive terms in the series is elementary abelian and that there is no group $R < H < G$ such that H/R is elementary abelian and H normal in G. The top of the series R is called the solvable radical and is the maximal normal solvable subgroup of G.

The parameter `LayerSizes` controls possible refinement of the series. As an example, take `LayerSizes := [ 2, 5, 3, 4, 7, 3, 11, 2, 17, 1]`. When constructing an elementary abelian series for the group, attempt to split 2-layers of size gt 2^5, 3-layers of size gt 3^4, etc. The implied exponent for 13 is 2 and for all primes greater than 17 the exponent is 1. Setting `LayerSizes` to `[2, 1]` will attempt to split all layers, resulting in a portion of a chief series for G.

`ElementaryAbelianSeriesCanonical(G)`

Gives a similar result to using `ElementaryAbelianSeries`, except the series returned depends only on the isomorphism type of the solvable radical, and consists of characteristic subgroups of G. This function may be slower than `ElementaryAbelianSeries`.

59.11.3 Composition and Chief Factors

The functions in this section enable the user to find the composition factors of a matrix group. They are restricted to matrix groups where a base and strong generating set can be constructed. The chief series and factors functions are further restricted to groups where the base ring is either a field or can be embedded into a field.

`CompositionFactors(G)`

Given a matrix group G, return a sequence S of tuples that represent the composition factors of G, ordered according to some composition series of G. Each tuple is a triple of integers f, d, q that defines the isomorphism type of the corresponding composition factor. A triple $< f, d, q >$ describes a simple group as follows. The integer f defines the family to which the group belongs, and d and q are the parameters of the family. The length of the sequence S is the number of composition factors of G. The numbering of the simple group families is given in Tables 1 and 2 of the chapter on permutation groups on page 1590.

`ChiefFactors(G)`

Given a group G, return a sequence of the isomorphism types $< f, d, q, m >$ of the chief factors. An isomorphism type in a chief factor should be understood as the direct product of m copies of the simple group described by $< f, d, q >$ (see `CompositionFactors` above). For the algorithm, see Unger [Ung].

`ChiefSeries(G)`

Given a group G, return the chief series of G and a sequence of the corresponding isomorphism types $< f, d, q, m >$ of the chief factors. An isomorphism type in a chief factor should be understood as the direct product of m copies of the simple group described by $< f, d, q >$ (see `CompositionFactors` above). The series will be organised to contain the soluble radical of G, and, if G is insoluble, the socle of the quotient of G by the soluble radical.

Example H59E25

We get the chief factors of a group of degree 4 defined over the cyclotomic field of order 8.

```
> L<zeta_8> := CyclotomicField(8);
> w := -( - zeta_8^3 - zeta_8^2 + zeta_8);
> // Define sqrt(q)
> rt2 := -1/6*w^3 + 5/6*w;
> // Define sqrt(-1)
> ii := -1/6*w^3 - 1/6*w;
> f := rt2;
> t := f/2 + (f/2)*ii;
> GL4L := GeneralLinearGroup(4, L);
>
> A := GL4L ! [ 1/2, 1/2, 1/2, 1/2,
>             1/2,-1/2, 1/2,-1/2,
```

```
>             1/2, 1/2,-1/2,-1/2,
>             1/2,-1/2,-1/2, 1/2 ];
>
> B := GL4L ! [ 1/f,   0, 1/f,   0,
>               0, 1/f,   0, 1/f,
>             1/f,   0,-1/f,   0,
>               0, 1/f,   0,-1/f ];
>
> g4 := GL4L ! [  1, 0, 0, 0,
>                 0, 1, 0, 0,
>                 0, 0, 1, 0,
>                 0, 0, 0,-1 ];
>
> D1 := GL4L ! [ 1, 0, 0, 0,
>               0,ii, 0, 0,
>               0, 0, 1, 0,
>               0, 0, 0,ii ];
>
> D3 := GL4L ! [ t, 0, 0, 0,
>               0, t, 0, 0,
>               0, 0, t, 0,
>               0, 0, 0, t ];
>
> G3 := sub< GL4L | A, B, g4, D1, D3 >;
> Order(G3);
92160
> ChiefFactors(G3);
    G
    |  Cyclic(2)
    *
    |  Alternating(6)
    *
    |  Cyclic(2) (4 copies)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

59.12 Coset Tables and Transversals

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

`CosetTable(G, H)`

The (right) coset table for the group G over subgroup H relative to its defining generators.

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a matrix group G and a subgroup H of G, this function returns

(a) A set of elements T of G forming a right transversal for G over H; and

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and $g \in G$, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

59.13 Presentations

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.13.1 Presentations

`FPGroup(G)`

Construct a presentation for the matrix group G on the set of defining generators and return the presentation in the form of a finitely presented group F that is isomorphic to G. The presentation is obtained by first computing the regular representation of G and then using the Todd-Coxeter Schreier algorithm to construct a presentation on the strong generators. In this situation the strong generators are identical to the defining generators.

A group homomorphism $\phi : F \to G$, defining G as a matrix representation of F, is also returned.

`FPGroupStrong(G)`

Construct a presentation for the matrix group G on a set of strong generators and return the presentation in the form of a finitely presented group F that is isomorphic to G. In MAGMA, the Todd-Coxeter Schreier algorithm is used to construct the presentation. If strong generators are not already known for G, they will be constructed. In the case in which strong generators are already known for G, the presentation will be on these strong generators.

The presentation will have the property that it contains presentations for all stabilizer subgroups defined by the BSGS.

The group homomorphism $f : F \to G$, defining G as a matrix representation of F, is also returned.

59.13.2 Matrices as Words

Consider a matrix group G defined on d generators. The *word group* of G is a free group W of rank d. Then we regard G as a homomorphic image of F with associated homomorphism $\phi : W \to G$. All operations involving words in the generators of G will be performed in W.

`WordGroup(G)`

Given a matrix group G defined on d generators, return (a) a free group W on d generators as an SLP-group, and (b) the homomorphism ϕ from W to G such that $W.i \to G.i$, for $i = 1, \ldots, d$. The group W associated with G by this function will be referred to as the *word group* for G.

`InverseWordMap(G)`

Given a matrix group G and its associated word group W with canonical homomorphism $\phi : W \to G$, construct the inverse mapping ρ. Thus, given a matrix g of G, $g@\rho$ returns an element in the preimage of g under ϕ. If the word group W does not already exist, it will be created.

59.14 Automorphism Groups

The automorphism group of a finite matrix group may be computed in MAGMA, subject to the same restrictions on the group as when computing maximal subgroups. (That is, all of the non-abelian composition factors of the group must appear in a certain database.) The methods used are those described in Cannon and Holt [CH03]. The existence of an isomorphism between a given matrix group and any other type of finite group (`GrpPerm` or `GrpPC`) may also be determined using similar methods.

`AutomorphismGroup(G: parameters)`

Given a finite matrix group G, construct the full automorphism group F of G. The function returns the full automorphism group of G as a group of mappings (i.e., as a group of type `GrpAuto`). The automorphism group F is also computed as a finitely presented group and can be accessed via the function `FPGroup(F)`. A function `PermutationRepresentation` is provided that when applied to F, attempts to construct a faithful permutation representation of reasonable degree. The algorithm described in Cannon and Holt [CH03] is used.

`SmallOuterAutGroup` RNGINTELT *Default :* 20000

`SmallOuterAutGroup :=` t: Specify the strategy for the backtrack search when testing an automorphism for lifting to the next layer. If the outer automorphism group O at the previous level has order at most t, then the regular representation of O is used, otherwise the program tries to find a smaller degree permutation representation of O.

`Print` RNGINTELT *Default :* 0

The level of verbose printing. The possible values are 0, 1, 2 or 3.

`PrintSearchCount` RNGINTELT *Default :* 1000

`PrintSearchCount := s`: If `Print := 3`, then a message is printed at each s-th iteration during the backtrack search for lifting automorphisms.

Further information about the construction of the automorphism group and a description of machinery for computing with group automorphisms may be found in Chapter 67.

Example H59E26

We construct a 3-dimensional matrix group over $GF(4)$ and determine the order of its automorphism group.

```
> k<w> := GF(4);
> G := MatrixGroup< 3, k |
> [w^2, 0, 0, 0, w^2, 0, 0, 0, w^2],
> [w^2, 0, w^2, 0, w^2, w^2, 0, 0, w^2],
> [1, 0, 0, 1, 0, w, w^2, w^2, 0],
> [w, 0, 0, w^2, 1, w^2, w, w, 0],
> [w, 0, 0, 0, w, 0, 0, 0, w] >;
> G;
MatrixGroup(3, GF(2^2))
Generators:
    [w^2   0   0]
    [  0 w^2   0]
    [  0   0 w^2]

    [w^2   0 w^2]
    [  0 w^2 w^2]
    [  0   0 w^2]

    [  1   0   0]
    [  1   0   w]
    [w^2 w^2   0]

    [  w   0   0]
    [w^2   1 w^2]
    [  w   w   0]

    [  w   0   0]
    [  0   w   0]
    [  0   0   w]
> #G;
576
> A := AutomorphismGroup(G);
> #A;
3456
> OuterOrder(A);
72
> F := FPGroup(A);
```

```
> P := DegreeReduction(CosetImage(F, sub<F|>));
> P;
Permutation group P acting on a set of cardinality 48
```

Thus, we see that G has an automorphism group of order 3456 and the quotient group of A consisting of outer automorphisms, has order 72. The automorphism group may be realised as a permutation group of degree 48.

`IsIsomorphic(G, H: parameters)`

Test whether or not the two finite groups G and H are isomorphic as abstract groups. If so, both the result `true` and an isomorphism from G to H is returned. If not, the result `false` is returned. The algorithm described in Cannon and Holt [CH03] is used.

`SmallOuterAutGroup` RngIntElt *Default :* 20000

`SmallOuterAutGroup :=` t: Specify the strategy for the backtrack search when testing an automorphism for lifting to the next layer. If the outer automorphism group O at the previous level has order at most t, then the regular representation of O is used, otherwise the program tries to find a smaller degree permutation representation of O.

`Print` RngIntElt *Default :* 0

The level of verbose printing. The possible values are 0, 1, 2 or 3.

`PrintSearchCount` RngIntElt *Default :* 1000

`PrintSearchCount :=` s: If `Print := 3`, then a message is printed at each s-th iteration during the backtrack search for lifting automorphisms.

Example H59E27

We construct a 3-dimensional point group of order 8 and test it for isomorphism with the dihedral group of order 8 given as a permutation group.

```
> n := 4;
> N := 4*n;
> K<z> := CyclotomicField(N);
> zz := z^4;
> i := z^n;
> cos := (zz+ComplexConjugate(zz))/2;
> sin := (zz-ComplexConjugate(zz))/(2*i);
> GL := GeneralLinearGroup(3, K);
> G := sub< GL | [ cos, sin, 0,
>                 -sin, cos, 0,
>                    0,   0, 1 ],
>
>                [  -1,   0, 0,
>                    0,   1, 0,
>                    0,   0, 1 ] >;
```

```
>
>  #G;
8
> D8 := DihedralGroup(4);
> D8;
Permutation group G acting on a set of cardinality 4
Order = 8 = 2^3
    (1, 2, 3, 4)
    (1, 4)(2, 3)
> #D8;
8
> bool, iso := IsIsomorphic(G, D8);
> bool;
true
> iso;
Homomorphism of MatrixGroup(3, K) of order 2^3 into
GrpPerm: D8, Degree 4, Order 2^3 induced by
    [ 0  1  0]
    [-1  0  0]
    [ 0  0  1] |--> (1, 2, 3, 4)

    [-1  0  0]
    [ 0  1  0]
    [ 0  0  1] |--> (1, 3)
```

59.15 Representation Theory

A set of functions are provided for computing with the characters of a group. Full details of these functions may be found in Chapter 91. For convenience we include here two of the more useful character functions.

Also, functions are provided for computing with the modular representations of a group. Full details of these functions may be found in Chapter 89. For the reader's convenience we include here the functions which may be used to define a $K[G]$-module for a matrix group.

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

`LinearCharacters(G)`

A sequence containing the linear characters for the group G.

`CharacterTable(G: parameters)`

Construct the table of ordinary irreducible characters for the group G.

`Al` MonStgElt *Default :* `"Default"`

This parameter controls the algorithm used. The string `"DS"` forces use of the Dixon-Schneider algorithm. The string `"IR"` forces the use of Unger's induction/reduction algorithm [Ung06]. The `"Default"` algorithm is to use Dixon-Schneider for groups of order ≤ 5000 and Unger's algorithm for larger groups. This may change in future.

`DSSizeLimit` RngIntElt *Default :* 0

When the default algorithm is selected, a positive value n for `DSSizeLimit` means that before using Unger's algorithm, the full character space is split by some passes of Dixon-Schneider, restricted to using class matrices corresponding to conjugacy classes with size at most n.

`PermutationCharacter(G, H)`

Given a group G and a subgroup H of G, construct the ordinary character afforded by the representation of G given by its action on the coset space of the subgroup H.

`GModule(G)`

The natural $R[G]$-module for the matrix group G.

`GModule(G, A)`

Let A be a matrix ring defined over the ring R and let G be a finite group defined on m generators. Let M denote the underlying module of A. Suppose there is a one-to-one correspondence between the generators of G and the generators $[A_1, \ldots, A_m]$ of A. The function `GModule` creates the $R[G]$-module corresponding to an action of G on M defined by A, where the action of the i-th generator of G on M is given by A_i.

`GModule(G, Q)`

Let A be a matrix ring defined over the ring R and let G be a finite group defined on m generators. Let M denote the underlying module of A. Given a sequence Q of m elements of A, the function `GModule` creates the $R[G]$-module corresponding to an action of G on M defined by Q, where the action of the i-th generator of G on M is given by $Q[i]$.

`GModule(G, A, B)`

Let A and B be normal subgroups of G such that B is contained in A. Further, assume that A/B is elementary abelian of order p^n, p a prime. Let K denote the field of p elements. This function constructs a $K[G]$-module corresponding to the action of the group G on the elementary abelian section A/B of G. The map from A to the $K[G]$-module's underlying vector space is also returned.

`PermutationModule(G, H, R)`

The permutation module for the matrix group G over the ring R defined by its action on the cosets of the subgroup H.

`ChangeOfBasisMatrix(G, S)`

Given a matrix group G and a submodule S of its natural module, return an invertible matrix with topmost rows a basis for S. Conjugating by the inverse of this matrix puts the generators of G into a block form that exhibits their action on S and the quotient module.

Example H59E28

We use the module machinery to refine an elementary abelian normal subgroup by finding a normal subgroup contained in it.

```
> G := MatrixGroup<4, IntegerRing(4) |
>  [ 3, 3, 1, 3, 0, 2, 2, 3, 3, 0, 1, 3, 3, 2, 2, 1 ],
>  [ 2, 2, 3, 3, 0, 3, 1, 1, 3, 0, 1, 1, 2, 0, 1, 2 ] >;
> #G;
660602880
> H := pCore(G, 2);
> FactoredOrder(H);
[ <2, 15> ]
> IsElementaryAbelian(H);
true
> M, f := GModule(G, H, sub<H|>);
> SM := Submodules(M);
> #SM;
3
```

One of these submodules is 0, one is all `M`, we are interested in the one in the middle. Note that the result returned by `Submodules` is sorted by dimension.

```
> N := SM[2] @@ f;
> N;
MatrixGroup(4, IntegerRing(4))
Generators:
  [3 0 0 0]
  [0 3 0 0]
  [0 0 3 0]
  [0 0 0 3]
```

We have found `N`, a normal subgroup of `G`, contained in the 2-core, with order 2.

59.16 Base and Strong Generating Set

59.16.1 Introduction

Computing structural information for a matrix group G requires, in most cases, a representation of the set of elements of G. MAGMA represents this set by means of a *base and strong generating set, or BSGS* for G. Suppose the group G has the natural module M. A *base* B for G is a sequence of distinct elements and submodules of M with the property that the identity is the only element of G that fixes B pointwise. A base B of length n determines a sequence of subgroups $G^{(i)}$, $1 \leq i \leq n+1$, where $G^{(i)}$ is the stabilizer of the first $i-1$ points of B. Given a base B for G, a subset S of G is said to be a *strong generating set* for G if $G^{(i)} = \langle S \cap G^{(i)} \rangle$, for $i = 1, \ldots, n$.

Unlike permutation groups, however, the orbits of the i-th base point under the stabilizer $G^{(i)}$ are not bounded by the degree, but rather, by (where the base point is a 1-dimensional subspace) $(q^n - 1)/(q - 1)$ where q is the cardinality of the coefficient field and n is the degree of G. Clearly, it is essential to find small orbits if one is to compute with matrix groups in this manner. Unfortunately, there are no methods which are guaranteed to find short orbits. There are, however, some heuristics developed by Scott Murray and Eamonn O'Brien which often find good base points. These heuristics are used in MAGMA if the most likely standard base point would generate an orbit longer than 10000 (this bound may be changed).

59.16.2 Controlling Selection of a Base

Given the difficulties in automatically finding a good base for a matrix group, it is possible to apply the Murray-O'Brien base point selection procedure and preset a suitable base manually.

`GoodBasePoints(G: parameters)`

Apply the Murray–O'Brien base point selection procedure and return a sequence of vectors or subspaces according to the parameters. The procedure computes and sorts a collection of eigenspaces $[V_1, \ldots, V_m]$ for a generating set for the matrix group G. The default action is then to return $[V_1.1, \ldots, V_m.1, V_1.2, \ldots]$ where each new vector is only added if it is not in the span of the preceding vectors.

`Slots` RNGINTELT *Default :* 10

Expand the number of generators to work with to `Slots` matrices by adding random words in the generators of G.

`NoCycle` RNGINTELT *Default :* `false`

If `NoCycle := true`, instead of cycling through the eigenspaces, return the sequence $[V_1.1, \ldots, V_1.(\dim V_1), V_2.1, \ldots]$, with the addition of each vector subject to the same condition above.

`Eigenspaces` RNGINTELT *Default :* `false`

If `Eigenspaces := true`, then return the subsequence of the eigenspaces where all the eigenspaces have dimension $d \leq 10$. If there are no such eigenspaces, all the eigenspaces are returned.

```
AssertAttribute(G, "Base", B)
```

Set the base of the matrix group G to be $[B[1], \ldots, B[n]]$ where the tuple B has n components. An error will be reported if the matrix group G already has a base set.

```
HasAttribute(G, "Base")
```

Return whether the matrix group G has a base set, and if so, the base.

```
AssertAttribute(GrpMat, "FirstBasicOrbitBound", n)
```

Set the limit for the size of the first basic orbit to be n. If n is non-zero and the orbit of the first base point (a 1-dimensional subspace generated by a standard basis vector) has length exceeding n, then the Murray-O'Brien base point selection procedure is used to find a point more likely to have a short orbit. This assertion will affect all matrix groups. If $n = 1$ then use of the Murray-O'Brien procedure is guaranteed.

```
HasAttribute(GrpMat, "FirstBasicOrbitBound")
```

Get the limit for the size of the first basic orbit. This will always return `true` and the limit.

59.16.3 Construction of a Base and Strong Generating Set

The functions described below give user control of the construction of a base and strong generating set (BSGS) of a finite matrix group.

Many functions described in this chapter require a group to have a BSGS. In case the given group does not have a BSGS, then one will be constructed using the default algorithm, which is equivalent to using the `BSGS` procedure described below.

It should be noted that if the user constructs a BSGS for a group G using the `RandomSchreier` procedure, then other functions that require a BSGS will assume that the random BSGS is a complete BSGS. If this is not the case then results will be unpredictable.

```
BSGS(G)
```

```
BSGS(G, str)
```

The general procedure for constructing a base and strong generating set for the matrix group G. This version uses the default algorithm choices. Currently this is as follows: if the order of the group is known to the program then a BSGS is constructed using the random Schreier algorithm, if not then the Sims-Todd-Coxeter-Schreier procedure is used. If `str` is the name of a sporadic group, we assume that G is a representation for this group and choose base points specific to this group. This should ensure better performance. Information on the progress of these algorithms may be obtained by setting the verbose flags `RandomSchreier` and `STCS` true.

`RandomSchreier(G: parameters)`

`RandomSchreier(G, str : parameters)`

`Run` RngIntElt *Default :* 40

Construct a probable base and strong generating set for the group G. The strong generators are constructed from a set of randomly chosen elements of G. The expectation is that if sufficiently many random elements are taken then, upon termination, the algorithm will have produced a BSGS for G. If the attribute Order is defined for G, the random Schreier will continue until a BSGS defining a group of the indicated order is obtained. In such circumstances this method is the fastest method of constructing a base and strong generating set for G. This is particularly so for groups of large degree. If nothing is known about G, the random Schreier algorithm provides a cheap way of obtaining lower bounds on the group's order. This procedure has one associated parameter `Run`, which takes a positive integer value. If the value of `Run` is n, then the algorithm terminates after n consecutive random elements are found to lie in the set defined by the current BSGS (default 40). This will happen even if the Order attribute is defined for G. It should be emphasized that unpredictable results may arise if the user uses the base and strong generators produced by this algorithm, when, in fact, it does not constitute a complete BSGS for G. The `Verify` procedure, described below, may be used to check the completeness of the BSGS constructed by this function.

If `str` is the name of a sporadic group, we assume that G is a representation for this group and choose base points specific to this group. This should ensure better performance.

Information on the progress of this algorithm may be obtained by setting the verbose flag `RandomSchreier` to true.

`ToddCoxeterSchreier(G)`

Construct a BSGS for the matrix group G using the Sims-Todd-Coxeter-Schreier algorithm. Information on the progress of this algorithm may be obtained by setting the verbose flag `STCS` to true.

`Verify(G)`

Given a matrix group G for which a possible BSGS is stored, verify the correctness of the BSGS. If it is not complete, proceed to complete it. The Sims-Todd-Coxeter-Schreier method is used.

If G has no BSGS stored, then use of `Verify` is equivalent to using the `BSGS` procedure described above.

Information on the progress of these algorithms may be obtained by setting the verbose flags `RandomSchreier` and `STCS` true.

59.16.4 Defining Values for Attributes

```
AssertAttribute(G, "Order", n)
```
```
AssertAttribute(G, "Order", Q)
```

Define the order of the matrix group G to be the integer n (factored integer Q).

```
AssertAttribute(G, "IsVerified", b)
```

If the boolean variable b is `true`, the existing pseudo strong generators for the matrix group G (possibly created by `RandomSchreier`) are to be taken as correct.

```
HasAttribute(G, "Order")
```
```
HasAttribute(G, "FactoredOrder")
```

Returns `true` iff the order of the group G is known. In that case, the order is also returned as the second value of the function.

```
HasAttribute(G, "IsVerified")
```

Returns `true` iff the matrix group G has a verified set of strong generators.

59.16.5 Accessing the Base and Strong Generating Set

```
Base(G)
```

A base for the matrix group G. The base is returned as a sequence of points of Ω. If a base is not known, one will be constructed.

```
BasePoint(G, i)
```

The i-th base point for the matrix group G. A base and strong generating set must be known for G.

```
BasicOrbit(G, i)
```

The basic orbit at level i as defined by the current base for the matrix group G. This function assumes that a BSGS is known for G.

```
BasicOrbitLength(G, i)
```

The length of the basic orbit at level i as defined by the current base for the matrix group G. This function assumes that a BSGS is known for G.

```
BasicOrbitLengths(G)
```

The lengths of the basic orbits as defined by the current base for the matrix group G. This function assumes that a BSGS is known for G. The lengths are returned as a sequence of integers.

`BasicStabilizer(G, i)`

`BasicStabiliser(G, i)`

Given a matrix group G for which a base and strong generating set are known, and an integer i, where $1 \leq i \leq k$ with k the length of the base, return the subgroup of G which fixes the first $i-1$ points of the base.

`BasicStabilizerChain(G)`

`BasicStabiliserChain(G)`

Given a matrix group G, return the stabilizer chain defined by the base as a sequence of subgroups of G. If a BSGS is not already known for G, it will be created.

`NumberOfStrongGenerators(G)`

`Nsgens(G)`

The number of elements in the current strong generating set for the matrix group G.

`StrongGenerators(G)`

A set of strong generators for the matrix group G. If they are not currently available, they will be computed.

59.17 Soluble Matrix Groups

The functions described in this section apply only to finite groups for which a base and strong generating set may be constructed.

59.17.1 Conversion to a PC-Group

`PolycyclicGenerators(G)`

Construct a polycyclic generating sequence for the soluble group G.

`PCGroup(G)`

Given a soluble group G, construct a group S in category `GrpPC`, isomorphic to G. In addition to returning S, the function returns an isomorphism $\phi : G \to S$.

59.17.2 Soluble Group Functions

`pCentralSeries(G, p)`

Given a soluble group G, and a prime p dividing $|G|$, return the lower p-central series for G. The series is returned as a sequence of subgroups.

59.17.3 p-group Functions

`IsSpecial(G)`

Given a p-group G, return `true` if G is special, `false` otherwise.

`IsExtraSpecial(G)`

Given a p-group G, return `true` if G is extraspecial, `false` otherwise.

`FrattiniSubgroup(G)`

Given a p-group G, return the Frattini subgroup.

`JenningsSeries(G)`

Given a p-group G, return the Jennings series for G. The series is returned as a sequence of subgroups.

59.17.4 Abelian Group Functions

`AbelianInvariants(G)`

`Invariants(G)`

Given an abelian group G, return a sequence Q containing the types of each p-primary component of G.

59.18 Bibliography

[**But76**] Gregory Butler. The Schreier Algorithm for Matrix Groups. In *Proceedings of SYMSAC '76*, pages 167–170, 1976.

[**CCH01**] J.J. Cannon, B. Cox, and D.F. Holt. Computing the subgroups of a permutation group. *J. Symb. Comp.*, 31:149–161, 2001.

[**CH03**] J.J. Cannon and D.F. Holt. Automorphism group computation and isomorphism testing in finite groups. *J. Symbolic Comp.*, 35(3):241–267, 2003.

[**CLG97**] Frank Celler and Charles R. Leedham-Green. Calculating the Order of an Invertible Matrix. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 55–60. AMS, 1997.

[**CLGM+95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

[**ELGO02**] Bettina Eick, C.R. Leedham-Green, and E.A. O'Brien. Constructing automorphism groups of a p-groups. *Comm. Algebra*, 30:2271–2295, 2002.

[**KP02**] J. Kuzmanovich and A. Pavlichenkov. Finite groups of matrices whose entries are integers. *Amer. Math. Monthly*, 109(2):173–186, 2002.

[**LGO02**] C.R. Leedham-Green and E.A. O'Brien. Recognising tensor-induced matrix groups. *J. Algebra*, 253:14–30, 2002.

[**LGPS91**] C.R. Leedham-Green, C.E. Praeger, and L.H. Soicher. Computing with group homomorphisms. *J. Symbolic Comp.*, 12(4/5):527–532, 1991.

[**MN89**] M. Mecky and J. Neubüser. Some remarks on the computation of conjugacy classes of soluble groups. *Bull. Austral, Math. Soc.*, 40(2):281–292, 1989.

[**MO95**] Scott H. Murray and E. A. O'Brien. Selecting base points for the Schreier-Sims algorithm for matrix groups. *J. Symbolic Comp.*, 6:577–584, 1995.

[**O'B90**] E.A. O'Brien. The p-group generation algorithm. *J. Symbolic Comput.*, 9:677–698, 1990.

[**RD04**] Colva M. Roney-Dougal. Conjugacy of subgroups of the general linear group. *Experiment. Math.*, 13:151–163, 2004.

[**Sch00**] Ruth Schwingel. *Two matrix group algorithms with applications to computing the automorphism group of a finite p-group.* PhD thesis, Queen Mary and Westfield College, University of London, 2000.

[**Ung**] W.R. Unger. Computing chief series of a large permutation group. In preparation.

[**Ung06a**] W.R. Unger. Computing the character table of a finite group. *J. Symbolic Comp.*, 41(8):847–862, 2006.

[**Ung06b**] W.R. Unger. Computing the solvable radical of a permutation group. *J. Algebra*, 300(1):305–315, 2006.

60 MATRIX GROUPS OVER FINITE FIELDS

Chapter 60
MATRIX GROUPS OVER FINITE FIELDS

60.1 Introduction

If a matrix group G is defined over a finite field then, provided that the group is not too large, we can construct a BSGS-representation for G and consequently apply the standard algorithms for group structure as described in Chapter 59. However, there are many examples of groups having moderately small dimension where we cannot find a BSGS-representation.

In this chapter we describe techniques for computing with matrix groups that do not assume that a BSGS-representation is available. Thus, the techniques described here apply to matrix groups possibly having much larger order or much larger dimension than those that can be handled with the techniques of Chapter 59.

The `CompositionTree` package introduced in Section 60.6, which includes the collection of LMG (large matrix group) functions described in Section 60.7, provides a framework for such investigations. The package was prepared by Henrik Bäärnhielm, Derek Holt, C.R. Leedham-Green and E.A. O'Brien, and includes code prepared by Peter Brooksbank, Elliot Costi, Heiko Dietrich, Alice Niemeyer, and Csaba Schneider.

For recent surveys of work in this area, we refer the reader to [O'B06, O'B11].

The techniques described in this chapter fall roughly into two categories.

(a) Functions based on Aschbacher's theorem classifying maximal subgroups of the general linear group. The main thrust of this work is to devise a framework for computing arbitrary structural information for a matrix group without the use of a BSGS-representation.

(b) Functions which employ Monte Carlo and Las Vegas algorithms to determine some property of the group.

60.2 Finding Elements with Prescribed Properties

`RandomElementOfOrder(G, n : parameters)`

`Central`	BoolElt	*Default :* `false`
`Proof`	BoolElt	*Default :* `true`
`Randomiser`	GrpRandProc	*Default :*
`MaxTries`	RngIntElt	*Default :* 100

Given a finite matrix group G, this intrinsic attempts to locate an element x of order n in G by random search. If such an element is found, then the return values are the boolean value `true`, the element x, and an SLP for this element.

If `Central` is `true`, then an element is sought which has order n modulo the centre of G. If `Proof` is `false`, then the element returned may have order a multiple of n. In either case, the final return value indicates whether the element returned is known to have the precise order. The parameter `MaxTries` specifies the maximum number of random elements that are chosen. The parameter `Randomiser` specifies the random process that is to be used to construct the element and the SLP returned for the element is in the word group associated with this process. The default value of `Randomiser` is the process `RandomProcessWithWords(G)`.

`RandomElementOfNormalClosure(G, N)`

Given a group G and a subgroup N of G, this intrinsic returns a random element of the normal closure of N in G. Note that G may be a permutation or matrix group. The algorithm is due to Leedham-Green and O'Brien [LGO02].

`InvolutionClassicalGroupEven(G : parameters)`

`SmallCorank`	BoolElt	*Default :* `false`
`Case`	MonStgElt	*Default :* "*unknown*"

Let G be a quasisimple classical group in its natural representation and in even characteristic. If G is of type Ω^+ or Ω^- then it must have even degree at least 4 and be defined over a field with at least 4 elements. The corank of an involution I is the rank of $I - \mathtt{Identity}(G)$. This function returns an involution I of corank in $[d/4, \ldots, d/2]$, the SLP for I in `WordGroup`(G), and the corank of the involution. The parameter `Case` should be one of `"SL"`, `"Sp"`, `"SU"`, `"Omega-"`, or `"Omega+"`. If `SmallCorank` is `true`, then accept involution of small corank. The algorithm used to construct the involution is described in [DLLGO13]; it was implemented by Heiko Dietrich.

60.3 Monte Carlo Algorithms for Subgroups

`CentraliserOfInvolution(G, g : parameters)`

`Central`	BoolElt	*Default :* `false`
`NumberRandom`	RngIntElt	*Default :* 100
`CompletionCheck`	UserProgram	*Default :*

Given an involution g in G, this function returns the centraliser C of g in G using an algorithm of John Bray [Bra00]. Since it is Monte Carlo, it may return only a subgroup of the centraliser. If `Central` is `true`, the projective centraliser of g will be constructed: its elements commute with g modulo the centre of G.

The optional argument `CompletionCheck` is a function which can be used to determine when we have constructed the centraliser. It takes the following arguments: the parent group G; the proposed centraliser C; the involution g. By default, the algorithm completes when 20 generators have been found for the centraliser or when `NumberRandom` elements have been considered.

`CentraliserOfInvolution(G, g, w : parameters)`

`Randomiser`	GrpRandProc	*Default :*
`Central`	BoolElt	*Default :* `false`
`NumberRandom`	RngIntElt	*Default :* 100
`CompletionCheck`	UserProgram	*Default :*

Given an involution g in a matrix group G together with a SLP w corresponding to g, this function returns the centraliser C of g in G and SLPs for the generators of C. The algorithm used is due to John Bray [Bra00]. Since it is Monte Carlo, it may return a proper subgroup of the centraliser. If `Central` is `true`, the projective centraliser of g is constructed: its elements commute with g modulo the centre of G.

The parameter `Randomiser` specifies the random process that is to be used to construct the centraliser. By default `Randomiser` is the value returned by `RandomProcessWithWords (G)`. The SLP for g must lie in the word group of this process. The optional argument `CompletionCheck` is a function which can be used to determine when we have constructed the centraliser. It takes four arguments: the parent group G; the proposed centraliser C; the involution g; and the list of the SLPs for the generators of C. By default, the algorithm completes when 20 generators have been found for the centraliser or when `NumberRandom` elements have been considered.

`AreInvolutionsConjugate(G, x, wx, y, wy : parameters)`

`Randomiser`	GrpRandProc	*Default :*
`MaxTries`	RngIntElt	*Default :* 100

This Monte Carlo algorithm attempts to construct an element c of the group G which conjugates the involution x to the involution y. The corresponding SLPs for x and y are wx and wy respectively. If such an element c is found, then three values are returned: `true`, c and the SLP for c. Otherwise, the boolean value `false` is returned. At most `MaxTries` random elements are considered.

The parameter `Randomiser` specifies the random process to be used. By default `Randomiser` is the value returned by `RandomProcessWithWords (G)`. The SLPs for x and y must lie in the word group of this process and the SLP for c will also lie in this word group.

`NormalClosureMonteCarlo(G, H )`

`NormalClosureMonteCarlo(G, H : parameters)`

`slpsH`	[]	*Default :* []
`ErrorProb`	FltRatElt	*Default :* 9/10
`SubgroupChainLength`	RngIntElt	*Default :* `Degree(H)`

This Monte Carlo algorithm constructs the normal closure N of H in G. If SLPs of the generators of H in the generators of G are supplied via the parameter

slpsH, then the function also returns SLPs for the generators of N. The parameter SubgroupChainLength is used to specify an upper bound on the length of any subgroup chain in H. The probability that N is a proper subgroup of the normal closure is bounded above by ErrorProb, assuming that SubgroupChainLength is correctly set.

DerivedGroupMonteCarlo(G : *parameters*)

Randomiser	GrpRandProc	*Default :*
NumberGenerators	RngIntElt	*Default :* 10
MaxGenerators	RngIntElt	*Default :* 100

Given a matrix group G defined over a finite field, this intrinsic returns the derived group of G, and a list of SLPs of its generators in the generators of G. The SLPs are elements of the word group of the random process. The algorithm is Monte Carlo and may return a proper subgroup of the derived group. The parameter Randomiser specifies the random process to be used. By default Randomiser is the value returned by RandomProcessWithWords (G). At least NumberGenerators and at most MaxGenerators will be constructed for the derived group.

IsProbablyPerfect(G : *parameters*)

NumberRandom	RngIntElt	*Default :* 100

This intrinsic attempts to prove that a matrix or permutation group G is perfect by establishing that its generators are in G'. Since it is Monte-Carlo, there is a small probability of error. If the function returns true, then G is perfect; if it returns false, then G might still be perfect. Each call considers NumberRandom random elements.

The algorithm is due to Leedham-Green and O'Brien [LGO02] and it uses NormalSubgroupRandomElement.

Example H60E1

We illustrate IsProbablyPerfect with a subgroup of $GU(4, 9)$.

```
> G := GU(4, 9);
> N := sub<G | (G.1, G.2)>;
```

The generators of N have been chosen to be a normal generating set for the derived group of G.

```
> IsProbablyPerfect(N);
false
> x := NormalSubgroupRandomElement(G, N);
> x;
[$.1^68 $.1^34 $.1^26 $.1^55]
[$.1^23 $.1^78 $.1^16 $.1^72]
[$.1^42  $.1^2 $.1^24      2]
[$.1^11 $.1^66 $.1^13 $.1^29]
>  L := sub< G | N, x>;
```

```
> IsProbablyPerfect(L);
true
```

We now consider $SO(7,5)$ and $\Omega(7,5)$.

```
> G := SO(7, 5);
> IsProbablyPerfect(G);
false
> G := Omega(7, 5);
> IsProbablyPerfect(G);
true
```

60.4 Aschbacher Reduction

60.4.1 Introduction

An on-going international research project seeks to develop algorithms to explore the structure of groups having either large order or large degree. The approach relies on the following theorem of Aschbacher [Asc84]:

A matrix group G acting on the finite dimensional $K[G]$-module V over a finite field K satisfies at least one of the following conditions (which we have simplified slightly for brevity):

(i) G acts reducibly on V;

(ii) G acts semilinearly over an extension field of K;

(iii) G acts imprimitively on V;

(iv) G preserves a nontrivial tensor-product decomposition of V;

(v) G has a normal subgroup N, acting absolutely irreducibly on V, which is an extraspecial p-group or 2-group of symplectic type;

(vi) G preserves a tensor-induced decomposition of V;

(vii) G acts (modulo scalars) linearly over a proper subfield of K;

(viii) G contains a classical group in its natural action over K;

(ix) G is almost simple modulo scalars.

The philosophy underpinning the research program is to attempt to decide that G lies in at least one of the above categories, and to calculate the associated isomorphism or decomposition explicitly.

Groups in Category (i) can be recognised easily by means of the Meataxe functions described in the chapter on R-modules.

Groups which act irreducibly but not absolutely irreducibly on V fall theoretically into Category (ii), and furthermore act linearly over an extension field of K. In fact, absolute irreducibility can be tested using the built-in MAGMA functions and, by redefining their

field to be an extension field L of K and reducing, they can be rewritten as absolutely irreducible groups of smaller dimension, but over L instead of K. We can therefore concentrate on absolutely irreducible matrix groups.

The `CompositionTree` package currently includes functions which seek to decide membership in all categories.

60.4.2 Primitivity

Let G be a subgroup of $\mathrm{GL}(d, q)$ and assume that G acts irreducibly on the underlying vector space V. Then G acts *imprimitively* on V if there is a non-trivial direct sum decomposition

$$V = V_1 \oplus V_2 \oplus \ldots \oplus V_r$$

where $V_1, \ldots, V_r$ are permuted by G. In such a case, each block V_i has the same dimension or size, and we have the *block system* $\{V_1, \ldots, V_r\}$. If no such system exists, then G is *primitive*.

Theoretical details of the algorithm used may be found in Holt, Leedham-Green, O'Brien, and Rees [HLGOR96b].

`SetVerbose ("Smash", 1)` will provide information on the progress of the algorithm.

`IsPrimitive(G:` *parameters*`)`

`BlockSizes` [RngIntElt] *Default :* []

Given a matrix group G defined over a finite field, this intrinsic returns `true` if G is primitive, `false` if G is not primitive, or `"unknown"` if no decision can be reached.

If `BlockSizes` is supplied, then the search is restricted to systems of imprimitivity whose block sizes are given in the sequence `BlockSizes` only. Otherwise all valid sizes will be considered.

`ImprimitiveBasis(G)`

Given a matrix group G defined over a finite field which is imprimitive, this intrinsic returns the change-of-basis matrix which exhibits the block structure for G.

`Blocks(G)`

Given a matrix group G defined over a finite field which is imprimitive, this intrinsic returns the blocks of imprimitivity of G.

`BlocksImage(G)`

Given a matrix group G defined over a finite field which is imprimitive, this intrinsic returns the group induced by the action of G on the system of imprimitivity.

`ImprimitiveAction(G, g)`

Given a matrix group G defined over a finite field which is imprimitive and an element g of G, this intrinsic returns action of g on blocks of imprimitivity as a permutation.

Example H60E2

We construct an imprimitive group by taking the wreath product of $GL(4, 7)$ with S_3.

```
> MG := GL (4, 7);
> PG := Sym (3);
> G := WreathProduct (MG, PG);
>
> IsPrimitive (G);
false
```

We investigate the block system for G.

```
> B := Blocks (G);
> B;
> #B;
4
> B[1];
Vector space of degree 12, dimension 4 over GF(7)
Generators:
(0 0 0 0 1 0 0 0 0 0 0 0)
(0 0 0 0 0 1 0 0 0 0 0 0)
(0 0 0 0 0 0 1 0 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0 0)
Echelonized basis:
(0 0 0 0 1 0 0 0 0 0 0 0)
(0 0 0 0 0 1 0 0 0 0 0 0)
(0 0 0 0 0 0 1 0 0 0 0 0)
(0 0 0 0 0 0 0 1 0 0 0 0)
```

Now we obtain a permutation representation of G in its action on the blocks.

```
> P := BlocksImage (G);
> P;
Permutation group P acting on a set of cardinality 3
    (1, 2, 3)
    (2, 3)
> g := G.4 * G.3;
> ImprimitiveAction (G, g);
(1, 2)
```

60.4.3 Semilinearity

Let G be a subgroup of $\mathrm{GL}(d, q)$ and assume that G acts absolutely irreducibly on the underlying vector space V. Assume that a normal subgroup N of G embeds in $\mathrm{GL}(d/e, q^e)$, for $e > 1$, and a $d \times d$ matrix C acts as multiplication by a scalar λ (a field generator of $\mathbf{F}_{q^e}$) for that embedding.

We say that G acts as a *semilinear* group of automorphisms on the d/e-dimensional space if and only if, for each generator g of G, there is an integer $i = i(g)$ such that $Cg = gC^i$, that is, g corresponds to the field automorphism $\lambda \to \lambda^i$. If so, we have a map from G to the (cyclic) group $\mathrm{Aut}(GF(q^e))$, and C centralises the kernel of this map, which thus lies in $\mathrm{GL}(d, q^e)$.

Theoretical details of the algorithm used may be found in Holt, Leedham-Green, O'Brien and Rees [HLGOR96a].

`SetVerbose ("SemiLinear", 1)` will provide information on the progress of the algorithm.

`IsSemiLinear(G)`

Given a matrix group G defined over a finite field, this intrinsic returns `true` if G is semilinear, `false` if G is not semilinear, or `"unknown"` if no decision can be reached.

`DegreeOfFieldExtension(G)`

Let G be a subgroup of $K = \mathrm{GL}(d, q)$. The intrinsic returns the degree e of the extension field of F_q over which G is semilinear.

`CentralisingMatrix(G)`

Let G be a semilinear subgroup of $K = \mathrm{GL}(d, q)$. The intrinsic returns the matrix C which centralises the normal subgroup of G which acts linearly over the extension field of F_q.

`FrobeniusAutomorphisms(G)`

Let G be a semilinear subgroup of $K = \mathrm{GL}(d, q)$ and let C be the corresponding centralising matrix. The intrinsic returns a sequence S of positive integers, one for each generator g_i of G. The element $S[i]$ is the least positive integer such that $g_i^{-1} C g_i = C^{S[i]}$.

`WriteOverLargerField(G)`

Let G be a semilinear subgroup of $\mathrm{GL}(d, q)$ with extension degree e. This intrinsic returns:

(i) The normal subgroup N of the matrix group G which is the kernel of the map from G to C_e; this subgroup acts linearly over the extension field of K and is precisely the centraliser of C in G.

(ii) A cyclic group E of order e which is isomorphic to G/N.

(iii) A sequence of images of the generators of G in E.

Example H60E3

We analyse a semilinear group.

```
> P := GL(6,3);
> g1 := P![0,1,0,0,0,0,-1,0,0,0,0,0,
>          0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1];
> g2 := P![-1,0,0,0,1,0,0,-1,0,0,0,1,
>          -1,0,0,0,0,0,0,-1,0,0,0,0,0,0,-1,0,0,0,0,0,0,-1,0,0];
> g3 := P![1,0,0,0,0,0,0,-1,0,0,0,0,
>          0,0,1,0,0,0,0,0,0,-1,0,0,0,0,0,0,1,0,0,0,0,0,0,-1];
> G := sub <P | g1, g2, g3 >;
>
> IsSemiLinear (G);
true
> DegreeOfFieldExtension (G);
2
> CentralisingMatrix (G);
[2 2 0 0 0 0]
[1 2 0 0 0 0]
[0 0 2 2 0 0]
[0 0 1 2 0 0]
[0 0 0 0 2 2]
[0 0 0 0 1 2]
> FrobeniusAutomorphisms (G);
[ 1, 1, 3 ]
> K, E, phi := WriteOverLargerField (G);
```

The group K is the kernel of the homomorphism from G into E.

```
> K.1;
[0 1 0 0 0 0]
[2 0 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
```

The return value E is the cyclic group of order e while *phi* gives the sequence of images of $G.i$ in E.

```
> E;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*E.1 = 0
>
> phi;
[ 0, 0, E.1 ]
```

60.4.4 Tensor Products

Let G be a subgroup of $\mathrm{GL}(d, K)$, where $K = GF(q)$, and let V be the natural $K[G]$-module. We say that G preserves a tensor decomposition of V as $U \otimes W$ if there is an isomorphism of V onto $U \otimes W$ such that the induced image of G in $\mathrm{GL}(U \otimes W)$ lies in $\mathrm{GL}(U) \circ \mathrm{GL}(W)$.

Theoretical details of the algorithm used may be found in Leedham-Green and O'Brien [LGO97b, LGO97a].

The verbose flag `SetVerbose ("Tensor", 1)` will provide information on the progress of the algorithm.

`IsTensor(G: parameters)`

`Factors` [SeqEnum] *Default :* []

Given a matrix group G defined over a finite field, this intrinsic returns `true` if G preserves a non-trivial tensor decomposition, `false` if G is does not preserve a tensor decomposition, or `"unknown"` if no decision can be reached.

A sequence of valid dimensions for potential factors may be supplied using the parameter `Factors`. Then for each element $[u, w]$ of the sequence `Factors`, the algorithm will search for decompositions of V as $U \otimes W$, where U must have dimension u and W must have dimension w only. If this parameter is not set, then all valid factorisations will be considered.

`TensorBasis(G)`

Given a matrix group G defined over a finite field that admits a tensor decomposion, this intrinsic returns the change-of-basis matrix which exhibits the tensor decomposition of G.

`TensorFactors(G)`

Given a matrix group G defined over a finite field that admits a tensor decomposion, this intrinsic returns two groups which are the tensor factors of G.

`IsProportional(X, k)`

Given a matrix group G defined over a finite field that admits a tensor decomposition, this intrinsic returns `true` if and only if the matrix X is composed of $k \times k$ blocks which differ only by scalars. If this is indeed the case, the tensor decomposition of X is also returned.

Example H60E4

We define a subgroup of $GL(6, 3)$ which admits a non-trivial tensor decomposition.

```
> P := GL(6, 3);
>
> g := P![ 0, 1, 1, 2, 1, 0, 2, 2, 1, 2, 1, 1, 1, 0, 2, 1, 2, 2, 1, 2, 2,
>          2, 2, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 2, 2 ];
>
> h := P![ 1, 0, 2, 1, 1, 2, 0, 0, 2, 0, 0, 2, 2, 0, 1, 0, 2, 1, 2, 1, 2,
```

```
>          2, 1, 1, 0, 2, 0, 1, 0, 0, 0, 0, 0, 2, 1, 2 ];
>
> G := sub< P | g, h >;
> IsTensor(G);
true
> C := TensorBasis(G);
```

So C is the change-of-basis matrix. If we conjugate $G.1$ by C, we obtain a visible Kronecker product.

```
> G.1^C;
[0 0 2 0 2 0]
[0 0 2 2 2 2]
[2 0 0 0 2 0]
[2 2 0 0 2 2]
[0 0 0 0 1 0]
[0 0 0 0 1 1]
>
```

We use the function `IsProportional` to verify that $G.1^C$ is a Kronecker product.

```
> IsProportional(G.1^C, 2);
true
<
 [2 0]
 [2 2],
 [0 1 1]
 [1 0 1]
 [0 0 2]
>
```

Finally, we display the tensor factors.

```
> A := TensorFactors(G);
> A[1];
MatrixGroup(2, GF(3))
Generators:
    [1 2]
    [2 2]

    [2 0]
    [2 2]
> A[2];
MatrixGroup(3, GF(3))
Generators:
    [0 1 0]
    [1 2 1]
    [1 2 0]

    [0 1 1]
    [1 0 1]
```

```
[0 0 2]
```

60.4.5 Tensor-induced Groups

Let G be a subgroup of $\mathrm{GL}(d, K)$, where $K = \mathbf{F}_q$ and $q = p^e$ for some prime p, and let V be the natural $K[G]$-module. Assume that d has a proper factorisation as u^r. We say that G is tensor-induced if G preserves a decomposition of V as

$$U_1 \otimes U_2 \otimes \cdots \otimes U_r$$

where each U_i has dimension $u > 1$, $r > 1$, and the set of U_i is permuted by G. If G is tensor-induced, then there is a homomorphism of G into the symmetric group S_r.

Theoretical details of the algorithm used may be found in Leedham-Green and O'Brien [LGO02].

`SetVerbose ("TensorInduced", 1)` will provide information on the progress of the algorithm.

`IsTensorInduced(G : parameters)`

`InducedDegree` RngIntElt *Default :* "*All*"

Given a matrix group G defined over a finite field, return `true` if G is tensor-induced, `false` if G is not tensor-induced, and `"unknown"` if no decision can be reached.

If the value of the parameter `InducedDegree` is set to r, then the algorithm will search for homomorphisms into the symmetric group of degree r only. Otherwise is will consider all valid degrees.

`TensorInducedBasis(G)`

Given a matrix group G defined over a finite field that is tensor-induced, this intrinsic returns the change-of-basis matrix which exhibits that G is tensor-induced.

`TensorInducedPermutations(G)`

Given a matrix group G defined over a finite field that is tensor-induced, this intrinsic returns a sequence whose i-th entry is the homomorphic image of $G.i$ in S_r.

`TensorInducedAction(G, g)`

Given a matrix group G defined over a finite field that is tensor-induced, this intrinsic returns the tensor-induced action of the element $g \in G$.

Example H60E5

We illustrate the use of the functions for determining if a matrix group is tensor-induced.

```
> G := GL(2, 3);
> S := Sym(3);
> G := TensorWreathProduct(G, S);
> IsTensorInduced(G);
true
```

We next recover the permutations.

```
> TensorInducedPermutations(G);
[
    Id(S),
    Id(S),
    (1, 2, 3),
    (1, 2)
]
```

Hence $G.1$ and $G.2$ are in the kernel of the homomorphism from G to S. We extract the change-of-basis matrix C and then conjugate $G.1$ by C, thereby obtaining a visible Kronecker product.

```
> C := TensorInducedBasis(G);
> x := G.1^C;
> x;
[2 0 0 0 0 0 0 0]
[0 2 0 0 0 0 0 0]
[0 0 2 0 0 0 0 0]
[0 0 0 2 0 0 0 0]
[1 0 0 0 1 0 0 0]
[0 1 0 0 0 1 0 0]
[0 0 1 0 0 0 1 0]
[0 0 0 1 0 0 0 1]
```

Finally, we verify that $x = G.1^C$ is a Kronecker product for each of 2 and 4.

```
> IsProportional(x, 2);
true
<[2 0]
[0 2], [1 0 0 0]
[0 1 0 0]
[2 0 2 0]
[0 2 0 2]>
> IsProportional(x, 4);
true
<[2 0 0 0]
[0 2 0 0]
[0 0 2 0]
[0 0 0 2], [1 0]
[2 2]>
```

60.4.6 Normalisers of Extraspecial r-groups and Symplectic 2-groups

Let $G \leq GL(d, q)$, where $d = r^m$ for some prime r. If G is contained in the normaliser of an r-group R, of order either r^{2m+1} or 2^{2m+2}, then either R is extraspecial (in the first case), or R is a 2-group of symplectic type (that is, a central product of an extraspecial 2-group with the cyclic group of order 4).

If $d = r$ is an odd prime, we use the Monte Carlo algorithm of Niemeyer [Nie05] to decide whether or not G normalises such a subgroup. Otherwise, the corresponding intrinsic `IsExtraSpecialNormaliser` searches for elements of the normal subgroup, and can only reach a negative conclusion in certain limited cases. If it cannot reach a conclusion it returns `"unknown"`.

`IsExtraSpecialNormaliser(G)`

Given a matrix group G defined over a finite field, the intrinsic returns `true` if G normalises an extraspecial r-group or 2-group of symplectic type, `false` if G is known not to normalise an extraspecial r-group or a 2-group of symplectic type, or `"unknown"` if it cannot reach a conclusion.

`ExtraSpecialParameters(G)`

Given a matrix group G defined over a finite field that is known to normalise an extraspecial r-group or 2-group of symplectic type, this intrinsic returns a sequence of two integers, r and n, where the extraspecial or symplectic subgroup R normalised by G has order r^n.

`ExtraSpecialGroup(G)`

Given a matrix group G defined over a finite field that is known to normalise an extraspecial r-group or 2-group of symplectic type, this intrinsic returns the extraspecial or symplectic subgroup normalised by G.

`ExtraSpecialNormaliser(G)`

Given a matrix group G defined over a finite field that is known to normalise an extraspecial r-group or 2-group of symplectic type, this intrinsic returns the action of the generators of G on its normal extraspecial or symplectic subgroup as a sequence of matrices, each of degree $2r$, one for each generator of G.

`ExtraSpecialAction(G, g)`

Given a matrix group G defined over a finite field that is known to normalise an extraspecial r-group or 2-group of symplectic type, this intrinsic returns a matrix of degree $2r$ describing the action of element g on the extraspecial or symplectic group normalised by G.

`ExtraSpecialBasis(G)`

Given a matrix group G defined over a finite field, which is the odd prime degree case of G normalising an extraspecial r-group or 2-group of symplectic type, this intrinsic returns the change-of-basis matrix which conjugates the normal extraspecial subgroup into a "nice" representation, generated by a diagonal and a permutation matrix.

Example H60E6

For this example we construct a subgroup G of $\mathrm{GL}(7,8)$ that normalises an extraspecial r-group or 2-group of symplectic type.

```
> F:=GF(8);
> P:=GL(7,F);
> w := PrimitiveElement(F);
> g1:=P![
> w,0,w^2,w^5,0,w^3,w,w,1,w^6,w^3,0,w^4,w,w^2,w^6,w^4,1,w^3,w^3,w^5,
> w^6,w,w^3,1,w^5,0,w^4,1,w^6,w^3,w^6,w^3,w^2,w^2,w^3,w^6,w^6,w^4,1,w^2,w^4,
> w^5,w^4,w^2,w^6,1,w^5,w ];
> g2:=P![w^3,w^4,w^2,w^6,w,w,w^3,w^3,w^4,w,w,w^2,w^3,w^3,w,w^3,w^5,w,1,w^3,w,
> 0,w^2,w^6,w,w^5,1,w,w^6,0,w^3,0,w^4,w,w^5,w^3,w^3,1,w^3,w^5,w^5,w^3,
> w^4,w^6,w,w^6,w^4,w^4,0 ];
> g3:=P![w^5,w^6,w^2,w,w,w^4,w^6,w^6,w^6,w,w^6,w,1,w^3,w,w^6,w^2,w,w^6,w^3,w^6,
> w^2,w^6,w^6,w^3,w,w^6,w^5,0,w^4,w^6,w^6,w,w^2,0,w,w^3,w^5,w^2,w^3,w^4,w^6,
> 0,w^3,w,w^3,w^4,w^3,1];
> gens := [g1,g2,g3];
> G := sub< P | gens >;
> IsExtraSpecialNormaliser(G);
true
```

So G has the desired normaliser property.

```
> ExtraSpecialParameters (G);
[ 7, 3 ]
> N:=ExtraSpecialNormaliser(G);
> N;
[
    [3 4]
    [1 4],
    [4 3]
    [0 2],
    [1 0]
    [0 1]
]
```

60.4.7 Writing Representations over Subfields

The algorithm implemented by these functions is due to Glasby, Leedham-Green and O'Brien [GLGO05]. We also provide access to an earlier algorithm for the non-scalar case developed by Glasby and Howlett [GH97].

`IsOverSmallerField(G : parameters)`

`Scalars`	BoolElt	*Default :* `false`
`Algorithm`	MonStgElt	*Default :* "*GLO*"

Given an absolutely irreducible matrix group G defined over a finite field K, this intrinsic decides whether or not G has an equivalent representation over a subfield of K. If so, it returns `true` and the representation over the smallest possible subfield, otherwise it returns `false`. If the optional argument `Scalars` is `true` then decide whether or G modulo scalars has an equivalent representation over a subfield of K. If the optional argument `Algorithm` is set to `"GH"`, then the non-scalar case uses the original Glasby and Howlett algorithm. The default is the Glasby, Leedham-Green and O'Brien algorithm, specified by `"GLO"`.

`IsOverSmallerField(G, k : parameters)`

`Scalars`	BoolElt	*Default :* `false`
`Algorithm`	MonStgElt	*Default :* "*GLO*"

Given an absolutely irreducible matrix group G defined over a finite field K, and a positive integer k which is a proper divisor of the degree of K, this intrinsic decides whether or not G has an equivalent representation over a proper subfield of K having degree k over the prime field. If so, it returns `true` and the representation over this subfield, else it returns `false`. If the optional argument `Scalars` is `true` then it decides whether or not G modulo scalars has an equivalent representation over a degree k subfield of K. If the optional argument `Algorithm` is set to `"GH"`, then the non-scalar case uses the original Glasby and Howlett algorithm. The default is the Glasby, Leedham-Green and O'Brien algorithm, specified by `"GLO"`.

`SmallerField(G)`

Given an absolutely irreducible matrix group G defined over a finite field K, which can be written over a proper subfield of K (possibly modulo scalars), return the subfield.

`SmallerFieldBasis(G)`

Given an absolutely irreducible matrix group G defined over a finite field K, which can be written over a proper subfield of K (possibly modulo scalars), return the change of basis matrix for G which rewrites G over the smaller field.

`SmallerFieldImage(G, g)`

Given an absolutely irreducible matrix group G defined over a finite field K, which can be written over a proper subfield of K (possibly modulo scalars), return the image of $g \in G$ in the group defined over the subfield.

Example H60E7

We define a subgroup of $\mathrm{GL}(3,8)$ which can be written over $\mathbf{F}_2$.

```
> G := GL (2, GF (3, 2));
> H := GL (2, GF (3, 8));
> K := sub < H | G.1, G.2 >;
> K;
MatrixGroup(2, GF(3^8))
Generators:
    [ $.1^820        0]
    [       0        1]

    [       2        1]
    [       2        0]
> flag, M := IsOverSmallerField (K);
> flag;
true
> M;
MatrixGroup(2, GF(3^2))
Generators:
    [$.1^7 $.1^2]
    [    1     2]

    [$.1^7 $.1^6]
    [$.1^2 $.1^5]
> F := GF(3, 4);
> G := MatrixGroup<2, F | [ F.1^52, F.1^72, F.1^32, 0 ],
>                                 [ 1, 0, F.1^20, 2 ] >;
> flag, X := IsOverSmallerField (G);
> flag;
false
```

We now see if G has an equivalent representation modulo scalars.

```
> flag, X := IsOverSmallerField (G: Scalars := true);
> flag;
true
> X;
MatrixGroup(2, GF(3))
Generators:
    [2 1]
    [1 0]
    [2 1]
    [1 1]
> SmallerField (G);
Finite field of size 3
> SmallerFieldBasis (G);
[F.1^33 F.1^23]
[F.1^43 F.1^63]
```

```
> g := G.1 * G.2^2; g;
[F.1^52 F.1^72]
[F.1^32      0]
> SmallerFieldImage (G, g);
[1 2]
[2 0]
```

`WriteOverSmallerField(G, F)`

Given a group G of $d \times d$ matrices over a finite field E having degree e and a subfield F of E having degree f, write the matrices of G as de/f by de/f matrices over F and return the group and the isomorphism.

Example H60E8

We define the group $\mathrm{GL}(2, 4)$ and then rewrite in over $\mathbf{F}_2$ as a degree 4 matrix group.

```
> G := GL(2, 4);
> H := WriteOverSmallerField(G, GF(2));
> H;
MatrixGroup(4, GF(2))
Generators:
    [0 1 0 0]
    [1 1 0 0]
    [0 0 1 0]
    [0 0 0 1]

    [1 0 1 0]
    [0 1 0 1]
    [1 0 0 0]
    [0 1 0 0]
```

60.4.8 Decompositions with Respect to a Normal Subgroup

`SearchForDecomposition(G, S)`

Given a matrix group G defined over a finite field and a sequence S of elements of G, this intrinsic first constructs the normal closure N of S in G. It then seeks to decide whether or not G, with respect to N, has a decomposition corresponding to one of the categories (ii)–(vi) in the theorem of Aschbacher stated at the beginning of this section. Theoretical details of the algorithms used may be found in Holt, Leedham-Green, O'Brien, and Rees [HLGOR96a].

In summary, it tests for one of the following possibilities:

(ii) G acts semilinearly over an extension field L of K, and N acts linearly over L;

(iii) G acts imprimitively on V and N fixes each block of imprimitivity;

(iv) G preserves a tensor product decomposition $U \otimes W$ of V, where N acts as scalar matrices on U;

(v) N acts absolutely irreducibly on V and is an extraspecial p-group for some prime p, or a 2-group of symplectic type;

(vi) G preserves a tensor-induced decomposition $V = \otimes^m U$ of V for some $m > 1$, where N acts absolutely irreducibly on V and fixes each of the m factors.

If one of the listed decompositions is found, then the function *reports* the type found and returns `true`; if no decomposition is found with respect to N, then the function returns `false`. The answer provided by the function is conclusive for decompositions of types (ii)–(v), but a negative answer for (vi) is not necessarily conclusive.

Each test involves a decomposition of G with respect to the normal subgroup N (which may sometimes be trivial or scalar). In (ii), N is the subgroup of G acting linearly over the extension field irreducibly on V. In (iii), N is the subgroup which fixes each of the subspaces in the imprimitive decomposition of V. In (iv), it is the subgroup acting as scalar matrices on one of the factors in the tensor-product decomposition. In (v), N is already described, and in (vi), it is the subgroup fixing each of the factors in the tensor-induced decomposition (so N itself falls in Category (iv)).

If any one of these decompositions can be found, then it may be possible to obtain an explicit representation of G/N and hence reduce the study of G to a smaller problem. For example, in Category (iii), G/N acts as a permutation group on the subspaces in the imprimitive decomposition of V. Currently only limited facilities are provided to construct G/N.

Information about the progress of the algorithm can be output by setting the verbose flag `SetVerbose ("Smash", 1)`.

60.4.8.1 Accessing the Decomposition Information

The access functions described in the sections on Primitivity Testing, Semilinearity, Tensor Products, Tensor Induction, and Normalisers of extraspecial groups may be used to extract information about decompositions of type (ii), (iii), (iv), (v) and (vi). We illustrate such decompositions below.

Example H60E9

We begin with an example where no decomposition exists.

```
> G := GL(4, 5);
> SearchForDecomposition (G, [G.1]);
Smash: No decomposition found
false
```

The second example is of an imprimitive decomposition.

```
> M := GL (4, 7);
> P := Sym (3);
> G := WreathProduct (M, P);
> SearchForDecomposition (G, [G.1, G.2]);
Smash: G is imprimitive
true
> IsPrimitive (G);
false
> BlocksImage (G);
Permutation group acting on a set of cardinality 3
    Id($)
    Id($)
    (1, 2, 3)
    (1, 2)
```

The third example admits a semilinear decomposition.

```
> P := GL(6,3);
> g1 := P![0,1,0,0,0,0,-1,0,0,0,0,0,
>          0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1];
> g2 := P![-1,0,0,0,1,0,0,-1,0,0,0,1,
>          -1,0,0,0,0,0,0,-1,0,0,0,0,0,0,-1,0,0,0,0,0,0,-1,0,0];
> g3 := P![1,0,0,0,0,0,0,-1,0,0,0,0,
>          0,0,1,0,0,0,0,0,0,-1,0,0,0,0,0,0,1,0,0,0,0,0,0,-1];
> G := sub <P | g1, g2, g3 >;
>
> SearchForDecomposition (G, [g1]);
Smash: G is semilinear
true
> IsSemiLinear (G);
true
> DegreeOfFieldExtension (G);
2
```

```
> CentralisingMatrix (G);
[2 2 0 0 0 0]
[1 2 0 0 0 0]
[0 0 2 2 0 0]
[0 0 1 2 0 0]
[0 0 0 0 2 2]
[0 0 0 0 1 2]
> FrobeniusAutomorphisms (G);
[ 1, 1, 3 ]
```

The fourth example admits a tensor product decomposition.

```
> F := GF(5);
> G := GL(5, F);
> H := GL(3, F);
> P := GL(15, F);
> A := MatrixAlgebra (F, 5);
> B := MatrixAlgebra (F, 3);
> g1 := A!G.1; g2 := A!G.2;  g3 := A!Identity(G);
> h1 := B!H.1; h2 := B!H.2; h3 := B!Identity(H);
> w := TensorProduct (g1, h3);
> x := TensorProduct (g2, h3);
> y := TensorProduct (g3, h1);
> z := TensorProduct (g3, h2);
> G := sub < P | w, x, y, z>;
> SearchForDecomposition (G, [G.1, G.2]);
Smash: G is a tensor product
true
> IsTensor (G);
true
> TensorBasis (G);
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[4 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 4 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 4 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 4 0 1]
[0 0 0 0 0 0 0 0 0 4 0 1 0 0 0]
[1 4 4 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 4 4 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 4 4 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 4 4]
[0 0 0 0 0 0 0 0 0 1 4 4 0 0 0]
```

Our fifth example is of a tensor-induced decomposition.

```
> M := GL (3, GF(2));
```

```
> P := Sym (3);
> G := TensorWreathProduct (M, P);
> SearchForDecomposition (G, [G.1]);
Smash: G is tensor induced
true
>
> IsTensorInduced (G);
true
> TensorInducedPermutations (G);
[ Id(P), Id(P), (1, 3, 2), (1, 3) ]
```

Our final example is of a normaliser of a symplectic group.

```
> F := GF(5);
> P := GL(4,F);
> g1 := P![ 1,0,0,0,0,4,0,0,2,0,2,3,3,0,4,3];
> g2 := P![ 4,0,0,1,2,4,4,0,1,0,1,2,0,0,0,1];
> g3 := P![ 4,0,1,1,0,1,0,0,0,1,3,4,0,4,3,2];
> g4 := P![ 2,0,4,3,4,4,2,4,0,1,3,4,4,2,0,1];
> g5 := P![ 1,1,3,4,0,0,3,4,2,0,0,4,3,1,3,4];
> g6 := P![ 2,0,0,0,0,2,0,0,0,0,2,0,0,0,0,2];
> G := sub < P | g1, g2, g3, g4, g5, g6 >;
> SearchForDecomposition (G, [G.4]);
Smash: G is normaliser of symplectic 2-group
true
> IsExtraSpecialNormaliser (G);
true
> ExtraSpecialParameters (G);
[2, 6]
> g := G.1 * G.2;
> ExtraSpecialAction(G, g);
[0 1 0 0]
[1 1 0 0]
[0 1 1 1]
[1 1 1 0]
```

60.5 Constructive Recognition for Simple Groups

For each finite non-abelian simple group S, we designate a specific *standard copy* of S. The standard copy has a designated set of *standard generators*. For example, the standard copy of $\mathrm{Alt}(n)$ is on n points; its standard generators are $(1,2,3)$ and either of $(3,\ldots,n)$ or $(1,2)(3,\ldots,n)$ according to the parity of n. For a projective representation, the standard copy is the quotient of a matrix group by its scalar subgroup. For example, the standard copy of $\mathrm{PSL}(n,q)$ is the quotient of $\mathrm{SL}(n,q)$ by its scalar subgroup.

To compute in a copy G of S, we first construct effective isomorphisms between G and its standard copy. We do this by finding generators in G that correspond to the standard generators of S under an isomorphism. More formally, a *constructive recognition algorithm* for a non-abelian simple group G (possibly with decorations) solves the following problem: construct an isomorphism ϕ from G to a standard copy S of G, such that $\phi(g)$ can be computed efficiently for every $g \in G$. This is done by constructing standard generators in both G and its standard copy S.

A *rewriting algorithm* for G solves the *constructive membership problem*: given $g \in U \geq G = \langle X \rangle$, decide whether or not $g \in G$, and if so express g as an SLP in X. (Here U is the generic overgroup of G, such as $\mathrm{GL}(d,q)$ or $\mathrm{Sym}(n)$.) The rewriting algorithm is used to make the isomorphism between S and G effective. To compute the image of an arbitrary element s of S in G, we first write s as an SLP in the standard generators of S and then evaluate the SLP in the copy of the standard generators in G.

To verify that the homomorphism from S to G is an isomorphism, we can evaluate in G a *standard presentation* for S on its standard generators. If the copy of the standard generators in G satisfy the presentation, then we have proved that we have an isomorphism.

For a detailed discussion of these topics, see [O'B11, LGO09].

`ClassicalStandardGenerators(type, d, q)`

This intrinsic produces the standard generators of Leedham-Green and O'Brien for the quasisimple classical group of specified type in dimension d over field of size q. The type is designated by the argument "type" which must be one of the strings "SL", "Sp", "SU", "Omega", "Omega-", or "Omega+". The standard generators defined a specific copy of a classical group and are defined in [LGO09] and [DLLGO13].

`ClassicalConstructiveRecognition(G : parameters)`

`Case`	MonStgElt	*Default :* "*unknown*"
`Randomiser`	GrpRandProc	*Default :*

The argument G must be a matrix group defined over a finite field such that $G = \langle X \rangle$ is conjugate to a quasisimple classical group in its natural representation in dimension at least 2. The intrinsic constructs a copy $\mathcal{S}$ in G of the generators defined by `StandardGenerators`. If G is quasisimple and classical, then the function returns

`true`, the standard generators $\mathcal{S}$, and SLPs for these in X; otherwise it returns `false`.

The result returned by the intrinsic `ClassicalType (G)` can be supplied using the optional parameter `Case`.

The parameter `Randomiser` allows the user to specify the random process that is be used to construct random elements and the SLPs returned for the standard generators are in the word group of this process. The default value of `Randomiser` is `RandomProcessWithWords(G)`.

The implementations for even and odd characteristic were developed by Heiko Dietrich and Eamonn O'Brien respectively.

`ClassicalChangeOfBasis(G)`

G is a classical group in its natural representation; return a change-of-basis matrix to conjugate the generators returned by `ClassicalStandardGenerators` to those returned by `ClassicalConstructiveRecognition` (G). The latter intrinsic must have been applied to G.

`ClassicalRewrite(G, gens, type, dim, q, g` : *parameters*`)`

`Method` MonStgElt *Default : "choose"*

Let G be a classical group of type `type`, which is one of the strings "SL", "Sp", "SU", "Omega", "Omega-", or "Omega+", with dimension `dim` over the field $\mathbf{F}_q$ generated by `gens` which satisfy `ClassicalStandardPresentation (type, dim, q)`. Further, let g be an element of `Generic`(G).

If $g \in G$, then the function returns `true` and an SLP for g in `gens`; if $g \notin G$ then the function searches for an SLP w such that $g{\cdot}$`Evaluate`$(w, gens)^{-1}$ centralizes G; if it is successful, it returns `false` and w. Otherwise the function returns `false, false`.

The function chooses one of the following methods:

(i) If G is in its natural representation and `gens` is `ClassicalStandardGenerators (type, dim, q)` then Elliot Costi's implementation [Cos09] is used.

(ii) If (i) is not valid, but G is an absolutely irreducible representation in the defining characteristic, then another implementation developed by Csaba Schneider is used. This implementation is based on the description given by Costi [Cos09].

(iii) If neither of (i) and (ii) is valid, then a "black-box" method, independent of the representation of G, developed by Csaba Schneider is used. This case is not yet implemented for orthogonal groups.

The optional parameter `Method` can be used to override the default choice of method. The possible values of `Method` are `CharP` and `BB`.

A description of the algorithm used in the defining characteristic case appears in [Cos09]; a short description of the black-box algorithm appears in [AMPS10]. The code was prepared for distribution by Csaba Schneider.

`ClassicalRewriteNatural(type, CB, g)`

This is a faster specialized verson of the intrinsic `ClassicalRewrite` discussed above; it is designed for classical groups in their **natural representation**.

The argument `type` must be one of the strings "SL", "Sp", "SU", "Omega", "Omega-", or "Omega+". Both `CB` and g are elements of some $\mathrm{GL}(d, q)$.

If g is a member of the group generated by `ClassicalStandardGenerators (type, d, q)`$^{\mathbf{CB}}$ then the function returns `true` and an SLP w such that `Evaluate (w, ClassicalStandardGenerators (type, d, q)`$^{\mathbf{CB}}$`)` $= g$. Otherwise the function returns `false, false`.

This algorithm was developed and implemented by Elliot Costi; the code was prepared for distribution by Csaba Schneider.

`ClassicalStandardPresentation(type, d, q : parameters)`

`Projective` BoolElt *Default :* `false`

Given the specification `type, d, q` of a quasisimple group G, this intrinsic constructs a presentation on the standard generators for G. The string `type` must be one of "SL", "Sp", "SU", "Omega", "Omega-", or "Omega+", while d is the dimension and q is the cardinality of the finite field. The presentations are described in [LGO]. The relations are returned as SLPs together with the parent SLPGroup.

If the parameter `Projective` is set to true, the intrinsic constructs a presentation for the corresponding projective group.

Example H60E10

As our first illustration, we produce standard generators for $\mathrm{SL}(6, 5^3)$:

```
> E := ClassicalStandardGenerators ("SL", 6, 5^3);
> E;
[
    [      0       1       0       0       0       0]
    [      4       0       0       0       0       0]
    [      0       0       1       0       0       0]
    [      0       0       0       1       0       0]
    [      0       0       0       0       1       0]
    [      0       0       0       0       0       1],

    [      0       0       0       0       0       1]
    [      4       0       0       0       0       0]
    [      0       4       0       0       0       0]
    [      0       0       4       0       0       0]
    [      0       0       0       4       0       0]
    [      0       0       0       0       4       0],

    [      1       1       0       0       0       0]
    [      0       1       0       0       0       0]
    [      0       0       1       0       0       0]
```

```
    [      0       0       0       1       0       0]
    [      0       0       0       0       1       0]
    [      0       0       0       0       0       1],

    [    $.1       0       0       0       0       0]
    [      0 $.1^123       0       0       0       0]
    [      0       0       1       0       0       0]
    [      0       0       0       1       0       0]
    [      0       0       0       0       1       0]
    [      0       0       0       0       0       1]
]
```

We now perform constructive recognition on $\mathrm{SL}(6, 5^3)$, and so obtain S, conjugate to E in $\mathrm{GL}(6, 5^3)$. Observe that the change-of-basis matrix returned by `ClassicalChangeOfBasis` performs this conjugation.

```
> G := SL (6, 5^3);
> f, S, W := ClassicalConstructiveRecognition (G);
> f;
true
> CB := ClassicalChangeOfBasis (G);
> E^CB eq S;
true
```

Note that W is list of SLPs expressing S in terms of defining generators of G.

```
> S eq Evaluate (W, [G.i: i in [1..Ngens (G)]]);
true
```

We next express a random element of G as a SLP in S and then check that the standard generators satisfy the standard presentation.

```
> g := Random (G);
> f, w := ClassicalRewriteNatural ("SL", CB, g);
> Evaluate (w, S) eq g;
true
>
> P, R := ClassicalStandardPresentation ("SL", 6, 5^3);
> Set (Evaluate (R, S));
{
    [      1       0       0       0       0       0]
    [      0       1       0       0       0       0]
    [      0       0       1       0       0       0]
    [      0       0       0       1       0       0]
    [      0       0       0       0       1       0]
    [      0       0       0       0       0       1]
}
```

We perform constructive recognition on a random conjugate of $\mathrm{Sp}(10, 3^6)$ and again check that the standard generators satisfy the standard presentation.

```
> G := RandomConjugate (Sp(10, 3^6));
```

```
> f, S, W := ClassicalConstructiveRecognition (G);
> f;
true
```

The return variable W is list of SLPs expressing S in terms of defining generators of G.

```
> S eq Evaluate (W, [G.i: i in [1..Ngens (G)]]);
true
>
> g := Random (G);
> CB := ClassicalChangeOfBasis (G);
> f, w := ClassicalRewriteNatural ("Sp", CB, g);
> Evaluate (w, S) eq g;
true
>
> P, R := ClassicalStandardPresentation ("Sp", 10, 3^6);
> Set (Evaluate (R, S));
{
    [1 0 0 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0 0 0]
    [0 0 1 0 0 0 0 0 0 0]
    [0 0 0 1 0 0 0 0 0 0]
    [0 0 0 0 1 0 0 0 0 0]
    [0 0 0 0 0 1 0 0 0 0]
    [0 0 0 0 0 0 1 0 0 0]
    [0 0 0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 0 0 1 0]
    [0 0 0 0 0 0 0 0 0 1]
}
```

As another demonstration, we constructively recognise $\Omega^-(16, 2^6)$.

```
> G := RandomConjugate (OmegaMinus (16, 2^6));
> f, S, W := ClassicalConstructiveRecognition (G);
> f;
true
```

A random element of G is expressed as an SLP in S:

```
> g := Random (G);
> CB := ClassicalChangeOfBasis (G);
> f, w := ClassicalRewriteNatural ("Omega-", CB, g);
> Evaluate (w, S) eq g;
true
```

Finally, we illustrate using `ClassicalRewrite` to write an element of a classical group in a non-natural representation as an SLP in its standard generators.

```
> gens := [ExteriorSquare (x) : x in ClassicalStandardGenerators ("Sp", 6, 25)];
> G := sub<Universe (gens) | gens>;
> x := Random (G);
```

```
> f, w := ClassicalRewrite (G, gens, "Sp", 6, 25, x);
> f;
true
> Evaluate (w, gens) eq x;
true
> f, w := ClassicalRewrite (G, gens, "Sp", 6, 25, x : Method := "BB");
> f;
true
> Evaluate (w, gens) eq x;
true
```

60.6 Composition Trees for Matrix Groups

A *composition tree* for a group G can be viewed as a data structure that present a group in terms of its composition factors. The tree is constructed recursively and the data structure facilitates the rewriting of elements of G in terms of different generating sets.

The basic strategy for computing a *composition tree* of a matrix group employs a combination of a constructive version of Aschbacher's theorem [Asc84] and constructive recognition algorithms for finite simple groups. The basic algorithms are described in [LG01, O'B06, O'B11] while some new ideas introduced in [NS06] are incorporated. A detailed account of the entire *CompositionTree* procedure appears in [BHLGO11].

The algorithm to construct a composition tree proceeds as follows. Given a group G, either:

(i) Construct an effective homomorphism $\phi : G \to G_1$, for some group G_1. We call ϕ a *reduction* since G_1 is "smaller" than G in some respect – for example, its degree or field of definition.

(ii) Or deduce that G is cyclic, elementary abelian, or "close" to being non-abelian simple. Now G becomes a *leaf* in the tree.

Assume that Case (i) applies.

1. Now construct a composition tree for G_1.
2. Construct generators for $G_0 := \mathrm{Ker}(\phi)$. This requires a rewriting algorithm for G_1.
3. Construct a composition tree for G_0.
4. Combine the composition trees for G_1 and G_0 into a tree for G.

If $G \leq \mathrm{GL}(d, q)$, then we exploit Aschbacher's theorem [Asc84] in Step (1). This requires algorithms to decide if G lies in a certain Aschbacher class, and to construct the corresponding ϕ. Other homomorphisms, such as the determinant map, may also be used.

The group associated with a leaf need not be simple. It may be cyclic or elementary abelian, a soluble or non-abelian simple primitive permutation group, or an absolutely irreducible matrix group that is simple modulo its centre. The decisions on just which

groups are treated as a leaves are partly dictated by complexity considerations, and partly based on the quality of available algorithms to process a leaf. For example, we observe no practical advantage flowing from refining a cyclic group to its composition factors.

Once a composition tree for $G = \langle X \rangle$ has been constructed, then a second list Y of *nice generators* are stored with G. We call $\langle Y \rangle$ the *nice group*. The intrinsic `CompositionTree` constructs the nice generators Y as SLPs in X. The rewriting algorithm solves rewriting problems on Y and the resulting SLPs can then be rewritten to provide SLPs on X.

The verbose flag `SetVerbose("CompositionTree", n)` with $n = 1, \ldots, 10$ may be used to print increasing levels of information on the progress of the functions.

`CompositionTree(G : parameters)`

`Verify`	BoolElt	*Default :* `false`
`Scalar`	FldFinElt	*Default :* 1
`KernelBatchSize`	RngIntElt	*Default :* 5
`MandarinBatchSize`	RngIntElt	*Default :* 100
`MaxHomFinderFails`	RngIntElt	*Default :* 1
`MaxQuotientOrder`	RngIntElt	*Default :* 10^6
`FastTensorTest`	BoolElt	*Default :* `true`
`MaxBSGSVerifyOrder`	RngIntElt	*Default :* 2000
`AnalysePermGroups`	BoolElt	*Default :* `false`
`KnownLeaf`	BoolElt	*Default :* `false`
`NamingElements`	RngIntElt	*Default :* 200
`UnipotentBatchSize`	RngIntElt	*Default :* 100
`PresentationKernel`	BoolElt	*Default :* `true`

Given a matrix group G defined over a finite field, this intrinsic constructs a composition tree for G and returns the tree.

`Verify`: If `true`, then verify correctness of the tree during construction.

`KernelBatchSize`: The number of normal generators used to construct the kernel of homomorphism.

`MandarinBatchSize`: The number of random elements used to check correctness of the outcome of Monte-Carlo algorithms.

`MaxHomFinderFails`: Assume a negative answer after this many failures of certain Monte Carlo algorithms.

`AnalysePermGroups`: If `false`, then always treat the permutation group as a leaf, and do not analyse its structure.

`NamingElements`: The number of random elements used in calls to `LieType` and `RecogniseClassical`.

`MaxQuotientOrder`: A leaf with larger order will not be fully refined to its composition factors.

`FastTensorTest`: Use only the fast tensor product test.

`MaxBSGSVerifyOrder`: If `RandomSchreier` is used on a leaf and it has order less than `MaxBSGSVerifyOrder`, then `Verify` the calculation.

`PresentationKernel`: Use presentations to obtain kernels, where possible.

`UnipotentBatchSize`: Batch size for the unipotent kernels.

`CompositionTreeFastVerification(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. The intrinsic determines if correctness of the composition tree can be verified at modest cost using presentations. In effect, the intrinsic determines whether presentations on nice generators are known for all the leaves.

`CompositionTreeVerify(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. The intrinsic verifies the correctness of the composition tree by using it to construct a presentation for G. If G satisfies the presentation, then return `true`, and the relators of the presentation as SLPs; otherwise return `false`. The presentation is on the group returned by `CompositionTreeNiceGroup`(G).

`CompositionTreeNiceGroup(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. The intrinsic returns the nice group for G.

`CompositionTreeSLPGroup(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure and the associated nice group H has previously been constructed. The intrinsic returns the word group W for H, and the map from W to H.

`DisplayCompTreeNodes(G :` *parameters*`)`

`NonTrivial`	BoolElt	*Default :* `true`
`Leaves`	BoolElt	*Default :* `false`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic displays information about the nodes in the composition tree for G. The tree is traversed in-order. If parameter `NonTrivial` is `true`, then only non-trivial nodes will be displayed. If parameter `Leaves` is `true` then only leaves will be displayed.

`CompositionTreeNiceToUser(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns the coercion map from SLPs in nice generators of G to SLPs in input user generators of G, as well as the SLPs of the nice generators given in terms the user generators.

`CompositionTreeOrder(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns the order of G.

`CompositionTreeElementToWord(G, g)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. Given an element $g \in G$, return `true` and an SLP for g in the nice generators of G, otherwise return `false`.

`CompositionTreeCBM(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns a change-of-basis matrix that exhibits the Aschbacher reductions of G given by the composition tree.

`CompositionTreeReductionInfo(G, t)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns a string description of the reduction at the internal node t in the composition tree for G, as well as the image and kernel of this reduction.

`CompositionTreeSeries(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns:

1. A normal series of subgroups $1 = G_0 < G_1 < ... < G_k = G$.
2. Maps $G_i \mapsto S_i$, where S_i is the standard copy of G_i/G_{i-1}, where $i \geq 1$. The kernel of this map is G_{i-1}. Observe that S_i may be the standard copy plus scalars Z, and the map is then a homomorphism modulo scalars, so that the kernel is $(G_{i-1}.Z)/Z$.
3. Maps $S_i \mapsto G_i$.
4. Maps $S_i \mapsto$ `WordGroup`(S_i).
5. Boolean flag `true` or `false` to indicate if the series is a true composition series.
6. A sequence of the leaf nodes in the composition tree corresponding to each composition factor. All maps are defined by rules, so `Function` can be applied on them to avoid built-in membership testing.

`CompositionTreeFactorNumber(G, g)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic returns the minimal integer i such that g lies in the ith-term of the normal series returned by `CompositionTreeSeries` for G.

`HasCompositionTree(G)`

Given a matrix group G defined over a finite field, this intrinsic returns `true` if G has a composition tree and `false` otherwise.

`CleanCompositionTree(G)`

The argument G must be a matrix group over a finite field for which a composition tree datastructure has previously been constructed. This intrinsic removes all data related to the composition tree datastructure for G.

Example H60E11

We construct a composition tree for the conformal orthogonal group G of plus type of degree 4 over $\mathbf{F}_{5^2}$.

```
> G := CGOPlus(4, 5^2);
>
> T := CompositionTree(G);
>
> DisplayCompTreeNodes (G: Leaves:=true);
node = 2
parent = 1
depth = 1
scalar = 1
info = leaf, GrpAb, cyclic group, order 12
fast verify = true
----------
node = 4
parent = 3
depth = 2
scalar = 1
info = leaf, GrpAb, cyclic group, order 4
fast verify = true
----------
node = 7
parent = 6
depth = 4
scalar = 12
info = leaf, GrpAb, cyclic group, order 24
fast verify = true
----------
node = 8
parent = 6
depth = 4
scalar = 2
info = leaf, GrpMat, almost simple, <"A", 1, 25>
fast verify = true
----------
node = 9
```

```
parent = 5
depth = 3
scalar = 1
info = leaf, GrpMat, almost simple, <"A", 1, 25>
fast verify = true
----------
```

We now verify correctness of the composition tree. In order to show what is going on we illustrate various pieces of the verification process. We begin by setting up the nice group H for G and its associated SLP group; observe that $H = G$. After checking that verification can be done quickly, we perform the verification.

```
> H := CompositionTreeNiceGroup(G);
> W := CompositionTreeSLPGroup(G);
>
> CompositionTreeFastVerification(G);
true
>
> f, R := CompositionTreeVerify(G);
> #R;
73
```

At this point we have verified correctness. However, we now explicitly evaluate the relations R on the generators of H. This step has already been performed by `CompositionTreeVerify` so it is shown here just for demonstration purposes.

```
> Set(Evaluate(R, [H.i:i in [1..Ngens(H)]]));
{
    [     1      0      0      0]
    [     0      1      0      0]
    [     0      0      1      0]
    [     0      0      0      1]
}
```

Now that we know that the composition tree is correct, we ask for the order of G.

```
> CompositionTreeOrder(G);
11681280000
```

Express the element g of G as a SLP on the generators of the nice group H.

```
> g := Random(G);
> f, w := CompositionTreeElementToWord(G, g);
> Evaluate(w, [H.i:i in [1..Ngens(H)]]) eq g;
true
```

Rewrite the SLP in terms of the user-supplied generators for G.

```
> tau := CompositionTreeNiceToUser(G);
> tau;
```

```
Mapping from: GrpSLP: W to SLPGroup(5)
```

Images of elements of W under *tau* lie in `WordGroup`(G).

```
> v := tau(w);
> Evaluate (v, [G.i : i in [1..Ngens(G)]]) eq g;
true
```

Test a random element of the generic group for G for membership. (The generic group will be the general linear group $\mathrm{GL}(4, 5^2)$.)

```
> x := Random(Generic(G));
> f, w := CompositionTreeElementToWord(G, x);
> f;
false
```

Finally, we construct a normal series for G and locate a random element within this series.

```
> CS, _, _, _, flag := CompositionTreeSeries(G);
> "Series is composition series? ", flag;
Series is composition series?  true
> "Length is ", #CS;
Length is  10
>
> g := Random(G);
> CompositionTreeFactorNumber(G, g);
10
```

Example H60E12

In this example, we choose a maximal subgroup of the linear group $SL(10, 2^8)$ and compute its composition tree.

```
> X := ClassicalMaximals ("L", 10, 2^8);
> G := X[1];
>
> T := CompositionTree (G);
>
> DisplayCompTreeNodes (G: Leaves:=true, NonTrivial:=true);
node = 6
parent = 5
depth = 5
scalar = 1
info = leaf, GrpAb, cyclic group, order 255
fast verify = true
----------
node = 9
parent = 8
depth = 5
scalar = 1
info = leaf, GrpMat, almost simple, <"A", 8, 256>
```

```
fast verify = true
----------
node = 13
parent = 12
depth = 3
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 15
parent = 14
depth = 4
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 17
parent = 16
depth = 5
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 19
parent = 18
depth = 6
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 21
parent = 20
depth = 7
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 23
parent = 22
depth = 8
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 25
parent = 24
depth = 9
scalar = 1
```

```
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 27
parent = 26
depth = 10
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
node = 29
parent = 28
depth = 11
scalar = 1
info = leaf, GrpPC, abelian group, order 256
fast verify = true
----------
```

We now set up the nice group H for G and its associated SLP group; observe that $H = G$.

```
> H := CompositionTreeNiceGroup(G);
> "# of generators of H is ", Ngens(H);
# of generators of H is  77
> W := CompositionTreeSLPGroup(G);
```

After checking that correctness of the composition tree can be verified quickly, we perform verification.

```
> CompositionTreeFastVerification(G);
true
> f, R := CompositionTreeVerify(G);
> #R;
3028
```

Evaluate the relations on the generators of H.

```
> Set (Evaluate (R, [H.i:i in [1..Ngens (H)]]));
{
    [1 0 0 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0 0 0]
    [0 0 1 0 0 0 0 0 0 0]
    [0 0 0 1 0 0 0 0 0 0]
    [0 0 0 0 1 0 0 0 0 0]
    [0 0 0 0 0 1 0 0 0 0]
    [0 0 0 0 0 0 1 0 0 0]
    [0 0 0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 0 0 1 0]
    [0 0 0 0 0 0 0 0 0 1]
```

```
}
```

Express the element g of G as a SLP on the generators of the nice group H. Then rewrite the SLP in terms of user generators for G.

```
> g := Random (G);
> f, w := CompositionTreeElementToWord (G, g);
> Evaluate (w, [H.i:i in [1..Ngens (H)]]) eq g;
true
>
> tau := CompositionTreeNiceToUser (G);
> tau;
Mapping from: GrpSLP: W to SLPGroup(4)
>
> v := tau (w);
> Evaluate (v, [G.i : i in [1..Ngens (G)]]) eq g;
true
```

Next test a random element of the generic group of G for membership of G.

```
> x := Random (Generic (G));
> f, w := CompositionTreeElementToWord (G, x);
> f;
false
```

Finally, we construct a normal series for G.

```
> CS, _, _, _, flag := CompositionTreeSeries (G);
> "Series is composition series? ", flag;
Series is composition series?  true
> "Length is ", #CS;
Length is  78
```

60.7 The LMG functions

The LMG (large matrix group) functions are designed to provide a user-friendly interface to the `CompositionTree` package, and thereby enable the user to carry out a limited range of structural calculations in a matrix group that is too large for the use of BSGS methods.

By default, these methods have a small probability of failing or even of returning incorrect results. For most examples, at the cost of some extra time, the user can ensure that the computed results are verified as correct by calling `LMGInitialise` with the `Verify` flag on the group G before calling any of the other functions. (Once `CompositionTree` or any of the LMG functions has been called on a group, further calls of `LMGInitialise` will have no effect.)

Let G be a matrix group over a finite field. On the first call of any of the LMG functions on G, MAGMA decides whether it will use BSGS or Composition Tree based methods on G. It does this by carrying out a quick calculation to decide whether any of the basic orbit lengths would be larger than a constant `LMGSchreierBound`, which is

set to 40000 by default, but can be changed by the user. If all basic orbit lengths are at most `LMGSchreierBound`, then BSGS methods are used on G, and the LMG functions are executed using the corresponding standard MAGMA functions. Otherwise, Composition Tree methods are used, starting with a call of `CompositionTree`(G).

Composition Tree methods are also used if the user calls `CompositionTree` on the group before using th LMG functions, or if the user calls `LMGInitialize` with the `Al` option set to `"CompositionTree"`.

`SetVerbose("LMG", n)` with $n = 1, 2$ or 3 will provide increasing levels of information on the progress of the functions.

`SetLMGSchreierBound(n)`

Set the constant `LMGSchreierBound` to n.

`LMGInitialize(G : parameters)`

`LMGInitialise(G : parameters)`

`Al`	MONSTGELT	*Default :* ""
`Verify`	BOOLELT	*Default :* `false`
`RandomSchreierBound`	RNGINTELT	*Default :* `LMGSchreierBound`

It is not normally necessary to call this function but, by setting the optional parameters, it can be used to initialise G for LMG computations with a different value of `LMGSchreierBound` or, by setting `Al` to be `"CompositionTree"` (or `"CT"` or `"RandomSchreier"` (or `"RS"`), to force the use of either Composition Tree or BSGS methods on G.

If the `Verify` flag is set, then an attempt will be made to verify the correctness of the computed BSGS or Composition Tree. This will make the initialisation process slower, and for some groups the increased memory requirements will make verification impractical. In that case, a warning message is displayed.

`LMGOrder(G)`

Given a matrix group G defined over a finite field, this intrinsic returns the order of G.

`LMGFactoredOrder(G)`

Given a matrix group G defined over a finite field, this intrinsic returns the factored order of G.

`LMGIsIn(G, x)`

Given a matrix group G defined over a finite field $\mathbf{F}_{n,q}$ of G, the intrinsic returns `true` if x is in G and `false` otherwise.

`LMGIsSubgroup(G, H)`

Given a matrix group G defined over a finite field $\mathbf{F}_{n,q}$ of G, the intrinsic returns `true` if $H \leq G$ and `false` otherwise.

`LMGEqual(G, H)`

Given a matrix groups G and H belonging to a common overgroup $\mathrm{GL}(n, q)$, the intrinsic returns `true` if G and H are equal and `false` otherwise.

`LMGIndex(G, H)`

Given a matrix group G defined over a finite field, and a subgroup H of G, the intrinsic returns the index of H in G.

`LMGIsNormal(G, H)`

Given a matrix group G defined over a finite field, and a subgroup H of G, the intrinsic returns `true` if H is normal in G and `false` otherwise.

`LMGNormalClosure(G, H)`

Given a matrix group G defined over a finite field, and a subgroup H of G, the intrinsic returns the normal closure of H in G.

`LMGDerivedGroup(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the derived subgroup of G.

`LMGCommutatorSubgroup(G, H)`

Let g and H be subgroups of $\mathrm{GL}(n, q)$. This intrinsic returns the commutator subgroup of G and H as a subgroup of $\mathrm{GL}(n, q)$.

`LMGIsSoluble(G)`
`LMGIsSolvable(G)`

Given a matrix group G defined over a finite field, the intrinsic returns `true` if G is soluble and `false` otherwise.

`LMGIsNilpotent(G)`

Given a matrix group G defined over a finite field, the intrinsic returns `true` if G is nilpotent and `false` otherwise.

`LMGCompositionSeries(G)`

Given a matrix group G defined over a finite field, the intrinsic returns a composition series for G.

`LMGCompositionFactors(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the composition factors of G. The Handbook entry for `CompositionFactors(G)` of a finite group gives a detailed description of how to interpret the returned sequence.

`LMGChiefSeries(G)`

Given a matrix group G defined over a finite field, the intrinsic returns a chief series for G.

`LMGChiefFactors(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the chief factors G. The Handbook entry for `ChiefFactors(G)` of a finite group gives a detailed description of how to interpret the returned sequence.

`LMGUnipotentRadical(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the unipotent radical U of the matrix group G. A group P of type `GrpPC` and an isomorphism $U \to P$ are also returned.

`LMGSolubleRadical(G)`
`LMGSolvableRadical(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the soluble radical S of G. A group P of type `GrpPC` and an isomorphism $S \to P$ are also returned.

`LMGFittingSubgroup(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the Fitting subgroup S of G. A group P of type `GrpPC` and an isomorphism $S \to P$ are also returned.

`LMGCentre(G)`
`LMGCenter(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the centre of G.

`LMGSylow(G,p)`

Given a matrix group G defined over a finite field, the intrinsic returns a Sylow p-subgroup of G.

`LMGSocleStar(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the inverse image in G of the socle of G/S, where S is the soluble radical of G.

`LMGSocleStarFactors(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the simple direct factors of `LMGSocleStar(G)`/`LMGSolubleRadical(G)`, which may be represented projectively for large classical groups. A list of maps from the factors to G is also returned.

`LMGSocleStarAction(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the map ϕ representing the conjugation action of G on the simple direct factors of `LMGSocleStar(G)`/`LMGSolubleRadical(G)`. The image and kernel of ϕ are also returned.

`LMGSocleStarActionKernel(G)`

Given a matrix group G defined over a finite field, this intrinsic returns three values. The first is the kernel of the conjugation action of G on the simple direct factors of `LMGSocleStar(G)`/`LMGSolubleRadical(G)`. A group P of type `GrpPC` isomorphic to `LMGSocleStarActionKernel(G)`/`LMGSocleStar(G)` and the epimorphism $G \to P$ are also returned.

`LMGSocleStarQuotient(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the quotient group `G`/`LMGSocleStar(G)` represented as a permutation group, with associated epimorphism and kernel.

Example H60E13

We apply the LMG functions to a maximal subgroup of $\mathrm{SL}(12, 5)$.

```
> SetVerbose("LMG", 1);
> C := ClassicalMaximals("L", 12, 5);
> G := C[4];
> LMGFactoredOrder(G);
RandomSchreierBound is 40000
Using CompositionTree on this group
Composition tree computed
Composition series has length 40
Order of group is:  27845944957511377275508129969239234924316406250000000000000\
0000000000000000000
[ <2, 32>, <3, 7>, <5, 66>, <7, 1>, <11, 1>, <13, 3>, <31, 3>, <71, 1>, <313,
1>, <19531, 1> ]
> LMGChiefFactors(G);
Classifying composition factors
Defined PCGroup of solvable radical
Computed PCGroup of SocleKernel/SocleStar
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  A(3, 5)                = L(4, 5)
    *
    |  A(7, 5)                = L(8, 5)
    *
```

```
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(5) (32 copies)
    1
> D := LMGDerivedGroup(G);
RandomSchreierBound is 40000
Using CompositionTree on this group
Composition tree computed
Composition series has length 38
Order of group is:  6961486239377844318877032492309808731079101562500000000000\
0000000000000000000
> LMGIndex(G, D);
4
> SetVerbose("LMG", 0);
> LMGEqual( LMGDerivedGroup(D), D );
true
> S := LMGSolubleRadical(G);
> LMGFactoredOrder(S);
[ <2, 4>, <5, 32> ]
> LMGIsSoluble(G);
false
> LMGIsSoluble(S);
true
> LMGIsNilpotent(S);
false
> #LMGCentre(G);
4
> #LMGCentre(S);
4
```

We carelessly used the standard Magma Order function in the above two commands, but it did not matter, because it was small. We will be more careful next time!

```
> F := LMGFittingSubgroup(G);
> LMGFactoredOrder( LMGCentre(F) );
[ <2, 2>, <5, 32> ]
> P := LMGSylow(G, 5);
> LMGFactoredOrder(P);
[ <5, 66> ]
> LMGEqual( D, LMGNormalClosure(G,P) );
true
> facs, maps := LMGSocleStarFactors(G);
> #facs;
```

```
2
> LMGChiefFactors(facs[1]);
    G
    |  A(7, 5)                = L(8, 5)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

Note that, for large classical groups, the socle-star factors are represented projectively.

```
> I := sub< Generic(G) | [ facs[2].i @ maps[2] : i in [1..Ngens(facs[2])] ] >;
> LMGChiefFactors( LMGNormalClosure(G, I) );
    G
    |  A(3, 5)                = L(4, 5)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    *
    |  Cyclic(5) (4 copies)
    1
```

The remaining functions in this section will work only if a permutation representation can be computed for G/L, where L is the soluble radical of G. Apart from `LMGRadicalQuotient` itself, they all operate by solving the problem first in G/L and then lifting the solution through elementary abelian layers of L. Results are returned using the same formats as for other types of finite groups.

`LMGRadicalQuotient(G)`

Given a matrix group G defined over a finite field, the intrinsic returns a permutation group P isomorphic to G/L, where L is the soluble radical of G. An epimorphism $G \to P$ and its kernel L are also returned.

Of course, this will only work if G/L has such a representation of sufficiently small degree. This function is called implicitly as a first step in all of the remaining functions in this section.

`LMGCentraliser(G, g)`

`LMGCentralizer(G, g)`

Given a matrix group G defined over a finite field, the intrinsic returns the centraliser in the matrix group G of $g \in G$.

`LMGIsConjugate(G, g, h)`

Given a matrix group G defined over a finite field, the intrinsic returns `true` if the elements g, h in G are conjugate. If so, a conjugating element will also be returned.

`LMGClasses(G)`

`LMGConjugacyClasses(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the conjugacy classes of G.

`LMGNormaliser(G, H)`

`LMGNormalizer(G, H)`

Given a matrix group G defined over a finite field, and a subgroup H of G, the intrinsic returns the normaliser of H in G.

`LMGIsConjugate(G, H, K)`

Given a matrix group G defined over a finite field, and subgroups H and K of G, the intrinsic returns `true` if the subgroups H and K are conjugate in G. If so, a conjugating element will also be returned.

`LMGMaximalSubgroups(G)`

Given a matrix group G defined over a finite field, the intrinsic returns the maximal subgroups of G.

60.8 Unipotent Matrix Groups

The *power-conjugate presentation* is a very efficient way of representing a unipotent group; see Chapter 63 for more information. In this section we describe a number of functions for finding such a *PC-presentation* for a unipotent matrix group defined over a finite field.

The algorithm used is a straightforward echelonisation-like procedure.

`UnipotentMatrixGroup(G)`

Given a matrix group G defined over a finite field, the intrinsic constructs a known unipotent matrix group from G. Note that MAGMA does not at this stage check that G is in fact unipotent.

`WordMap(G)`

Given a unipotent matrix group G defined over a finite field, the intrinsic constructs the *word map* for G. The word map is a map from G to the group of straight-line programs on n generators, where n is the number of generators of G. More information on SLP-groups may be found in Chapter 76.

Example H60E14

We construct a unipotent matrix group, and use the word map.

```
> G := MatrixGroup<4, GF(5) | [1,1,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1],
>       [1,-1,0,0, 0,1,1,0, 0,0,1,0, 0,0,0,1]>;
> G;
MatrixGroup(4, GF(5))
Generators:
    [1 1 0 0]
    [0 1 0 0]
    [0 0 1 0]
    [0 0 0 1]

    [1 4 0 0]
    [0 1 1 0]
    [0 0 1 0]
    [0 0 0 1]
> IsUnipotent(G);
true
>
> G := UnipotentMatrixGroup(G);
> g := GL(4,5)![1,4,4,0, 0,1,3,0, 0,0,1,0, 0,0,0,1];
> g in G;
true
> phi := WordMap(G);
> phi;
Mapping from: GL(4, GF(5)) to SLPGroup(2) given by a rule [no inverse]
>
> assert g in G;
```

```
> wg := phi(g); wg;
function(G)
    w6 := G.1^4; w1 := G.1^-4; w2 := G.2 * w1; w7 := w2^3; w8 := w6 *
    w7; w3 := G.1^-1; w4 := G.1^w2; w5 := w3 * w4; w9 := w5^2; w10 :=
    w8 * w9; return w10;
end function
> Evaluate(wg, G);
[1 4 4 0]
[0 1 3 0]
[0 0 1 0]
[0 0 0 1]
> Evaluate(wg, G) eq g;
true
```

`PCPresentation(G)`

Given a unipotent matrix group G defined over a finite field, the intrinsic constructs a PC-presentation for G. It returns a finite soluble group H as first return value, a map from G to H as the second value, and a map from H to G as the third.

`Order(G)`
`#G`
`FactoredOrder(G)`

Given a unipotent matrix group G defined over a finite field, this intrinsic returns the order of G as an integer or as a factored integer (depending upon the choice of intrinsic). It is faster than the standard matrix group order intrinsic because of the use of the PC-presentation of G.

`g in G`

Given a matrix g and a unipotent matrix group G defined over a finite field, the intrinsic returns `true` if g is an element of G, and `false` otherwise. It is faster than the standard matrix group membership intrinsic because of the use of the PC-presentation of G.

Example H60E15

We construct the PC-presentation of some Sylow subgroup and demonstrate the use of the `FactoredOrder` function.

```
> G := UnipotentMatrixGroup(ClassicalSylow(GL(9,7), 7));
> H,phi,psi := PCPresentation(G);
> phi;
Mapping from: GrpMatUnip: G to GrpPC: H given by a rule [no inverse]
> psi;
Mapping from: GrpPC: H to GrpMatUnip: G
> phi(G.2);
```

```
H.9
> psi(H.3);
[1 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0]
[0 0 1 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 1]
> FactoredOrder(G);
[ <7, 36> ]
```

60.9 Bibliography

[**AMPS10**] S. Ambrose, S.H̃. Murray, C.Ẽ. Praeger, and C. Schneider. Constructive membership testing in black-box classical groups. In *Proceedings of The Third International Congress on Mathematical Software*, number 6327 in Lecture Notes in Computer Science, pages 54–57, Basel, 2010. Springer.

[**Asc84**] M. Aschbacher. On the maximal subgroups of the finite classical groups. *Invent. Math*, 76:469–514, 1984.

[**BHLGO11**] H. Bäärnhielm, Derek Holt, C.R. Leedham-Green, and E.A. O'Brien. A practical model for computation with matrix groups. *preprint*, 2011.

[**Bra00**] J.N. Bray. An improved method of finding the centralizer of an involution. *Arch. Math. (Basel)*, 74(1):241–245, 2000.

[**Cos09**] E. Costi. *Constructive membership testing in classical groups.* PhD thesis, Queen Mary, University of London, 2009.

[**DLLGO13**] Heiko Dietrich, Frank Lübeck, C.R. Leedham-Green, and E.A. O'Brien. Constructive recognition of classical groups in even characteristic. *J. Algebra*, 2013.

[**GH97**] S.P. Glasby and R.B. Howlett. Writing representations over minimal fields. *Comm. Algebra*, 25(6):1703–1711, 1997.

[**GLGO05**] S.P. Glasby, C.R. Leedham-Green, and E.A. O'Brien. Writing projective representations over subfields. *J. Algebra*, 295:51–61, 2005.

[**HLGOR96a**] Derek F. Holt, C.R. Leedham-Green, E.A. O'Brien, and Sarah Rees. Computing decompositions for modules with respect to a normal subgroup. *J. Algebra*, 184:818–838, 1996.

[**HLGOR96b**] Derek F. Holt, C.R. Leedham-Green, E.A. O'Brien, and Sarah Rees. Testing matrix groups for primitivity. *J. Algebra*, 184:795–817, 1996.

[**LG01**] Charles R. Leedham-Green. The computational matrix group project. In *Groups and computation, III (Columbus, OH, 1999)*, volume 8 of *Ohio State Univ. Math. Res. Inst. Publ.*, pages 229–247. de Gruyter, Berlin, 2001.

[**LGO**] C.R. Leedham-Green and E.A. O'Brien. Short presentations for classical groups. *preprint.*

[**LGO97a**] C.R. Leedham-Green and E.A. O'Brien. Recognising tensor products of matrix groups. *Internat. J. Algebra Comput.*, 7:541–559, 1997.

[**LGO97b**] C.R. Leedham-Green and E.A. O'Brien. Tensor Products are Projective Geometries. *J. Algebra*, 189:514–528, 1997.

[**LGO02**] C.R. Leedham-Green and E.A. O'Brien. Recognising tensor-induced matrix groups. *J. Algebra*, 253:14–30, 2002.

[**LGO09**] C.R. Leedham-Green and E.A. O'Brien. Constructive recognition of classical groups in odd characteristic. *J. Algebra*, 322:833–881, 2009.

[**Nie05**] Alice C. Niemeyer. Constructive recognition of normalisers of small extraspecial matrix groups. *Internat. J. Algebra Comput.*, 15:367–394, 2005.

[**NS06**] Max Neunhöffer and Ákos Seress. A data structure for a uniform approach to computations with finite groups. In *ISSAC 2006*, pages 254–261. ACM, New York, 2006.

[**O'B06**] E.A. O'Brien. Towards effective algorithms for linear groups. In *Finite Geometries, Groups and Computation*, pages 163–190. De Gruyer, 2006.

[**O'B11**] E.A. O'Brien. Algorithms for matrix groups. In *Groups St Andrews (Bath)*, volume 388 of *LMS Lecture Notes*, pages 297–323. Cambridge University Press, 2011.

61 MATRIX GROUPS OVER INFINITE FIELDS

Chapter 61
MATRIX GROUPS OVER INFINITE FIELDS

61.1 Overview

In this chapter we provide algorithms for computing with a group G given by a finite set $S = \{g_1, \ldots, g_r\}$ of invertible $n \times n$ matrices over an infinite field K. The algorithms are based on special techniques developed for computing in this class of groups ([DF08, DF09, DFO09, DEF09, DFO12, DFO11]), which rely on properties of finitely generated linear groups.

The group G is defined over the subring R of K generated by the entries of the matrices $g_i, {g_i}^{-1}$, $1 \leq i \leq r$. If ρ is an ideal of R, then it induces a congruence homomorphism from $GL(n, R)$ onto $GL(n, R/\rho)$, which replaces every entry of an element in S by its image in R/ρ. Our techniques depend on the construction of a congruence homomorphism with the property that all torsion elements of its kernel G_ρ (called a congruence subgroup) are unipotent. The existence of a normal subgroup of finite index in G with such a property was proved by Selberg and Wehrfritz. One advantage of the congruence homomorphism techniques is that they replace the ground domain by a domain that is more convenient for computing. In particular, if the ideal ρ is maximal, then we get a reduction to a finite field R/ρ. For more details on the method see [DF08, Section 3].

In this chapter we provide three sets of functions based on the above techniques.

(a) Functions which test finiteness of matrix groups over a wide range of infinite domains. These functions are an implementation of algorithms developed in [DF09, DFO09, DFO12]. Together with other currently available algorithms for deciding finiteness, they enable testing finiteness of a finitely generated linear group over an arbitrary field (subject to special representation of input data). Additionally, if a group is found to be finite, then we can construct an isomorphic copy over a finite field, and use that for further structural investigation of the group.

(b) Functions for testing various properties of infinite matrix groups. These functions test whether G is soluble-by-finite or soluble, nilpotent-by-finite or nilpotent, abelian-by-finite, or central-by-finite. In effect, they provide access to the first publicly available implementations of algorithms to decide the "Tits alternative" for a linear group. If G is soluble-by-finite we can test whether it is completely reducible. These functions are an implementation of algorithms developed in [DFO11].

(c) Functions for testing nilpotency and computing with nilpotent matrix groups. These functions are an implementation of algorithms developed in [DF08], which in turn are based on an implementation of algorithms in [DF06] for computing with nilpotent matrix groups over finite fields. The functions may also be used for investigating the structure of nilpotent matrix groups. In particular, special algorithms have been developed for deciding finiteness of nilpotent matrix groups. Functions are also available to decide irreducibility and primitivity for finite nilpotent matrix groups over number fields and function fields

in zero characteristic; these algorithms, developed and implemented by Tobias Rossmann, are described in [Ros10, Ros11].

Since G is finitely generated, it is defined over a finitely generated subfield of K. Hence, the main fields to be considered are finite degree extensions of $F(x_1, \ldots, x_m)$, where the $x_1, \ldots, x_m$ are algebraically independent indeterminates, $m \geq 0$, and the coefficient field F is an number field or a finite field.

For a recent survey of work in the area of computing with matrix groups over infinite fields, we refer to [DEF09].

Verbose output for these functions can be obtained with `SetVerbose ("Infinite", 1);`

61.2 Construction of Congruence Homomorphisms

In this section, K is a finite degree extension of $F(x_1, \ldots, x_m)$, where F is Q, a number field, or a finite field. Also $m \geq 0$ if $charF = 0$, and $m > 0$ otherwise.

`CongruenceImage(G : parameters)`

`Virtual`	BoolElt	*Default :* `false`
`Prime`	RngIntElt	*Default :* 3
`Limit`	RngIntElt	*Default :* 10
`ExtDegree`	RngIntElt	*Default :* 1

If G is a finitely generated subgroup of $\mathrm{GL}(n, K)$, then G has a normal subgroup N whose torsion elements are unipotent; so N is torsion-free if K has characteristic 0.

This function constructs a *congruence homomorphism* from G into $\mathrm{GL}(n, \mathbf{F}_q)$ for some prime power q; its kernel is N. If $charK$ is positive, then $\mathbf{F}_q$ has the same characteristic.

For a detailed description of the congruence homomorphisms see [DFO12, Section 3]. The function returns the congruence image H, the congruence homomorphism, and the list of images of generators of G.

If the optional parameter `Virtual` is set to `true` then the congruence homomorphism satisfies additional properties [DFO11]. In particular it can be used to test whether G satisfies the "virtual" properties described in Section 61.4.

The optional parameter `Prime` applies if K has characteristic 0: if `Prime` is positive, then it is a lower bound for the characteristic of the congruence image; if it is 0 then the function returns a congruence image defined over a field of characteristic 0.

The optional parameter `Limit` applies to groups defined over (rational) function fields. If $charK > 0$, then we consider extensions of F to degree `Limit` only; otherwise we examine tuples in the ring of integers mod `Limit`.

The optional parameter `ExtDegree` applies to groups defined over (algebraic) function fields of positive characteristic: we construct a congruence image over an extension of (at least) this degree of coefficient field.

61.3 Testing Finiteness

In this section, K is a finite degree extension of the field $F(x_1, \ldots, x_m)$, where F is Q, a number field, or a finite field. Also $m \geq 0$ if $charF = 0$, and $m > 0$ otherwise.

`IsFinite(G : parameters)`

`NumberRandom`	RngIntElt	*Default :* 10
`Presentation`	MonStgElt	*Default :* "*CT*"
`Small`	RngIntElt	*Default :* 10^5
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Algebra`	BoolElt	*Default :* `true`
`Nilpotent`	BoolElt	*Default :* `false`
`UseCongruence`	BoolElt	*Default :* `false`
`DetermineOrder`	BoolElt	*Default :* `false`
`Prime`	RngIntElt	*Default :* 3

Let G be a finitely generated subgroup of $\mathrm{GL}(n, K)$. If G is finite then the function returns `true`, otherwise `false`. The function is an implementation of algorithms from [DFO12, DF09, DFO09, DF08].

The algorithm first tests whether `NumberRandom` random elements of G have finite order.

If the optional parameter `Algebra` is `true` and K is a function field of characteristic zero (resp. positive characteristic), then we use the "algebra algorithm" of [DF09] (resp. [DFO09]) to decide finiteness.

Otherwise, we prove that G is finite by first constructing a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. If $charK = 0$, then the kernel should be trivial, otherwise the kernel is unipotent.

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

If K is Q or a number field and `UseCongruence` is `true`, then use congruence homomorphism machinery to decide; otherwise use default algorithm.

If G is known to be nilpotent then by setting the optional parameter `Nilpotent` to `true`, the function will call a special procedure for testing finiteness of nilpotent groups (see [DF08, Section 4.3]).

If the optional parameter `DetermineOrder` is set to `true`, and G is finite, then the function returns the order of G. This may sometimes be more expensive than deciding finiteness.

The optional parameter `Prime` applies if K has characteristic 0: if `Prime` is positive, then it is a lower bound for the characteristic of the congruence image; if it is 0 then the function constructs a congruence image defined over a field of characteristic 0.

`IsomorphicCopy(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`Small`	RngIntElt	*Default :* 10^5
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Verify`	BoolElt	*Default :* `false`
`Algebra`	BoolElt	*Default :* `false`
`StartDegree`	RngIntElt	*Default :* 1
`EndDegree`	RngIntElt	*Default :* 5
`CompletelyReducible`	BoolElt	*Default :* `false`

The input is a finite subgroup G of $\mathrm{GL}(n, K)$. If the function succeeds, then it returns `true` and an isomorphic copy of G in $\mathrm{GL}(n, \mathbf{F}_q)$ where q is a prime power; otherwise it returns `false`. A description of the method used is in [DFO12, Section 4.3]. If $charK$ is positive, then $\mathbf{F}_q$ has the same characteristic. Note that the function always succeeds if K has zero characteristic.

If the optional parameter `Algebra` is `true` and K is a function field of characteristic zero (resp. positive characteristic), then we use the "algebra algorithm" of [DF09] (resp. [DFO09]) to construct an isomorphic copy.

Otherwise we prove that a congruence homomorphism is an isomorphism by constructing a presentation for the congruence image and evaluating its relations to obtain normal generators for the congruence kernel.

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

If the optional parameter `Verify` is set to `true` then we first check whether G is finite.

If the characteristic of the coefficient field F is positive, then we investigate extensions of F in the range `StartDegree` ... `EndDegree`.

If the optional parameter `CompletelyReducible` is set to `true` then we use a more efficient algorithm to construct the isomorphic copy.

`Order(G : parameters)`

`Verify`	BoolElt	*Default :* `false`
`UseCongruence`	BoolElt	*Default :* `false`

Given a finite subgroup G of $\mathrm{GL}(n, K)$, the function returns the order of G by applying `IsomorphicCopy` to G.

If the optional parameter `Verify` is set to `true`, then we first check that G is finite.

If K is Q or a number field and `UseCongruence` is `true`, then use congruence homomorphism machinery to decide; otherwise use default algorithm.

61.4 Deciding Virtual Properties of Linear Groups

In this section, K is a finite degree extension of $F(x_1, \ldots, x_m)$, where F is Q, a number field, or a finite field. Also $m \geq 0$ if $charF = 0$, and $m > 0$ otherwise.

We describe algorithms to decide various "virtual" properties of a finitely generated linear group over an infinite field. Details of the algorithms can be found in [DFO11].

`IsSolubleByFinite(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over K, and tests whether G is soluble-by-finite. If so, it returns `true`, otherwise `false`. Note that currently the function is valid only for $p > n$ if K has characteristic $p > 0$.

The algorithm first constructs a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. For further details, see [DFO11, Section 3.2].

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

IsPolycyclicByFinite(G : *parameters*)

Presentation	MonStgElt	*Default :* "*CT*"
OrderLimit	RngIntElt	*Default :* 10^{12}
Small	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over Z, and tests whether G is polycyclic-by-finite. If so, it returns `true`, otherwise `false`. See [DFO11, Section 3.2] for details.

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

IsNilpotentByFinite(G : *parameters*)

Presentation	MonStgElt	*Default :* "*CT*"
OrderLimit	RngIntElt	*Default :* 10^{12}
Small	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over K, and tests whether G is nilpotent-by-finite. If so, it returns `true`, otherwise `false`. Here K must currently be Q, a number field, or an (algebraic) function field with a single indeterminate.

The algorithm first constructs a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. Further details of the algorithm can be found in [DFO11, Section 5.2].

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

`IsAbelianByFinite(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over K, and tests whether G is abelian-by-finite. If so, it returns `true`, otherwise `false`. As before, K must currently be Q, a number field, or an (algebraic) function field with a single indeterminate.

The algorithm first constructs a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. Further details of the algorithm can be found in [DFO11, Section 5.2].

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

`IsCentralByFinite(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5
`CompletelyReducible`	BoolElt	*Default :* `false`

This function takes as input a finitely generated matrix group G over a field K, and tests whether G is central-by-finite. If so, it returns `true`, otherwise `false`. Here K is (a finite degree extension of) $F(x_1, \ldots, x_m)$, where F is Q or a number field.

The algorithm first constructs a congruence homomorphism, then a presentation for the congruence image, and finally evaluates its relations to obtain normal generators for the congruence kernel. Further details of the algorithm can be found in [DFO11, Section 5.3].

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional

argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

If the optional parameter `CompletelyReducible` is set to `true` then we use a more efficient algorithm to test whether G is central-by-finite.

61.5 Other Properties of Linear Groups

In this section, K is a finite degree extension of $F(x_1, \ldots, x_m)$, where F is Q, a number field, or a finite field, and $m \geq 0$.

`IsCompletelyReducible(G : parameters)`

`SolubleByFinite`	BoolElt	*Default :* `false`
`NilpotentByFinite`	BoolElt	*Default :* `false`
`AbelianByFinite`	BoolElt	*Default :* `false`
`Nilpotent`	BoolElt	*Default :* `false`
`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5

This function takes as input a finitely generated matrix group G over K, and tests whether G is completely reducible. If so, it returns `true`, otherwise `false`.

The algorithm used is described in [DFO11, Section 4]. It applies only if G is soluble-by-finite, nilpotent-by-finite, or abelian-by-finite. Hence one (and only one) of the four optional arguments `SolubleByFinite, NilpotentByFinite, AbelianByFinite, Nilpotent` must be true. In particular, if `Nilpotent` is set to be true, then a more efficient algorithm (from [DF08]) is used.

In positive characteristic p, if p divides the order of the congruence image of G then currently the algorithm cannot decide complete reducibility of G.

The optional parameter `Presentation` is used to dictate how the presentation is constructed. If its value is "CT", then we use the presentation provided by `CompositionTreeVerify`. If its value is "PC" and the image is soluble, then we use a PC-presentation provided by `LMGSolubleRadical`. If its value is "FP" then we use the presentation provided by `FPGroup` or `FPGroupStrong`. If the order of the congruence image is less than the value of the optional argument `Small`, then we use `FPGroup` to construct the presentation; if it is less than the value of the optional argument `OrderLimit`, then we use `FPGroupStrong` to construct the presentation; otherwise we use the presentation provided by `CompositionTreeVerify`.

`IsUnipotent(G)`

This function takes as input a finitely generated matrix group G defined over an exact field F, and tests whether G is unipotent, i.e., whether it is conjugate in $\mathrm{GL}(n, F)$ to a group of upper unitriangular matrices. If G is unipotent then the function returns `true` and a change-of-basis matrix $c \in \mathrm{GL}(n, F)$ such that G^c is upper unitriangular, otherwise `false`. See [DF06, Section 2.1] for details of the algorithm.

`IsNilpotent(G)`

Let G be a finitely generated subgroup of $\mathrm{GL}(n, K)$. This function returns `true` if G is nilpotent; otherwise it returns `false`. If K is finite then the function is an implementation of the algorithm of [DF06]. If K is infinite then the function is similar to the algorithm in [DF08], and is based on the construction of a homomorphic image H of G via `CongruenceImage`.

`IsSoluble(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5
`UseCongruence`	BoolElt	*Default :* `false`

Let G be a finitely generated subgroup of $\mathrm{GL}(n, K)$. This function returns `true` if G is soluble; otherwise it returns `false`. If K is infinite and has characteristic $p > 0$, then the algorithm is applicable only for $p > n$. For details see [DFO11, Section 3.2].

If K is Q or a number field and `UseCongruence` is `true`, then use congruence homomorphism machinery to decide; otherwise use default algorithm.

The other optional arguments are those described above for `IsSolubleByFinite`.

`IsPolycyclic(G : parameters)`

`Presentation`	MonStgElt	*Default :* "*CT*"
`OrderLimit`	RngIntElt	*Default :* 10^{12}
`Small`	RngIntElt	*Default :* 10^5

This function takes as input a finite matrix group G over Z, and tests whether G is polycyclic. If so, it returns `true`, otherwise `false`.

The optional arguments are those described above for `IsSolubleByFinite`.

`HasFiniteOrder (g : parameters)`

`UseCongruence`	BoolElt	*Default :* `false`

Let g be an invertible matrix defined over Z, Q, a number field, a function field, or an algebraic function field.

If g has finite order, then return `true` and, if known, a multiplicative upper bound for the order of g; else return `false`.

If g is defined over Z, Q, or a number field and `UseCongruence` is `true`, then use congruence homomorphism machinery to decide; otherwise use default algorithm.

61.6 Other Functions for Nilpotent Matrix Groups

`SylowSystem(G : parameters)`

`Verify` BoolElt *Default :* `false`

Given a nilpotent matrix group G over a finite field, this function constructs one Sylow p-subgroup for each prime p dividing $|G|$ using the algorithm of [DF06]. If the optional parameter `Verify` is set to `true`, then we first verify that G is nilpotent.

The next two functions were developed and implemented by Tobias Rossmann.

`IsIrreducibleFiniteNilpotent(G : parameters)`

`DecideOnly` BoolElt *Default :* `false`

`Verify` BoolElt *Default :* `false`

Let G be a finite nilpotent matrix group over K, where K is a number field or a rational function field over a number field. The function returns `true` if G is irreducible or `false` and a proper submodule of `GModule(G)`. The construction of a submodule can be suppressed by setting `DecideOnly` to `true`. If the optional parameter `Verify` is set to `true`, then the function checks if G is nilpotent and finite. The algorithm used for irreducibility testing is described in [Ros10a].

`IsPrimitiveFiniteNilpotent(G : parameters)`

`DecideOnly` BoolElt *Default :* `false`

`Verify` BoolElt *Default :* `false`

Let G be an irreducible finite nilpotent matrix group over K, where K is a number field or a rational function field over a number field. The function returns `true` if G is primitive, or `false` and a system of imprimitivity for G given as a sequence of subspaces of `RSpace(G)`. The construction of a system of imprimitivity can be suppressed by setting `DecideOnly` to `true`. If the optional parameter `Verify` is set to `true`, then the function checks if G is nilpotent and finite. The algorithm used for primitivity testing is described in [Ros10b].

61.7 Examples

Example H61E1

```
> Q := Rationals ();
> F<t>:= RationalFunctionField (Q);
> M:= MatrixAlgebra (F, 3);
> a:= M![-1, 2*t^2, -2*t^4 - 2*t^3 - 2*t^2, 0, 1, 0, 0, 0, 1];
> b:= M![1, 0, 0, 1/t^2, -1, (2*t^3 - 1)/(t - 1), 0, 0, 1];
> c:= M![t, -t^3 + t^2, t^5 - t^2 - t, t^2, -t^4, (t^8 - t^5 + 1)/
> (t^2 - t), (t - 1)/t, -t^2 + t, t^4 - t];
> G:= sub<GL(3,F)|a,b,c>;
> IsFinite(G);
```

```
true
> flag, H := IsomorphicCopy(G);
> H;
MatrixGroup(3, GF(3))
Generators:
    [2 2 1]
    [0 1 0]
    [0 0 1]

    [1 0 0]
    [1 2 0]
    [0 0 1]

    [2 2 2]
    [1 2 0]
    [2 1 2]
> #H;
48
```

Example H61E2

```
> F<t>:= RationalFunctionField (GF(5));
> M:= MatrixAlgebra (F, 6);
> a:= M![2, 2*t^2, 4, 1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0,
> 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1];
> b:= M![(4*t + 4)/t, 4*t, (t + 1)/t, 0, t, t^2 + t, 0, 4, 0, 0, 0,
> 1/t, 4/t, t^2 + 4*t, 1/t, 0, 0, 0, 0, 4*t, 0, 0, 0, 0, 0, 0, 4, 4,
> 0, 0, 0, 0, 0, 4, 0, 0];
> G:= sub<GL(6,F)|a,b>;
> IsFinite(G);
true
> flag, H := IsomorphicCopy (G);
> flag;
true
> H;
MatrixGroup(6, GF(5)) of order 2^7 * 3 * 5^4 * 31
Generators:
    [2 2 4 1 0 0]
    [0 2 0 0 0 0]
    [0 0 1 1 0 0]
    [0 0 0 1 0 0]
    [0 0 0 0 1 1]
    [0 0 0 0 0 1]

    [3 4 2 0 1 2]
    [0 4 0 0 0 1]
    [4 0 1 0 0 0]
```

```
    [0 4 0 0 0 0]
    [0 0 4 4 0 0]
    [0 0 0 4 0 0]
> #H;
7440000
```

Example H61E3

```
>  L<t> := RationalFunctionField (GF (5^2));
>  G := GL (2, L);
>  a := G![t,1,0,-1];
>  b:= G![t/(t + 1), 1, 0, 1/t];
>  H := sub <GL(2, L) | a, b>;
> f :=IsFinite(H);
> f;
false
> IsSolubleByFinite (H);
true
> IsCompletelyReducible (H);
false
```

Example H61E4

```
> G := MatrixGroup<3, IntegerRing() |
>  [ 5608, 711, -711, 6048, 766, -765, 1071, 135, -134 ],
>  [ 1, -2415, 5475, 0, 4471, -10140, 0, 780, -1769 ],
>  [ 5743, -5742, 639, -576, 577, -72, -711, 711, -80 ],
>  [ 526168, -618507, 729315, 621984, -731138, 862125,
>        274455, -322620, 380419      ]  ,
>  [ 648226, -4621455, 9226791, 660687, -4710305, 9404184,
>        85626, -610473, 1218820 ],
>  [ 32581, -39465, 46350, 53100, -64319, 75540, 24210,
>        -29325, 34441 ]>;
> IsFinite (G);
false
> IsSolubleByFinite (G);
false
> IsNilpotentByFinite (G);
false
> time IsCentralByFinite (G);
false
> IsAbelianByFinite (G);
false
```

Example H61E5

```
> Q<z> := QuadraticField(5);
> O<w> := sub< MaximalOrder(Q) | 7 >;
> G := GL(2, Q);
> x := G![1,1+w,0,w];
> y :=  G![-1/2, 2, 2 + w, 5 + w^2];
> H:=sub<G | x, y>;
> IsFinite (H);
false
> IsSolubleByFinite (H);
false
```

Example H61E6

```
>  R<x> := PolynomialRing(Integers());
>  K<y> := NumberField(x^4-420*x^2+40000);
>  G := GL (2, K);
>  a := G![y,1,0,-1];
>  b:= G![y/(y + 1), 1, 0, 1/y];
>  H := sub <GL(2, K) | a, b>;
> time IsFinite(H);
false
```

Example H61E7

```
> /* example over algebraic extension of a function field */
>
> R<u> := FunctionField (Rationals ());
> v := u; w := -2 * v;
> Px<X> := PolynomialRing (R);
> Py<Y> := PolynomialRing (R);
> f :=  Y^2- 3 * u * X * Y^2  + v * X^3;
> facs := Factorisation (f);
> F:=ext <R | facs[2][1]>;
> F;
Algebraic function field defined over Univariate rational function
field over Rational Field by Y - 1/2/u
>
> n := 3;
> G:= GL(n,F);
> Z := 4 * X * Y;
> MA:= MatrixAlgebra(F,n);
> h1:= Id(MA);
> h1[n][n]:= (X^2+Y+Z+1);
> h1[1][n]:= X+1;
```

```
> h1[1][n]:= X+1;
> h1[1][1]:=(Z^5-X^2*Z+Z*X*Y);
> h1[2][1]:=1-X*Y*Z;
> h1[2][n]:= X^20+X*Y^15+Y^10+Z^4*Y*X^5+1;
> h2:= Id(MA);
> h2[n][n]:= (X^7+Z^6+1);
> h2[1][n]:= X^2+X+1;
> h2[1][1]:=(Y^3+X^2+X+1);
> h2[1][1]:=(Y^3+X^2+X+1);
> h2[2][1]:=1-X^2;
> h2[2][n]:= X^50+Y^35+X^20+X^13+Y^2+1;
> G := sub< GL(n, F) | h1, h2>;
> G;
MatrixGroup(3, F)
Generators:
    [1/u^10 0 (u + 1/2)/u]
    [(u^4 - 1/4)/u^4 1 (u^20 + 1/1024*u^10 + 1/64*u^6 + 1/65536*u^4 +
        1/1048576)/u^20]
    [0 0 (u^2 + 1/2*u + 5/4)/u^2]

    [(u^3 + 1/2*u^2 + 1/4*u + 1/8)/u^3 0 (u^2 + 1/2*u + 1/4)/u^2]
    [(u^2 - 1/4)/u^2 1 (u^50 + 1/4*u^48 + 1/8192*u^37 + 1/1048576*u^30 +
        1/34359738368*u^15 + 1/1125899906842624)/u^50]
    [0 0 (u^12 + 1/128*u^5 + 1)/u^12]
> time IsFinite(G);
false
Time: 0.010
> time IsSolubleByFinite (G);
true
```

Example H61E8

```
> F := GF(2);
> P := PolynomialRing (F);
> P<t> := PolynomialRing (F);
> F := ext < F | t^2+t+1>;
> G := GL (2, FunctionField (F));
> a := G![1,1/t, 0, 1];
> b := [1,1/(t + 1), 0, 1];
> c := [1,1/(t^2 + t + 1), 0, 1];
> d := [1,1/(t^2 + t), 0, 1];
> G := sub < G | a,b,c,d>;
> time IsFinite (G);
true
> f, I, tau := IsomorphicCopy (G);
> f;
```

```
true
```

Example H61E9

```
> // irreducible but (evidently) imprimitive
> K<w> := QuadraticField (2);
> G := MatrixGroup< 8, K |
>     [1/2*w,1/2*w,0,0,0,0,0,0,-1/2*w,1/2*w,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,
>      0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
>      0,0,0,1],
>     [1,0,0,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,
>      0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1],
>     [0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,
>     0,0,0,0,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0] >;
> G;
MatrixGroup(8, K)
Generators:
    [ 1/2*w  1/2*w        0        0        0        0        0        0]
    [-1/2*w  1/2*w        0        0        0        0        0        0]
    [       0        0        1        0        0        0        0        0]
    [       0        0        0        1        0        0        0        0]
    [       0        0        0        0        1        0        0        0]
    [       0        0        0        0        0        1        0        0]
    [       0        0        0        0        0        0        1        0]
    [       0        0        0        0        0        0        0        1]

    [ 1  0  0  0  0  0  0  0]
    [ 0 -1  0  0  0  0  0  0]
    [ 0  0  1  0  0  0  0  0]
    [ 0  0  0  1  0  0  0  0]
    [ 0  0  0  0  1  0  0  0]
    [ 0  0  0  0  0  1  0  0]
    [ 0  0  0  0  0  0  1  0]
    [ 0  0  0  0  0  0  0  1]

    [0 0 1 0 0 0 0 0]
    [0 0 0 1 0 0 0 0]
    [0 0 0 0 1 0 0 0]
    [0 0 0 0 0 1 0 0]
    [0 0 0 0 0 0 1 0]
    [0 0 0 0 0 0 0 1]
    [1 0 0 0 0 0 0 0]
    [0 1 0 0 0 0 0 0]
> IsIrreducibleFiniteNilpotent(G);
true
> r, B := IsPrimitiveFiniteNilpotent(G);
> r;
```

```
false
> #B;
2
```

Example H61E10

```
> M:= MatrixAlgebra (GF(17), 4);
> a:= M![5, 5, 3, 3, 0, 5, 0, 3, 16, 16, 14, 14, 0, 16, 0, 14];
> b:= M![9, 9, 0, 0, 0, 9, 0, 0, 10, 10, 8, 8, 0, 10, 0, 8];
> G:= sub<GL(4,17)|a,b>;
> IsNilpotent(G);
true
> SylowSystem (G);
[
    MatrixGroup(4, GF(17))
    Generators:
        [ 5  0  3  0]
        [ 0  5  0  3]
        [16  0 14  0]
        [ 0 16  0 14]

        [ 9  0  0  0]
        [ 0  9  0  0]
        [10  0  8  0]
        [ 0 10  0  8],

    MatrixGroup(4, GF(17))
    Generators:
        [ 1  1  0  0]
        [ 0  1  0  0]
        [ 0  0  1  1]
        [ 0  0  0  1]
]
> Order(G);
8704
```

Example H61E11

```
> R<s>:= QuadraticField(-1);
> F<t>:= FunctionField(R);
> M:= MatrixAlgebra (F, 2);
> a:= M![-s*t^2 + 1, s*t^3, -s*t, s*t^2 + 1];
> b:= M![t^2 - 3*t + 1, 0, 0, t^2 - 3*t + 1];
> G:= sub<GL(2,F)|a,b>;
> IsNilpotent(G);
true
```

```
> IsFinite(G);
false
```

61.8 Bibliography

[**DEF09**] A. S. Detinko, B. Eick, and D. L. Flannery. Computing with matrix groups over infinite fields. *preprint (15pp.)*, 2009.

[**DF06**] A. S. Detinko and D. L. Flannery. Computing in nilpotent matrix groups. *LMS J. Comput. Math.*, 9:104–134 (electronic), 2006.

[**DF08**] A. S. Detinko and D. L. Flannery. Algorithms for computing with nilpotent matrix groups over infinite domains. *J. Symbolic Comput.*, 43:8–26, 2008.

[**DF09**] A. S. Detinko and D. L. Flannery. On deciding finiteness of matrix groups. *J. Symbolic Comput.*, 44:1037–1043, 2009.

[**DFO09**] A. S. Detinko, D. L. Flannery, and E. A. O'Brien. Deciding finiteness of matrix groups in positive characteristic. *J. Algebra*, 322:4151–4160, 2009.

[**DFO11**] A. S. Detinko, D. L. Flannery, and E. A. O'Brien. Algorithms for the Tits alternative and related problems. *J. Algebra*, 344:397–406, 2011.

[**DFO12**] A. S. Detinko, D. L. Flannery, and E. A. O'Brien. Recognition of finite matrix groups over infinite fields. *J. Symbolic Comput.*, 2012.

[**Ros10**] T. Rossmann. Irreducibility testing of finite nilpotent linear groups. *J. Algebra*, 324:1114–1124, 2010.

[**Ros11**] T. Rossmann. Primitivity testing of finite nilpotent linear groups. *LMS JCM*, 14:87–98, 2011.

62 MATRIX GROUPS OVER Q AND Z

Chapter 62
MATRIX GROUPS OVER Q AND Z

62.1 Overview

In addition to the functionality explained in Chapter 61 and the functions that are available for all finite (matrix) groups, MAGMA can also compute normalizers and centralizers of a finite integral matrix group G in $\mathrm{GL}_n(\mathbf{Z})$ as well as decide conjugacy in $\mathrm{GL}_n(\mathbf{Z})$ and $\mathrm{GL}_n(\mathbf{Q})$.

These algorithms are based on the sublattice machinery (see Section 31.3.5) and the enumeration of G-perfect forms. They are explained in [OPS98, Opg01]. The algorithms perform very well, as long as the space of G-invariant symmetric forms has small dimension (say less than 15) and the index of the groups in their Bravais groups is not too large.

The databases of maximal finite irreducible rational, integral, symplectic and quaternionic matrix groups are explained in Chapter 66.

62.2 Invariant Forms

Let G be a finite matrix group $G < \mathrm{GL}_n(\mathbf{Q})$. A matrix $F \in M_n(\mathbf{Q})$ is G-invariant if $gFg^{tr} = F$ for all $g \in G$.

`PositiveDefiniteForm(G)`

For a finite integral or rational matrix group G, return a positive definite symmetric G-invariant form.

`InvariantForms(G)`
`SymmetricForms(G)`
`AntisymmetricForms(G)`

For an integral or rational matrix group G, return a basis for the space of G-linear forms or for the subspace of (anti-) symmetric forms respectively.

The first form returned by `InvariantForms` and `SymmetricForms` will be positive definite.

`InvariantForms(G, n)`
`SymmetricForms(G, n)`
`AntisymmetricForms(G, n)`

For an integral or rational matrix group G, return a sequence consisting of $n \geq 0$ G-invariant (symmetric or antisymmetric) bilinear forms for G.

`NumberOfInvariantForms(G)`

`NumberOfSymmetricForms(G)`

`NumberOfAntisymmetricForms(G)`

For an integral or rational matrix group G or a G-lattice L, return the dimension of the space of (symmetric or anti-symmetric) invariant bilinear forms for G.

The algorithm uses a modular method which is much faster than the actual computation of the forms.

62.3 Endomorphisms

`EndomorphismRing(G)`

For an integral or rational matrix group G, return the endomorphism ring (i.e. the commuting algebra) of G as a subalgebra of $M_n(\mathbf{Z})$ or $M_n(\mathbf{Q})$ respectively.

`CentreOfEndomorphismRing(G)`

`CentreOfEndomorphismAlgebra(G)`

For an integral or rational matrix group G, return the center of the endomorphism ring (i.e. the commuting algebra) of G as a subalgebra of $M_n(\mathbf{Z})$ or $M_n(\mathbf{Q})$ respectively.

`DimensionOfEndomorphismRing(G)`

Return the dimension of the endomorphism ring of an integral or rational matrix group G by a modular method.

`DimensionOfCentreOfEndomorphismRing(G)`

Return the dimension of the centre of the endomorphism ring of an integral or rational matrix group G by a modular method.

`Endomorphisms(G, n)`

For an integral or rational matrix group G, return a sequence containing n independent endomorphisms of G. n must be in the range $[0..d]$, where d is the dimension of the endomorphism ring of G.

`CentralEndomorphisms(G, n)`

For an integral or rational matrix group G, return a sequence containing n independent central endomorphisms of G. n must be in the range $[0..d]$, where d is the dimension of the centre of the endomorphism ring of G.

62.4 New Groups From Others

`BravaisGroup(G)`

For a finite integral matrix group G, compute its Bravais group which is the integral group fixing all symmetric bilinear forms fixed by G.

`IntegralGroup(G)`

Return the action of the finite rational matrix group G on an invariant lattice as an integral matrix group, thus giving an equivalent integral group H, together with the transformation matrix T from the standard lattice to the invariant lattice. Thus $H = T \cdot G \cdot T^{-1}$.

62.5 Perfect Forms and Normalizers

`PerfectForms(G)`

`Limit` RngIntElt *Default :* ∞

A positive definite symmetric G-invariant form F is called G-perfect if for every nonzero symmetric G-invariant form F' there exists some shortest vector x of F such that $F'x^{tr}x$ has nonzero trace.

The normalizer of the Bravais group of G in $\mathrm{GL}_n(\mathbf{Z})$ acts on the set of integral G-perfect forms whose entries have GCD 1 and the number of orbits is finite. This function returns a sequence of representatives of these orbits.

If `Limit` is set to a positive integer m, then the algorithm stops after m orbits have been enumerated.

`NormalizerGLZ(G)`

`CentralizerGLZ(G)`

`IsBravais` BoolElt *Default :* `false`

Given a finite subgroup G of $\mathrm{GL}_n(\mathbf{Z})$, returns the normalizer or centralizer of G in $\mathrm{GL}_n(\mathbf{Z})$.

If G is know to be equal to its Bravais group, one can set `IsBravais` to `true` to speed up the computation.

The algorithm employed is a variation of Opgenorth's normalizer algorithm [Opg01].

62.6 Conjugacy

`ZClasses(G)`

`Homogeneously` BoolElt *Default :* `false`

Given a finite integral or rational matrix group G, its $\mathrm{GL}_n(\mathbf{Q})$-conjugacy class splits into finitely many $\mathrm{GL}(n, \mathbf{Z})$-conjugacy classes. Representatives of these classes are constructed as the action of G on some G-invariant sublattices. More precisely, the $\mathrm{GL}(n, \mathbf{Z})$-conjugacy classes are in bijection with the orbits of G-invariant lattices under the normalizer N of G in $\mathrm{GL}(n, \mathbf{Q})$.

A G-lattice L' belongs to a G-lattice L if $L = \sum_i L'e_i$ where $e_1, \ldots, e_r$ denote the central idempotents of the endomorphism ring of G. Further, L is called *homogeneously decomposable* if L belongs to itself.

The algorithm will first compute representatives $L_1, \ldots, L_k$ of the orbits of homogeneously decomposable G-lattices under the action of N.

In a second step, it will then compute the G-lattices $L_{i,j}$ belonging to L_i up to the action of N.

The second return value will then consist of a sequence of k sequences $T_1, \ldots, T_k$. The first element $T_i[1]$ is the basis matrix of L_i, the following entries are basis matrices of the lattices $L_{i,j}$.

The first return value is a sequence of integral matrix groups describing the action of G on the lattices $L_{1,1}, L_{1,2}, \ldots$. Hence these groups correspond to the $\mathrm{GL}_n(\mathbf{Z})$-conjugacy classes of G.

If `Homogeneously` is set to `true`, the function will only compute the homogeneously decomposable lattices $L_1, \ldots, L_k$ and the corresponding matrix groups. (If G is reducible, this option is much faster, but will not yield all conjugacy classes / orbits of lattices.)

`IsGLZConjugate(G, H)`

Tests whether the finite integral matrix groups G and H are conjugate in $\mathrm{GL}_n(\mathbf{Z})$. If so, a matrix x such that $G^x = H$ is also returned.

`IsBravaisEquivalent(G, H)`

Given two finite integral matrix groups G and H, tests whether their Bravais groups $B(G)$ and $B(H)$ are conjugate in $\mathrm{GL}_n(\mathbf{Z})$. If so, a matrix x such that $B(G)^x = B(H)$ is also returned.

Note that this function does not need to compute the Bravais groups and hence it is faster than calling `IsGLZConjugate` on the Bravais groups directly.

If G and H are known to be Bravais groups, this function is usually more efficient than calling `IsGLZConjugate`.

`IsGLQConjugate(G, H)`

`Al` MonStgElt *Default :*

Tests whether the finite rational matrix groups G and H are conjugate in $\mathrm{GL}_n(\mathbf{Q})$. If so, a matrix x such that $G^x = H$ is also returned.

There are currently two algorithms available. If the optional parameter `Al` equals `"Aut"`, Magma will use the `GModule`-machinery together with the outer automorphism group of H. If `Al` is set to `"ZClasses"`, Magma splits the $\mathrm{GL}(n, \mathbf{Q})$-conjugacy class of H into $\mathrm{GL}_n(\mathbf{Z})$-conjugacy classes and then decides whether an integral copy of G lies in one of these classes by several calls to `IsGLZConjugate`.

If `Al` is not provided, a sensible choice is made by the system.

62.7 Conjugacy Tests for Matrices

Given two $n \times n$ matrices A and B with rational or integral entries, Magma can test whether A is conjugate to B in $\mathrm{GL}_n(Z)$.

Currently, the implementation is limited to the cases where A, B have finite order or where $n = 2$. This limitation will be removed in future versions.

`IsGLZConjugate(A, B)`

`IsSLZConjugate(A, B)`

Tests whether two rational or integral matrices A and B are conjugate in $\mathrm{GL}_n(\mathbf{Z})$ or $\mathrm{SL}_n(\mathbf{Z})$. If so, a matrix x such that $A^x = B$ is also returned.

`CentralizerGLZ(A)`

Given a rational or integral matrix A, this function returns its centralizer in $\mathrm{GL}_n(\mathbf{Z})$. The current implementation is limited to the cases where either A has finite order or A is a 2×2 matrix.

62.8 Examples

Example H62E1

We split the $\mathrm{GL}_3(\mathbf{Q})$-conjugacy class of the following faithful representation of the dihedral group with 12 elements.

```
> G := MatrixGroup< 3, Integers() |
>  [ 1, -1, 0, 0, -1, 0, 0, 0, 1 ],
>  [ 1, -1, 0, 1, 0, 0, 0, 0, -1 ] >;
> Z, T:= ZClasses(G);
> #Z;
3
> < #t : t in T >;
```

```
<1, 2>
```

So there are 2 classes of homogeneously decomposable lattices represented by `T[1,1]` and `T[2,1]`. The third lattice `T[2,2]` belongs to `T[2,1]` as we check.

```
> Q := Rationals();
> GQ := ChangeRing(G, Q);
> Ids := CentralIdempotents(EndomorphismRing(GQ));
> L := VerticalJoin([ Matrix(Integers(), T[2,2] * i) : i in Ids]);
> Image(L) eq Image(Matrix(Integers(), T[2,1]));
true
```

Finally, we check that the 3 $\mathrm{GL}_3(\mathbf{Z})$-conjugacy classes stored in `Z` correspond to the 3 lattices in `T`.

```
> TT := &cat T;
> [ GQ eq ChangeRing(Z[i], Q)^(GL(3, Q) ! TT[i]) : i in [1..#Z] ];
[ true, true, true ]
```

Example H62E2

We test that the automorphism groups of the lattices B_8 and D_8 are conjugate in $\mathrm{GL}_8(\mathbf{Q})$ but not in $\mathrm{GL}_8(\mathbf{Z})$.

```
> G := AutomorphismGroup( Lattice("B", 8) );
> H := AutomorphismGroup( Lattice("D", 8) );
> ok, x := IsGLQConjugate(G, H); ok, x;
true
[ 1 -1  0  0  0  0  0  0]
[ 1 -1 -2  0  0  0  0  0]
[-1  1  2  2  2  2  2  2]
[ 1  1  0  0  0  0  0  0]
[-1  1  2  2  2  2  2  0]
[ 1 -1 -2 -2 -2  0  0  0]
[-1  1  2  2  2  2  0  0]
[-1  1  2  2  0  0  0  0]
> Determinant(x);
-128
> IsGLZConjugate(G,H);
false
```

Example H62E3

Let C be the companion matrix of the fifth cyclotomic polyomial. We find a unimodular matrix that induces the automorphism $C -> C^2$.

```
> C:= CompanionMatrix(CyclotomicPolynomial(5));
> ok, h:= IsGLZConjugate(C, C^2); ok;
true
> C^2 eq h^-1 * C * h;
```

```
true
```

We now check by hand that this automorphism cannot be realized by a matrix of determiant 1.

```
> Determinant(h);
-1
> G:= CentralizerGLZ(C);
> [ Determinant(g) : g in Generators(G) ];
[1, 1, 1]
```

Of course, we could also just ask:

```
> IsSLZConjugate(C, C^2);
false
```

62.9 Bibliography

[**Opg01**] J. Opgenorth. Dual Cones and the Voronoi Algorithm. *Exp. Math.*, 10(4):599–608, 2001.

[**OPS98**] J. Opgenorth, W. Plesken, and T. Schulz. Crystallographic Algorithms and Tables. *Acta Crystallographica*, A54:517–531, 1998.

63 FINITE SOLUBLE GROUPS

Chapter 63
FINITE SOLUBLE GROUPS

63.1 Introduction

Any finite soluble group has a subnormal series with cyclic factors. Such a series gives rise to various polycyclic presentations. These polycyclic presentations are useful because the word problem in such presentations can be solved in an algorithmic fashion. In MAGMA, we use the specific form called a *power-conjugate presentation (pc-presentation)*, which is described below. The MAGMA category of groups represented by a power-conjugate presentation (pc-groups for short) is called `GrpPC`.

This chapter describes how to use polycyclic presentations to compute with p-groups and other finite soluble groups in MAGMA. While most functions apply to any soluble group, a small number of functions specific to p-groups are identified in the text.

Over the past two decades a considerable body of efficient algorithms has been developed for computing with soluble groups defined in terms of pc-presentations. It is recommended that the `GrpPC` representation of a soluble group be used whenever intensive calculation with that group is necessary.

63.1.1 Power-Conjugate Presentations

Let G be a finite soluble group. A presentation for G of the form

$$< a_1, \ldots, a_n \mid a_j^{p_j} = w_{jj}, \quad 1 \leq j \leq n, \quad a_j^{a_i} = w_{ij}, \quad 1 \leq i < j \leq n >$$

where

(i) p_j is the least prime such that $a_j^{p_j} \in < a_{j+1}, \ldots, a_n >$ for $j < n$, and $a_j^{p_j}$ is the identity for $j = n$, and

(ii) w_{ij} is a word in the generators $a_{i+1}, \ldots, a_n$, will be called a *power-conjugate presentation (pc-presentation)* for G. The generators of G corresponding to $a_1, \ldots, a_n$ in this presentation are known as a *power-conjugate generating sequence (pc-generators)* for G.

It is easy to show that every finite soluble group possesses a pc-presentation. If such a presentation satisfies a certain additional condition (the *consistency condition*) then every element a of G can be written uniquely in the *normal form*

$$a_1^{\alpha_1} \ldots a_n^{\alpha_n}, 0 \leq \alpha_i < p_i \quad \text{for } i = 1, \ldots, n.$$

Given such a pc-presentation for G there exists an algorithm (the *collection algorithm*), which given an arbitrary word in the pc-generators $a_1, \ldots, a_n$, will determine the corresponding normal word. In particular, collection can be used to compute the normal word which is equal to the product of two given normal words, thus implementing the group multiplication.

63.2 Creation of a Group

A user can create a GrpPC representation of a finite soluble group in a variety of ways. There are several built-in construction functions for creating standard examples such as cyclic or dihedral groups. For greater flexibility, it is possible to define a group directly from a power-commutator presentation. One can also build new groups out of old groups using standard constructions such as direct product. Finally, there are several conversion functions which will automatically compute a pc-presentation for an existing soluble group in some other category (such as permutation group or matrix group). We will start with the first two styles of construction and describe the remaining two in later sections.

In each case, regardless of how the group was originally defined, MAGMA will store the group internally as a pc-presentation and will display the pc-presentation whenever the group is printed. Normally when printing a pc-presentation, trivial conjugate relations are omitted. In the case of a p-group, then trivial power relations (those indicating that a generator has order p) are also omitted. The one exception to this policy is in the case of elementary abelian p-groups (which would have no relations displayed under the above policies). In the elementary abelian case, MAGMA will display the power relations, even though they are trivial.

63.2.1 Construction Functions

The simplest method of producing a pc-presentation for a group is to use one of the built-in construction functions. By specifying the category GrpPC as the first parameter of each function, we produce the desired representation.

It is also possible to obtain a pc-presentation for many small soluble groups by using the function SmallGroup described in Chapter 66.

`CyclicGroup(GrpPC, n)`

The cyclic group of order n as a pc-group.

`AbelianGroup(GrpPC, Q)`

Construct the abelian group defined by the sequence $Q = [n_1, \ldots, n_r]$ of positive integers as a pc-group. The function returns the abelian group which is the direct product of the cyclic groups $C_{n_1} \times C_{n_2} \times \cdots \times C_{n_r}$.

`DihedralGroup(GrpPC, n)`

The dihedral group of order $2 * n$ as a pc-group.

`ExtraSpecialGroup(GrpPC, p, n : parameters)`

Given a small prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1} in the category GrpPC. The isomorphism type of G may be selected using the parameter Type.

Type MONSTGELT *Default :* "+"

Possible values for this parameter are "+" (default) and "−".

If `Type` is set to "+", the function returns, for $p = 2$, the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If `Type` is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

Example H63E1

A pc-representation for the cyclic group C_{12} can be computed as follows.

```
> G := CyclicGroup(GrpPC, 12);
```

We can then check various properties of G.

```
> Order(G);
12
> IsAbelian(G);
true
> IsSimple(G);
false
```

If we simply print G, we will see the presentation which MAGMA has generated for this group.

```
> G;
GrpPC : G of order 12 = 2^2 * 3
PC-Relations:
    G.1^2 = G.2,
    G.2^2 = G.3,
    G.3^3 = Id(G)
```

Or, we could build a slightly different group.

```
> H := AbelianGroup(GrpPC, [2,2,3]);
> Order(H);
12
> IsCyclic(H);
false
```

63.2.2 Definition by Presentation

While the standard construction functions are convenient, most groups cannot be defined in that way. Complete flexibility in defining a soluble group can be obtained by directly specifying the group's pc-presentation.

One uses a power-conjugate presentation to define a soluble group by means of the `PolycyclicGroup` constructor, or the `quo` constructor for finitely presented groups.

`PolycyclicGroup< x1, ..., xn | R : parameters >`

`Check`	BOOLELT	*Default :* `true`
`ExponentLimit`	RNGINTELT	*Default :* 20
`Class`	MONSTGELT	*Default :*

Construct the soluble group G defined by the power-conjugate presentation $< x_1, \ldots, x_n \mid R >$.

The construct $x_1, \ldots, x_n$ defines names for the generators of G that are local to the constructor, i.e. they are used when writing down the relations to the right of the bar. However, no assignment of *values* to these variables is made. If the user wants to refer to the generators by these (or other) names, then the *generators assignment* construct must be used on the left hand side of an assignment statement.

The construct R denotes a list of pc-relations. Thus, an element of R must be one of:

(a) A *power relation* $a_j^{p_j} = w_{jj}$, $1 \leq j \leq n$, where w_{jj} is 1 or a word in generators $a_{j+1}, \ldots, a_n$ for $j < n$, and $w_{jj} = 1$ for $j = n$, and p_j a prime.

(b) A *conjugate relation* $a_j^{a_i} = w_{ij}$, $1 \leq i < j \leq n$, where w_{ij} is a word in the generators $a_{i+1}, \ldots, a_n$.

(c) A *power* $a_j^{p_j}$, $1 \leq j \leq n$ and p_j a prime, which is treated as the *power relation* $a_j^{p_j} = Id(F)$.

(d) A set of (a) – (c).

(e) A sequence of (a) – (c).

Note the following points:

(i) A power relation must be present for each generator a_i, $i = 1, \ldots, n$;

(ii) Conjugate relations involving commuting generators (i.e. of the form $y^x = y$) may be omitted;

(iii) The words w_{ij} must be in normal form.

In addition, one can alternatively specify a power-commutator presentation using commutator relations rather than conjugate relations.

(b') A *commutator relation* $(a_j, a_i) = w_{ij}$, $1 \leq i < j \leq n$, where w_{ij} is a word in the generators $a_{i+1}, \ldots, a_n$. However, commutators and conjugates cannot be mixed in a single presentation.

A map f from the free group of rank n to G is returned as well.

The parameters `Check` and `ExponentLimit` may be used. `Check` indicates whether or not the presentation is checked for consistency. `ExponentLimit` determines the amount of space that will be used by the group to speed calculations. Given `ExponentLimit := e`, the group will precompute and store normal words for appropriate products $a^i * b^j$ where a and b are generators and i and j are in the range 1 to e.

If the construction of an object in the category `GrpPC` fails because R is not a valid power-conjugate presentation, an attempt is made to construct a group in the

category GrpGPC (cf. Chapter 72). This feature can be turned off by setting the parameter Class to "GrpPC"; an invalid power-conjugate presentation then causes a runtime error. Since, by default, the constructor always returns a group in the category GrpPC if possible, this is the only effect of setting the parameter Class to "GrpPC".

quo< GrpPC : F | R : *parameters* >

Check	BoolElt	*Default :* true
ExponentLimit	RngIntElt	*Default :* 20

Given a free group F of rank n with generating set X, and a collection R of pc-relations on X, construct the soluble group G defined by the power-conjugate presentation $< X|R >$.

The construct R denotes a list of pc-relations. The syntax and semantics for the relations clause is identical to that appearing in the PolycyclicGroup-construct.

This constructor returns a pc-group because the category GrpPC is stated. If no category were stated, it would return an fp-group.

The parameters Check and ExponentLimit may be used as described in the PolycyclicGroup-construct.

The natural homomorphism, $F \to G$, is also returned.

Example H63E2

Consider the group of order 80 defined by the presentation

$$< a, b, c, d, e \mid a^2 = c, b^2, c^2 = e, d^5, e^2, b^a = b * e, c^a = c, c^b = c,$$
$$d^a = d^2, d^b = d, d^c = d^4, e^a = e, e^b = e, e^c = e, e^d = e > .$$

Giving the relations in the form of a list, this presentation would be specified as follows:

```
> G<a,b,c,d,e> := PolycyclicGroup<a, b, c, d, e |
>  a^2 = c, b^2, c^2 = e, d^5, e^2,
>  b^a = b*e, d^a = d^2, d^c = d^4 >;
```

Starting from a free group and giving the relations in the form of a set of relations, this presentation would be specified as follows:

```
> F<a,b,c,d,e> := FreeGroup(5);
> rels := { a^2 = c, b^2 = Id(F), c^2 = e, d^5 = Id(F), e^2 = Id(F),
>  b^a = b*e, d^a = d^2, d^c = d^4 };
> G<a,b,c,d,e> := quo< GrpPC : F | rels >;
```

Notice that here we have redefined the variables $a, \ldots, e$ to be the pc-generators in G. Thus, when G is printed, Magma displays the following presentation:

```
> G;
GrpPC : G of order 80 = 2^4 * 5
PC-Relations:
    a^2 = c,
```

```
    b^2 = Id(G),
    c^2 = e,
    d^5 = Id(G),
    e^2 = Id(G),
    b^a = b * e,
    d^a = d^2,
    d^c = d^4
> Order(G);
80
> IsAbelian(G);
false
```

63.2.3 Possibly Inconsistent Presentations

The `PolycyclicGroup` and `quo` constructors accept a parameter `Check` which enables the user to suppress the automatic consistency checking for input presentations. This is primarily intended to be used when it is certain that the input presentation is consistent, in order to save time. For instance, the presentation may have been generated from some other reliable program, or even from an earlier MAGMA session. This parameter should be used with care, since all of the MAGMA functions assume that every `GrpPC` group is consistent. The user will encounter numerous bizarre results if an attempt is made to compute with an inconsistent presentation.

On occasion, a user may wish to "try out" a series of pc-presentations, some of which may not be consistent. The `Check` parameter can be used, along with the function `IsConsistent`, to test a presentation for consistency.

`IsConsistent(G)`

Returns `true` if G has a consistent presentation, `false` otherwise.

Example H63E3

The following example demonstrates generating a family of presentations, and then checking consistency. Of course, it is easy to predict the outcome in this simple example.

```
> F := FreeGroup(2);
> for p in [n: n in [3..10] | IsPrime(n)] do
>   r := [F.1^3=Id(F), F.2^p=Id(F), F.2^F.1=F.2^2];
>   G := quo<GrpPC: F | r: Check:=false>;
>   if IsConsistent(G) then
>     print "For p=",p," the group is consistent.";
>   else
>     print "For p=",p," the group is inconsistent.";
>   end if;
> end for;
For p= 3  the group is inconsistent.
For p= 5  the group is inconsistent.
```

```
For p= 7  the group is consistent.
```

63.3 Basic Group Properties

63.3.1 Infrastructure

The functions described here provide access to basic information stored for a pc-group G.

`G . i`

The i-th pc-generator for G. A negative subscript indicates that the inverse of the generator is to be created. $G.0$ is `Identity(G)`.

`Generators(G)`

A set containing the defining generators for G. If G is a p-group, this is guaranteed to be a minimal set of generators. For non-p-groups, this will be the set of pc-generators.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for G.

`PCGenerators(G)`

An indexed set containing the pc-generators for G.

`NumberOfPCGenerators(G)`

`NPCGenerators(G)`

`NPCgens(G)`

The number of pc-generators for G.

`PCPrimes(G)`

A sequence $[p_1, \ldots, p_n]$ containing the primes associated with the pc-generators of G. The i-th term of the sequence contains the prime associated with generator a_i of G for $i = 1, \ldots, n$.

63.3.2 Numerical Invariants

MAGMA has built-in functions to compute the order and exponent of a group.

`Order(G)`
`#G`

The order of the group G, returned as an ordinary integer.

`FactoredOrder(G)`

The factored order of the group G.

`Exponent(G)`

The exponent of the group G.

63.3.3 Predicates

MAGMA has built-in functions to check standard group properties.

`IsAbelian(G)`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsNilpotent(G)`

Returns `true` if the group G is nilpotent, `false` otherwise.

`IsPerfect(G)`

Returns `true` if the group G is perfect, `false` otherwise. A soluble group G is perfect only if it is trivial.

`IsSimple(G)`

Returns `true` if the group G is simple, `false` otherwise.

`IsSoluble(G)`
`IsSolvable(G)`

Returns `true` if the group G is soluble, `false` otherwise. It always returns the value `true` for a pc-group.

`IsTrivial(G)`

Returns `true` if the group G has order 1, `false` otherwise.

`IsSpecial(G)`

Given a p-group G, return `true` if G is special, `false` otherwise.

`IsExtraSpecial(G)`

Given a p-group G, return `true` if G is extra-special, `false` otherwise e.

Example H63E4

We use a presentation to define an extraspecial 3-group of exponent 9.

```
> E := PolycyclicGroup<a1,a2,b1,b2,z|a1^3,a2^3,b1^3=z,b2^3=z,
>   z^3,b1^a1=b1*z,b2^a2=b2*z>;
```

The sequence of base, exponent pairs from `FactoredOrder` shows us that the group has order 3^5.

```
> FactoredOrder(E);
[ <3, 5> ]
> Exponent(E);
9
```

As well as with the `Order` function, one can get the size of a group by using the `#` shorthand.

```
> D3 := DihedralGroup(GrpPC, 3);
> #D3;
6
> IsNilpotent(D3);
false
```

63.4 Homomorphisms

Arbitrary homomorphisms can be defined between pc-groups by using the `hom<>` constructor. For pc-groups, this constructor has features not generally available for user-defined homomorphisms. In addition to defining the map by giving images for the pc-generators, a homomorphism can be defined by giving images for any generating set of the domain. MAGMA will verify that the specified images define a homomorphism and will compute the kernel and inverse images for the defined map. Note that the value returned for an inverse image of an element is simply one element from the preimage, not the complete coset.

`hom< G -> H | L >`

`Check` BOOLELT *Default :* `true`

Construct a homomorphism $\phi : G \to H$ defined by the images specified by the list L.

The list L must be one of the following:

(a) a list, set, or sequence of 2-tuples $< g_i, h_i >$ (order not important);

(b) a list, set, or sequence of arrow pairs $g_i -> h_i$ (order not important);

(c) a list or sequence of images $h_1, \ldots, h_n$ (order is important).

The elements g_i and h_i must be elements of G and H, respectively, in each case. For the cases (a) and (b), the elements g_i must generate G and the homomorphism will satisfy $\phi(g_i) = h_i$. For case (c), n must be the number of pc-generators of G and the g_i are implicitly defined to be the pc-generators.

The parameter `Check` can be set to false in order to turn off the check that the map defined is a homomorphism. This should only be done when one is certain that the map is a homomorphism, since later results will most likely be incorrect if it is not.

`IsHomomorphism(G, H, L)`

This is a conditional form of the `hom`-constructor. The argument L must be a set or sequence of pairs (as in case (a) of the `hom`-constructor), or a sequence of images in H for the pc-generators of G (as in case (c) of the `hom`-constructor). If the specified images define a homomorphism, the value true and the resulting map are returned. Otherwise, false is returned.

`IdentityHomomorphism(G)`

The identity map from G to G.

`Kernel(f)`

Given a homomorphism f from one pc-group to another, return the kernel of f. This will be a pc-group which is a subgroup of the domain of f.

`Homomorphisms(G, H)`

Given finite *abelian* groups G and H, return a sequence containing all elements of $\mathrm{Hom}(G, H)$. The elements are returned as actual (MAGMA map type) homomorphisms. Note that this function simply uses `Hom`, transferring each element of the returned group to the actual MAGMA map type homomorphism.

Example H63E5

Let G be a pc-representation of the symmetric group S_4, and N be $O_2(G)$.

```
> G := PCGroup(Sym(4));
> G;
GrpPC : G of order 24 = 2^3 * 3
PC-Relations:
    G.1^2 = Id(G),
    G.2^3 = Id(G),
    G.3^2 = Id(G),
    G.4^2 = Id(G),
    G.2^G.1 = G.2^2,
    G.3^G.1 = G.3 * G.4,
    G.3^G.2 = G.4,
    G.4^G.2 = G.3 * G.4
> N := pCore(G,2);
> Order(N);
```

```
4
```

Let us define H to be a complement of N in G.

```
> H := sub<G|G.1*G.4,G.2*G.4>;
> Order(H);
6
> H meet N;
GrpPC of order 1
PC-Relations:
```

We now wish to define the projection homomorphism from G to H. This will map each element of N to the identity and each element of H to itself. We can define the map directly using these properties.

```
> pairs := [];
> for n in Generators(N) do
>   pairs cat:= [<G!n,Id(H)>];
> end for;
> for h in Generators(H) do
>   pairs cat:= [<G!h, h>];
> end for;
> proj := hom<G -> H|pairs>;
> proj;
Mapping from: GrpPC: G to GrpPC: H
> proj(G.1);
H.1
> proj(N);
GrpPC of order 1
PC-Relations:
```

We can also compute inverse images and can verify that N is the kernel of the map. Note that the preimage of a single element is just one element from the preimage, not the complete coset. Of course, one can use the kernel to compute the full coset if desired.

```
> y := (H.1)@@proj;
> y;
G.1
> Kernel(proj) eq N;
true
> {y*k: k in Kernel(proj)};
{ G.1 * G.3 * G.4, G.1 * G.4, G.1, G.1 * G.3 }
```

63.5 New Groups from Existing

`DirectProduct(G, H)`

The direct product K of the pc-groups G and H. The second argument returned is a sequence containing the inclusion maps $I_G : G \to K$ and $I_H : H \to K$. The third argument returned is a sequence containing the projection maps $P_G : K \to G$ and $P_H : K \to H$. Furthermore, the (user-) presentation of K is arranged so that the first pc-generators correspond to those of G and the remaining generators correspond to those of H.

`DirectProduct(Q)`

The direct product of pc-groups in the non-empty sequence Q, and the inclusion and projection maps.

`Extension(G, H, f)`

The split extension K of the pc-group G by the pc-group H, where the action of H on G is given by the homomorphism $\phi : H \to \mathrm{Aut}(G)$ specified by f. The extension K will have a normal subgroup $G^\sim$ isomorphic to G, while the quotient group $K/G^\sim$ is isomorphic to H.

The homomorphism ϕ is given by the sequence of maps f. Suppose that the pc-generators for H are $h_1, \ldots, h_s$. The i-th entry of f defines the action of h_i on G. That is, $f[i](x) = h_i^{-1} \cdot x \cdot h_i$, for $x \in G$.

`Extension(M, H)`

The split extension K of the G-module M by the pc-group H. We use the action of H on M to define the action of H on an elementary abelian p-group of order p^d where M is a d-dimensional module over $GF(p)$, p prime.

`Extension(G, H, f, t)`

The non-split extension K of the pc-group G by the pc-group H, where the action of H on G is given by the homomorphism $\phi : H \to \mathrm{Aut}(G)$ and the tails for H are given as the set of tuples t. The extension K will have a normal subgroup $G^\sim$ isomorphic to G, while the quotient group $K/G^\sim$ is isomorphic to H.

The homomorphism ϕ is given by the sequence of maps f. Suppose that the pc-generators for H are $h_1, \ldots, h_s$. The i-th entry of f defines the action of h_i on G. That is, $f[i](x) = h_i^{-1} \cdot x \cdot h_i$, for $x \in G$.

The specification of t involves giving the relations $h_j^{-1} h_i h_j = w_{ij}$, where w_{ij} is a word in K for $1 \le j < i \le s$. For $i = j$, we need the relation ${h_i}^{p_i} = w_{ii}$, where w_{ii} is a word in K for $1 \le i \le s$. Each w_{ij} is the RHS of the relation from H with the tail x_{ij}. The tails are given by the sequence t in the order $t = [x_{11}, x_{21}, x_{22}, x_{31}, \ldots, x_{ss}]$. Alternatively, t can be given as a set of tuples $< i, j, x_{ij} >$ for non-trivial x_{ij}.

Note that if $x_{ij} =$`Id(G)`, for $1 \le i \le s$ and $1 \le j \le i$, then K will just be the split extension of G and H.

`Extension(M, H, t)`

The non-split extension K of the G-module M by the pc-group H. We use the action of H on M to define the action of H on an elementary abelian p-group of order p^d where M is a d-dimensional module over $GF(p)$, p prime.

The specification of t is similar to that for t in the preceding description.

`IsExtension(G, H, f)`

`IsExtension(M, H)`

`IsExtension(G, H, f, t)`

`IsExtension(M, H, t)`

For each `Extension` variation, there is a corresponding function `IsExtension` which attempts to construct the specified group and returns a boolean value indicating whether or not the construction succeeded. If the construction succeeds, the extension group is also returned.

The `Extension` functions will generate a runtime error if the specified construction is not legal. The `IsExtension` function allows the user to detect this error condition and continue.

`WreathProduct(G, H)`

The wreath product of the pc-groups G and H, where the regular permutation representation of H is used to define the action.

`WreathProduct(G, H, f)`

The wreath product of the pc-groups G and H where the action of H is given by f, which may be either a homomorphism from H into a permutation group P or a sequence of permutations defining a homomorphism from H into P. If f is a sequence, the homomorphism $\phi : H \to P$ is defined by $H.i \to f[i]$ for $i = 1, \ldots, s$.

Example H63E6

To demonstrate some of the versions of `Extension` we first build a split extension of a cyclic group of order 4 acting on an elementary abelian group of order 9.

```
> C4 := CyclicGroup(GrpPC,4);
> E9 := AbelianGroup(GrpPC,[3,3]);
> f1 := hom<E9->E9|[E9.1*E9.2^2, E9.1^2*E9.2^2]>;
> f2 := hom<E9->E9|[E9.1^2,E9.2^2]>;
> G := Extension(E9,C4,[f1,f2]);
> G;
GrpPC : G of order 36 = 2^2 * 3^2
PC-Relations:
    G.1^2 = G.2,
    G.2^2 = Id(G),
    G.3^3 = Id(G),
    G.4^3 = Id(G),
```

```
    G.3^G.1 = G.3 * G.4^2,
    G.3^G.2 = G.3^2,
    G.4^G.1 = G.3^2 * G.4^2,
    G.4^G.2 = G.4^2
```

Then, we define a module for this group and use it to build a nonsplit extension.

```
> MR := MatrixRing(GF(3),2);
> m1 := MR![1,1,1,2];
> m2 := MR![2,0,0,2];
> V := GModule(G,[m1,m2,Id(MR),Id(MR)]);
> IsIrreducible(V);
true
> v0 := V!0;
> tails := [v0,v0,v0,v0,V![1,0],V![2,0],V![1,2],V![0,2],v0,V![0,1]];
> H := Extension(V,G,tails);
> H;
GrpPC : H of order 324 = 2^2 * 3^4
PC-Relations:
    H.1^2 = H.2,
    H.2^2 = Id(H),
    H.3^3 = H.5^2,
    H.4^3 = H.6,
    H.5^3 = Id(H),
    H.6^3 = Id(H),
    H.3^H.1 = H.3 * H.4^2,
    H.3^H.2 = H.3^2 * H.5,
    H.4^H.1 = H.3^2 * H.4^2 * H.5 * H.6^2,
    H.4^H.2 = H.4^2 * H.6^2,
    H.5^H.1 = H.5 * H.6,
    H.5^H.2 = H.5^2,
    H.6^H.1 = H.5 * H.6^2,
    H.6^H.2 = H.6^2
```

Notice that the relations of H involving the first four generators are those of G with the specified tails appended. We are then ready to compute various properties of H.

```
> [N`order:N in NormalSubgroups(H)];
[ 1, 9, 81, 162, 324 ]
```

Example H63E7

In this example we verify an example of Cossey and Hawkes in [CH00]. The paper shows that the largest size of a conjugacy class in an abelian by nilpotent finite group is at least as large as the product of the largest class sizes for the Sylow subgroups. The example is a group having derived length 3 in which this fails.
We start with a dihedral group of order 10 acting on a cyclic group of order 8.

```
> E := DihedralGroup(GrpPC,5);
```

```
> A := CyclicGroup(GrpPC,8);
```

Define an action of E on A and create the split extension.

```
> f1 := hom<A->A|A.1->(A.1)^-1>;
> f2 := hom<A->A|A.1->A.1>;
> H := Extension(A, E, [f1, f2]);
```

Then construct a certain H-module...

```
> QH := SylowSubgroup(H,2);
> t := TrivialModule(QH, FiniteField(5));
> B := Induction(t, H);
```

...and form the split extension of H acting on that module.

```
> G := Extension(B, H);
> print G;
GrpPC : G of order 250000 = 2^4 * 5^6
PC-Relations:
    G.1^2 = Id(G),
    G.2^5 = Id(G),
    G.3^2 = G.4,
    G.4^2 = G.5,
    G.5^2 = Id(G),
    G.6^5 = Id(G),
    G.7^5 = Id(G),
    G.8^5 = Id(G),
    G.9^5 = Id(G),
    G.10^5 = Id(G),
    G.2^G.1 = G.2^4,
    G.3^G.1 = G.3 * G.4 * G.5,
    G.4^G.1 = G.4 * G.5,
    G.6^G.2 = G.10,
    G.7^G.1 = G.10,
    G.7^G.2 = G.6,
    G.8^G.1 = G.9,
    G.8^G.2 = G.7,
    G.9^G.1 = G.8,
    G.9^G.2 = G.8,
    G.10^G.1 = G.7,
    G.10^G.2 = G.9
> print DerivedLength(G);
3
```

Now check the relevant class sizes.

```
> P := SylowSubgroup(G,5);
> Q := SylowSubgroup(G,2);
> print  Maximum({x[2]:x in Classes(G)});
1250
```

```
> print  Maximum({x[2]:x in Classes(P)});
625
> print  Maximum({x[2]:x in Classes(Q)});
4
```

Note that 1250 is less than the product 625*4.

63.6 Elements

Elements of a pc-group are written in terms of the generators. The pc-generators of a group G can always be written as `G.1`, `G.2`, ... Any variables naming the generators, either assigned during the definition of the group, or later using standard assignment statements, can also be used to express the generators. An arbitrary element can be written as a word in the generators using the various element operations.

63.6.1 Definition of Elements

A *word* is defined inductively as follows:

(i) A generator is a word;

(ii) The expression (u) is a word, where u is a word;

(iii) The product $u * v$ of the words u and v is a word;

(iv) The conjugate u^v of the word u by the word v is a word (u^v expands into the word $v^{-1} * u * v$);

(v) The power of a word u^n, where u is a word and n is an integer, is a word;

(vi) The commutator (u, v) of the words u and v is a word ((u, v) expands into the word $u^{-1} * v^{-1} * u * v$).

A group element is always printed by MAGMA as a normal word in the pc-generators of its parent group.

It is also possible to create an element of a group G from its exponent vector. That is, the sequence $[e_1, e_2, \ldots, e_n]$ corresponds to the element $G.1^{e_1} * G.2^{e_2} * \cdots * G.n^{e_n}$. The coercion operator `!` is used to convert the sequence to the element.

`G ! Q`

Given the pc-group G and a sequence Q of length n, containing the distinct positive integers α_i, $0 \leq \alpha_i < p_i$ for $i = 1, \ldots, n$, construct the element x of G given by

$$x = a_1^{\alpha_1} \ldots a_n^{\alpha_n}, \quad 0 \leq \alpha_i < p_i \quad \text{for } i = 1, \ldots, n.$$

`ElementToSequence(x)`

`Eltseq(x)`

Given an element x belonging to the pc-group G, where

$$x = a_1^{\alpha_1} \ldots a_n^{\alpha_n}, \quad 0 \leq \alpha_i < p_i \quad \text{for } i = 1, \ldots, n,$$

return the sequence Q of n integers defined by $Q[i] = \alpha_i$, for $i = 1, \ldots, n$.

Identity(G)

Id(G)

G ! 1

Construct the identity element of the pc-group G.

Example H63E8

Given a pc-group, we can define elements as words in the generators or as more general expressions.

```
> G := PolycyclicGroup<a,b,c|a^3,b^2,c^2,b^a=c,c^a=b*c>;
> G;
GrpPC : G of order 12 = 2^2 * 3
PC-Relations:
    G.1^3 = Id(G),
    G.2^2 = Id(G),
    G.3^2 = Id(G),
    G.2^G.1 = G.3,
    G.3^G.1 = G.2 * G.3
> x := G.1^2*G.3;
> x;
G.1^2 * G.3
> x^2;
G.1 * G.2 * G.3
> x^3;
Id(G)
```

MAGMA will print the element in normal form even if it is not entered that way.

```
> G.2*G.1;
G.1 * G.3
```

When coercing a sequence into a group element, the sequence is always interpreted as an exponent vector for a normal word.

```
> y := G![0,1,1];
> y;
G.2 * G.3
> x*y;
G.1^2 * G.2
> y*x;
G.1^2
> (x,y);
G.2
```

An element can also be converted into a sequence.

```
> x^y;
G.1^2 * G.2 * G.3
> Eltseq(x^y);
[ 2, 1, 1 ]
```

63.6.2 Arithmetic Operations on Elements

New elements can be computed from existing elements using standard operations.

`g * h`

Product of the element g and the element h, where g and h belong to some common subgroup G of a pc-group U. If g and h are given as elements belonging to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h are given as elements belonging to distinct subgroups H and K of U, then the product is returned as an element of G, where G is the smallest subgroup of U known to contain both elements.

`g *:= h`

Replace g with the product of element g and element h.

`g ^ n`

The n-th power of the element g, where n is a positive or negative integer.

`g ^:= n`

Replace g with the n-th power of the element g.

`g / h`

Quotient of the element g by the element h, i.e. the element $g * h^{-1}$. Here g and h must belong to some common subgroup G of a pc-group U. The rules for determining the parent group of g/h are the same as for $g * h$.

`g /:= h`

Replace g with the quotient of the element g by the element h.

`g ^ h`

Conjugate of the element g by the element h, i.e. the element $h^{-1} * g * h$. Here g and h must belong to some common subgroup G of a pc-group U. The rules for determining the parent group of g^h are the same as for $g * h$.

`g ^:= h`

Replace g with the conjugate of the element g by the element h.

`(g1, ..., gn)`

Given the n words $g_1, \ldots, g_n$ belonging to some common subgroup G of a pc-group U, return the commutator. If $g_1, \ldots, g_n$ are given as elements belonging to the same proper subgroup G of U, then the result will be returned as an element of G; if $g_1, \ldots, g_n$ are given as elements belonging to distinct subgroups of U, then the product is returned as an element of G, where G is the smallest subgroup of U known to contain all elements. Commutators are *left-normed*, so that they are evaluated from left to right.

63.6.3 Properties of Elements

`Order(x)`

Order of the element x.

`Parent(x)`

The parent group G of the element x.

63.6.4 Predicates for Elements

Elements in the same group can be compared using `eq` and `ne`.

`g eq h`

Given elements g and h belonging to a common pc-group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given elements g and h belonging to a common pc-group, return `true` if g and h are distinct elements, `false` otherwise.

`IsIdentity(g)`
`IsId(g)`

Returns `true` if g is the identity element, `false` otherwise.

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value `true` if g and h are conjugate in G. The function also returns a second value in the event that the elements are conjugate: an element z such that $g^z = h$.

Example H63E9

We check if one element commutes with another.

```
> G<a,b,c> := PolycyclicGroup<a,b,c|a^3,b^2,c^2,b^a=c,c^a=b*c>;
> b^a eq b;
false
```

The same information can also be obtained by checking the commutator.

```
> IsIdentity((b,a));
false
```

If we assign the result of `IsConjugate` to a single variable, it will store the boolean result.

```
> r := IsConjugate(G, c, b);
> r;
```

```
true
```

If we simply print `IsConjugate`, the boolean value and the conjugating element (if any) are displayed. On the other hand, using the multiple assignment, we can capture both of those values.

```
> IsConjugate(G, c, b);
true a^2
> r, x := IsConjugate(G, c, b);
> x, r;
a^2 true
> c^x;
b
```

63.6.5 Set Operations

These functions allow one to work with the set of elements of G, possibly without much knowledge of the structure of G.

`NumberingMap(G)`

A bijective mapping from the group G onto the set of integers $\{1...|G|\}$. The actual mapping depends upon the current presentation for G.

`Random(G)`

An element, randomly chosen, from the group G. This function uses an entirely different procedure than that used by `RandomProcess` (see below). A group element is chosen with uniform probability by generating (pseudo-)random integers in the proper range to form a legal exponent vector for G. The corresponding element is returned. This is an extremely efficient process and is the recommended method for producing random elements of a pc-group.

`RandomProcess(G)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 20

Create a process to generate randomly chosen elements from the group G. The process uses an 'expansion' procedure to construct a set of elements corresponding to fairly long words in the generators of G. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G) + 1$. Initially, these are simply the generators of G and products of pairs of generators of G. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses an element x stored by the process and returns it, replacing x with the product of x and another random element (multiplied on the left or the right). Setting `Scramble := m` causes m such operations to be performed initially.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the finite group G, construct a random element of G by forming a random product over the expanded generating set constructed when the process was created.

`Representative(G)`

`Rep(G)`

A representative element of G. For a pc-group, this always returns the identity element.

Example H63E10

The `NumberingMap` function assigns a number to each group element.

```
> G := DihedralGroup(GrpPC,4);
> num_map := NumberingMap(G);
> for x in G do
>   print x,"->",num_map(x);
> end for;
Id(G) -> 1
G.3 -> 2
G.2 -> 3
G.2 * G.3 -> 4
G.1 -> 5
G.1 * G.3 -> 6
G.1 * G.2 -> 7
G.1 * G.2 * G.3 -> 8
```

The inverse map can be used to obtain the group element corresponding to a particular number.

```
> 6 @@ num_map;
G.1 * G.3
```

The `Random` function is sometimes useful to create a statistical profile of a group. To demonstrate, we take two groups of order 3^6 from the `SmallGroup` database.

```
> G1 := SmallGroup(3^6, 60);
> G2 := SmallGroup(3^6, 392);
```

We want to build a histogram of element orders for each group. Since these are 3-groups, each order will be a power of 3 and we use `Ilog` to get the exponent of the order. First, we define a short function to compute the histogram.

```
> function hist(G, trials)
>   // Given a 3-group G, of exponent <= 3^5,
>   // return a sequence whose ith term is the
>   // number of elements of order p^(i-1) out
>   // of trials randomly chosen elements.
>   table := [0,0,0,0,0,0];
>   for i := 1 to trials do
```

```
>     x := Random(G);
>     n := Ilog(3, Order(x));
>     table[n+1] +:= 1;
>   end for;
>   return table;
> end function;
```

Now, we use this function to compute order distributions for 100 elements in each group.

```
> t1 := hist(G1,100);
> t1;
[ 0, 0, 5, 28, 67, 0 ]
> t2 := hist(G2,100);
> t2;
[ 0, 5, 5, 25, 65, 0 ]
```

We can even display them with simple character graphics.

```
> for e in t1 do print ":","@"^e; end for;
:
:
: @@@@@
: @@@@@@@@@@@@@@@@@@@@@@@@@@@@
: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
:
> for e in t1 do print ":","@"^e; end for;
:
: @@@@@
: @@@@@
: @@@@@@@@@@@@@@@@@@@@@@@@@
: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
:
```

Example H63E11

Given the subgroups H and K of G, construct the set product of the groups H and K.

```
> set_product := func<G, H, K | { G | x * y : x in H, y in K }>;
```

Given a subgroup H of the pc-group G, construct H as a set of elements of G.

```
> elements := func<G, H | { G | x : x in H }>;
```

63.7 Conjugacy

`Class(H, g)`

`Conjugates(H, g)`

`g ^ H`

Given a group H and an element g belonging to a group K such that H and K are subgroups of some covering group, this function returns the set of conjugates of g under the action of H. If $H = K$, the function returns the conjugacy class of g in H.

`ConjugacyClasses(G)`

`Classes(G)`

Construct a set of representatives for the conjugacy classes of G. The classes are returned as a sequence of tuples containing the order of the elements in the class, the class length and a representative element for the class. For non-p-groups, the classes are computed using the homomorphism principle down a series with elementary abelian factors and orbit-stabilizer in each quotient. See [MN89] for details. For p-groups an algorithm based on linear algebra developed by Charles Leedham-Green is used.

`ClassMap(G)`

The class map $M : G \to \{1, \ldots, n\}$ for the group G, where n is the number of conjugacy classes of G.

`ClassRepresentative(G, x)`

The designated representative for the conjugacy class of G containing the element x (relative to existing conjugacy classes).

`IsConjugate(G, g, h)`

Given a group G and elements g and h belonging to G, return the value true if g and h are conjugate in G. The function also returns a second value in the event that the elements are conjugate: an element z which conjugates g into h.

`NumberOfClasses(G)`

`Nclasses(G)`

The number of conjugacy classes of elements of the group G.

`PowerMap(G)`

The power map M associated with the conjugacy classes of G. The map M describes where the elements of the conjugacy classes of G move under powers. That is, $< c, n >@M$ returns the class number where class c moves under the power n. The value of c must be in the range $[1 \ldots$`Nclasses(G)`$]$.

$M : \{1 \ldots n\} \times \mathbf{Z} \to \{1 \ldots n\}$

Example H63E12

Let G be a pc-representation of SL(2,3). We can compute the conjugacy classes of G. Notice that the conjugacy class object has a special printing routine, but you can still access individual entries.

```
> G := PCGroup(SpecialLinearGroup(2,GF(3)));
> G;
GrpPC : G of order 24 = 2^3 * 3
PC-Relations:
    G.1^3 = Id(G),
    G.2^2 = G.4,
    G.3^2 = G.4,
    G.4^2 = Id(G),
    G.2^G.1 = G.3 * G.4,
    G.3^G.1 = G.2 * G.3 * G.4,
    G.3^G.2 = G.3 * G.4
> Nclasses(G);
7
> cc := Classes(G);
> cc;
Conjugacy Classes of group G
----------------------------
[1]     Order 1       Length 1
        Rep Id(G)
[2]     Order 2       Length 1
        Rep G.4
[3]     Order 3       Length 4
        Rep G.1
[4]     Order 3       Length 4
        Rep G.1^2
[5]     Order 4       Length 6
        Rep G.2
[6]     Order 6       Length 4
        Rep G.1 * G.4
[7]     Order 6       Length 4
        Rep G.1^2 * G.4
> cc[3];
<3, 4, G.1>
> x := cc[3][3];
> Class(G,x);
{ G.1 * G.2 * G.3 * G.4, G.1 * G.2 * G.4, G.1, G.1 * G.3 }
7
>
```

We can use the `ClassMap` function to compute class multiplication constants (structure constants for the center of the group algebra). For example, we compute the decomposition of class 3 times class 5.

```
> cm := ClassMap(G);
```

```
> cm(G.1);
3
> i := 3; j := 5;
> t := [0: c in cc];
> for x in Class(G,cc[i][3]), y in Class(G,cc[j][3]) do
>   t[cm(x*y)] +:= 1;
> end for;
> t;
[ 0, 0, 12, 0, 0, 12, 0 ]
```

To get the actual structure constants, we need to divide each entry in `t` by the corresponding class size.

```
> [ t[i]/cc[i][2]: i in [1..#t] ];
[ 0, 0, 3, 0, 0, 3, 0 ]
```

63.8 Subgroups

Subgroups of pc-groups are treated as independent pc-groups in their own right, with the subgroup relationship maintained in internal data structures. Thus, a subgroup has generators and a pc-presentation and one can apply any of the functions described earlier for groups. Furthermore, there are a variety of functions and operations specifically involving subgroups.

63.8.1 Definition of Subgroups by Generators

The most flexible method of defining a subgroup is to list generators or normal generators for the subgroup.

`sub< G | L >`

Construct the subgroup H of the pc-group G generated by the elements specified by the terms of the *generator list* L.

A term $L[i]$ of the generator list may consist of any of the following objects:

(a) An element liftable to G (in particular, any element of G);

(b) A subgroup of G;

(c) A set or sequence of (a), or (b).

The collection of words and groups specified by the list must all belong to the group G and H will be constructed as a subgroup of G.

The subgroup H is defined to be generated by the words specified directly by terms $L[i]$ together with the stored generating words for any groups specified by terms of $L[i]$. MAGMA will compute a set of pc-generators for H and, if H is a p-group, a minimal generating set.

The inclusion map from H to G is returned as well.

`ncl< G | L >`

Construct the subgroup N of the pc-group G as the normal closure of the subgroup generated by the elements specified by the terms of the *generator list* L.

The possible forms of a term $L[i]$ of the generator list are the same as for the `sub`-constructor.

The inclusion map from N to G is returned as well.

Example H63E13

We define G to be Z_5 wr Z_3 and then create two subgroups. Notice that the `ncl`-constructor builds a larger subgroup in this case.

```
> G<a,b,c,d> := PolycyclicGroup<a,b,c,d| a^3, b^5, c^5, d^5,
>   b^a = c, c^a = d, d^a = b>;
> H := sub<G| b,c>;
> Order(H);
25
> IsAbelian(H);
true
> IsNormal(G, H);
false
> N := ncl<G| b,c>;
> IsNormal(G, N);
true
> Order(N);
125
```

63.8.2 Membership and Coercion

There are several functions and operators which allow one to take advantage of the subgroup relationship to rewrite elements from one presentation to another. That is, if x is an element of H which is a subgroup of G, then x has a representation as a normal word in the pc-generators of H, but also has a representation as a (different) normal word in the pc-generators of G. The coercion operator and inclusion map allow one to compute one of these words based on the other, thus shifting where we view the element in question.

MAGMA keeps track of the various relationships between subgroups in a group. Thus, if H is a subgroup of K which is a subgroup of G, then H can also be considered a subgroup G. Similarly, in situations involving elements of two groups, A and B, MAGMA will often try to find a *covering group* C which contains both of A and B. In this case, the elements may be automatically coerced into the covering group.

`g in G`

Given an element g and a group G, return `true` if g is an element of G, `false` otherwise. In order for this comparison to make sense, both g and G must be contained in some covering group.

`g notin G`

Given an element g and a group G, return `true` if g is not an element of G, `false` otherwise. In order for this comparison to make sense, both g and G must be contained in some covering group.

`G ! g`

Given an element g belonging to some subgroup H of the group G, rewrite g as an element of G.

`H ! g`

Given an element g belonging to the group G, and given a subgroup H of G containing g, rewrite g as an element of H.

`K ! g`

Given an element g belonging to the group H, and a group K, such that H and K are subgroups of a covering group G, and both H and K contain g, rewrite g as an element of K.

Example H63E14

We create two subgroups of a dihedral group.

```
> G := DihedralGroup(GrpPC, 10);
> C := sub<G| G.2>;
> H := sub<G| G.1, G.3>;
> H.1 in C;
false
> H.2 in C;
true
> Parent(H.1);
GrpPC : H of order 10 = 2 * 5
PC-Relations:
    H.1^2 = Id(H),
    H.2^5 = Id(H),
    H.2^H.1 = H.2^4
> G!(H.1);
G.1
```

MAGMA will compute appropriate covering groups as needed.

```
> H.1*C.1;
G.1 * G.2 * G.3^2
> x := (H.1, C.2);
> x;
G.3^2
> H!x;
H.2^2
> C!x;
```

```
C.2^2
> C!(H.2);
C.2
```

63.8.3 Inclusion and Equality

`S subset G`

Given an group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G`

Given a group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G`

Given groups G and H, subgroups of some covering group, return `true` if H is a subgroup of G, `false` otherwise.

`H notsubset G`

Given groups G and H, subgroups of some covering group, return `true` if H is not a subgroup of G, `false` otherwise.

`G eq H`

Given groups G and H, subgroups of some covering group, return `true` if G and H are the same group, `false` otherwise.

`G ne H`

Given groups G and H, subgroups of some covering group, return `true` if G and H are distinct groups, `false` otherwise.

`InclusionMap(G, H)`

The map from the subgroup H of G to G.

63.8.4 Standard Subgroup Constructions

The operators and functions which construct a subgroup of a pc-group always return the subgroup as a pc-group.

```
H ^ g
```

```
Conjugate(H, g)
```

Construct the conjugate $g^{-1} * H * g$ of the group H under the action of the element g. The group H and the element g must belong to a common group.

```
H meet K
```

The intersection of groups H and K. The algorithm used for non-p-groups is described in [GS90].

```
H meet:= K
```

Replace H with the intersection of groups H and K.

```
CommutatorSubgroup(G, H, K)
```

```
CommutatorSubgroup(H, K)
```

Construct the commutator subgroup of groups H and K, where H and K are subgroups of a common group G.

```
Centralizer(G, g)
```

```
Centraliser(G, g)
```

The centralizer of the element g in the group G.

```
Centralizer(G, H)
```

```
Centraliser(G, H)
```

The centralizer of the subgroup H in the group G.

```
Core(G, H)
```

The maximal normal subgroup of G that is contained in the subgroup H of G.

```
H ^ G
```

```
NormalClosure(G, H)
```

The normal closure of the subgroup H in the group G.

```
Normalizer(G, H)
```

```
Normaliser(G, H)
```

The normalizer of the subgroup H of the group G. The algorithm used for non-p-groups is described in [GS90].

Example H63E15

We'll consider various subgroups of a direct product of a cyclic group of order 6 and dihedral group of order 10.

```
> G := DirectProduct(CyclicGroup(GrpPC,6), DihedralGroup(GrpPC,5));
> x := G.3;
> C := Centralizer(G,x);
> C;
GrpPC : C of order 12 = 2^2 * 3
PC-Relations:
    C.1^2 = Id(C),
    C.2^2 = Id(C),
    C.3^3 = Id(C)
> H := sub<G|G.2,G.4>;
> Order(H);
15
```

We can compute the intersection using the `meet` operator.

```
> K := H meet C;
> K;
GrpPC : K of order 3
PC-Relations:
    K.1^3 = Id(K)
```

To get the join of two subgroups, we simply use the `sub`-constructor.

```
> J := sub<G|H, C>;
> J eq G;
true
```

63.8.5 Properties of Subgroups

`Index(G, H)`

The index of the subgroup H in the group G, returned as an ordinary integer.

`FactoredIndex(G, H)`

The factored index of the subgroup H in the group G.

63.8.6 Predicates for Subgroups

`IsCentral(G, H)`

Returns `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

`IsConjugate(G, H, K)`

Given a group G and subgroups H and K belonging to G, return the value `true` if H and K are conjugate in G. The function returns a second value in the event that the subgroups are conjugate: an element z which conjugates H into K.

`IsMaximal(G, H)`

Returns `true` if the subgroup H of the group G is a maximal subgroup of G, `false` otherwise.

`IsNormal(G, H)`

Returns `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

`IsSelfNormalizing(G, H)`

Returns `true` if the subgroup H of the group G is self-normalizing in G, `false` otherwise.

`IsSubnormal(G, H)`

Returns `true` if the subgroup H of the group G is subnormal in G, `false` otherwise.

Example H63E16

```
> G := PCGroup(Sym(4));
> G;
GrpPC : G of order 24 = 2^3 * 3
PC-Relations:
    G.1^2 = Id(G),
    G.2^3 = Id(G),
    G.3^2 = Id(G),
    G.4^2 = Id(G),
    G.2^G.1 = G.2^2,
    G.3^G.1 = G.3 * G.4,
    G.3^G.2 = G.4,
    G.4^G.2 = G.3 * G.4
> U := sub<G|G.4>;
> IsNormal(G,U);
false
> IsSubnormal(G,U);
```

```
true
```

Now, we try to construct a subnormal chain by taking normalizers.

```
> N1 := Normalizer(G,U);
> Index(G,N1);
3
> N2 := Normalizer(G,N1);
> Index(G,N2);
3
> N1 eq N2;
true
```

We're stuck. However, we can work our way down with `NormalClosure`.

```
> M1 := NormalClosure(G,U);
> U subset M1;
true
> M1 subset U;
false
> M2 := NormalClosure(M1,U);
> M2 eq U;
true
```

Now, we work inside the Sylow 2-subgroup and look for complements of the cyclic group of order 4 by brute force.

```
> S := Sylow(G,2);
> S;
GrpPC : S of order 8 = 2^3
PC-Relations:
    S.2^S.1 = S.2 * S.3
> T := sub<S|S.1*S.2>;
> list := [];
> for x in S do
>   if (Order(x) ne 2) or (x in T) then
>     continue;
>   end if;
>   Append(~list, sub<S|x>);
> end for;
> #list;
4
> for i in [1..3], j in [i+1..4] do
>   print i,j,IsConjugate(S,list[i],list[j]);
> end for;
1 2 true S.1
1 3 false
1 4 false
2 3 false
2 4 false
```

```
3 4 true S.2
```

We see that T has two conjugacy classes of complements.

63.8.7 Hall π-Subgroups and Sylow Systems

The functions given here all assume that G is a soluble group having order $p_1^{e_1}p_2^{e_2}\cdots p_k^{e_k}$.

`ComplementBasis(G)`

A complement basis of the soluble group G. This is a sequence of k subgroups of G, where the i-th subgroup has order $p_1^{e_1}\ldots p_{i-1}^{e_{i-1}}p_{i+1}^{e_{i+1}}\ldots p_k^{e_k}$, i.e. the complements of the k Sylow subgroups of G.

`HallSubgroup(G, S)`

The Hall π-subgroup of G, where π is defined by S. The argument S may be a set of primes, a single prime, or the negation of a single prime. If $S = -p$, then the Hall p'-subgroup of G is returned.

`pCore(G, S)`

The core of the Hall π-subgroup, where π is defined by the argument S, which has the same interpretation as for `HallSubgroup`.

`SylowBasis(G)`

A Sylow basis for the soluble group G. This is a sequence of k subgroups of G, having orders $p_1^{e_1},\ldots,p_k^{e_k}$, i.e. the k Sylow subgroups of G.

`SylowSubgroup(G, p)`

`Sylow(G, p)`

A Sylow p-subgroup for the group G.

`SystemNormalizer(G)`

`SystemNormaliser(G)`

The system normalizer for the group G. The system normalizer of the complement basis $\Sigma = \{H_1, ..., H_k\}$ is defined to be the intersection of the normalizers in G of each H_i, ie. $N(\Sigma) = \cap_{i=1}^{k} N_G(H_i)$. The algorithm used is derived directly from the definition.

Example H63E17

Given the group $D_3 \wr D_5$, we can construct the Hall 2-subgroup as follows:

```
> H := DihedralGroup(GrpPerm, 5);
> G := WreathProduct(DihedralGroup(GrpPC, 3), DihedralGroup(GrpPC, 5),
>         [H.2, H.1]);
> H2 := HallSubgroup(G, 2);
> Order(H2);
64
```

The Hall $2'$-subgroup of the same group is constructed as follows:

```
> H35 := HallSubgroup(G, -2);
> Order(H35);
1215
```

63.8.8 Conjugacy Classes of Subgroups

MAGMA has functions for computing the subgroups of a group that return the subgroups either as a list of conjugacy class representatives as or as a poset. Details of these functions may be found in Chapter 57. Here we mention the basic functions for convenience.

`SubgroupClasses(G)`

`Subgroups(G)`

Conjugacy class representatives for all subgroups of G. The algorithm was developed by M. Slattery and is essentially that of [Hul99] without the action of automorphisms.

`AbelianSubgroups(G)`

`CyclicSubgroups(G)`

`ElementaryAbelianSubgroups(G)`

`NilpotentSubgroups(G)`

Conjugacy class representatives for all subgroups of the indicated type in G. The algorithm used is essentially that of [Hul99] without the action of automorphisms. Appropriate filters are applied to select the desired groups at each successive quotient in the computation.

`MaximalSubgroups(G)`

A sequence of conjugacy class representatives for the maximal subgroups of G. The algorithm, developed by Charles Leedham-Green, relies on computing a special presentation for G.

`SubgroupLattice(G)`

The lattice of conjugacy classes of subgroups of G.

`BurnsideMatrix(G)`

The Burnside matrix corresponding to the lattice of subgroups of G.

`DisplayBurnsideMatrix(G)`

Pretty-print the Burnside matrix corresponding to the lattice of subgroups of G.

Example H63E18

To show a bit about subgroup classes, we look at the direct product of C_3 and D_3. First we list out the normal subgroups of G.

```
> G := DirectProduct(CyclicGroup(GrpPC,3),
>                    DihedralGroup(GrpPC,3));
> ns := NormalSubgroups(G);
> ns;
Conjugacy classes of subgroups
------------------------------
[1]     Order 1            Length 1
        GrpPC of order 1
        PC-Relations:
[2]     Order 3            Length 1
        GrpPC of order 3
        PC-Relations:
            $.1^3 = Id($)
[3]     Order 3            Length 1
        GrpPC of order 3
        PC-Relations:
            $.1^3 = Id($)
[4]     Order 6            Length 1
        GrpPC of order 6 = 2 * 3
        PC-Relations:
            $.1^2 = Id($),
            $.2^3 = Id($),
            $.2^$.1 = $.2^2
[5]     Order 9            Length 1
        GrpPC of order 9 = 3^2
        PC-Relations:
            $.1^3 = Id($),
            $.2^3 = Id($)
[6]     Order 18           Length 1
        GrpPC of order 18 = 2 * 3^2
        PC-Relations:
            $.1^2 = Id($),
            $.2^3 = Id($),
            $.3^3 = Id($),
```

```
            $.3^$.1 = $.3^2
```

The normal subgroups sequence has a special printing routine. Each entry in the sequence is actually a record.

```
> ns[2];
rec<recformat<order, length, subgroup, presentation> |
 order := 3, length := 1, subgroup := GrpPC of order 3
PC-Relations:
    $.1^3 = Id($)>
```

We extract the two normal subgroups of order 3. Each of them turn out to have one conjugacy class of complements in G. However, one of the complements is normal and the other is not.

```
> N1 := ns[2]`subgroup;
> N2 := ns[3]`subgroup;
> c1 := Complements(G,N1);
> c1;
[
    GrpPC of order 6 = 2 * 3
    PC-Relations:
        $.1^2 = Id($),
        $.2^3 = Id($),
        $.2^$.1 = $.2^2
]
> c2 := Complements(G,N2);
> c2;
[
    GrpPC of order 6 = 2 * 3
    PC-Relations:
        $.1^2 = Id($),
        $.2^3 = Id($)
]
> Index(G,Normalizer(G,c1[1]));
1
> Index(G,Normalizer(G,c2[1]));
3
```

We can look at the full list of classes of subgroups of G to see that there are three classes of non-normal subgroups as well as the normal subgroups. There are two non-normal subgroups of order 3 in addition to $N1$ and $N2$.

```
> Subgroups(G);
Conjugacy classes of subgroups
------------------------------
[ 1]    Order 1            Length 1
        GrpPC of order 1
        PC-Relations:
[ 2]    Order 2            Length 3
        GrpPC of order 2
        PC-Relations:
```

```
          $.1^2 = Id($)
[ 3]    Order 3           Length 1
        GrpPC of order 3
        PC-Relations:
          $.1^3 = Id($)
[ 4]    Order 3           Length 1
        GrpPC of order 3
        PC-Relations:
          $.1^3 = Id($)
[ 5]    Order 3           Length 2
        GrpPC of order 3
        PC-Relations:
          $.1^3 = Id($)
[ 6]    Order 6           Length 1
        GrpPC of order 6 = 2 * 3
        PC-Relations:
          $.1^2 = Id($),
          $.2^3 = Id($),
          $.2^$.1 = $.2^2
[ 7]    Order 6           Length 3
        GrpPC of order 6 = 2 * 3
        PC-Relations:
          $.1^2 = Id($),
          $.2^3 = Id($)
[ 8]    Order 9           Length 1
        GrpPC of order 9 = 3^2
        PC-Relations:
          $.1^3 = Id($),
          $.2^3 = Id($)
[ 9]    Order 18          Length 1
        GrpPC of order 18 = 2 * 3^2
        PC-Relations:
          $.1^2 = Id($),
          $.2^3 = Id($),
          $.3^3 = Id($),
          $.3^$.1 = $.3^2
```

63.9 Quotient Groups

63.9.1 Construction of Quotient Groups

One of the strengths of representing groups with polycyclic or power-conjugate presentations is that arbitrary quotient groups can be computed. Given (generators for) a normal subgroup of a pc-group, MAGMA will compute a pc-presentation for the quotient and the corresponding canonical homomorphism.

The `pQuotient` function, which can be used to find a prime-power quotient of a finitely-presented group, can also be used to compute quotients of pc-groups.

`quo< G | L >`

Construct the quotient Q of the pc-group G by the normal subgroup N, where N is the smallest normal subgroup of G containing the elements specified by the terms of the *generator list* L.

The possible forms of a term $L[i]$ of the generator list are the same as for the `sub`-constructor.

The quotient group Q and the corresponding natural homomorphism $f : G \to Q$ are returned.

`G / N`

Given a normal subgroup N of the pc-group G, construct the quotient of G by N.

Example H63E19

We will compute $O_{3',3}(G)$, where G is a pc-representation of the symmetric group S_4. The subgroup is defined by $O_{3',3}(G)/O_{3'}(G) = O_3(G/O_{3'}(G))$.

```
> G := PCGroup(Sym(4));
> N := pCore(G,-3);
> Q,f := quo<G|N>;
> Q;
GrpPC : Q of order 6 = 2 * 3
PC-Relations:
    Q.1^2 = Id(Q),
    Q.2^3 = Id(Q),
    Q.2^Q.1 = Q.2^2
> S := pCore(Q,3);
> H := S @@ f;
> H;
GrpPC : H of order 12 = 2^2 * 3
PC-Relations:
    H.1^3 = Id(H),
    H.2^2 = Id(H),
    H.3^2 = Id(H),
    H.2^H.1 = H.2 * H.3,
    H.3^H.1 = H.2
```

63.9.2 Abelian and p-Quotients

A number of standard quotients may be constructed.

`AbelianQuotient(G)`

The maximal abelian quotient G/G' of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism $\pi : G \to G/G'$ is returned as second value.

`AbelianQuotientInvariants(G)`

`AQInvariants(G)`

A sequence of integers giving the abelian invariants of the maximal abelian quotient of G.

`ElementaryAbelianQuotient(G, p)`

The maximal p-elementary abelian quotient Q of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism $\pi : G \to Q$ is returned as second value.

`pQuotient(G, p, c : parameters)`

`Workspace`	RngIntElt	*Default :* 5000000
`Metabelian`	BoolElt	*Default :* `false`
`Exponent`	RngIntElt	*Default :* 0
`Print`	RngIntElt	*Default :* 0

Given a pc-group G, a prime p, and a positive integer c, this function constructs a consistent power-conjugate presentation for the largest p-quotient P of G having lower exponent-p class at most c. If c is given as zero, then the limit 127 is placed on the class.

The function also returns the natural homomorphism π from G to P, a sequence S describing the definitions of the pc-generators of P and a flag indicating whether P is the maximal p-quotient of G.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $P.k$ of P as follows.

- If $S[k] = [0, r]$, then $P.k$ is defined via the image of $G.r$ under π.
- If $S[k] = [r, 0]$, then $P.k$ is defined via the power relation for $P.r$.
- If $S[k] = [r, s]$, then $P.k$ is defined via the conjugate relation involving $P.r^{P.s}$.

63.10 Normal Subgroups and Subgroup Series

63.10.1 Characteristic Subgroups

`Centre(G)`
`Center(G)`

The centre of the group G.

`CommutatorSubgroup(G)`
`DerivedSubgroup(G)`
`DerivedGroup(G)`

The derived subgroup of the group G.

`FittingSubgroup(G)`
`FittingGroup(G)`

The Fitting subgroup of the group G.

`FrattiniSubgroup(G)`

The Frattini subgroup of the group G.

`Hypercentre(G)`
`Hypercenter(G)`

The hypercentre of the group G, i.e. the stationary term in the upper central series for G.

`MinimalNormalSubgroups(G)`

A sequence containing all minimal normal subgroups of G.

`pCore(G, S)`

The maximal normal π-subgroup of G, $O_\pi(G)$, where π is defined by S. The argument S may be a set of primes, a single prime, or the negation of a single prime. If $S = -p$, then $O_{p'}(G)$ is returned.

`Socle(G)`

The socle of G.

63.10.2 Subgroup Series

`AbelianBasis(G)`

Given an abelian group G, return sequences B and I such that order$(B[i]) = I[i]$ and $\langle B \rangle = G$ and the terms of I give the types of each p-primary component of G.

`AbelianInvariants(G)`

`Invariants(G)`

The abelian invariants of the abelian group G as a sequence of integers.

`ChiefSeries(G)`

A chief series for the group G. The series is returned as a sequence of subgroups of G.

`CompositionSeries(G)`

A composition series for the group G. The series is returned as a sequence of subgroups of G. The i-th term of the composition series has a presentation given by the generators $G.i$ through $G.$`NPCgens`(G) and relations involving those generators only.

`CompositionFactors(G)`

A sequence of integer tuples that describe the composition factors, ordered according to some composition series for the group G. Since each factor will be a cyclic group of prime order, the tuples will each be of the form $< 19, 0, q >$ representing the cyclic group of order q. The sequence has a custom print routine.

`CompositionSeries(G, i)`

The $i+1$-th entry of the composition series for the group G. Its presentation is given by the generators $G.(i+1)$ through $G.m$, where m is the number of pc-generators of G and relations involving these generators only.

`DerivedSeries(G)`

The derived series of the group G. The series is returned as a sequence of subgroups.

`DerivedLength(G)`

The derived length of the group G.

`ElementaryAbelianSeries(G)`

An elementary abelian series is a chain of normal subgroups with the property that the quotient of each pair of successive terms in the series is elementary abelian. The elementary abelian series for the group G is returned as a sequence of subgroups.

`ElementaryAbelianSeriesCanonical(G)`

Gives a similar result to using `ElementaryAbelianSeries`, except the series returned depends only on the isomorphism type of the group, and consists of characteristic subgroups. This function may be slower than `ElementaryAbelianSeries`.

`LowerCentralSeries(G)`

The lower central series for the group G. The series is returned as a sequence of subgroups.

`NilpotencyClass(G)`

If G is nilpotent, return the nilpotence class of G. Otherwise, -1 is returned.

`pCentralSeries(G, p)`

The p-central series for G, where p is a prime dividing $|G|$. The series is returned as a sequence of subgroups. The p-central series $P_1 \triangleright P_2 \triangleright \cdots \triangleright P_i$ of a soluble group G is defined inductively as follows:

$P_1 = G$,

$P_{i+1} = (G, P_i)P_i^p$, for $i > 0$.

`SubnormalSeries(G, H)`

Given a group G and a subgroup H of G, return a sequence of subgroups commencing with G and terminating with H, such that each subgroup is normal in the previous one. If H is not subnormal in G, the empty sequence is returned.

`UpperCentralSeries(G)`

The upper central series of G. The series is returned as a sequence of subgroups.

Example H63E20

The elementary abelian series of the group $D_3 \wr D_5$ has terms of the following orders:

```
> H := DihedralGroup(GrpPerm, 5);
> G := WreathProduct(DihedralGroup(GrpPC, 3), DihedralGroup(GrpPC, 5),
>       [H.2, H.1]);
> EAS := ElementaryAbelianSeries(G);
> for i := 1 to #EAS do
>    print FactoredOrder(EAS[i]);
> end for;
    [ <2, 6>, <3, 5>, <5, 1> ]
    [ <2, 4>, <3, 5>, <5, 1> ]
    [ <2, 4>, <3, 5> ]
    [ <3, 5> ]
    []
```

Hence the elementary abelian factors can be seen to have sizes 2^2, 5, 2^4, and 3^5, reading from top to bottom.

63.10.3 Series for p-groups

The following functions are only defined for a pc-group which is a p-group.

`Agemo(G, i)`

Given a p-group G, return the characteristic subgroup of G generated by the elements x^{p^i}, $x \in G$, where i is a positive integer.

`Omega(G, i)`

Given a p-group G, return the characteristic subgroup of G generated by the elements of order dividing p^i, where i is a positive integer.

`JenningsSeries(G)`

Given a p-group G, return the Jennings series for G. The series is returned as a sequence of subgroups. The Jennings series $J_1 \triangleright J_2 \triangleright \cdots \triangleright J_i \cdots$ of a p-group G is defined inductively as follows:

$J_1 = G$,

$J_{i+1} =< (J_i, G), J_k^p >$, with $k = \lceil (i+1)/p \rceil$, $i > 0$.

`pClass(G)`

The lower exponent-p class of the p-group G.

`pRanks(G)`

A sequence whose i-th entry is the number of pc-generators for the lower exponent-p class i quotient of the p-group G.

63.10.4 Normal Subgroups and Complements

`NormalSubgroups(G)`

The collection of all normal subgroups of G returned as a sequence.

`NormalLattice(G)`

The lattice of normal subgroups of G.

`MinimalNormalSubgroup(G)`

An elementary abelian minimal normal subgroup of the soluble group G.

`MinimalNormalSubgroup(G, N)`

Given a non-trivial, normal subgroup N of G, return an elementary abelian minimal normal subgroup of G contained in N.

`Complements(G, N)`

Given a normal subgroup N of G, return conjugacy class representatives of all complements of N in G. This function implements the first cohomology computation described in [CNW90].

`NormalComplements(G, N)`

Given a normal subgroup N of G, return all normal complements of N in G. This function implements the first cohomology computation described in [CNW90].

`NormalComplements(G, H, N)`

Given a normal subgroup N of G, and a normal subgroup H of G containing N, return all complements of N in H which are normal in G. This function implements the first cohomology computation described in [CNW90].

Example H63E21

We define the direct product of an extraspecial group of order 3^3 and D_3 and let N be the first factor of this product. Inside the Sylow 3-subgroup, we see that N has 11 classes of complements, three of which are normal.

```
> A := ExtraSpecialGroup(GrpPC,3,1);
> B := DihedralGroup(GrpPC,3);
> G,f,p := DirectProduct(A,B);
> N := f[1](A);
> S3 := Sylow(G,3);
> cS := Complements(S3,N);
> [Index(S3,Normalizer(S3,t)):t in cS];
[ 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3 ]
```

We can compute only the normal complements by using `NormalComplements`.

```
> ncS := NormalComplements(S3,N);
> #ncS;
3
```

We can check that precisely one of these three complements is actually normal in G.

```
> [IsNormal(G,t):t in ncS];
[ true, false, false ]
```

Since N has a G-normal complement in $S3$, we must have $S3$ normal in G. We can verify this. Using the three-parameter version of `NormalComplements` we can directly compute the G-normal complements of N in $S3$.

```
> IsNormal(G,S3);
true
> ncG := NormalComplements(G,S3,N);
> #ncG;
1
> #NormalComplements(G,N);
```

```
1
```

63.11 Cosets

63.11.1 Coset Tables and Transversals

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a group G and a subgroup H of G, this function returns

(a) An indexed set of elements T of G forming a right transversal for G over H; and

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and g in G, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

`CosetTable(G, H)`

Given a group G and a subgroup H of G of index r, return a mapping $M : \langle\{1..r\}, G\rangle \to \{1..r\}$ describing the action of G on the (right) cosets of H.

`Transversal(G, H, K)`

An indexed set of representatives for the double cosets HuK in G, and the corresponding transversal mapping. The algorithm used is described in [Sla01].

`ShortCosets(p, H, G)`

Computes a set of representatives for the transversal of G modulo H of all cosets that contain p. This computation does not do a full transversal of G modulo H and may therefore be used even if the index of $(G : H)$ is very large.

63.11.2 Action on a Coset Space

`CosetAction(G, H)`

Given a subgroup H of the group G, construct the permutation representation of G given by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The natural homomorphism $f : G \to L$;

(b) The induced group L;

(c) The kernel K of the action (a subgroup of G).

`CosetImage(G, H)`

Given a subgroup H of the group G, construct the image L of G given by the action of G on the set of (right) cosets of H in G. L is returned as a permutation group.

`CosetKernel(G, H)`

Given a subgroup H of the group G, construct the kernel of the action of G on the set of (right) cosets of H in G.

63.12 Automorphism Group

63.12.1 General Soluble Group

In the case of a soluble non-p-group there are two algorithms available. By default, a lifting-based algorithm developed by M. Smith [Smi94] and extended by Smith and Slattery to use second cohomology is used. Alternatively, there is also an algorithm developed by D. Howden [How12], which uses the automorphism group of a Sylow p-subgroup of G to construct the automorphism group of G.

63.12.1.1 Lifting Algorithm

The automorphism group is computed step-by-step considering a series of factors of G by terms of a characteristic series

$$G = G_1 > G_2 > \cdots > G_k > 1$$

such that G_i/G_{i+1} is elementary abelian for each i. We compute the automorphism group $\mathrm{Aut}(G/G_2) \cong \mathrm{GL}(d, p)$ for some integer d and prime p, and then lift through each of the elementary abelian layers $G/G_3, \ldots, G/G_{k-1}$, finally arriving at $\mathrm{Aut}(G/G_k) = \mathrm{Aut}(G)$.

A group of type `GrpAuto` is returned. Details of the computation can be seen by setting the verbose flag `AutomorphismGroup` to `true`, and the characteristic series is available as the attribute `CharacteristicSeries` on the returned group.

In addition to the usual properties of `GrpAuto` (such as `Order`, `Ngens`, etc.), two special fields, `GenWeights` and `WeightSubgroupOrders`, are provided for automorphism groups of (non p-group) pc-groups. These each relate to weight subgroups of the automorphism group. Let i be the largest subscript such that the automorphism acts trivially on G/G_i. Then the automorphism is said to have weight $2i + 1$ if it acts non-trivially on G_i/G_{i+1}, and weight $2i + 2$ if it acts trivially on G_i/G_{i+1}. Note that there are no automorphisms of weight 2. The automorphisms of weight greater than or equal to a given value form a normal subgroup of A.

`AutomorphismGroup(G)`

Given a soluble group G presented by a pc-presentation, this function returns the automorphism group of G as a group of type `GrpAuto`.

`HasAttribute(A, "GenWeights")`

If the attribute `GenWeights` is defined for A then the function returns `true`, and a sequence of integers. This integer sequence indicates where each generator lies in the normal series of A corresponding to the action of the group on G (as described at the beginning of the section). If the attribute is not set then the sequence is unassigned. The function call `AutomorphismGroup(G)`, where G has type `GrpPC`, always returns an automorphism group with this attribute set. In this case the sequence may also be obtained by the short form `A‘GenWeights`.

```
HasAttribute(A, "WeightSubgroupOrders")
```

If the attribute `WeightSubgroupOrders` is defined for A then the function returns `true`, and a sequence of integers. This sequence of integers gives the orders for the normal series of weight subgroups described at the beginning of the section. If the attribute is not set then the sequence is unassigned. The function call `AutomorphismGroup(G)`, where G has type `GrpPC`, always returns an automorphism group with this attribute set. In this case the sequence may also be obtained by the short form `A‘WeightSubgroupOrders`.

Example H63E22

An example using `AutomorphismGroup` and some related features. We build a group based on the structure of a finite field (multiplicative group acting on the additive group) and then compute its automorphism group. First, we set up the field.

```
> E := GF(2);
> F := GF(8);
> V,phi := VectorSpace(F,E);
> d := Dimension(V);
> x := PrimitiveElement(F);
```

Then, define a pc-group to act and define the action based on the multiplication in the field. Compute the matrix by mapping the vectors back to the field, multiplying by x, and then recording the result.

```
> C := CyclicGroup(GrpPC,Order(x));
> MR := MatrixRing(E, d);
> s := [];
> for i := 1 to d do
>  y := ((V.i)@@phi)*x;
>  s cat:= Eltseq(y);
> end for;
```

Turn the sequence of image components into a matrix and use the matrix to create a C-module. Then use that module to create the split extension.

```
> t := MR!s;
> M := GModule(C,[t]);
> G := Extension(M,C);
> G;
GrpPC : G of order 56 = 2^3 * 7
PC-Relations:
    G.1^7 = Id(G),
    G.2^2 = Id(G),
    G.3^2 = Id(G),
    G.4^2 = Id(G),
    G.2^G.1 = G.3,
    G.3^G.1 = G.4,
```

```
    G.4^G.1 = G.2 * G.3
```

Then we can compute the automorphism group of G.

```
> A := AutomorphismGroup(G);
> A;
A group of automorphisms of GrpPC : G
Generators:
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1^2
        G.2 |--> G.3 * G.4
        G.3 |--> G.2 * G.4
        G.4 |--> G.3
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1
        G.2 |--> G.2 * G.3
        G.3 |--> G.3 * G.4
        G.4 |--> G.2 * G.3 * G.4
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1 * G.2 * G.3
        G.2 |--> G.2
        G.3 |--> G.3
        G.4 |--> G.4
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1 * G.3 * G.4
        G.2 |--> G.2
        G.3 |--> G.3
        G.4 |--> G.4
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1 * G.4
        G.2 |--> G.2
        G.3 |--> G.3
        G.4 |--> G.4
> [Order(x):x in Generators(A)];
[ 3, 2, 7, 2, 2 ]
```

Next, we can use the automorphisms to create an extension of G.

```
> b := A.1;
> Order(b);
3
> tau := hom<G->G|[b(G.i):i in [1..NPCgens(G)]]>;
> D := CyclicGroup(GrpPC,Order(b));
> K := Extension(G,D,[tau]);
> K;
GrpPC : K of order 168 = 2^3 * 3 * 7
PC-Relations:
    K.1^3 = Id(K),
    K.2^7 = Id(K),
    K.3^2 = Id(K),
```

```
    K.4^2 = Id(K),
    K.5^2 = Id(K),
    K.2^K.1 = K.2^2,
    K.3^K.1 = K.4 * K.5,
    K.3^K.2 = K.4,
    K.4^K.1 = K.3 * K.5,
    K.4^K.2 = K.5,
    K.5^K.1 = K.4,
    K.5^K.2 = K.3 * K.4
> #Classes(K);
8
```

Finally, we examine information about the weight subgroups. We list only the orders of the terms of the characteristic series in G in order to save space.

```
> [Order(H): H in A‘CharacteristicSeries];
[ 56, 8, 1 ]
> A‘GenWeights;
[ 1, 3, 4, 4, 4 ]
> A‘WeightSubgroupOrders;
[ 168, 56, 56, 8 ]
```

63.12.1.2 Lifting from the Automorphism Group of a Sylow p-subgroup

The algorithm developed by D. Howden [How12] follows a different strategy to the Smith-Slattery lifting approach. Given a soluble group G, the algorithm determines a suitable Sylow p-subgroup P of G, and uses the automorphism group of P (using the algorithm for p-groups detailed below) to construct the automorphism group of G.

In cases where the automorphism group is soluble, the algorithm automatically constructs a pc-representation for it. Solublility of the returned group can then be tested via the `IsSoluble` intrinsic. The pc-representation obtained using the `PCGroup` intrinsic, and pc-generators (as automorphism maps) via the `PCGenerators` intrinsic.

In some cases where the automorphism group is not soluble, the algorithm will construct a permutation representation during its construction.

A group of type `GrpAuto` is returned. Details of the computation can be seen by setting the verbose flag `AutomorphismGroupSolubleGroup` to 1.

A variation of this algorithm can be used for isomorphism testing.

`AutomorphismGroupSolubleGroup(G: parameters)`

Given a soluble group G presented by a pc-presentation, this function returns the automorphism group of G as a group of type GrpAuto.

p [RngIntElt] *Default :* 1;

A prime p dividing the order of G. The automorphism group of the Sylow p-subgroup is then used to construct the automorphism group of G. If the p-core of G is trivial

then an error is given. The default value of $p = 1$ indicates that the algorithm should take p to be the prime giving the largest Sylow p-subgroup of G.

`IsIsomorphicSolubleGroup(G, H: parameters)`

Returns `true` if the soluble groups G, H presented by pc-presentations, are isomorphic, and `false` otherwise. Where the groups are isomorphic, a mapping $G \to H$ is also returned.

`p` [RngIntElt] *Default :* 1;

A prime p dividing the order of G (assuming that the orders of G and H are the same). The algorithm then tests Sylow p-subgroups of G and H for isomorphism and attempts to extend these isomorphisms to an isomorphism $G \to H$. If the p-cores of G and H are trivial then an error is given. The default value of $p = 1$ indicates that the algorithm should take p to be the prime giving the largest Sylow p-subgroup of G.

Example H63E23

An example using `AutomorphismGroupSolubleGroup` and some related features. We use a group from the `solgps` library.

```
> load solgps;
> G := G10();
> FactoredOrder(G);
[ <2, 18>, <7, 4> ]
> time A := AutomorphismGroupSolubleGroup(G);
Time: 18.220
> time R_A, phi_A := PCGroup(A);
Time: 0.000
> FactoredOrder(R_A);
[ <2, 20>, <3, 2>, <7, 6> ]
```

63.12.2 p-group

For a description of the algorithm used to construct the automorphism group of a p-group, see [ELGO02].

While it is difficult to state very firm guidelines for the performance of the algorithm, our experience suggests that it has most difficulty in constructing automorphism groups of p-groups of "large" Frattini rank (say rank larger than about 6) and p-class 2. If the group has larger p-class, then it usually has more characteristic structure and the algorithm exploits this. The *order* of a group is *not* a useful guide to the difficulty of the computation.

`SetVerbose ("AutomorphismGroup", 1)` provides information on the progress of the algorithm.

`AutomorphismGroup(G: parameters)`

The group G is a p-group described by a pc-presentation. The function returns the automorphism group of G as a group of type `GrpAuto`.

`CharacteristicSubgroups`

[GrpPC] *Default :* [];

A list of known characteristic subgroups of G; these may improve the efficiency of the construction. Note that the algorithm simply accepts that the supplied subgroups are fixed under the action of the automorphism group; it does *not* verify that they are in fact characteristic.

Example H63E24

```
> G := SmallGroup (64, 78);
> A := AutomorphismGroup (G);
> #A;
1024
> A.1;
    Automorphism of GrpPC : G which maps:
        G.1 |--> G.1
        G.2 |--> G.2
        G.3 |--> G.1 * G.3
        G.4 |--> G.4
        G.5 |--> G.5
        G.6 |--> G.4 * G.6
> Order (A.1);
4
> a := A.1^2; [a (G.i): i in [1..6]];
[ G.1, G.2, G.3 * G.5, G.4, G.5, G.6 ]
```

`OrderAutomorphismGroupAbelianPGroup(A)`

Order of automorphism group of abelian p-group G where $A = [a_1, a_2, \ldots]$ and $G = C_{a_1} \times C_{a_2} \times \ldots$.

Example H63E25

Subgroups of $C_4 \times C_8 \times C_{64}$.

```
> NumberOfSubgroupsAbelianPGroup ([4, 8, 64]);
[ 7, 35, 91, 139, 171, 171, 139, 91, 35, 7, 1 ]
```

Hence, for example, there are 7 subgroups of order 2 and 139 subgroups of order 2^4.

```
> OrderAutomorphismGroupAbelianPGroup ([4, 8, 64]);
4194304
```

63.12.3 Isomorphism and Standard Presentations

The `pQuotient` command returns a power-conjugate presentation for a given p-group but this presentation depends on the user-supplied description of the group. The Standard Presentation algorithm computes a "canonical" presentation for the p-group, which is independent of the user-supplied description. For a description of this algorithm, see [O'B94].

The canonical or *standard* presentation of a given p-group is the power-conjugate presentation obtained when a description of the group is computed using the default implementation of the p-group generation algorithm.

Hence, two groups in the same isomorphism class have identical standard presentations. Given two p-groups, if their standard presentations are identical, then the groups are isomorphic, otherwise they are not. Hence to decide whether two groups are isomorphic, we can first construct the standard presentation of each using the `StandardPresentation` function and then compare these presentations using the `IsIdenticalPresentation` function.

While it is difficult to state very firm guidelines for the performance of the algorithm, our experience suggests that the difficulty of deciding isomorphism between p-groups is governed by their Frattini rank and is most practical for p-groups of rank at most 5. The *order* of a group is *not* a useful guide to the difficulty of the computation.

`SetVerbose ("Standard", 1)` will provide information on the progress of the algorithm.

`StandardPresentation(G)`

`StandardPresentation(G:` *parameters*`)`

The group G is a p-group presented by an arbitrary pc-presentation. The group H defined by its standard presentation is returned together with a map from G to H.

`StartClass` RngIntElt *Default :* 1

If `StartClass` is k, then use `pQuotient` to construct the class $k-1$ p-quotient of G and standardize the presentation only from class k onwards.

`IsIdenticalPresentation(G, H)`

Returns `true` if G and H have identical presentations, `false` otherwise.

`IsIsomorphic(G, H)`

The function returns `true` if the p-groups G and H are isomorphic, `false` otherwise. It constructs their standard presentations class by class, and checks for equality. If they are isomorphic, it also returns an isomorphism from G to H.

Example H63E26

In the next two examples, we investigate whether particular p-quotients of fp-groups are isomorphic.

```
> F<x, y, t> := FreeGroup(3);
```

```
> G := quo< F | x*y^2*x^-1=y^-2, y*x^2*y^-1=x^-2, x^2=t^2, y^2=(t^-1*x)^2,
>                       t*(x*y)^2=(x*y)^2*t >;
> Q1 := pQuotient(G, 2, 3: Print := 1);
Lower exponent-2 central series for G
Group: G to lower exponent-2 central class 1 has order 2^3
Group: G to lower exponent-2 central class 2 has order 2^7
Group: G to lower exponent-2 central class 3 has order 2^11
> H := quo< F | x*y^2*x^-1=y^-2, y*x^2*y^-1=x^-2, x^2=t^2*(x*y)^2,
>                       y^2=(t^-1*x)^2, t*(x*y)^2=(x*y)^2*t >;
> Q2 := pQuotient(H, 2, 3: Print := 1);
Lower exponent-2 central series for H
Group: H to lower exponent-2 central class 1 has order 2^3
Group: H to lower exponent-2 central class 2 has order 2^7
Group: H to lower exponent-2 central class 3 has order 2^11
```

Now check whether the class 3 2-quotients are isomorphic.

```
> IsIsomorphic(Q1, Q2);
false
```

In the next example, we construct an explicit isomorphism between two 5-groups.

```
> F<a, b> := Group<a, b | a^5, b^5, (a * b * a)^5 = (b, a, b) >;
> G := pQuotient (F, 5, 6 : Print := 1);
Lower exponent-5 central series for F
Group: F to lower exponent-5 central class 1 has order 5^2
Group: F to lower exponent-5 central class 2 has order 5^3
Group: F to lower exponent-5 central class 3 has order 5^4
Group: F to lower exponent-5 central class 4 has order 5^5
Group: F to lower exponent-5 central class 5 has order 5^7
Group: F to lower exponent-5 central class 6 has order 5^8
> G;
GrpPC : G of order 390625 = 5^8
PC-Relations:
G.2^G.1 = G.2 * G.3,
G.3^G.1 = G.3 * G.4,
G.3^G.2 = G.3 * G.6 * G.7^4 * G.8^4,
G.4^G.1 = G.4 * G.5,
G.4^G.2 = G.4 * G.7 * G.8,
G.4^G.3 = G.4 * G.7^4 * G.8,
G.5^G.1 = G.5 * G.6,
G.5^G.2 = G.5 * G.7,
G.5^G.3 = G.5 * G.8^2,
G.6^G.2 = G.6 * G.8,
G.7^G.1 = G.7 * G.8^3
> K<a, b> := Group<a, b | a^5, b^5, (b * a * b)^5 = (b, a, b) >;
> H := pQuotient (K, 5, 6 : Print := 1);
Lower exponent-5 central series for K
Group: K to lower exponent-5 central class 1 has order 5^2
Group: K to lower exponent-5 central class 2 has order 5^3
```

```
Group: K to lower exponent-5 central class 3 has order 5^4
Group: K to lower exponent-5 central class 4 has order 5^5
Group: K to lower exponent-5 central class 5 has order 5^7
Group: K to lower exponent-5 central class 6 has order 5^8
> H;
GrpPC : H of order 390625 = 5^8
PC-Relations:
H.2^H.1 = H.2 * H.3,
H.3^H.1 = H.3 * H.4,
H.3^H.2 = H.3 * H.6^2 * H.7^2 * H.8^2,
H.4^H.1 = H.4 * H.5,
H.4^H.2 = H.4 * H.7,
H.4^H.3 = H.4 * H.7^4 * H.8,
H.5^H.1 = H.5 * H.6,
H.5^H.2 = H.5 * H.7,
H.5^H.3 = H.5 * H.8^2,
H.6^H.2 = H.6 * H.8,
H.7^H.1 = H.7 * H.8^3
> flag, phi := IsIsomorphic (G, H);
> flag;
true
> for g in PCGenerators (G) do print g, "--->", phi (g); end for;
G.1 ---> H.1
G.2 ---> H.2^3 * H.4^3 * H.5^3 * H.6^2 * H.7^4 * H.8^3
G.3 ---> H.3^3 * H.5^3 * H.6^4 * H.8^3
G.4 ---> H.4^3 * H.6^3 * H.7^2 * H.8^3
G.5 ---> H.5^3 * H.8
G.6 ---> H.6^3
G.7 ---> H.7^4
G.8 ---> H.8^4
```

The functions `IsIsomorphic` and `StandardPresentation` are expensive. Here we have a list of groups and we want to find any isomorphisms among the collection. Rather than repeatedly applying `IsIsomorphic`, we first construct and store standard presentations for each group in the sequence, and then quickly compare these using `IsIdenticalPresentation`.

```
> F<a, b> := FreeGroup (2);
> p := 7;
> Q := [];
> for k := 1 to p - 1 do
>       G := quo< F | a^p = (b, a, a), b^p = a^(k*p), (b, a, b)>;
>       H := pQuotient (G, p, 10);
>       Q[k] := StandardPresentation (H);
> end for;
```

Now run over the list of standard presentations and check for equality.

```
> for i in [2..p - 1] do
>    for j in [1.. i - 1] do
>       if IsIdenticalPresentation (Q[i], Q[j]) then
```

```
>          print "Standard Presentations  ", i, " and ", j, " are identical";
>       end if;
>    end for;
> end for;
Standard Presentations   2  and  1  are identical
Standard Presentations   4  and  1  are identical
Standard Presentations   4  and  2  are identical
Standard Presentations   5  and  3  are identical
Standard Presentations   6  and  3  are identical
Standard Presentations   6  and  5  are identical
```

63.13 Generating p-groups

The p-central series of a group G is the descending sequence of subgroups

$$G = P_0(G) \geq \ldots \geq P_{i-1}(G) \geq P_i(G) \geq \ldots \geq$$

where $P_i(G) = [P_{i-1}(G), G]P_{i-1}(G)^p$ for $i \geq 1$.

If $P_c(G) = 1$ and c is the smallest such integer then G has *p-class c*. A group with p-class c is nilpotent and has nilpotency class at most c.

Let G be a finite p-group with Frattini rank d and class c. A group H is a *descendant* of G if H has Frattini rank d and the quotient $H/P_c(H)$ is isomorphic to G. A group is an *immediate descendant* of G if it is a descendant of G and has class $c+1$.

The p-group generation algorithm allows the construction of (immediate) descendants of a p-group. For a description of this algorithm, see [New77, O'B90].

`SetVerbose ("GeneratepGroups", 1)` will provide information on the progress of the algorithm.

`GeneratepGroups (p, d, c : parameters)`

Generate all d-generator p-class at most c p-groups.

`Exponent` RngIntElt *Default :* 0

All groups constructed satisfy the supplied exponent.

`OrderBound` RngIntElt *Default :* 0

Given `OrderBound := n`, all groups constructed have order at most p^n.

`StepSizes` [RngIntElt] *Default :* []

Construct descendants of order $p^{(n+s)}$ of a group of order p^n only for s in `StepSizes`.

`All` BoolElt *Default :* `true`

If `true`, return all groups. Otherwise, return only the capable groups (those which have descendants).

`Descendants(G : parameters)`

`Descendants(G, c : parameters)`

Construct descendants of G having p-class at most c; if c is not supplied, it is assumed to be one larger than the p-class of G. This function supports the same variable arguments as `GeneratepGroups`.

Example H63E27

```
> G := DihedralGroup(GrpPC, 16);
> T := Descendants (G, 8);
> #T;
12
> H := T[5];
> H;
GrpPC : H of order 128 = 2^7
PC-Relations:
    H.1^2 = H.7,
    H.2^2 = H.3 * H.4,
    H.3^2 = H.4 * H.5,
    H.4^2 = H.5 * H.6,
    H.5^2 = H.6 * H.7,
    H.6^2 = H.7,
    H.2^H.1 = H.2 * H.3,
    H.3^H.1 = H.3 * H.4,
    H.4^H.1 = H.4 * H.5,
    H.5^H.1 = H.5 * H.6,
    H.6^H.1 = H.6 * H.7
```

Example H63E28

What is the soluble length of a 2-generator group of exponent 4? We construct the 2-generator 2-groups having exponent 4.

```
> T := GeneratepGroups(2, 2, 10: Exponent := 4);
> "The number of 2-generator exponent 4 groups is ", # T;
The number of 2-generator exponent 4 groups is  26
```

What are their soluble lengths?

```
> for i := 1 to #T do
>       "Group ", i, " has soluble length ", DerivedLength (T[i]);
> end for;
Group  1  has soluble length  1
Group  2  has soluble length  2
Group  3  has soluble length  2
Group  4  has soluble length  1
Group  5  has soluble length  2
Group  6  has soluble length  2
```

```
Group  7  has soluble length  2
Group  8  has soluble length  2
Group  9  has soluble length  2
Group  10  has soluble length  2
Group  11  has soluble length  2
Group  12  has soluble length  2
Group  13  has soluble length  2
Group  14  has soluble length  2
Group  15  has soluble length  2
Group  16  has soluble length  2
Group  17  has soluble length  2
Group  18  has soluble length  2
Group  19  has soluble length  2
Group  20  has soluble length  2
Group  21  has soluble length  3
Group  22  has soluble length  3
Group  23  has soluble length  3
Group  24  has soluble length  3
Group  25  has soluble length  3
Group  26  has soluble length  3
```

Example H63E29

Can we find all 2-generator 3-groups of abundance zero? Such groups have order at most 3^5. First, we define a function which checks the number of conjugacy classes of a group (to determine abundance).

```
> IsGoodGroup := function(G, k)
>
>    ncl := # Classes(G);
>
>    O := FactoredOrder(G);
>    p := O[1][1];
>    m := O[1][2];
>    n := Floor(m / 2);
>    e := m - n * 2;
>    Desired := n * (p^2 - 1) + p^e + k * (p - 1) * (p^2 - 1);
>
>    return (Desired eq ncl);
>
> end function;
```

Then, we generate the potential candidates and check each.

```
> a := GeneratepGroups (3, 2, 4 : OrderBound := 5);
> #a;
42
>
> for i := 1 to #a do
```

```
>        G := a[i];
>        if IsGoodGroup(G, 0) then
>           "Group ", i, " of order ", Order(G), " has abundance 0";
>        end if;
> end for;
Group  1  of order  9  has abundance 0
Group  3  of order  27  has abundance 0
Group  4  of order  27  has abundance 0
Group  11  of order  81  has abundance 0
Group  12  of order  81  has abundance 0
Group  13  of order  81  has abundance 0
Group  14  of order  81  has abundance 0
Group  40  of order  243  has abundance 0
Group  41  of order  243  has abundance 0
Group  42  of order  243  has abundance 0
```

`ClassTwo(p, d :` *parameters*`)`

`ClassTwo(p, d, Step :` *parameters*`)`

`ClassTwo(p, d, s :` *parameters*`)`

Count the d-generator p-groups of p-class 2. If s or *Step* is supplied, then count only those of order $p^{(d+s)}$ or $p^{(d+m)}$ for $m \in Step$. In the first two invocations, the sequence returns a sequence of length $\binom{d}{2}$, whose m-th entry is the number of groups of $p^{(d+m)}$. (Some additional entries may be deduced on the basis of duality.) The last invocation returns the number of groups of $p^{(d+s)}$. For details of the algorithm used see [EO99].

`Exponent` RngIntElt *Default :* 0

If `Exponent` is `true`, count those groups which have exponent p. The directive `SetVerbose ("ClassTwo", 1)` will provide information on the progress of the algorithm.

Example H63E30

Count the number of 3-generator p-class 2 5-groups.

```
> ClassTwo(5, 3);
[ 4, 19, 42, 19, 4, 1 ]
```

For example, the number of 3-generator 5-groups of order 5^6 and p-class 2 is precisely 42.
Count the number of 4-generator p-class 2 5-groups of order 5^7.

```
> ClassTwo(5, 4, 3);
6598
```

63.14 Representation Theory

Chapter 91 on characters describes many functions for computing with partial character tables or individual characters.

`CharacterDegrees(G)`

`CharacterDegrees(G, z, p)`

Given a finite pc-group G, return the sequence $[\langle d_1, c_1\rangle, \langle d_2, c_2\rangle, \ldots]$, where c_i is the number of irreducible characters of G having degree d_i. For details of the algorithm see Conlon [Con90b].

The second form requires z to be a central element of G and p to be a prime or zero. The sequence returned enumerates the number of absolutely irreducible characters of G in characteristic p, lying over some faithful linear character of $\langle z\rangle$.

`CharacterDegrees(G)`

Given a finite p-group G, return the sequence $[\langle d_1, c_1\rangle, \langle d_2, c_2\rangle, \ldots]$, where c_i is the number of irreducible characters of G having degree d_i. For details of the algorithm see [Sla86].

`CharacterDegreesPGroup(G)`

Given a finite p-group G, return the sequence $[C_0, C_1, \ldots]$, where C_i is the number of irreducible characters of G having degree p^i. For details of the algorithm see [Sla86].

`CharacterTable(G:` *parameters*`)`

Construct the table of ordinary irreducible characters for the group G.

`Al` MonStgElt *Default :* "*Default*"

This parameter controls the algorithm used. The string `"DS"` forces use of the Dixon-Schneider algorithm. The string `"IR"` forces the use of Unger's induction/reduction algorithm [Ung06]. The `"Default"` algorithm is to use Dixon-Schneider for groups of order ≤ 5000 and Unger's algorithm for larger groups. This may change in future.

`DSSizeLimit` RngIntElt *Default :* 10^4

When the default algorithm is selected, a positive value n for `DSSizeLimit` means that before using Unger's algorithm, the full character space is split by some passes of Dixon-Schneider, restricted to using class matrices corresponding to conjugacy classes with size at most n.

`CharacterTableConlon(G)`

Given a finite p-group G, return the character table of G. The algorithm is due to Conlon, as described in [Con90].

`GModule(G, M)`

The G-module for the action of G on the vector space defined by the matrix ring M.

`GModule(G, A)`

A KG-module M corresponding to the action of the group G on the elementary abelian subgroup A of G is constructed. The map from A to the vector space underlying M is also returned.

`GModule(G, A, B)`

A KG-module M corresponding to the action of the group G on the elementary abelian section A/B of G is constructed. The map from A to the vector space underlying M is also returned.

`AbsolutelyIrreducibleRepresentationsSchur(G, k:` *parameters*`)`

`AbsolutelyIrreducibleModulesSchur(G, k:` *parameters*`)`

Compute the absolutely irreducible representations of the group G over appropriate extensions or sub-fields of the given field k. The representations returned are inequivalent and consist of all distinct representations, subject to the conditions imposed. The field k may be a finite field, the rationals or a cyclotomic field. In the case when k is a finite field, the Glasby-Howlett algorithm is used to determine the minimal field over which a representation may be realised. If k has characteristic 0, the field over which a representation is realised may not be minimal.

The representations are found using Schur's method of climbing the composition series for G defined by the pc-presentation. If the argument i is given then the algorithm will calculate only representations of the ith subgroup of the composition series.

The "Representations" function returns a list of homomorphisms $\rho : G \to GL(n, K)$, where K is a field compatible with k. The "Modules" version returns an equivalent list of G-modules.

`Process` BoolElt *Default :* `true`

If the parameter `Process` is set true then the list is a list of pairs comprising an integer and a representation. This list or any sublist of it is a suitable value for the argument L in the last versions of the function, and in this case only the representations in L will be extended up the series. This allows the user to inspect the representations produced along the way and cull any that are uninteresting.

`GaloisAction` MonStgElt *Default :* "Yes"

Possible values are `"Yes"`, `"No"` and `"Relative"` The default is `"Yes"` for intermediate levels and `"No"` for the whole group. The value `"Yes"` means that it only lists one representation from each orbit of the action of the absolute Galois group $Gal(K/\text{primefield}(K))$. Setting this parameter to `"No"` turns this reduction off (thus listing all inequivalent representations), while setting it to `"Relative"` uses the group $Gal(K/k)$.

MaxDimension RngIntElt *Default :*

Restrict the representations to those of dimension $\leq$ MaxDimension. Default is no restriction.

ExactDimension SetEnum *Default :*

If ExactDimension is assigned a set S of positive integers, attention is restricted to representations having dimensions lying in the set S. The default is equivalent to taking the set of all positive integers.

If both MaxDimension and ExactDimension are assigned values, then representations having dimensions that are either bounded by MaxDimension or contained in ExactDimension are produced.

IrreducibleRepresentationsSchur(G, k: *parameters*)

IrreducibleModulesSchur(G, k: *parameters*)

Compute irreducible representations of G over the given field k. All arguments and parameters are as for the absolutely irreducible case.

The computation proceeds by first computing the absolutely irreducible representations subject to the given parameters, then rewriting over the field k, with a consequent change of dimension of the representation.

Example H63E31

We compute representations of the dihedral group of order 20.

```
> G := DihedralGroup(GrpPC, 10);
> FactoredOrder(G);
[ <2, 2>, <5, 1> ]
```

First some modular representations with characteristic 2.

```
> r := IrreducibleModulesSchur(G, GF(2));
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2)
*]
> r := AbsolutelyIrreducibleModulesSchur(G, GF(2));
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(2^2),
    GModule of dimension 2 over GF(2^2)
*]
> r := AbsolutelyIrreducibleModulesSchur(G, GF(2) : GaloisAction:="Yes");
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(2^2)
```

```
*]
```

The irreducible representation of dimension 4 is not absolutely irreducible, as over $GF(4)$ it splits into two Galois-equivalent representations.
Getting irreducible representations over the complex field presents no problem, despite not being able to use the complex field as an argument to the function call. We could specify the field to be the cyclotomic field with degree equal to Exponent(G), but it is preferable to ask for absolutely irreducible representations over the rationals.

```
> r := AbsolutelyIrreducibleRepresentationsSchur(G, Rationals());
> r;
[*
    Mapping from: GrpPC: G to GL(1, RationalField()),
    Mapping from: GrpPC: G to GL(1, RationalField()),
    Mapping from: GrpPC: G to GL(1, RationalField()),
    Mapping from: GrpPC: G to GL(1, RationalField()),
    Mapping from: GrpPC: G to GL(2, CyclotomicField(5)),
    Mapping from: GrpPC: G to GL(2, CyclotomicField(5)),
    Mapping from: GrpPC: G to GL(2, CyclotomicField(5)),
    Mapping from: GrpPC: G to GL(2, CyclotomicField(5))
*]
> r[6](G.2);
[zeta_5^3        0]
[       0 zeta_5^2]
```

63.15 Central Extensions

We now describe functions to construct $H^2(G, U)$ for a finite soluble group G and finite abelian group U (a trivial G-module). We also present functions to construct central extensions of U by G.

Denote by $Z^2(G, U)$ the abelian group of all cocycles from G to U, under pointwise multiplication. The values $\psi(g, h)$ of $\psi \in Z^2(G, U)$ may be represented as a "cocyclic matrix" with entries in U.

If $\phi : G \to U$ is a set map with $\phi(1_G) = 1_U$, then there is a coboundary $\partial\phi \in Z^2(G, U)$ defined by $\partial\phi(g, h) = \phi(g)\phi(h)\phi(gh)^{-1}$. The group of all coboundaries from G to U is denoted $B^2(G, U)$, and we have $H^2(G, U) = Z^2(G, U)/B^2(G, U)$. Then $H^2(G, U) = I \times T$, where I is the (faithful) image of $\mathrm{Ext}(G/G', U) \leq H^2(G/G', U)$ under inflation, and T is the (faithful) image of $\mathrm{Hom}(H_2(G), U)$ under a certain transgression homomorphism. Here we provide functions which construct representatives for the elements in a generating set for each of these two factors.

For details of the theory and the algorithm used, see [FO00].

`SetVerbose ("Cocycle", 1)` will provide additional information on the calculations in the functions.

`ExtGenerators(G, U)`

Given a soluble group G and an abelian group U (both defined by pc-presentations) the function returns a sequence of tuples describing generators for $\mathrm{Ext}(G/G', U)$ as cocyclic matrices; the first entry in each tuple is a representative of a generator, the second is the order of the coset of the representative in $H^2(G, U)$.

`HomGenerators(G, U)`

Given a soluble group G and an abelian group U (both defined by pc-presentations) the function returns a sequence of tuples describing generators for $\mathrm{Hom}(H_2(G), U)$ as cocyclic matrices; the first entry in each tuple is a representative of a generator, the second is the order of the coset of the representative in $H^2(G, U)$.

`ElementSequence(G)`

For a soluble group G, the function returns an indexed set of elements of G listed in the order used by `ExtGenerators` and `HomGenerators`.

`RepresentativeCocycles(G, U, Ext, Hom)`

Let G be a soluble group G and U be an abelian group both defined by pc-presentations. Let Ext and Hom be the values returned by calling `ExtGenerators` and `HomGenerators` respectively. The function `RepresentativeCocycles` returns a complete and irredundant set of representatives for the elements of $H^2(G, U)$ as cocyclic matrices.

`CentralExtension(G, U, A)`

Let G be a soluble group G and U be an abelian group, both defined by pc-presentations. Further, let A be a cocyclic matrix (as determined by the function `RepresentativeCocycles`). Then, this function returns the central extension of U by G determined by the cocyclic matrix A.

`CentralExtensions(G, U, Q)`

If G is a soluble group G and U is an abelian group, both defined by pc-presentations, and Q is a sequence of cocyclic matrices (as determined by the function `RepresentativeCocycles`), this function returns the corresponding sequence of central extension of U by G determined by the sequence of cocyclic matrices A. Note that the central extensions thereby constructed need not be mutually non-isomorphic.

`CentralExtensionProcess(G, U)`

Given a soluble group G and an abelian group U (both defined by pc-presentations) the function creates a process P for central extensions of U by G. Note that the list of central extensions constructed by this process will contain all isomorphism types but the extensions need not be mutually non-isomorphic.

`NextExtension(~P)`

Given a central extension process P, construct the next central extension determined by P.

`IsEmpty(P)`

Return `true` if all central extensions determined by the process P have been constructed; otherwise return `false`.

Example H63E32

We compute the abelian invariants of $H^2(D_4, C_2)$.

```
> G := DihedralGroup(GrpPC, 4);
> U := AbelianGroup(GrpPC, [2]);
>
> Ext := ExtGenerators(G, U);
> Ext[1];
    <[Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) Id(U) Id(U) Id(U) U.1 U.1 U.1 U.1]
    [Id(U) Id(U) Id(U) Id(U) U.1 U.1 U.1 U.1]
    [Id(U) Id(U) Id(U) Id(U) U.1 U.1 U.1 U.1]
    [Id(U) Id(U) Id(U) Id(U) U.1 U.1 U.1 U.1], 2>,
>
> Hom := HomGenerators(G, U);
> Hom;
[
    <[Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) U.1 U.1 Id(U) Id(U) U.1 U.1 Id(U)]
    [Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U) Id(U)]
    [Id(U) U.1 U.1 Id(U) Id(U) U.1 U.1 Id(U)]
    [Id(U) Id(U) Id(U) U.1 Id(U) U.1 U.1 U.1]
    [Id(U) U.1 Id(U) Id(U) U.1 U.1 Id(U) U.1]
    [Id(U) Id(U) Id(U) U.1 Id(U) U.1 U.1 U.1]
    [Id(U) U.1 Id(U) Id(U) U.1 U.1 Id(U) U.1], 2>
]
>
> AbelianInvariants(Ext, Hom);
[ 2, 2, 2 ]
```

We now compute the central extension of U by G determined by a single cocyclic matrix.

```
> A := RepresentativeCocycles(G, U, Ext, Hom);
> E := CentralExtension(G, U, A[2]);
> E;
GrpPC : E of order 16 = 2^4
PC-Relations:
```

```
    E.1^2 = E.4,
    E.2^2 = E.3 * E.4,
    E.2^E.1 = E.2 * E.3
```

Alternatively we can build all central extensions of U by G.

```
> E := CentralExtensions(G, U, A);
> "Number of extensions is ", #E;
Number of extensions is  8
```

Next, we provide an example of using the central extension process. Firstly, we create the groups and initialize the process.

```
> G := SmallGroup(12, 5);
> U := AbelianGroup(GrpPC, [2, 3]);
> P := CentralExtensionProcess(G, U);
```

Now we run over the central extensions and count conjugacy classes.

```
> C := [];
> while IsEmpty(P) eq false do
>    NextExtension(~P, ~E);
>    Append(~C, #Classes (E));
> end while;
> "# conjugacy classes is ", C;
# conjugacy classes is  [ 45, 72, 45, 72, 45, 72, 45, 72, 45, 72, 45,
72, 45, 72, 45, 72, 45, 72, 45, 72, 45, 72, 45, 72 ]
```

63.16 Transfer Between Group Categories

63.16.1 Transfer to GrpPC

The `PolycyclicGroup`-constructor allows complete flexibility in defining a pc-group. However, it is often more convenient to have MAGMA compute a pc-presentation based on some other description of the group. The `PCGroup` function will produce a pc-presentation for a finite group in various categories such as `GrpPerm` and `GrpMat`. Converting from a `GrpFP` group is trickier, since the original group need not be finite. There are two functions provided to produce pc-presentations for certain quotients of finitely-presented groups. The `pQuotient` function constructs a pc-presentation for the largest p-group quotient having specified lower exponent-p class. Similarly, `SolubleQuotient` will compute the largest soluble quotient subject to certain restrictions. Each of these functions also provides a homomorphism (isomorphism in the case of `PCGroup`) from the original group to the new pc-group. More information on each of the two quotient functions can be found in Chapter 71.

`PCGroup(G)`

A `GrpPC` representation of the group G and the isomorphism.

`pQuotient( F, p, c : parameters)`

`Workspace`	RngIntElt	*Default :* 1000000
`Metabelian`	BoolElt	*Default :* `false`
`Exponent`	RngIntElt	*Default :* 0
`Print`	RngIntElt	*Default :* 0

Given a finitely presented group F, a prime p, and a positive integer c, this function constructs a consistent power-conjugate presentation for the largest p-quotient H of F having lower exponent-p class at most c). If c is given as zero, then the limit 127 is placed on the class. The function returns both the p-quotient H defined by a pc-presentation and the homomorphism from F to H.

`SolubleQuotient(G)`

`SolvableQuotient(G)`

A `GrpPC` representation P of the largest solvable quotient of G and the homomorphism $\phi : G \to P$.

Example H63E33

We use `PCGroup` to produce a pc-presentation for a matrix group.

```
> GL := GeneralLinearGroup(4,GF(3));
> S3 := Sylow(GL,3);
> P := PCGroup(S3);
> P;
GrpPC : P of order 729 = 3^6
PC-Relations:
    P.2^P.1 = P.2 * P.4^2,
    P.3^P.1 = P.3 * P.5^2,
    P.3^P.2 = P.3 * P.6^2,
    P.5^P.2 = P.4 * P.5,
    P.6^P.1 = P.4 * P.6
```

63.16.2 Transfer from GrpPC

Given a pc-group, it is straight-forward to convert it to a GrpFP or GrpGPC representation by using the appropriate transfer function. If one wishes to have a permutation representation of the group, this requires more cleverness. The `CosetAction` function can be used to compute the permutation representation of a group on a subgroup. If the subgroup is chosen to have trivial core, then the permutation group obtained will be isomorphic to the original group.

`AbelianGroup(G)`

Given an abelian pc-group G, return a `GrpAb` group H isomorphic to G and an isomorphism $\phi : G \to H$.

FPGroup(G)

A GrpFP representation F of G and the isomorphism from G to F.

GPCGroup(G)

A GrpGPC representation F of G and the isomorphism from G to F.

Example H63E34

Take one of the groups of order $2^6 * 3^2$.

```
> G := SmallGroup(576, 4123);
> G;
GrpPC : G of order 576 = 2^6 * 3^2
PC-Relations:
    G.1^2 = Id(G),
    G.2^2 = Id(G),
    G.3^2 = G.5,
    G.4^3 = Id(G),
    G.5^2 = G.7,
    G.6^2 = G.7,
    G.7^2 = Id(G),
    G.8^3 = Id(G),
    G.2^G.1 = G.2 * G.6,
    G.6^G.1 = G.6 * G.7,
    G.6^G.2 = G.6 * G.7,
    G.8^G.1 = G.8^2
```

Since G is small, we can search for a minimum degree permutation presentation by brute force. First we build a set containing all the subgroups.

```
> SL := Subgroups(G);
> T := {X`subgroup: X in SL};
> #T;
243
```

Then, we select those subgroups with trivial core, and find one with the smallest index.

```
> TrivCore := {H:H in T| #Core(G,H) eq 1};
> mdeg := Min({Index(G,H):H in TrivCore});
> Good := {H: H in TrivCore| Index(G,H) eq mdeg};
> #Good;
3
> H := Rep(Good);
```

We then use CosetAction to construct the permutation representation on the cosets of H.

```
> f,P,K := CosetAction(G,H);
> #K;
1
> IsPrimitive(P);
false
```

63.17 More About Presentations

Each pc-group can have up to three pc-presentations associated with it. If the user specifies a consistent presentation in the `PolycyclicGroup`-constructor, then this presentation (the "user" presentation) will be used for all printing and interpretation of element input. If the specified presentation is inconsistent, a runtime error is generated.

Internally, MAGMA uses a "conditioned" presentation for computation. The composition series associated with this presentation is guaranteed to refine a normal series with elementary abelian factors. If G is a p-group, then the composition series is guaranteed to be a central series and the first d pc-generators are a minimal set of generators for the group. Hence, their images generate the Frattini factor group. If the user presentation satisfies these conditions, then it is used as the conditioned presentation. Otherwise, a separate presentation is computed automatically.

Several algorithms rely on a "special" presentation for the group. This presentation exhibits Hall π-subgroups and a characteristic series with elementary abelian factors. When needed, such a presentation is computed and elements are automatically translated between presentations.

The "compact" presentation is not a presentation used in computation. Rather it provides an efficient means to input and output large pc-groups. This is especially useful for stored collections of groups (libraries or databases).

63.17.1 Conditioned Presentations

MAGMA will compute a pc-presentation which will be used for internal computation, but the user's presentation will be used for all input and output. The recommended way to access the conditioned internal presentation is via the intrinsic `ConditionedGroup`.

63.17.1.1 Structure Operations

`ConditionedGroup(G)`

The internally used, conditioned presentation of the pc-group G. The returned group is recorded as a subgroup of G in the relationship tables, so coercion can be used to move between presentations.

`IsConditioned(G)`

Reutrns `true` if G uses the user presentation as the internal presentation, `false` otherwise.

63.17.1.2 Element Operations

`LeadingTerm(x)`

Given an element x of a pc-group G with n pc-generators and a conditioned presentation, where x is of the form $a_1^{\alpha_1} \dots a_n^{\alpha_n}$, return $a_i^{\alpha_i}$ for the smallest i such that $\alpha_i > 0$. If x is the identity of G, then the identity is returned.

`LeadingGenerator(x)`

Given an element x of a pc-group G with n pc-generators and a conditioned presentation, where x is of the form $a_1^{\alpha_1} \dots a_n^{\alpha_n}$, return a_i for the smallest i such that $\alpha_i > 0$. If x is the identity of G, then the identity is returned.

`LeadingExponent(x)`

Given an element x of a pc-group G with n pc-generators and a conditioned presentation, where x is of the form $a_1^{\alpha_1} \dots a_n^{\alpha_n}$, return α_i for the smallest i such that $\alpha_i > 0$. If x is the identity of G, then 0 is returned.

`Depth(x)`

Given an element x of a pc-group G with n pc-generators and a conditioned presentation, where x is of the form $a_1^{\alpha_1} \dots a_n^{\alpha_n}$, return the smallest i such that $\alpha_i > 0$. If x is the identity of G, then 0 is returned.

`PCClass(x)`

`WeightClass(x)`

The weight class of the element x. The `WeightClass` of an arbitrary element of a pc-group G is defined to be k if $x \in G_{\delta_{k-1}}$ and $x \notin G_{\delta_k}$. If x is the identity of G, then `WeightClass` returns $n + 1$.

63.17.2 Special Presentations

A special presentation is one which has several properties described by C. R. Leedham-Green:

(1) The composition series defined by the pc-generators refines the LG-series. The LG-series is a characteristic series which refines the nilpotent series. Within each nilpotent section, it refines the series of successive Frattini factors. Factors of successive terms in the LG-series are elementary abelian p-groups, with p increasing through each Frattini factor.

(2) The presentation exhibits a Sylow system. By this we mean that if π is a set of primes, then the pc-generators whose corresponding prime lies in π will generate a Hall π-subgroup.

(3) The presentation exhibits "head splittings". These are certain complements in factors of the group as follows: If N is a term of the nilpotent series of G, M the next term (so N/M is a maximal nilpotent factor of N), and F/M is the Frattini subgroup of N/M, then it is possible to show that N/F has a complement in G/F. We say the presentation exhibits this complement (or "splitting") if the pc-generators of G which are not in N generate a complement for N mod F.

Several algorithms rely on having a special presentation for the given group. In these cases, MAGMA will automatically compute a special presentation. However, if the user wishes to have a special presentation as the user presentation for a group, the function

`SpecialPresentation` can be used. This is typically used when implementing new algorithms which rely on the properties of a special presentation. The other functions allow one to identify specific characteristics of a special presentation. They are not defined for arbitrary presentations.

`SpecialPresentation(G)`

Returns a new group H which is defined by a special presentation. H is in fact a subgroup of G (equal to G) and so one can use the coercion operator (!) to translate elements between the two presentations. Furthermore, any subgroup of H is automatically a subgroup of G. For instance, if one computed the center Z of H (using some algorithm relying on the special presentation), Z would be a subgroup of G, and would be the center of G.

`SpecialWeights(G)`

A sequence of triples of integers is returned, with one triple corresponding to each pc-generator. The first integer in a triple gives the number of the nilpotent section containing the generator, the second gives the number of the square-free exponent abelian section of that nilpotent section containing it, and the third gives the number of the elementary abelian p-group layer that contains the generator. The prime for the generator is not included in the triple (see `PCPrimes`).

`NilpotentLength(G)`

The number of nilpotent factors in the nilpotent series.

`NilpotentBoundary(G,i)`

The subscript of the last generator in the ith nilpotent section, where i lies between 1 and `NilpotentLength(G)`.

`MinorLength(G,i)`

The number of minor sections (Frattini factors) in the ith nilpotent section of G.

`MinorBoundary(G,i,j)`

The subscript of the last generator in the jth minor section of the ith nilpotent section, where j lies between 1 and `MinorLength(G,i)`.

`LayerLength(G,i,j)`

The number of elementary abelian p-group layers in the jth minor section of the ith nilpotent section of G.

`LayerBoundary(G,i,j,k)`

The subscript of the last generator in the kth elementary abelian p-group layer of the jth minor section of the ith nilpotent section, where k lies between 1 and `LayerLength(G,i,j)`.

Example H63E35

We show how user presentations and special presentations can differ. If we define a wreath product using `PolycyclicGroup`, the given presentation becomes the user presentation, but this is not a special presentation for the group.

```
> T := PolycyclicGroup<a,b,c,d|a^3,b^3,c^3,d^3,
>                          b^a=c, c^a=d, d^a=b>;
> T;
GrpPC : T of order 81 = 3^4
PC-Relations:
    T.2^T.1 = T.3,
    T.3^T.1 = T.4,
    T.4^T.1 = T.2
> S := SpecialPresentation(T);
> S;
GrpPC : S of order 81 = 3^4
PC-Relations:
    S.2^S.1 = S.2 * S.3^2 * S.4,
    S.3^S.1 = S.3 * S.4^2
```

Here we build another wreath product and construct a special presentation.

```
> C6 := CyclicGroup(GrpPC,6);
> C2 := CyclicGroup(GrpPC,2);
> G := WreathProduct(C2,C6);
> G;
GrpPC : G of order 384 = 2^7 * 3
PC-Relations:
    G.1^2 = G.2,
    G.2^3 = Id(G),
    G.3^2 = Id(G),
    G.4^2 = Id(G),
    G.5^2 = Id(G),
    G.6^2 = Id(G),
    G.7^2 = Id(G),
    G.8^2 = Id(G),
    G.3^G.1 = G.8,
    G.3^G.2 = G.5,
    G.4^G.1 = G.6,
    G.4^G.2 = G.3,
    G.5^G.1 = G.7,
    G.5^G.2 = G.4,
    G.6^G.1 = G.3,
    G.6^G.2 = G.8,
    G.7^G.1 = G.4,
    G.7^G.2 = G.6,
    G.8^G.1 = G.5,
    G.8^G.2 = G.7
> H := SpecialPresentation(G);
```

```
> H;
GrpPC : H of order 384 = 2^7 * 3
PC-Relations:
    H.1^2 = Id(H),
    H.2^2 = Id(H),
    H.3^3 = Id(H),
    H.4^2 = Id(H),
    H.5^2 = Id(H),
    H.6^2 = Id(H),
    H.7^2 = Id(H),
    H.8^2 = Id(H),
    H.2^H.1 = H.2 * H.4,
    H.5^H.3 = H.6,
    H.6^H.3 = H.5 * H.6,
    H.7^H.1 = H.6 * H.7,
    H.7^H.3 = H.8,
    H.8^H.1 = H.5 * H.6 * H.8,
    H.8^H.3 = H.7 * H.8
```

We can coerce between the presentations.

```
> G!(H.2), H!(G.2);
G.6 * G.7 * G.8 H.3
```

Look at some specific features of the presentation.

```
> SpecialWeights(H);
[ <1, 1, 1>, <1, 1, 1>, <1, 1, 2>, <1, 2, 1>, <2, 1, 1>, <2, 1, 1>, <2, 1, 1>,
<2, 1, 1> ]
> MinorLength(H,1);
2
> MinorBoundary(H,1,1);
3
```

63.17.3 CompactPresentation

When the MAGMA parser reads in large group presentations of the form

```
S4 := PolycyclicGroup< a, b, c, d | a^2 = 1, b^3 = 1, c^2 = 1,
       d^2 = 1, b^a = b^2, c^a = c * d, c^b = c * d, d^b = c >;
```

a large amount of memory and time is used to build all of the expressions involved in the statement. This time is most noticeable when loading in large libraries of MAGMA code containing many large presentations. The following intrinsics provide a way to avoid this overhead.

`CompactPresentation(G)`

Given a pc-group G, return a sequence of integers that contains the information needed to define the group's presentation.

PCGroup(Q : *parameters*)

Check	BoolElt	*Default :* false
ExponentLimit	RngIntElt	*Default :* 20

Return a group G in category GrpPC, whose presentation is provided by the integer sequence Q. Constructing the group from the integer sequence has very low overhead in the parser. The time taken to construct the group is less when the presentation is conditioned.

The parameter Check indicates whether or not the presentation is checked for consistency. Leaving the Check parameter set to false speeds the construction of the group, but will be disastrous if the sequence Q does not represent a consistent pc-presentation.

Parameter ExponentLimit determines the amount of space that will be used by the group to speed calculations. Given ExponentLimit := e, the group will store the products $a^i * b^j$ where a and b are generators and i and j are in the range 1 to e.

Example H63E36

If the user wants to store the definition of a group in a library, the following may be done.

```
> S4 := PolycyclicGroup< a, b, c, d | a^2 = 1, b^3 = 1, c^2 = 1, d^2 = 1,
>  b^a = b^2, c^a = c * d, c^b = c * d, d^b = c >;
> Q := CompactPresentation( S4 );
> Q;
[ 4, -2, -3, -2, 2, 33, 218, 114, 55 ]
```

The library code would then be

```
> Make:=func< | PCGroup(\[4, 2, 3, 2, 2, 33, 218, 114, 55] : Check := false) >;
```

Note the use of a literal sequence here — see Chapter 10.

63.18 Optimizing Magma Code

63.18.1 PowerGroup

If the user is working with enumerated sets of pc-groups that are all subgroups of a common over-group G, then the following optimization is strongly recommended. Define the set to have the universe PowerGroup(G). For any subgroup H of G, we can find a canonical form for the generators of H. This allows us to have a very good hashing function for the subgroups.

Example H63E37

The following example illustrates the optimization.

```
> G := ExtraSpecialGroup( GrpPC, 3, 3 );
> P := PowerGroup(G);
> time s1 := {  P | sub< G | Random(G), Random(G) > : x in { 1..500}  };
Time: 1.140
> time s2 := { Parent(G) | sub< G | Random(G), Random(G) > : x in { 1..500}  };
Time: 9.769
```

63.19 Bibliography

[**CH00**] John Cossey and Trevor Hawkes. On the largest conjugacy class size in a finite group. *Rend. Sem. Mat. Univ. Padova*, 103:171–179, 2000.

[**CNW90**] F. Celler, J. Neubüser, and C.R.B. Wright. Some remarks on the computation of complements and normalizers in soluble groups. *Acta Appl. Math.*, 21:57–76, 1990.

[**Con90a**] S. B. Conlon. Calculating characters of p-groups. *J. Symbolic Comp.*, 9:535–550, 1990.

[**Con90b**] S. B. Conlon. Computing modular and projective character degrees of soluble groups. *J. Symbolic Comp.*, 9:551–570, 1990.

[**ELGO02**] Bettina Eick, C.R. Leedham-Green, and E.A. O'Brien. Constructing automorphism groups of a p-groups. *Comm. Algebra*, 30:2271–2295, 2002.

[**EO99**] Bettina Eick and E.A. O'Brien. Enumerating p-groups. *J. Austral. Math. Soc.*, 67:191–205, 1999.

[**FO00**] D.L. Flannery and E.A. O'Brien. Computing 2-cocycles for central extensions and relative difference sets. *Comm. Algebra*, 28:1935–1955, 2000.

[**GS90**] S.P. Glasby and Michael C. Slattery. Computing intersections and normalizers in soluble groups. *J. Symbolic Comp.*, 9:637–651, 1990. Computational group theory, Part 1.

[**How12**] David J. A. Howden. *Computing automorphism groups and isomorphism testing in finite groups*. PhD thesis, University of Warwick, 2012.

[**Hul99**] Alexander Hulpke. Computing subgroups invariant under a set of automorphisms. *J. Symbolic Comp.*, 27:415–427, 1999.

[**MN89**] M. Mecky and J. Neubüser. Some remarks on the computation of conjugacy classes of soluble groups. *Bull. Austral, Math. Soc.*, 40(2):281–292, 1989.

[**New77**] M.F. Newman. Determination of groups of prime-power order. In *Group Theory (Canberra, 1975)*, volume 573 of *Lecture Notes in Mathematics*, pages 73–84. Springer-Verlag, Berlin-Heidelberg-New York, 1977.

[**O'B90**] E.A. O'Brien. The p-group generation algorithm. *J. Symbolic Comput.*, 9:677–698, 1990.

[**O'B94**] E.A. O'Brien. Isomorphism testing for p-groups. *J. Symbolic Comp.*, 17:133–147, 1994.

[**Sla86**] Michael C. Slattery. Computing character degrees in p-groups. *J. Symbolic Comp.*, 2:51–58, 1986.

[**Sla01**] Michael C. Slattery. Computing double cosets in soluble groups. *J. Symbolic Comp.*, 31:179–192, 2001. Computational algebra and number theory (Milwaukee, WI, 1996).

[**Smi94**] Michael J. Smith. *Computing automorphisms of finite soluble groups.* PhD thesis, Australian National University, 1994.

[**Ung06**] W.R. Unger. Computing the character table of a finite group. *J. Symbolic Comp.*, 41(8):847–862, 2006.

64 BLACK-BOX GROUPS

Chapter 64
BLACK-BOX GROUPS

64.1 Introduction

This Chapter describes the category of black-box groups (BB-groups). The name in MAGMA for the category of BB-groups is `GrpBB`.

64.2 Construction of an SLP-Group and its Elements

64.2.1 Structure Constructors

Magma's black-box groups are built on Magma's other group types. The basic constructor takes a group and returns a corresponding black-box group. The element set of the black-box group is essentially the same as the element set of the original group, and the group operations are inherited from the original group.

`NaturalBlackBoxGroup(H)`

Construct the natural black-box group from the concrete group H.

64.2.2 Construction of an Element

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity element for the BB-group G.

64.3 Arithmetic with Elements

`u * v`

Construct the product of elements u and v of the BB-group G.

`u ^ m`

Given an integer m and u, an element of BB-group G, return the element of G corresponding to the m-th power of u.

`u ^ v`

Given u and v, elements of BB-group G, return the element of G corresponding to the conjugate of u by v, i.e. $v^{-1} * u * v$.

`(u, v)`

Commutator of the elements u and v, i.e. the element $u^{-1} * v^{-1} * u * v$. Here u and v must belong to the same BB-group G.

64.3.1 Accessing the Defining Generators

The functions described here provide access to basic information stored for a BB-group G.

`G . i`

The i-th generator for G.

`Generators(G)`

A set containing the generators for G.

`NumberOfGenerators(G)`
`Ngens(G)`

The number of generators for B.

64.4 Operations on Elements

64.4.1 Equality and Comparison

`u eq v`

Returns `true` if and only if the underlying concrete group elements for u and v are equal.

`u ne v`

Returns `true` if and only if the underlying concrete group elements for u and v are not equal.

64.4.2 Attributes of Elements

`Parent(u)`

The parent group G of the element u.

`UnderlyingElement(u)`

The concrete group element corresponding to the BB-group element u.

`Order(u)`

The order of the underlying concrete group element of u.

Example H64E1

The following function takes a black box group isomorphic to M_{24} and finds standard generators. It is taken from the ATLAS of Finite Group Representations page on M_{24}.

```
> m24_standard := function(B)
> repeat a := PseudoRandom(B); until Order(a) eq 10;
> a := a ^ 5;
> repeat b := PseudoRandom(B); until Order(b) eq 15;
> b := b ^ 5;
> repeat b := b ^ PseudoRandom(B); ab := a*b;
> until Order(ab) eq 23;
> x := ab*(ab^2*b)^2*ab*b;
> if Order(x) eq 5 then b := b^-1; end if;
> return a,b;
> end function;
```

We take a group which must be M_{24} and find these generators.

```
> G := PermutationGroup<24 |
> [ 20, 4, 10, 3, 15, 9, 7, 1, 11, 22, 21, 19, 8, 2, 24, 5,
> 12, 18, 13, 16, 14, 23, 6, 17 ],
> [ 12, 18, 3, 2, 7, 11, 5, 21, 19, 22, 23, 1, 14, 17, 10,
> 8, 4, 13, 24, 20, 9, 15, 6, 16 ]>;
> #G;
244823040
> Transitivity(G);
5
> B := NaturalBlackBoxGroup(G);
> a,b := m24_standard(B); a,b;
GrpBBElt (1, 16)(2, 22)(3, 14)(4, 15)(5, 11)(6, 24)(7,
10)(8, 18)(9, 19)(12, 17)(13, 20)(21, 23)
GrpBBElt (1, 14, 17)(2, 18, 13)(5, 16, 20)(7, 22, 9)(8, 24,
15)(19, 23, 21)
```

The printing of the GrpBBElts shows the underlying concrete group elements. These may be extracted using the `UnderlyingElement` intrinsic for use within G.

64.5 Set-Theoretic Operations

64.5.1 Membership and Equality

`g in G`

Return true if and only if G is the parent group of g or the parent group of g is a subgroup of G.

64.5.2 Set Operations

`PseudoRandom(G)`

Return a pseudo-random element of the BB-group G. The method used is product-replacement with accumulator.

`Rep(G)`

A representative element of G.

64.5.3 Coercions Between Related Groups

`G ! g`

Given an element g belonging to a subgroup of the BB-group G, rewrite g as an element of G.

65 ALMOST SIMPLE GROUPS

Chapter 65
ALMOST SIMPLE GROUPS

65.1 Introduction

65.1.1 Overview

This chapter describes a set of tools for working with finite almost-simple groups (AS-groups). In the program for computing with non-soluble finite groups, the goal is to reduce the solution of many problems concerning a non-soluble group G to that of solving the same problem for the non-abelian simple composition factors of G. We are concerned with very specific types of computation with AS-groups.

The techniques described in this chapter are under development and are very incomplete in their coverage. The material falls roughly into two main categories.

(a) Functions which try to identify a particular group S known to be almost simple with a standard copy T of that AS-group. In addition, if such an isomorphism is found, it is often desirable to explicitly construct it so that questions concerning S can be answered by mapping them into the "standard" group T. Hence the recognition functions are divided into those which perform *non-constructive recognition* (they assert the existence of an isomorphism between S and T) and those that perform *constructive recognition* (an explicit isomorphism between S and T is returned).

(b) Functions which allow the user to determine information about an AS-group. These functions are usually implemented separately for each family of simple groups. Thus, for each family of simple groups our goal is to provide machinery for constructing key properties of any group T in that family in the context of a standard representation of the group. Using the isomorphism constructed in (a), this information can then be transferred back to the user's group S. Examples of such information include, information about element conjugacy, maximal subgroups, and Sylow p-subgroups.

The functions described in this chapter do not assume that a BSGS-representation can be constructed available. Thus, the techniques described here apply to groups possibly having both much larger order and/or much larger dimension than those that can be handled with the techniques of Chapters 58 and 59.

65.2 Creating Finite Groups of Lie Type

Several functions are provided which construct various classical groups and other groups of Lie type. The effect of these functions is to define the group in terms of a set of generating matrices.

As shown by Chevalley, for each simple Lie algebra L over the complex field and for each finite field $\mathbf{F}_q$ there is an associated matrix group $L(q)$. In general, these groups are perfect but not simple. To obtain the simple group, it is necessary to form the quotient by the centre. Similarly, as Steinberg, Ree and others have shown, if the associated Coxeter graph has an automorphism, of order t say, then there will be a 'twisted' version ${}^tL(q)$ of $L(q)$.

Generators for the series A, C, ^{2}A and ^{2}B are described in [Tay87]. Generators for the series B, D and ^{2}D are as given by Rylands and Taylor [RT98]. Generators for the exceptional groups of Lie type are described by Howlett, Rylands and Taylor in [HRT01].

65.2.1 Generic Creation Function

`ChevalleyGroup(X, n, K:` *parameters*`)`

`ChevalleyGroup(X, n, q:` *parameters*`)`

`Irreducible` BoolElt *Default :* `false`

Construct a matrix group over the field K (or over $\mathbf{F}_q$) which has the adjoint Chevalley group of Lie series `X` and Lie rank n as the quotient modulo scalar matrices. In most cases the group returned is the universal Chevalley group $X_n(q)$; however, for series `B`, `D` and `2D` the universal group is the spin group and the matrix group returned by `ChevalleyGroup` is $\Omega(2n+1,q)$, $\Omega^+(2n,q)$ or $\Omega^-(2n,q)$.

For the twisted groups the meaning of the parameter q is consistent with the (abbreviated) notation in the 'Atlas of Finite Groups' and in the monograph series 'The Classification of the Finite Simple Groups' by Gorenstein, Lyons and Solomon. For a Chevalley group of rank n and type X with an automorphism of order t the Atlas defines the *twisted Chevalley group* ${}^tX_n(q,q^t)$ to be the set of elements of $X_n(q^t)$ fixed by the quotient of the twisting automorphism and the field automorphism induced by $x \mapsto x^q$ of $\mathbf{F}_{q^t}$. In the Atlas the abbreviated notation for the twisted group is ${}^tX_n(q)$ but in Carter [Car72] it is ${}^tX_n(q^t)$. The first signature of the intrinsic expects the field $\mathbf{F}_{q^t}$ but the second signature expects the parameter q.

For example, for the series `"2A"`, the group ${}^2A_n(q)$ is $\mathrm{SU}(n+1,q)$ but, in the first form of the signature, K must be the field $\mathbf{F}_{q^2}$. Similarly the first form of the signature for the groups ${}^3D_4(q)$ and ${}^2E_6(q)$ requires the fields $\mathbf{F}_{q^3}$ and $\mathbf{F}_{q^2}$, respectively.

The possible series and the groups returned are:

`"A"` : $n \geq 0$, $A_n(q)$, the special linear group $\mathrm{SL}(n+1,q)$.

`"B"` : $n \geq 1$, $B_n(q)$, the orthogonal group $\Omega(2n+1,q)$.

`"C"` : $n \geq 1$, $C_n(q)$, the symplectic group $\mathrm{Sp}(2n,q)$.

`"D"` : $n \geq 1$, $D_n(q)$, the orthogonal group $\Omega^+(2n, q)$.

`"E"` : $n \in \{6, 7, 8\}$, the exceptional groups $E_n(q)$. $E_6(q)$ is represented as a matrix group of degree 27. It is simple unless $q \equiv 1 \bmod 3$, in which case its centre has order 3. $E_7(q)$ is represented as a matrix group of degree 56. It is simple unless $q \equiv 1 \bmod 2$, in which case its centre has order 2. $E_8(q)$ is represented as a matrix group of degree 248.

`"F"` : $n = 4$, the exceptional group $F_4(q)$ represented as a matrix group of degree 26. If $q = 3^k$ then this representation is reducible. An irreducible representation is not yet available.

`"G"` : $n = 2$, the exceptional group $G_2(q)$ represented as a matrix group of degree 7. If $q = 2^k$ then this representation is reducible. An irreducible representation of degree 6 can be obtained by setting the parameter `Irreducible := true`.

`"2A"` : $n \geq 1$, $K = \mathbf{F}_{q^2}$, the special unitary group ${}^2A_n(q) = \mathrm{SU}(n+1, q)$.

`"2B"` : $n = 2$, $K = \mathbf{F}_q$, $q = 2^{2k+1}$, the Suzuki group ${}^2B_2(q) = \mathrm{Sz}(q)$.

`"2D"` : $n \geq 1$, $K = \mathbf{F}_q$, ${}^2D_n(q)$, the orthogonal group $\Omega^-(2n, q)$.

`"3D"` : $n = 4$, $K = \mathbf{F}_{q^3}$, the exceptional group ${}^3D_4(q)$.

`"2E"` : $n = 6$, $K = \mathbf{F}_{q^2}$, the exceptional group ${}^2E_6(q)$.

`"2F"` : $n = 4$, $K = \mathbf{F}_q$, $q = 2^{2k+1}$, the Ree group ${}^2F_4(q)$, simple except when $q = 2$ when the derived group is simple and is returned by the function `TitsGroup`.

`"2G"` : $n = 2$, $K = \mathbf{F}_q$, $q = 3^{2k+1}$, the Ree group ${}^2G_2(q)$, simple except when $q = 3$.

65.2.2 The Orders of the Chevalley Groups

`ChevalleyOrderPolynomial(type, n: parameters)`

The orders of the universal Chevalley groups $X_n(q)$ and ${}^tX_n(q)$ are polynomials in q. For the twisted groups of types ${}^2A_n$, ${}^3D_4$ and ${}^2E_6$ the parameter q is the order of the fixed field of the Frobenius automorphism.

Other versions of Chevalley groups are quotients of universal Chevalley groups modulo a subgroup of the centre.

`FactoredChevalleyGroupOrder(type, n, F: parameters)`

`FactoredChevalleyGroupOrder(type, n, q: parameters)`

`Proof`	BoolElt	*Default :* `true`
`Version`	MonStgElt	*Default :* "*Default*"

`ChevalleyGroupOrder(type, n, F: parameters)`

`ChevalleyGroupOrder(type, n, q: parameters)`

`Version` MonStgElt *Default : "Default"*

The (factored) order of the Chevalley group of a given type and rank over the field F (or $\mathbf{F}_q$). The default is the order of the group returned by `ChevalleyGroup`, which except for types B_n, D_n and ${}^2D_n$ is the universal group. The orders of the universal and adjoint Chevalley group can be obtained by setting the parameter `Version` to `Universal` or `Adjoint`. In the factored version the value of `Proof` is passed to the Magma's factorisation function (q.v.).

65.2.3 Classical Groups

Magma offers several functions to construct the classical groups. For most of these functions, it is possible to specify the particular group by giving one of the following combinations of arguments:

(i) The degree n and the coefficient field K of the desired matrix group;

(ii) The degree n of the desired matrix group and a prime power q which relates the group to the appropriate Lie algebra. With the exception of the unitary groups (which will be defined over $\mathbf{F}_{q^2}$), the resulting group will be defined over $\mathbf{F}_q$; or,

(iii) A full vector space $V = K^n$ on which the desired matrix group should act naturally.

65.2.3.1 Linear Groups

`GeneralLinearGroup(n, q)`

`GeneralLinearGroup(n, K)`

`GeneralLinearGroup(V)`

`GL(n, q)`

`GL(n, K)`

`GL(V)`

Here n is a positive integer, q is the power of a prime, K is a finite field $\mathbf{F}_q$, and V is an n-dimensional vector space over K. This function constructs the general linear group $\mathrm{GL}(n, q)$ (resp. $\mathrm{GL}(n, K)$, $\mathrm{GL}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GL`.

`SpecialLinearGroup(n, q)`

`SpecialLinearGroup(n, K)`

`SpecialLinearGroup(V)`

`SL(n, q)`

`SL(n, K)`

`SL(V)`

Here n is a positive integer, q is the power of a prime, K is a finite field $\mathbf{F}_q$, and V is an n-dimensional vector space over K. This function constructs the special linear group $\mathrm{GL}(n, q)$ (resp. $\mathrm{GL}(n, K)$, $\mathrm{GL}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SL`.

```
AffineGeneralLinearGroup(GrpMat, n, q)
AffineGeneralLinearGroup(GrpMat, n, K)
AffineGeneralLinearGroup(GrpMat, V)
AGL(GrpMat, V)
```

Here n is a positive integer greater than or equal to 2, q is the power of a prime, K is a finite field $\mathbf{F}_q$, and V is an n-dimensional vector space over K. This function constructs the affine general linear group $\mathrm{AGL}(n, q)$ (resp. $\mathrm{AGL}(n, K)$, $\mathrm{AGL}(V)$) as a subgroup of $\mathrm{GL}(n+1, K)$. If the category name `GrpMat` is omitted the affine group will be returned as a permutation group. The intrinsic name may be abbreviated to `AGL`.

```
AffineSpecialLinearGroup(GrpMat, n, q)
AffineSpecialLinearGroup(GrpMat, n, K)
AffineSpecialLinearGroup(GrpMat, V)
ASL(GrpMat, V)
```

Here n is a positive integer greater than or equal to 2, q is the power of a prime, K is a finite field $\mathbf{F}_q$, and V is an n-dimensional vector space over K. This function constructs the affine special linear group $\mathrm{ASL}(n, q)$ (resp. $\mathrm{ASL}(n, K)$, $\mathrm{ASL}(V)$) as a subgroup of $\mathrm{SL}(n+1, K)$. If the category name `GrpMat` is omitted, the affine group will be returned as a permutation group. The intrinsic name may be abbreviated to `ASL`.

65.2.3.2 Unitary Groups

```
ConformalUnitaryGroup(n, q)
ConformalUnitaryGroup(n, K)
ConformalUnitaryGroup(V)
CU(n, q)
CU(n, K)
CU(V)
```

Here $n \geq 2$ is a positive integer, q is the power of a prime, K is the finite field $\mathbf{F}_{q^2}$, and V is the n-dimensional vector space over K. This function constructs the conformal unitary group $\mathrm{CU}(n, q)$ (resp. $\mathrm{CU}(n, K)$, $\mathrm{CU}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `CU`. A conformal unitary group is the group that preserves a unitary form up to a constant.

```
GeneralUnitaryGroup(n, q)
GeneralUnitaryGroup(n, K)
GeneralUnitaryGroup(V)
GU(n, q)
GU(n, K)
GU(V)
```

Here $n \geq 2$ is a positive integer, q is the power of a prime, K is the finite field $\mathbf{F}_{q^2}$, and V is the n-dimensional vector space over K. This function constructs the general unitary group $\mathrm{GU}(n, q)$ (resp. $\mathrm{GU}(n, K)$, $\mathrm{GU}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GU`.

```
SpecialUnitaryGroup(n, q)
SpecialUnitaryGroup(n, K)
SpecialUnitaryGroup(V)
SU(n, q)
SU(n, K)
SU(V)
```

Here n is an integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_{q^2}$, and V is the n-dimensional vector space over K. This function constructs the special unitary group $\mathrm{SU}(n, q)$ (resp. $\mathrm{SU}(n, K)$, $\mathrm{SU}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SU`.

65.2.3.3 Symplectic Groups

```
ConformalSymplecticGroup(n, q)
ConformalSymplecticGroup(n, K)
ConformalSymplecticGroup(V)
CSp(n, q)
CSp(n, K)
CSp(V)
```

Here n is an even integer greater than or equal to 4, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the conformal symplectic group $\mathrm{CSp}(n, q)$ (resp. $\mathrm{CSp}(n, K)$, $\mathrm{CSp}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `CSp`. A conformal symplectic group is the group that preserves a symplectic form up to a constant.

```
SymplecticGroup(n, q)
SymplecticGroup(n, K)
SymplecticGroup(V)
Sp(n, q)
Sp(n, K)
Sp(V)
```

Here n is an even integer greater than or equal to 4, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the symplectic group $\mathrm{Sp}(n, q)$ (resp. $\mathrm{Sp}(n, K)$, $\mathrm{Sp}(V)$) in terms of two generating matrices. The intrinsic name may be abbreviated to `Sp`.

65.2.3.4 Orthogonal and Spin Groups

```
ConformalOrthogonalGroup(n, q)
ConformalOrthogonalGroup(n, K)
ConformalOrthogonalGroup(V)
CO(n, q)
CO(n, K)
CO(V)
```

Here n is an odd integer greater than or equal to 3, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the conformal orthogonal group $\mathrm{CO}(n, q)$ (resp. $\mathrm{CO}(n, K)$, $\mathrm{CO}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `CO`.

```
GeneralOrthogonalGroup(n, q)
GeneralOrthogonalGroup(n, K)
GeneralOrthogonalGroup(V)
GO(n, q)
GO(n, K)
GO(V)
```

Here n is an odd integer greater than or equal to 3, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the general orthogonal group $\mathrm{GO}(n, q)$ (resp. $\mathrm{GO}(n, K)$, $\mathrm{GO}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GO`.

```
SpecialOrthogonalGroup(n, q)
SpecialOrthogonalGroup(n, K)
```

```
SpecialOrthogonalGroup(V)
SO(n, q)
SO(n, K)
SO(V)
```

Here n is an odd integer greater than or equal to 3, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the special orthogonal group $\mathrm{SO}(n, q)$ (resp. $\mathrm{SO}(n, K)$, $\mathrm{SO}(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SO`.

```
ConformalOrthogonalGroupPlus(n, q)
ConformalOrthogonalGroupPlus(n, K)
ConformalOrthogonalGroupPlus(V)
COPlus(n, q)
COPlus(n, K)
COPlus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the conformal orthogonal group $\mathrm{CO}^+(n, q)$ (resp. $\mathrm{CO}^+(n, K)$, $\mathrm{CO}^+(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `COPlus`.

```
GeneralOrthogonalGroupPlus(n, q)
GeneralOrthogonalGroupPlus(n, K)
GeneralOrthogonalGroupPlus(V)
GOPlus(n, q)
GOPlus(n, K)
GOPlus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the general orthogonal group $\mathrm{GO}^+(n, q)$ (resp. $\mathrm{GO}^+(n, K)$, $\mathrm{GO}^+(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GOPlus`.

```
SpecialOrthogonalGroupPlus(n, q)
SpecialOrthogonalGroupPlus(n, K)
SpecialOrthogonalGroupPlus(V)
SOPlus(n, q)
SOPlus(n, K)
SOPlus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the special orthogonal group $\mathrm{SO}^+(n,q)$ (resp. $\mathrm{SO}^+(n,K)$, $\mathrm{SO}^+(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SOPlus`.

```
ConformalOrthogonalGroupMinus(n, q)
ConformalOrthogonalGroupMinus(n, K)
ConformalOrthogonalGroupMinus(V)
COMinus(n, q)
COMinus(n, K)
COMinus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the conformal orthogonal group $\mathrm{CO}^-(n,q)$ (resp. $\mathrm{CO}^-(n,K)$, $\mathrm{CO}^-(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `COMinus`.

```
GeneralOrthogonalGroupMinus(n, q)
GeneralOrthogonalGroupMinus(n, K)
GeneralOrthogonalGroupMinus(V)
GOMinus(n, q)
GOMinus(n, K)
GOMinus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the general orthogonal group $\mathrm{GO}^-(n,q)$ (resp. $\mathrm{GO}^-(n,K)$, $\mathrm{GO}^-(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `GOMinus`.

```
SpecialOrthogonalGroupMinus(n, q)
SpecialOrthogonalGroupMinus(n, K)
SpecialOrthogonalGroupMinus(V)
SOMinus(n, q)
SOMinus(n, K)
SOMinus(V)
```

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the special orthogonal group $\mathrm{SO}^-(n,q)$ (resp. $\mathrm{SO}^-(n,K)$, $\mathrm{SO}^-(V)$) in terms of generating matrices. The intrinsic name may be abbreviated to `SOMinus`.

```
Omega(n, q)
```

`Omega(n, K)`

`Omega(V)`

Here n is an odd integer greater than or equal to 3, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the orthogonal group $\Omega(n, K)$ (resp. $\Omega(n, K)$, $\Omega(V)$) in terms of two generating matrices. The group $\Omega(n, K)$ is the kernel of the spinor norm map on $\mathrm{SO}(n, K)$).

`OmegaPlus(n, q)`

`OmegaPlus(n, K)`

`OmegaPlus(V)`

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the orthogonal group $\Omega^+(n, q)$ (resp. $\Omega^+(n, K)$, $\Omega^+(V)$) in terms of two generating matrices. The group $\Omega^+(n, K)$ is the kernel of the spinor norm map on $\mathrm{SO}^+(n, K)$.

`OmegaMinus(n, q)`

`OmegaMinus(n, K)`

`OmegaMinus(V)`

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the orthogonal group $\Omega^-(n, q)$ (resp. $\Omega^-(n, K)$, $\Omega^-(V)$) in terms of two generating matrices. The group $\Omega^-(n, K)$ is the kernel of the spinor norm map on $\mathrm{SO}^-(n, K)$.

`Spin(n, q)`

`Spin(n, K)`

`Spin(V)`

Here n is an odd integer greater than or equal to 1, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the spin group $\mathrm{Spin}(n, K)$ (resp. $\mathrm{Spin}(n, K)$, $\mathrm{Spin}(V)$).

`SpinPlus(n, q)`

`SpinPlus(n, K)`

`SpinPlus(V)`

Here n is an even integer greater than or equal to 2, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the spin group $\mathrm{Spin}^+(n, K)$ (resp. $\mathrm{Spin}^+(n, K)$, $\mathrm{Spin}^+(V)$).

```
SpinMinus(n, q)
SpinMinus(n, K)
SpinMinus(V)
```

Here n is an even integer greater than or equal to 4, q is the power of a prime, K is the finite field $\mathbf{F}_q$, and V is the n-dimensional vector space over K. This function constructs the spin group $\mathrm{Spin}^-(n, K)$ (resp. $\mathrm{Spin}^-(n, K)$, $\mathrm{Spin}^-(V)$).

65.2.4 Exceptional Groups

65.2.4.1 Suzuki Groups

The Suzuki groups are specified slightly differently, as the degree of the group is always four. Thus for this family of groups, the possible combinations of arguments are:

(i) A finite field $K = \mathbf{F}_{2^{2m+1}}$, over which the resulting matrix group is defined;

(ii) An integer $q = 2^{2m+1}$, corresponding to the field $K = \mathbf{F}_q$ over which the resulting matrix group is defined; or,

(iii) A vector space $V = K^4$ where $K = \mathbf{F}_{2^{2m+1}}$ on which the resulting matrix group acts naturally. which the resulting

```
SuzukiGroup(q)
SuzukiGroup(K)
SuzukiGroup(V)
```

Here q is a prime power of the form 2^{2n+1}, K is the finite field $\mathbf{F}_q$, and V is the 4-dimensional vector space over K. This function constructs the Suzuki simple group $\mathrm{Sz}(q)$ (resp. $\mathrm{Sz}(K)$, $\mathrm{Sz}(V)$) in terms of two generating matrices. The intrinsic name may be abbreviated to `Sz`.

Example H65E1

We create the 10-dimensional symplectic group over $\mathbf{F}_8$:

```
> F<u> := FiniteField(8);
> G := SymplecticGroup(10, F);
> G;
MatrixGroup(10, GF(2, 3))
Generators:
[  u   0   0   0   0   0   0   0   0   0]
[  0   1   0   0   0   0   0   0   0   0]
[  0   0   1   0   0   0   0   0   0   0]
[  0   0   0   1   0   0   0   0   0   0]
[  0   0   0   0   u   0   0   0   0   0]
[  0   0   0   0   0   u   0   0   0   0]
[  0   0   0   0   0   0   1   0   0   0]
[  0   0   0   0   0   0   0   1   0   0]
```

```
[  0   0   0   0   0   0   0   0   1   0]
[  0   0   0   0   0   0   0   0   0 u^6]

[0 0 0 1 1 1 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 1 0 1 0 0 0]
[0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 1 0 0 0 0 0]
```

Example H65E2

We create the Suzuki group over $\mathbf{F}_{128}$:

```
> F<w> := FiniteField(128);
> V := VectorSpace(F, 4);
> S := SuzukiGroup(V);
> S;
MatrixGroup(4, GF(2, 7))
Generators:
[0 0 0 1]
[0 0 1 0]
[0 1 0 0]
[1 0 0 0]

[  w^8     0     0     0]
[    0 w^120     0     0]
[    0     0   w^7     0]
[    0     0     0 w^119]

[   1    0    0    0]
[ w^8    1    0    0]
[   0    w    1    0]
[w^17  w^9  w^8    1]
> Order(S);
34093383680
> FactoredOrder(S);
[ <2, 14>, <5, 1>, <29, 1>, <113, 1>, <127, 1> ]
```

65.2.4.2 Small Ree Groups

The Ree groups (${}^2G_2(q)$) are given in an irreducible matrix representation of degree seven. The possible combinations of arguments are:

(i) A finite field $K = \mathbf{F}_{3^{2m+1}}$ with $m > 0$, over which the matrix group is defined.

(ii) An integer $q = 3^{2m+1}$ with $m > 0$, corresponding to the field $K = \mathbf{F}_q$ over which the group is defined; or,

(iii) A vector space $V = K^7$ where $K = \mathbf{F}_{3^{2m+1}}$ with $m > 0$, on which the matrix group acts naturally.

```
ReeGroup(q)
ReeGroup(K)
ReeGroup(V)
```

Here q is a prime power of the form $q = 3^{2m+1}$ with $m > 0$, K is the finite field $\mathbf{F}_q$, and V is the 7-dimensional vector space over K. This function constructs the Ree group ${}^2G_2(q)$ (resp. ${}^2G_2(K)$, ${}^2G_2(V)$) in terms of standard generating matrices. The intrinsic name may be abbreviated to `Ree`.

65.2.4.3 Large Ree Groups

The Ree groups (${}^2F_4(q)$) are given in an irreducible matrix representation of degree twenty-six. The possible combinations of arguments are:

(i) A finite field $K = \mathbf{F}_{2^{2m+1}}$ with $m > 0$, over which the matrix group is defined.

(ii) An integer $q = 2^{2m+1}$ with $m > 0$, corresponding to the field $K = \mathbf{F}_q$ over which the group is defined; or,

(iii) A vector space $V = K^{26}$ where $K = \mathbf{F}_{2^{2m+1}}$ with $m > 0$, on which the matrix group acts naturally.

```
LargeReeGroup(q)
LargeReeGroup(K)
LargeReeGroup(V)
```

Here q is a prime power of the form $q = 2^{2m+1}$ with $m > 0$, K is the finite field $\mathbf{F}_q$, and V is the 26-dimensional vector space over K. This function constructs the Ree group ${}^2F_4(q)$ (resp. ${}^2F_4(K)$, ${}^2F_4(V)$) in terms of standard generating matrices. The intrinsic name may be abbreviated to `LargeRee`.

65.3 Group Recognition

65.3.1 Constructive Recognition of Alternating Groups

`RecogniseAlternatingOrSymmetric(G, n)`

Constructive recognition of the group G, which will succeed with probability $\geq 1-e^5$ if G is isomorphic to either the alternating or symmetric group of degree $n > 11$. The method is that of Beals *et al* [BLGN$^+$03], implemented by Colva Roney-Dougal.

The return values start with a flag indicating success or failure. If the algorithm was successful, then there are three more return values: a flag which is true when G is symmetric and false when alternating, and two programs. The first program takes an element x of an overgroup of G and produces a boolean to indicate whether $x \in G$ and a permutation representing x in the natural action of S_n (if such a permutation exists). The second taking a permutation to the corresponding element of G. The programs define mutually inverse group isomorphisms, implemented as MAGMA functions.

Example H65E3

We give an example of `RecogniseAlternatingOrSymmetric` in use.

```
> a:= AlternatingGroup(13);
> h:= Stabiliser(a, {1,2});
> k:= CosetImage(a, h);
> Degree(k);
78
> worked, is_sym, bb_to_perm, perm_to_bb:=
> RecogniseAlternatingOrSymmetric(k, 13);
> worked;
true
> is_sym;
false
> x:= Sym(78)!(1, 35, 16, 28, 14, 26, 69, 5, 74)(2, 54,
> 67, 18, 51, 63, 6, 50, 77)(3, 33, 78, 12, 34, 29, 19, 15, 73)
> (4, 52, 61, 24, 49, 60, 68, 38, 64)(7, 20, 71, 17,
> 32, 11, 72, 8, 36)(9, 76, 47, 31, 56, 62, 13, 53, 59)
> (10, 70, 57, 23, 37, 22, 21, 27, 25)(30, 45, 46, 43, 42,
> 44, 40, 41, 75)(39, 55, 65)(48, 66, 58);
> x in k;
true;
> in_k, perm_image:= bb_to_perm(x);
> in_k;
true
> perm_image;
(1, 2, 3)(4, 7, 12, 6, 10, 11, 13, 9, 8)
> perm_to_bb(perm_image) eq x;
true
```

`RecogniseSymmetric(G, n: parameters)`

`maxtries`	RngIntElt	*Default :* $100n + 5000$
`Extension`	BoolElt	*Default :* `false`

The group G should be known to be isomorphic to the symmetric group S_n for some $n \geq 8$. The Bratus-Pak algorithm [BP00] (implemented by Derek Holt) is used to define an isomorphism between G and S_n. If successful, return `true`, homomorphism from G to S_n, homomorphism from S_n to G, the map from G to its word group and the map from the word group to G.

If the optional parameter `Extension` is set, then the group G should be known to be isomorphic either to S_n or to a perfect central extension $2.S_n$. In that case, the first two maps returned will be a homomorphism from G to S_n and a map from S_n to G that induces a homomorphism onto $G/Z(G)$. The sixth value returned will be `true`, if $G \cong 2.S_n$ and `false`, if $G \cong 2.A_n$.

If unsuccessful, `false` is returned. This will always occur if the input group is not isomorphic to S_n (or $2.S_n$ when `Extension` is set) with $n \geq 8$, and may occur occasionally even when G is isomorphic to S_n. The optional parameter `maxtries` (default $100n + 5000$) can be used to control the number of random elements chosen before giving up.

`SymmetricElementToWord (G, g)`

If g is an element of G which has been constructively recognised to be isomorphic to S_n (or $2.S_n$), then return `true` and element of word group for G which evaluates to g. Otherwise return `false`. This facilitates membership testing in G.

`RecogniseAlternating(G, n: parameters)`

`maxtries`	RngIntElt	*Default :* $100n + 5000$
`Extension`	BoolElt	*Default :* `false`

The group G should be known to be isomorphic to the alternating group A_n for some $n \geq 9$. The Bratus-Pak algorithm [BP00] (implemented by Derek Holt) is used to define an isomorphism between G and A_n. If successful, return `true`, homomorphism from G to A_n, homomorphism from A_n to G, the map from G to its word group and the map from the word group to G.

If the optional parameter `Extension` is set, then the group G should be known to be isomorphic either to A_n or to a perfect central extension $2.A_n$. In that case, the first two maps returned will be a homomorphism from G to A_n and a map from A_n to G that induces a homomorphism onto $G/Z(G)$. The sixth value returned will be `true`, if $G \cong 2.A_n$ and `false`, if $G \cong 2.A_n$.

If unsuccessful, `false` is returned. This will always occur if the input group is not isomorphic to A_n (or $2.A_n$ when `Extension` is set) with $n \geq 9$, and may occur occasionally even when G is isomorphic to A_n. The optional parameter tt maxtries (default $100n + 5000$) can be used to control the number of random elements chosen before giving up.

`AlternatingElementToWord (G, g)`

If g is an element of G which has been constructively recognised to be isomorphic to A_n (or $2.A_n$), then return `true` and element of word group for G which evaluates to g. Otherwise return `false`. This facilitates membership testing in G.

`GuessAltsymDegree(G: parameters)`

`maxtries`	RngIntElt	*Default :* 5000
`Extension`	BoolElt	*Default :* `false`

The group G should be believed to be isomorphic to S_n or A_n for some $n > 6$, or to $2.S_n$ or $2.A_n$ if the optional parameter `Extension` is set. This function attempts to determine n and whether G is symmetric or alternating. It does this by sampling orders of elements. It returns either `false`, if it is unable to make a decision after sampling `maxtries` elements (default 5000), or `true`, *type* and n, where *type* is "Symmetric" or "Alternating", and n is the degree. If G is not isomorphic to S_n or A_n (or $2.S_n$ or $2.A_n$ when `Extension` is set) for $n > 6$, then the output is meaningless - there is no guarantee that `false` will be returned. There is also a small probability of a wrong result or `false` being returned even when G is S_n or A_n with $n > 6$. This function was written by Derek Holt.

Example H65E4

For a group G which is believed to be isomorphic to S_n or A_n for some unknown value of $n > 6$, the function `GuessAltsymDegree` can be used to try to guess n, and then `RecogniseSymmetric` or `RecogniseAlternating` can be used to confirm the guess.

```
> G:= sub< GL(10,5) |
> PermutationMatrix(GF(5),Sym(10)![2,3,4,5,6,7,8,9,1,10]),
> PermutationMatrix(GF(5),Sym(10)![1,3,4,5,6,7,8,9,10,2]) >;
> GuessAltsymDegree(G);
true Alternating 10
> flag, m1, m2, m3, m4  := RecogniseAlternating(G,10);
> flag;
true
> x:=Random(G); Order(x);
8
> m1(x);
(1, 2, 4, 9, 10, 8, 6, 3)(5, 7)
> m2(m1(x)) eq x;
true
> m4(m3(x)) eq x;
true
> flag, w := AlternatingElementToWord(G,x);
> flag;
true
> m4(w) eq x;
true
```

```
> y := Random(Generic(G));
> flag, w := AlternatingElementToWord(G,y);
> flag;
false
> flag, m1, m2, m3, m4 := RecogniseAlternating(G,11);
> flag;
false
> flag, m1, m2, m3, m4 := RecogniseSymmetric(G,10);
> flag;
false
```

The nature of the `GuessAltsymDegree` function is that it assumes that its input is either an alternating or symmetric group and then tries to guess which one and the degree. As such, it is almost always correct when the input is an alternating or symmetric group, but will often return a bad guess when the input group is not of this form, as in the following example.

```
> GuessAltsymDegree(Sym(50));
true Symmetric 50
> GuessAltsymDegree(Alt(73));
true Alternating 73
> GuessAltsymDegree(PSL(5,5));
true Alternating 82
```

65.3.2 Determining the Type of a Finite Group of Lie Type

Given a finite quasisimple group of Lie type in any representation, the functions in this section apply probabilistic algorithms to determine its defining characteristic and type as a Lie group.

`LieCharacteristic(G : parameters)`

`NumberRandom`	RngIntElt	*Default :* 100
`Verify`	BoolElt	*Default :* `true`

Given a finite quasisimple permutation or matrix group G which is of Lie type, determine its defining characteristic. The Monte Carlo algorithm implemented by this function is that of Liebeck and O'Brien [LO07]. Since it is Monte Carlo, there is a small probability of error. The number of random elements considered is `NumberRandom`. If `Verify` is `true`, then we first verify that G is perfect by applying `IsProbablyPerfect`.

Example H65E5

```
> F := GF (4);
> w := PrimitiveElement (F);
> a := [
> 0,w^3,0,0,0,
> w^3,0,0,0,0,
```

```
> 0,0,0,w^3,0,
> 0,0,w^3,0,0,
> w^2,w^2,w^3,w^3,w^3];
> b := [
> 0,0,w^3,0,0,
> w^1,w^2,w^2,0,0,
> w^2,w^1,w^2,0,0,
> 0,0,0,0,w^3,
> w^2,w^2,w^2,w^3,w^3];
> G := sub <GL(5, F) | a, b>;
> LieCharacteristic(G);
11
```

`LieType(G, p : parameters)`

`LieType(G, p : parameters)`

`NumberRandom` RngIntElt *Default :* 100

If the matrix or permutation group G is nearly simple, and its non-abelian composition factor is isomorphic to a group of Lie type in characteristic p, then this function returns `true` and its standard Chevalley name. Otherwise it returns `false`.

The algorithm is that of Babai, Kantor, Pálfy and Seress [BKPS02]; this implementation was developed by Malle and O'Brien. Since it is Monte Carlo, there is a small probability of error. The number of random elements considered is `NumberRandom`.

The standard name is a tuple that defines the isomorphism type of the composition factor. It is similar to that employed by `CompositionFactors`, described in the Permutation Groups chapter.

If the composition factor is a group of Lie type, then the tuple is $< s, n, q >$ and it defines the adjoint Chevalley group of Lie series s and Lie rank n over $GF(q)$. The tuple entries are valid arguments for `ChevalleyGroup`.

If the composition factor is an alternating group, and so lies in family 17, then the tuple is $< 17, n, 0 >$ and it defines the alternating group of degree n.

If the composition factor is a sporadic group and so lies in family 18, then the tuple is $< 18, n, s >$; the string s is its standard Atlas name and n is the number of the group in family 18.

`SimpleGroupName(G : parameters)`

`SimpleGroupName(G : parameters)`

`NumberRandom` RngIntElt *Default :* 100

If the matrix or permutation group G is nearly simple, this function returns `true` and a list of possible names for its non-abelian simple composition factor; otherwise it returns `false`. Since it is Monte Carlo, there is a small probability of error.

The number of random elements considered is `NumberRandom`. The list of standard names follows the convention described above.

The algorithm and implementation were developed by Malle and O'Brien; it uses `LieType` and `LieCharacteristic`.

Example H65E6

We create the classical group $\Omega(7,5)$ in its natural representation and apply `SimpleGroupName` to it.

```
> SetSeed(1);
> G := Omega(7, 5);
> flag, name := SimpleGroupName(G);
> name;
[* <B, 3, 5> *]
```

We create a certain 5-dimensional matrix group over $GF(3)$ and determine which simple group it is.

```
> F := GF(3);
> P := GL(5,F);
> gens := [
> P![2,1,2,1,2,2,0,0,0,2,0,2,0,0,0,0,1,2,0,1,1,0,2,2,1],
> P![2,1,0,2,1,1,2,0,2,2,1,1,2,1,1,0,2,0,1,1,1,1,2,2,2]];
> G := sub <P | gens>;
> flag, name := SimpleGroupName(G);
> flag;
true
> name;
[* <18, 1, M11> *]
```

¿ /* naming an alternating group */

```
> G := MatrixGroup<4, GF(2) |
>    [ 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0 ],
>    [ 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0 ] >;
> flag, name := SimpleGroupName(G);
> flag;
true
> /* this is A5 */
> name;
[* <17, 5, 0> *]
> /* naming a classical group */
> F := GF(7^2);
> P := GL (6,F);
> w := PrimitiveElement (F);
> gens := [
> P![w^12,w^36,  0,  5,  2,  0,w^44,w^36,  0,  6,  2,  0,
> w^42,w^42,w^28,w^22,w^22,  3,  4,  3,  0,w^36,w^12,  0,
> 2,  3,  0,w^20,w^12,  0,w^14,w^14,  1,w^18,w^18, w^4],
```

```
>
> P![w^38,w^26,w^25,w^21, w^9,  3,w^21,w^45,w^33, w^4,w^28,
> 2,  6,  4, w^1, w^7,w^15,  4,  1,w^36,w^35, w^5,w^41,  5,
> w^31, w^7,w^43,w^36,w^12,  1,w^34,w^42,w^11,w^39,w^47,  2]
> ];
> G := sub <P | gens>;
> flag, name := LieType(G, 5);
> flag;
true
> name;
<A, 1, 5>
> /* so this is SL(2, 5) */
```

65.3.3 Classical Forms

Let G be an absolutely irreducible subgroup of $\mathrm{GL}(d, q)$. The following functions compute symplectic, unitary and orthogonal forms of the underlying vector space V left invariant by the action of G.

A *bilinear* form is a bilinear function κ from $V \times V \to F$. It is G-invariant modulo scalars if for each $g \in G$ there is a $\mu_g \in F$ such that $\kappa(vg, wg) = \mu_g \kappa(v, w)$ for all $v, w \in V$.

Now suppose that $a \mapsto \bar{a}$ is an automorphism of F of order 2. A *sesquilinear* form is a biadditive function κ from $V \times V \to F$ such that $\kappa(au, bv) = a\bar{b}\kappa(u, v)$ for all $u, v \in V$ and $a, b \in F$. It is G-invariant modulo scalars if for each $g \in G$ there is a $\mu_g \in F$ such that $\kappa(vg, wg) = \mu_g \kappa(v, w)$ for all $v, w \in V$.

A quadratic form is a function $\chi : V \to F$ such that

(1) $\chi(av) = a^2\chi(v)$ for all $a \in F$, $v \in V$; and

(2) the form κ, defined by $\kappa(u, v) = \chi(u + v) - \chi(u) - \chi(v)$ for all $u, v \in V$, is bilinear.

It is G-invariant if for each $g \in G$, $\chi(vg) = \chi(v)$ for all $v \in V$. It is G-invariant modulo scalars if for each $g \in G$ there is a $\mu_g \in F$ such that $\chi(vg) = \mu_g \chi(v)$ for all $v \in V$.

A bilinear form which is G-invariant (modulo scalars) is represented by a matrix B such that $g * B * g^{tr} = \mu_g B$ for all $g \in G$ and is unique up to multiplication by an element of F. Assume F has an automorphism $a \mapsto \bar{a}$ of order 2; a sesquilinear form is a matrix B such that $g * B * \bar{g}^{tr} = \mu_g B$ for all $g \in G$ and is unique up to multiplication by an element of F (where $\bar{g}$ denotes the matrix obtained from g by replacing each entry g_{ij} by $\bar{g_{ij}}$). A quadratic form is represented by an upper triangular matrix Q such that the matrix $g * Q * g^{tr}$, normalized into an upper triangular matrix, equals $\mu_g Q$.

The functions below will exit with an error message if the input group G is reducible. They may also exit with error if G is not absolutely irreducible, or if `Scalars` is `true` and the derived subgroup $[G, G]$ of G is not absolutely irreducible. They may however sometimes succeed in finding a fixed form when G is irreducible but not absolutely irreducible.

`ClassicalForms(G: parameters)`

`Scalars` BoolElt *Default :* `false`

Given as input a matrix group G acting absolutely irreducibly on the underlying vector space V over the field F, `ClassicalForms` will try to find a classical form which is G-invariant or prove that no such form exists. If the optional argument `Scalars` is `true` then it will look for a form which is G-invariant modulo scalars. When `Scalars` is `true`, it is only guaranteed to succeed when $[G, G]$ acts absolutely irreducibly on V. If it finds a fixed form, then it will stop and will not look for alternative fixed forms of different types.

The classical forms are: *symplectic* (non-degenerate, alternating bilinear), *unitary* (non-degenerate sesquilinear) or *orthogonal* (a *symmetric bilinear form* and a *quadratic* form).

The function `ClassicalForms` returns a record `forms` which contains the components `formType`, `sign`, `bilinearForm`, `sesquilinearForm`, `quadraticForm` and `scalars`. Depending on the entry `formType` the record components are set to indicate:

`"unknown"` : it is not known whether G fixes a classical form.

`"linear"` : it is known that G does not fix a classical form modulo scalars.

`"symplectic"` : G fixes a symplectic form modulo scalars. The matrix of the form is stored in `bilinearForm` and the scalars for each generator of G are stored in `scalars`. In characteristic two this also implies that no quadratic form is fixed.

`"unitary"` : G fixes a unitary form (modulo scalars). The matrix of the form is stored in `sesquilinearForm`. The scalars for each generator of G are stored in `scalars`.

`"orthogonalcircle"` :

`"orthogonalplus"` :

`"orthogonalminus"` : G fixes an orthogonal form modulo scalars. The matrix of the bilinear form is stored in `bilinearForm` and the corresponding quadratic form in `quadraticForm`. The scalars for each generator of G are stored in `scalars`. In the orthogonal case, `sign` is set to 0, 1, or -1 when `formType` is `"orthogonalcircle"`, `"orthogonalplus"`, or `"orthogonalminus"`, respectively.

`SymplecticForm(G: parameters)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves a symplectic form (modulo scalars if the optional argument `Scalars` is `true`), this function returns `true` and the matrix of the form. If it is known that G does not preserve such a form it returns `false`.

If it cannot decide (perhaps because the group does not act absolutely irreducibly), then it exits with an error message. If `Scalars` is `true`, then the list of scalars for the generators of G is also returned.

`SymmetricBilinearForm(G: parameters)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves an orthogonal form (modulo scalars if the optional argument `Scalars` is `true`), then this function returns `true`, the matrix of the symmetric bilinear form, and the type of the form (as in `ClassicalForms`). If it is known that G does not preserve such a form, it returns `false`. If it cannot decide, then it exits with an error message. If `Scalars` is `true`, then the list of scalars for the generators of G is also returned.

`QuadraticForm(G)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves a quadratic form (modulo scalars if the optional argument `Scalars` is `true`), this function returns `true`, the matrix of the form in upper triangular form, and the type of the form (as in `ClassicalForms`). If it is known that G does not preserve such a form it returns `false`. If it cannot decide, then it exits with an error message. If `Scalars` is `true`, then the list of scalars for the generators of G is also returned.

`UnitaryForm(G)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves a unitary form (non-degenerate sesquilinear) (modulo scalars if the optional argument `Scalars` is `true`), then this function returns `true` and the matrix of the form. If it is known that G does not preserve such a form, it returns false. If it cannot decide, then it exits with an error message. If `Scalars` is `true`, then the list of scalars for the generators of G is also returned.

`FormType(G)`

`Scalars` BoolElt *Default :* `false`

If the absolutely irreducible group G preserves a classical form (modulo scalars if the optional argument `Scalars` is `true`), this function returns its type (see `ClassicalForms`). Otherwise it returns `"unknown"`.

Example H65E7

```
> G := Omega( 9, 11 );
> ClassicalForms( G );
rec<recformat<bilinearForm, quadraticForm, sesquilinearForm, bilinFlag,
sesquiFlag, scalars, formType, bc, n> | bilinearForm :=
[ 0  0  0  0  0  0  0  0  1]
[ 0  0  0  0  0  0  0  1  0]
[ 0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  1  0  0  0]
[ 0  0  0  0  6  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0]
[ 1  0  0  0  0  0  0  0  0],
quadraticForm :=
[ 0  0  0  0  0  0  0  0  1]
[ 0  0  0  0  0  0  0  1  0]
[ 0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  1  0  0  0]
[ 0  0  0  0  3  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0],
sesquilinearForm := false, bilinFlag := true,  sesquiFlag := false,
scalars := [ 1, 1 ], formType := orthogonalcircle, sign := 0>
> FormType( G );
orthogonalcircle
> SymplecticForm( G );
false
```

`TransformForm(form, type)`

Return a matrix m such that G^m lies in the classical group returned by the MAGMA function `GU`, `Sp`, or `GO(Plus/Minus)`. The argument *form* should be a classical form of type *type* fixed by an absolutely irreducible subgroup G of $GL(d, q)$. It should be the bilinear or sesquilinear form fixed by G, except when G is orthogonal in characteristic 2, in which case it should be the quadratic form. The argument *type* should be as in the `formType` component of the record returned by `ClassicalForms`; i.e. one of `"symplectic"`, `"unitary"`, `"orthogonalcircle"`, `"orthogonalplus"`, or `"orthogonalminus"`.

`TransformForm(G)`

`Scalars`	BoolElt	*Default :* `false`

This function calls `ClassicalForms` to find a form fixed by the absolutely irreducible subgroup G of $GL(d, q)$. If `Scalars` is `true`, then `ClassicalForms` is called with `Scalars` set to `true`, so that a form fixed module scalars is found. If a form $form$ of type *type* is fixed, then it returns `TransformForm(form, type)`. Otherwise it returns `false`.

`SpinorNorm(g, form)`

The spinor norm of g with respect to the given form . $form$ must be the matrix of an orthogonal form (ie, it must be symmetric and nonsingular), and g an element of the general orthogonal group `GO(Plus/Minus)` fixing that form. Note that the form is ignored in even characteristic, since the spinor norm of g is just equal to the rank modulo 2 of $g - I$ in that case.

65.3.4 Recognizing Classical Groups in their Natural Representation

Let G be an irreducible subgroup of $\mathrm{GL}(d, q)$. The following algorithm is designed to test whether G contains the corresponding classical group Ω and is contained in Δ. Here Ω and Δ are defined as follows:

Case "linear": $\Delta = \mathrm{GL}(d, q)$, $\Omega = \mathrm{SL}(d, q)$

Case "symplectic": $\Delta = \mathrm{GSp}(d, q)$, $\Omega = \mathrm{Sp}(d, q)$

Case "orthogonalplus": $\Delta = \mathrm{GO}^+(d, q)$, $\Omega = \Omega^+(d, q)$

Case "orthogonalminus": $\Delta = \mathrm{GO}^-(d, q)$, $\Omega = \Omega^-(d, q)$

Case "orthogonalcircle":$\Delta = \mathrm{GO}^\circ(d, q)$, $\Omega = \Omega^\circ(d, q)$

Case "unitary": $\Delta = \mathrm{GU}(d, q)$, $\Omega = \mathrm{SU}(d, q)$

`RecognizeClassical( G : parameters)`

`Case`	MonStgElt	*Default :* "*unknown*"
`NumberOfElements`	RngIntElt	*Default :* 25
Verbose	`Classical`	*Maximum :* 3

`RecognizeClassical` takes as input a group G, which is a subgroup of $\mathrm{GL}(d, q)$.

The parameter `Case` is one of `"linear"`, `"symplectic"`, `"orthogonalplus"`, `"orthogonalminus"`, `"orthogonalcircle"`, `"unitary"` or `"unknown"`; if `Case` is supplied, then the algorithm seeks to decide for this case only.

The parameter `NumberOfElements` is the number of random elements selected from G during the execution of the algorithm.

The output of `RecognizeClassical` is either `true`, `false` or `"Does not apply"`. If the algorithm returns `true`, then we know with certainty that G contains Ω and is contained in Δ. Note that the proof of correctness of the algorithm depends on the finite simple group classification. If it returns `false` then either G does not contain

Ω, or G is not contained in Δ, or G is not irreducible, or there is a small chance that G is contained in Δ and contains Ω. More precisely, if the irreducible group G is contained in Δ and really does contain Ω then the probability with which the algorithm returns `false` is less than ε, where ε is a real number between 0 and 1. The smaller the value of ε, the larger `NumberOfElements` must be. If the algorithm returns `"Does not apply"` then it is not applicable to the given group.

If `"Classical"` is set to verbose then, where `RecognizeClassical` returns `true`, it also prints the statement "Proved that the group contains a classical group of type *case* in n random selections", where n is the number of selections needed. If it returns `false`, it prints a statement giving some indication of why the algorithm reached this conclusion.

Theoretical details of the algorithms used may be found in Niemeyer & Praeger [NP97][NP98][NP99] and Praeger [Pra99]. Its approach is based on the SL-recognition algorithm (Neumann & Praeger, [NP92]).This implementation also uses algorithms described in Celler & Leedham-Green [CLG97a][CLG97b] and Celler et al. [CLGM$^+$95].

For small fields ($q < 2^{16}$), the cost of this implementation for a given value of `NumberOfElements` is $O(d^3 \log d)$ bit operations.

`IsLinearGroup(G)`

This function tests whether the subgroup G of $\mathrm{GL}(d, q)$ contains $\mathrm{SL}(d, q)$. If the function can establish this fact, it returns `true` and otherwise `false`. Hence, if `IsLinearGroup` returns `false`, there is a small chance that G nevertheless contains $\mathrm{SL}(d, q)$. See `RecognizeClassical` for more details.

`IsSymplecticGroup(G)`

This function tests whether the subgroup G of $\mathrm{GSp}(d, q)$ contains $\mathrm{Sp}(d, q)$. If the function can establish this fact, it returns `true` and otherwise `false`. Hence, if `IsSymplecticGroup` returns `false`, there is a small chance that G nevertheless contains $\mathrm{Sp}(d, q)$. See `RecognizeClassical` for more details.

`IsOrthogonalGroup(G)`

This function tests whether the subgroup G of $\mathrm{GO}^\epsilon(d, q)$ contains $\Omega^\epsilon(d, q)$. If the function can establish this fact, it returns `true` and otherwise `false`. Hence, if `IsOrthogonalGroup` returns `false`, there is a small chance that G nevertheless contains $\Omega^\epsilon(d, q)$. See `RecognizeClassical` for more details.

`IsUnitaryGroup(G)`

This function tests whether the subgroup G of $\mathrm{GU}(d, q)$ contains $\mathrm{SU}(d, q)$. If the function can establish this fact, it returns `true` and otherwise `false`.

`ClassicalType(G)`

If G is known to be a classical subgroup of $\mathrm{GL}(d, q)$ this function returns the appropriate classical type as a string, i.e. `"linear"`, `"symplectic"`, `"orthogonalplus"`, `"orthogonalminus"`, `"orthogonalcircle"`, or `"unitary"`. Otherwise the function returns `false`.

Example H65E8

```
> G := SU (60, 9);
> SetVerbose( "Classical", true );
> RecognizeClassical( G );
true
> IsLinearGroup( G );
false
> IsUnitaryGroup( G );
true
> IsSymplecticGroup( G );
false
> IsOrthogonalGroup( G );
false
> ClassicalType( G );
unitary
> G := Sp (462, 3);
> time RecognizeClassical( G );
true
Time: 7.630
```

65.3.5 Constructive Recognition of Linear Groups

The functions in this section recognise whether of not a given group G is a specified linear group T. If it is, then an isomorphism between G and T is returned.

`RecognizeSL2(G)`

`RecognizeSL2(G)`

`RecognizeSL2(G, q)`

`RecognizeSL2(G, q)`

If G, a matrix or permutation group, is isomorphic, possibly modulo scalars, to $(P)SL(2, q)$, then homomorphisms between G and $(P)SL(2, q)$ are constructed. The function returns a homomorphism from G to $(P)SL(2, q)$, a homomorphism from $(P)SL(2, q)$ to G, the map from G to its word group, and the map from the word group to G.

If q, the cardinality of the defining field for G, is known, it *should* be supplied. Otherwise, the function `SL2Characteristic` is used to determine q; if q is large, this calculation may be **expensive**.

SL2ElementToWord(G, g)

SL2ElementToWord(G, g)

If g is an element of the matrix or permutation group G which has been constructively recognised to have central quotient isomorphic to $PSL(2, q)$, then return `true` and element of word group for G which evaluates to g, else `false`. This facilitates membership testing in G.

SL2Characteristic(G : *parameters*)

SL2Characteristic(G : *parameters*)

NumberRandom	RngIntElt	*Default :* 100
Verify	BoolElt	*Default :* `true`

Subject to the *assumption* that the group G has central quotient $(P)SL(2, q)$, determine its characteristic and field size. The Monte Carlo algorithm implemented by this function is that of Liebeck and O'Brien [LO07]. Since it is Monte Carlo, there is a small probability of error. The number of random elements considered is `NumberRandom`. If `Verify` is `true`, then we first verify that G is perfect by applying `IsProbablyPerfect`.

The constructive recognition algorithms for $SL(2, q)$ were developed by Conder, Leedham-Green and O'Brien [CLGO06]. The algorithm used for other representations was developed by Brooksbank and O'Brien.

Example H65E9

Our first example uses $G = \mathrm{SL}(2, 3^2)$ in its natural representation. We first recognise the group and then express a random matrix of G as a word in the generators of G.

```
> G := SL(2, 3^2);
> flag, phi, tau, gamma, delta := RecogniseSL2(G, 3^2);
> g := G![1, 2, 0, 1];
> w := gamma(g);
> delta(w) eq g;
true
```

Example H65E10

We now consider a representation of a 2-dimensional linear group inside $GL(6, \mathbf{F}_{57})$.

```
> K<w> := GF(5, 7);
> G :=
> MatrixGroup<6, GF(5, 7) |
> [w^19035, w^14713, w^50617, w^14957, w^51504, w^48397, w^16317, w^3829,
>  w^35189, w^2473, w^19497, w^77192, w^46480, w^6772, w^29577, w^61815,
>  w^54313, w^16757, w^43765, w^64406, w^58788, w^30789, w^13579, w^66728,
>  w^7733, w^45434, w^42411, w^61613, w^12905, w^6889, w^50116, w^16117,
>  w^56717, w^25226, w^49940, w^36836 ],
```

```
> [w^63955, w^40568, w^45004, w^11642, w^39536, w^11836, w^52594, w^71166,
>  w^47015, w^74450, w^32373, w^37021, w^76381, w^18155, w^57943, w^31194,
>  w^62524, w^65864, w^11868, w^76867, w^26483, w^41335, w^64856, w^41125,
>  w^43990, w^40104, w^24842, w^3153, w^23777, w^60024, w^14454, w^68648,
>  w^43403, w^26710, w^39779, w^22074 ] >;
>
> flag, phi, tau, gamma, delta := RecogniseSL2(G, 5^7);
> phi;
Mapping from: GrpMat: G to SL(2, GF(5, 7)) given by a rule [no inverse]
> g := Random(G);
> h := phi (g);
> h;
[w^40430   w^970]
[ w^5607 w^11606]
> k := tau(h);
> w := gamma(k);
> m := delta(w);
```

Recall that we are working modulo scalars.

```
> IsScalar(m * g^-1);
true
> H := SL(2, 5^7);
> h := H![1,1,0,1];
> g := tau(h);
> Order(g);
5
```

We now test a random element of $GL(6, \mathbf{F}_{57})$ for membership of our group.

```
> g := Random(GL(6, 5^7));
> SL2ElementToWord(G, g);
false
```

`RecogniseSL3(G)`

`RecogniseSL3(G, q : parameters)`

`Verify` BoolElt *Default :* `true`

If $G \leq GL(d, F)$, is isomorphic, possibly modulo scalars, to $(P)SL(3, q)$, then construct homomorphisms between G and $(P)SL(3, q)$. Return homomorphism from G to $(P)SL(3, q)$, homomorphism from $(P)SL(3, q)$ to G, the map from G to its word group and the map from the word group to G.

If q, the cardinality of the defining field for G, is known, it *should* be supplied. Otherwise, it is computed using the functions `LieCharacteristic` and `LieType`.

If `Verify` is `false`, then assume G is isomorphic, possibly modulo scalars, to $(P)SL(3, q)$.

`SL3ElementToWord (G, g)`

If g is an element of G which has been constructively recognised to have central quotient isomorphic to $PSL(3, q)$, then return `true` and element of word group for G which evaluates to g, else `false`. This facilitates membership testing in G.

The constructive recognition algorithms for $SL(3, q)$ were developed by Lübeck, Magaard, and O'Brien [LMO07]. Its current implementation, which is part of the `CompositionTree` package, was developed by Bäärnhielm and O'Brien.

Example H65E11

We create $SL(3, 5^4)$ in its natural representation and recognise it. We then form its symmetric square and apply the recognition machinery to that.

```
> G := SL(3, 5^4);
> flag, phi, tau, gamma, delta := RecogniseSL3(G);
> w := PrimitiveElement (GF(5^4));
> g := GL(3, 5^4)! [1,2,1,0,w,1,0,0,w^-1];
> w := gamma (g);
> delta (w) eq g;
true
> G := ActionGroup(SymmetricSquare(GModule(G)));
> flag, phi, tau, gamma, delta := RecogniseSL3(G);
> phi;
Mapping from: GL(6, GF(5, 4)) to SL(3, GF(5, 4)) given by a rule [no inverse]
> g := Random(G);
> h := phi(g);
> h;
[$.1^40430   $.1^970]
[ $.1^5607 $.1^11606]
> k := tau(h);
> w := gamma(k);
> m := delta(w);
```

Recall that we are working modulo scalars. We conclude by testing whether a random element of $GL(6, 5^4)$ is contained in our group.

```
> IsScalar(m * g^-1);
true
> g := Random(GL(6, 5^4));
> SL3ElementToWord(G, g);
false
```

`RecogniseSL(G, d, q)`

`RecognizeSL(G, d, q)`

Use the Kantor-Seress algorithm to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{SL}(d, q)$ or $\mathrm{PSL}(d, q)$. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms (modulo scalars if $G \cong \mathrm{PSL}(d, q)$) from G to $\mathrm{SL}(d, q)$ and from $\mathrm{SL}(d, q)$ to G.
Warning: This function often returns `false` even when G is isomorphic to $\mathrm{SL}(d, q)$ or $\mathrm{PSL}(d, q)$, so it should be called repeatedly until it returns `true`!

65.3.6 Constructive Recognition of Symplectic Groups

`RecogniseSpOdd(G, d, q)`

`RecognizeSpOdd(G, d, q)`

Use the Kantor-Seress algorithm to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{Sp}(d, q)$ or $\mathrm{PSp}(d, q)$ for odd q. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms (modulo scalars if $G \cong \mathrm{PSp}(d, q)$) from G to $\mathrm{Sp}(d, q)$ and from $\mathrm{Sp}(d, q)$ to G.
Warning: This function often returns `false` even when G is isomorphic to $\mathrm{Sp}(d, q)$ or $\mathrm{PSp}(d, q)$, so it should be called repeatedly until it returns `true`!

`RecogniseSp4Even(G, q)`

`RecognizeSp4Even(G, q)`

Use an algorithm of Peter Brooksbank to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{Sp}(4, q)$ for even q. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms from G to $\mathrm{Sp}(d, q)$ and from $\mathrm{Sp}(d, q)$ to G. The third and fourth return values are mutually inverse homomorphisms from G to the word group W of G and from W to G.

65.3.7 Constructive Recognition of Unitary Groups

`RecogniseSU3(G, d, q)`

`RecognizeSU3(G, d, q)`

Use an algorithm of Peter Brooksbank to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{SU}(3, q)$ or $\mathrm{PSU}(3, q)$ for $q > 2$. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms (modulo scalars if $G \cong \mathrm{PSU}(3, q)$) from G to $\mathrm{SU}(3, q)$ and from $\mathrm{SU}(3, q)$ to G. The third and fourth return values are mutually inverse homomorphisms from G to the word group W of G and from W to G.

`RecogniseSU4(G, d, q)`

`RecognizeSU4(G, d, q)`

Use an algorithm of Peter Brooksbank to try to find an isomorphism between the finite group G (regarded as a black-box group) and $\mathrm{SU}(4,q)$ or $\mathrm{PSU}(4,q)$. The first return value indicates whether the attempt was successful. If so, then the second and third return values are mutually inverse homomorphisms (modulo scalars if $G \cong \mathrm{PSU}(4,q)$) from G to $\mathrm{SU}(4,q)$ and from $\mathrm{SU}(4,q)$ to G. The third and fourth return values are mutually inverse homomorphisms from G to the word group W of G and from W to G.

65.3.8 Constructive Recognition of $\mathrm{SL}(d,q)$ in Low Degree

Let $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ with $q = p^f$, where V is the natural H-module. Let H act on an irreducible $\mathbf{F}_q$-module W of dimension at most d^2. Magaard, O'Brien & Seress [MOAS08] describe algorithms which, given as input the irreducible representation of H on W, construct a d-dimensional projective representation of H. Their implementations, prepared by Eamonn O'Brien, are described below.

`RecogniseSymmetricSquare (G)`

G is symmetric square representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ and $d \geq 4$. Reconstruct H; if successful, then return `true` and H, otherwise `false`.

`SymmetricSquarePreimage (G, g)`

G is symmetric square representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$; return preimage of g in H.

`RecogniseAlternatingSquare (G)`

G is alternating square representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ and $d \geq 3$. Reconstruct H; if successful, then return `true` and H, otherwise `false`.

`AlternatingSquarePreimage (G, g)`

G is alternating square representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$; return preimage of g in H.

`RecogniseAdjoint (G)`

G is adjoint representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ and $d \geq 3$. Reconstruct H; if successful, then return `true` and H, otherwise `false`.

`AdjointPreimage (G, g)`

G is adjoint representation of H, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$; return preimage of g in H.

`RecogniseDelta (G)`

G is absolutely irreducible representation of $H \otimes H^{p^e}$, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$ and $d \geq 4$. Reconstruct H; if successful, then return `true` and H, otherwise `false`.

`DeltaPreimage (G, g)`

G is absolutely irreducible representation of $H \otimes H^{(}p^e)$, where $\mathrm{SL}(d,q) \leq H \leq \mathrm{GL}(d,q)$; return preimage of g in H.

Example H65E12

```
> G := SL(4, 3^2);
> G := SL(4, 9);
> M := GModule (G);
> M := SymmetricPower (M, 2);
> G := MatrixGroup (M);
> G := RandomConjugate (G);
> f, H := RecogniseSymmetricSquare (G);
> f;
true
> H;
MatrixGroup(4, GF(3^2))
Generators:
    [    0     1     0     0]
    [    0     0     0     1]
    [$.1^6     2     2   $.1]
    [    2   $.1     0   $.1]
    [    0     0     1     0]
    [$.1^2 $.1^7     1 $.1^6]
    [    1     2 $.1^6 $.1^6]
    [  $.1     0 $.1^7     0]
> g := Random (G);
>  h := SymmetricSquarePreimage (G, g);
> h;
[$.1^6     0     0 $.1^2]
[$.1^6     0 $.1^3     0]
[$.1^2 $.1^5     2   $.1]
[    0 $.1^3 $.1^5   $.1]
```

65.3.9 Constructive Recognition of Suzuki Groups

65.3.9.1 Introduction

A description of the functionality for constructive recognition and constructive membership testing of the Suzuki groups $\mathrm{Sz}(q)$, with $q = 2^{2m+1}$ for some $m > 0$ follows.

The main intrinsics of the package are `RecogniseSz(G)` which performs constructive recognition of $G \cong \mathrm{Sz}(q)$, `SzElementToWord(G, g)` which returns a `GrpSLPElt` for g in the generators of G, and `IsSuzukiGroup(G)` which is a non-constructive test for isomorphism between G and $\mathrm{Sz}(q)$.

Informative printing can be obtained using one of a number of verbose flags:

`SuzukiGeneral`, for the general routines.

`SuzukiStandard`, for the routines related to the standard copy.

`SuzukiConjugate`, for the routines related to conjugation.

`SuzukiTensor`, for the routines related to tensor decomposition.

`SuzukiMembership`, for the routines related to membership testing.

`SuzukiCrossChar`, for the routines related to cross-characteristic representations.

`SuzukiTrick`, for the routines related to the double coset trick.

`SuzukiNewTrick`, for the routines related to the stabiliser trick.

For each of the flags, the verbose level takes any value up to 10, with higher values resulting in more output.

65.3.9.2 Recognition Functions

`IsSuzukiGroup(G)`

Given a matrix group G, this function determines (non-constructively) whether or not G is isomorphic to $\mathrm{Sz}(q)$. The corresponding finite field cardinality q is also returned.

If the group G is defined over a field of odd characteristic or has degree greater than 4, the Monte Carlo algorithm of `LieType` is used. If G has degree 4 and is over a field of characteristic 2, then a fast Las Vegas algorithm is used, described in [Bää06a].

`RecogniseSz(G : parameters)`

`RecognizeSz(G : parameters)`

`Verify`	BoolElt	*Default :* `true`
`FieldSize`	RngIntElt	*Default :*
`Optimise`	BoolElt	*Default :* `false`

Let G be a group that is absolutely irreducible and is defined over a minimal field. This function constructively recognises G as a Suzuki group. If G is isomorphic to $\mathrm{Sz}(q)$, where q is the size of the defining field of G, then return:

Isomorphism from G to $\mathrm{Sz}(q)$.

Isomorphism from $\mathrm{Sz}(q)$ to G.

Map from G to the word group of G.

Map from the word group of G to G.

The isomorphisms are composed of maps that are defined by rules, so `Function` should be used on each component to avoid unnecessary built-in membership testing. The word group is the `GrpSLP` group which is the parent of the elements returned by `SzElementToWord`. In general this is not the same as `WordGroup(G)`, but is created from it using `AddRedundantGenerators`.

If `Verify` is true, then it is checked if G is isomorphic to $\mathrm{Sz}(q)$, using `IsSuzukiGroup`. In that case, `FieldSize` must be set to the correct value of q. Constructive recognition of 2.Sz(8) is also handled.

If `Optimise` is true, then the third map returns element in an optimised word group (using `AddRedundantGenerators`). Then each invocation of the map will be faster, but the initialisation will take longer.

The algorithms used for constructive recognition are described in [Bää06a] and [Bää05].

`SzElementToWord(G, g)`

If G has been constructively recognised as a Suzuki group, and if g is an element of G, then return `true` and a `GrpSLPElt` from the word group of G which evaluates to g, else return `false`.

This facilitates membership testing in G.

`SzPresentation(q)`

If $q = 2^{2m+1}$ for some $m > 0$, return a short presentation of $\mathrm{Sz}(q)$ on the MAGMA standard generators, i.e. the generators returned by the `Sz` intrinsic.

`SatisfiesSzPresentation(G)`

G is constructively recognised as $\mathrm{Sz}(q)$ for some q. Verify that it satisfies a presentation for this group.

`SuzukiIrreducibleRepresentation(F, twists :` *parameters*`)`

`CheckInput` BOOLELT *Default :* `true`

Let F be a finite field of cardinality $q = 2^{2m+1}$ for some $m > 0$, and let *twists* be a sequence of n distinct integers in the range $[0 \dots 2m]$. The function returns an absolutely irreducible representation of $\mathrm{Sz}(q)$ having dimension 4^n, being a tensor product of twisted powers of the copy returned by the `Sz` intrinsic, where the twists are given by the input sequence.

If `CheckInput` is true, then it is verified that F and *twists* satisfy the above requirements. Otherwise this is not checked.

Example H65E13

We illustrate the basic facilities starting with a random conjugate of the standard version of the Suzuki group $Sz(32)$. We first perform non-constructive recognition.

```
> G := Sz(32);
> G ^:= Random(Generic(G));
> flag, q := SuzukiRecognition(G);
> flag, q eq 32;
true true
```

The next step is to perform constructive recognition. The explicit isomorphisms will be the values of `iso` and `inv`.

```
> flag, iso, inv, g2slp, slp2g := RecognizeSz(G);
> flag;
true
> iso, inv;
Mapping from: GrpMat: G to MatrixGroup(4, GF(2^5)) given by a rule [no inverse]
Mapping from: MatrixGroup(4, GF(2^5)) to GrpMat: G given by a rule [no inverse]
```

We now experiment with membership testing. We use Function to avoid MAGMA's built-in membership testing but in doing so, we may not obtain the shortest possible SLP.

```
> w := Function(g2slp)(G.1);
> #w;
284
```

The algorithm is probabilistic, so different executions will most likely give different results.

```
> ww := Function(g2slp)(G.1);
> w eq ww;
false
```

Note that the resulting SLPs are from a word group that is not the word group W corresponding to the defining generators of G. However, they can be coerced into W.

```
> W := WordGroup(G);
> NumberOfGenerators(Parent(w)), NumberOfGenerators(W);
7 3
> flag, ww := IsCoercible(W, w);
> flag;
true
> slp2g(w) eq Evaluate(ww, UserGenerators(G));
true
```

So there are two ways to get the element back. An alternative is to use the intrinsic `SzElementToWord`, which is better if the elements are not known to lie in the group.

```
> flag, ww := SzElementToWord(G, G.1);
> flag, slp2g(w) eq slp2g(ww);
```

```
true true
```

We take an element just outside the group.

```
> H := Sp(4, 32);
> flag, ww := SzElementToWord(G, H.1);
> flag;
false
> // in this case we will not get an SLP
> ww := Function(g2slp)(H.1);
> ww;
false
> SatisfiesSzPresentation(G);
true
```

Example H65E14

As a variation we apply the machinery to $2.Sz(8)$. We demonstrate constructive recognition and constructive membership testing.

```
> A := ATLASGroup("2Sz8");
> reps := MatRepKeys(A);
> G := MatrixGroup(reps[3]);
> Degree(G), CoefficientRing(G);
40 Finite field of size 7
> flag, iso, inv, g2slp, slp2g := RecognizeSz(G);
> flag;
true
> R := RandomProcess(G);
> g := Random(R);
> w := Function(g2slp)(g);
> slp2g(w) eq g;
true
```

Example H65E15

For the next example we consider a case where the dimension is large. We construct the Suzuki group in a 64-dimensional matrix representation and then take a random conjugate and also rewrite is over a smaller field.

```
> F := GF(2, 9);
> twists := [0, 3, 6];
> G := SuzukiIrreducibleRepresentation(F, twists);
> Degree(G), IsAbsolutelyIrreducible(G);
64 true
> G ^:= Random(Generic(G));
> flag, GG := IsOverSmallerField(G);
> flag, CoefficientRing(GG);
```

```
true Finite field of size 2^3
```

Non-constructive recognition is harder in this case and will give us the defining field size. Constructive recognition will decompose the tensor product.

```
> time SuzukiRecognition(GG);
true 512
Time: 2.330
> time flag, iso, inv, g2slp, slp2g := RecogniseSz(GG);
Time: 4.800
> iso;
Mapping from: GrpMat: GG to MatrixGroup(4, GF(2^9)) given by a rule [no inverse]
```

Constructive membership is again easy

```
> R := RandomProcess(GG);
> g := Random(R);
> time w := Function(g2slp)(g);
Time: 0.020
> // but SLP evaluation is harder in large dimensions
> time slp2g(w) eq g;
true
Time: 0.370
> time SatisfiesSzPresentation(GG);
true
Time: 10.930
```

Example H65E16

The final example will be in cross characteristic. We build a representation of $Sz(8)$ in cross characteristic.

```
> G := Sz(8);
> _, P := SuzukiPermutationRepresentation(G);
> // for example over GF(9)
> M := PermutationModule(P, GF(3, 2));
> factors := CompositionFactors(M);
> exists(m64){f : f in factors | Dimension(f) eq 64};
true
> m64;
GModule m64 of dimension 64 over GF(3^2)
> H := ActionGroup(m64);
> IsAbsolutelyIrreducible(H);
true
> flag, G := IsOverSmallerField(H);
> Degree(G), CoefficientRing(G);
64 Finite field of size 3
```

We actually end up with a group in characteristic 3.

```
> time flag, iso, inv, g2slp, slp2g := RecogniseSz(G);
```

```
Time: 3.490
> iso;
Mapping from: GrpMat: G to MatrixGroup(4, GF(2^3)) given by a rule [no inverse]
> R := RandomProcess(G);
> g := Random(R);
> time w := Function(g2slp)(g);
Time: 0.010
> time slp2g(w) eq g;
true
Time: 0.110
> time SatisfiesSzPresentation(G);
true
Time: 0.330
```

65.3.10 Constructive Recognition of Small Ree Groups

65.3.10.1 Introduction

This machinery provides functionality for constructive recognition and constructive membership testing of the small Ree groups ${}^2\mathrm{G}_2(q) = \mathrm{Ree}(q)$, with $q = 3^{2m+1}$ for some $m > 0$.

The important intrinsics are `RecogniseRee` which performs constructive recognition of $G \cong \mathrm{Ree}(q)$, `ReeElementToWord` which returns a `GrpSLPElt` for g in the generators of G, and `IsReeGroup` which is a non-constructive test for isomorphism between G and $\mathrm{Ree}(q)$.

There are a few verbose flags used in the package.

`ReeGeneral`, for the general routines.

`ReeStandard`, for the routines related to the standard copy.

`ReeConjugate`, for the routines related to conjugation.

`ReeTensor`, for the routines related to tensor decomposition.

`ReeMembership`, for the routines related to membership testing.

`ReeCrossChar`, for the routines related to cross-characteristic representations.

`ReeTrick`, for the routines related to the stabiliser trick.

`ReeInvolution`, for the routines related to involution centralisers.

`ReeSymSquare`, for the routines related to symmetric square decomposition.

All the flags can be set to values up to 10, with higher values resulting in more output.

65.3.10.2 Recognition Functions

`RecogniseRee(G : parameters)`

`RecognizeRee(G : parameters)`

`Verify`	BoolElt	*Default :* `true`
`FieldSize`	RngIntElt	*Default :*
`Optimise`	BoolElt	*Default :* `false`

G is absolutely irreducible and defined over minimal field. Constructively recognise G as a Ree group. If G is isomorphic to Ree(q) where q is the size of the defining field of G, then return:

Isomorphism from G to Ree(q).

Isomorphism from Ree(q) to G.

Map from G to the word group of G.

Map from the word group of G to G.

The isomorphisms are composed of maps that are defined by rules, so `Function` should be used on each component to avoid unnecessary built-in membership testing.

The word group is the `GrpSLP` which is the parent of the elements returned by `ReeElementToWord`. In general this is not the same as `WordGroup(G)`, but is created from it using `AddRedundantGenerators`.

If `Verify` is true, then it is checked that G is isomorphic to Ree(q), using `IsReeGroup`, otherwise this is not checked. In that case, `FieldSize` must be set to the correct value of q.

If `Optimise` is true, then the third map returns element in an optimised word group (using `AddRedundantGenerators`). Then each invocation of the map will be faster, but the initialisation will take longer.

The algorithms for constructive recognition are those of [Bää06b].

`ReeElementToWord(G, g)`

If G has been constructively recognised as a Ree group, and if g is an element of G, then return `true` and a `GrpSLPElt` from the word group of G which evaluates to g, else return `false`.

This facilitates membership testing in G.

`IsReeGroup(G)`

Determine (non-constructively) if G is isomorphic to Ree(q). The corresponding q is also returned.

If G is over a field of characteristic not 3 or has degree greater than 7, the Monte Carlo algorithm of `LieType` is used. If G has degree 7 and is over a field of characteristic 3, then a fast Las Vegas algorithm is used.

`ReeIrreducibleRepresentation(F, twists : parameters)`

`CheckInput` BoolElt *Default :* `true`

The finite field F must have size $q = 3^{2m+1}$ for some $m > 0$, and *twists* should be a sequence of n distinct pairs of integers (i, j) where i is 7 or 27 and j in the range $[0 \dots 2m]$.

Return an absolutely irreducible representation of $\mathrm{Ree}(q)$, a tensor product of twisted powers of the representation of dimension 7 or 27, where the twists are given by the input sequence.

If `CheckInput` is `true`, then it is verified that F and *twists* satisfy the above requirements. Otherwise this is not checked.

Example H65E17

Our first example shows off the recognition machinery for the Ree group defined over $\mathbf{F}_{27}$.

```
> SetSeed(1);
> F := GF(3, 3);
> G := ReeGroup(F);
> G ^:= Random(Generic(G));
> flag, q := ReeRecognition(G);
> flag, q eq #F;
true true
> flag, iso, inv, g2slp, slp2g := RecognizeRee(G);
> flag;
true
> iso, inv;
Mapping from: GrpMat: G to MatrixGroup(7, GF(3^3)) given by a rule [no inverse]
Mapping from: MatrixGroup(7, GF(3^3)) to GrpMat: G given by a rule [no inverse]
```

We now experiment with membership testing. As the algorithm is probabilistic, different executions will most likely give different results.

```
> w := Function(g2slp)(G.1);
> #w;
342
> ww := Function(g2slp)(G.1);
> w eq ww;
false
```

The resulting SLPs are from another word group but can be coerced into W.

```
> W := WordGroup(G);
> NumberOfGenerators(Parent(w)), NumberOfGenerators(W);
7 3
> flag, ww := IsCoercible(W, w);
> flag;
true
> // so there are two ways to get the element back
> slp2g(w) eq Evaluate(ww, UserGenerators(G));
```

```
true
```

If the elements are not known to lie in the group, a better alternative is to use the intrinsic `ReeElementToWord`. We take a generator of $\Omega(7, F)$ as an example of an element not lying in $G_2(27)$.

```
> flag, ww := ReeElementToWord(G, G.1);
> flag, slp2g(w) eq slp2g(ww);
true true
> H := Omega(7, #F);
> flag, ww := ReeElementToWord(G, H.1);
> flag;
false
> ww := Function(g2slp)(H.1);
> ww;
false
```

65.3.11 Constructive Recognition of Large Ree Groups

65.3.11.1 Introduction

This machinery provides functionality for constructive recognition and constructive membership testing of the large Ree groups ${}^2\mathrm{F}_4(q) = \mathrm{LargeRee}(q)$, with $q = 2^{2m+1}$ for some $m > 0$.

The important intrinsics are `RecogniseLargeRee` which performs constructive recognition of $G \cong \mathrm{LargeRee}(q)$, `LargeReeElementToWord` which returns a `GrpSLPElt` for g in the generators of G, and `IsLargeReeGroup` which is a non-constructive test for isomorphism between G and $\mathrm{LargeRee}(q)$.

There are a few verbose flags used in the package.

`LargeReeGeneral`, for the general routines.

`LargeReeStandard`, for the routines related to the standard copy.

`LargeReeConjugate`, for the routines related to conjugation.

`LargeReeRyba`, for the routines related to membership testing.

`LargeReeTrick`, for the routines related to the stabiliser trick.

`LargeReeInvolution`, for the routines related to involution centralisers.

All the flags can be set to values up to 10, with higher values resulting in more output.

65.3.11.2 Recognition Functions

`RecogniseLargeRee(G : parameters)`

`RecognizeLargeRee(G : parameters)`

`Verify`	BoolElt	*Default :* `true`
`FieldSize`	RngIntElt	*Default :*
`Optimise`	BoolElt	*Default :* `false`

G is absolutely irreducible and defined over minimal field. Constructively recognise G as a Large Ree group. If G is isomorphic to LargeRee(q) where q is the size of the defining field of G, then return:

Isomorphism from G to LargeRee(q).

Isomorphism from LargeRee(q) to G.

Map from G to the word group of G.

Map from the word group of G to G.

The isomorphisms are composed of maps that are defined by rules, so `Function` should be used on each component to avoid unnecessary built-in membership testing.

The word group is the `GrpSLP` which is the parent of the elements returned by `LargeReeElementToWord`. In general this is not the same as `WordGroup(G)`, but is created from it using `AddRedundantGenerators`.

If `Verify` is true, then it is checked that G is isomorphic to LargeRee(q), using `IsLargeRee`, otherwise this is not checked. In that case, `FieldSize` must be set to the correct value of q.

If `Optimise` is true, then the third map returns element in an optimised word group (using `AddRedundantGenerators`). Then each invocation of the map will be faster, but the initialisation will take longer.

`LargeReeElementToWord(G, g)`

If G has been constructively recognised as a Large Ree group, and if g is an element of G, then return `true` and a `GrpSLPElt` from the word group of G which evaluates to g, else return `false`.

This facilitates membership testing in G.

`IsLargeReeGroup(G)`

Determine (non-constructively) if G is isomorphic to LargeRee(q). The corresponding q is also returned.

If G is over a field of characteristic not 2 or has degree greater than 26, the Monte Carlo algorithm of `LieType` is used. If G has degree 26 and is over a field of characteristic 2, then a fast Las Vegas algorithm is used.

65.4 Properties of Finite Groups Of Lie Type

65.4.1 Maximal Subgroups of the Classical Groups

The `ClassicalMaximals` function, written by Derek Holt and Colva Roney-Dougal, returns a list of the maximal subgroups of the classical quasisimple groups in their natural representations, as returned by the MAGMA functions `SL, Sp, SU, Omega, OmegaPlus, OmegaMinus`. The list should be complete for dimensions up to 12 apart from a few omissions in $\Omega^+(8,q)$ which will be rectified in the near future.

There are also options to return the normalisers of these subgroups in various groups, such as $\mathrm{GL}(n,q)$, $\mathrm{GU}(n,q)$, that lie between the quasisimple group and its normaliser in the general linear group. These should be sufficient to enable the skilled user to determine the maximal subgroups of any group lying between the quasisimple groups and its normaliser.

According to the theorem of Aschbacher [Asc84] discussed earlier in this chapter, the maximal subgroups of a quasisimple classical group over a finite field lie in (at least) one of nine categories, which were listed in the Aschbacher Reduction section.

The subgroups in the first eight of these categories are said to be of *geometric type* and can be described in a uniform fashion. This description is the topic of the book [KL90]. They are returned in all dimensions by `ClassicalMaximals`. There is no such uniform description of the subgroups in the ninth class, which have to be classified separately in each dimension. The lists in the papers [HM01], [HM02] and [LÖ1] contain sufficient information in theory to compute these subgroups up to dimension 250, but currently this has been carried out only up to dimension 12.

`ClassicalMaximals(type, d, q : parameters)`

`classes`	SETENUM	*Default :* $\{1 \ldots 9\}$
`all`	BOOLELT	*Default :* `true`
`special`	BOOLELT	*Default :* `true`
`general`	BOOLELT	*Default :* `true`
`normaliser`	BOOLELT	*Default :* `true`
`novelties`	BOOLELT	*Default :* `false`

Return a list of representatives of the conjugacy classes of maximal subgroups of the quasisimple group of the specified type in dimension d over the field of order q. The string *type* must be one of `L, S, U, O, O+, O-`.

If the optional parameter `classes` is set to a proper subset of $\{1 \ldots 9\}$, then only the subgroups lying in the corresponding Aschbacher categories will be returned.

If the option `all` is set `false`, then representatives of the conjugacy classes under the action of the full automorphism group of the simple classical group will be returned: so this option will usually result in fewer subgroups in the returned list!

The option `special` only has effect for types `O, O+, O-`. When this is set to `true`, the normalisers of the subgroups in the appropriate group $\mathrm{SO}(d,q)$, $\mathrm{SO}^+(d,q)$ or $\mathrm{SO}^-(d,q)$ will be returned.

If the option `general` is set to `true`, then the normalisers of the subgroups in the appropriate group $\mathrm{GL}(d,q)$, $\mathrm{GU}(d,q)$, $\mathrm{GO}(d,q)$, $\mathrm{GO}^+(d,q)$ or $\mathrm{GO}^-(d,q)$ will be returned. (This option has not effect for type `S`.)

If the option `normaliser` is set to `true`, then the normalisers of the subgroups in the full normaliser of the quasisimple group in the general linear group (i.e. the group preserving the relevant form modulo scalars) will be returned. (For type `L` this has the same effect as setting *general* to `true`.)

If the option `novelties` is set `true`, then the intersections with the quasisimple group of any novelty maximal subgroups of any groups lying between the simple group and its full automorphism group will be returned. Use this option with caution, because the results are not guaranteed to be reliable!

65.4.2 Maximal Subgroups of the Exceptional Groups

Here follows some intrinsics for creating and conjugating maximal subgroups of Suzuki and Ree groups. The flags `SuzukiMaximals` and `ReeMaximals` may be used to produce verbose output.

`SuzukiMaximalSubgroups(G)`

If G has been constructively recognised as a Suzuki group, return a sequence of representatives of the maximal subgroups of G. Also returns sequences of `GrpSLPElt` of the generators of the subgroups, from the word group of G.

`SuzukiMaximalSubgroupsConjugacy(G, R, S)`

If G has been constructively recognised as a Suzuki group and if R and S are conjugate maximal subgroups of G, then return an element g of G that conjugates R to S. A `GrpSLPElt` from the word group of G, that evaluates to g, is also returned.

`ReeMaximalSubgroups(G)`

If G has been constructively recognised as a Ree group, return a sequence of representatives of the maximal subgroups of G. Also returns sequences of `GrpSLPElt` of the generators of the subgroups, from the word group of G.

`ReeMaximalSubgroupsConjugacy(G, R, S)`

If G has been constructively recognised as a Ree group and if R and S are conjugate maximal subgroups of G, then return an element g of G that conjugates R to S. A `GrpSLPElt` from the word group of G, that evaluates to g, is also returned. This is not implemented if R, S are Frobenius groups.

65.4.3 Sylow Subgroups of the Classical Groups

The MAGMA ClassicalSylow package written by Mark Stather provides functionality for constructing and conjugating the Sylow p-subgroups of the classical groups over finite fields in their natural representation, for any prime p. The classical groups may be created in MAGMA using the `GL, SL, Sp, GO, GOPlus, GOMinus, SO, SOPlus, SOMinus, Omega, OmegaPlus, OmegaMinus, GU, SU` intrinsics.

This package makes use of code to compute the classical form fixed by a group written by Derek Holt, and code to conjugate classical forms written by Colva Roney-Dougal.

The algorithms in this package are described in [Sta], which in turn makes use of the descriptions of the Sylow subgroups of the classical groups given in [Wei55], [CF64], [R. R57] and [Car72]. The conjugation algorithms make use of only the Meataxe, Smash, basic linear algebra and the solution of norm equations over finite fields.

`ClassicalSylow(G,p)`

The argument G must be a classical group in its natural representation, up to conjugation, with the exception of $GO(2m+1, 2^e)$. More precisely, it must be a conjugate of a group returned by one of the intrinsics `GL, SL, Sp, GO, GOPlus, GOMinus, SO, SOPlus, SOMinus, Omega, OmegaPlus, OmegaMinus, GU, SU`. p must be a prime number. The intrinsic returns a Sylow p-subgroup of G as a matrix group.

`ClassicalSylowConjugation(G,P,S)`

The argument G must be a classical group in its natural representation, up to conjugation, with the exception of $GO(2m+1, 2^e)$. More precisely, it must be a conjugate of a group returned by one of the intrinsics `GL, SL, Sp, GO, GOPlus, GOMinus, SO, SOPlus, SOMinus, Omega, OmegaPlus, OmegaMinus, GU, SU`. The groups P and S must be Sylow p-subgroups of G. The intrinsic returns an element $g \in G$ with $P^g = S$.

`ClassicalSylowNormaliser(G,P)`

In this case G must the full classical group in its natural representation, up to conjugation, with the exception of $GO(2m + 1, 2^e)$. More precisely, it must be a conjugate of a group returned by one of the intrinsics `GL, Sp, GO, GOPlus, GOMinus, GU`. The subgroup P must be a Sylow p-subgroup of G. The intrinsic returns the normaliser of P in G.

`ClassicalSylowToPC(G,P)`

The argument G must be a classical group in its natural representation, up to conjugation, with the exception of $GO(2m+1, 2^e)$. More precisely, it must be a conjugate of a group returned by one of the intrinsics `GL, SL, Sp, GO, GOPlus, GOMinus, SO, SOPlus, SOMinus, Omega, OmegaPlus, OmegaMinus, GU, SU`. The group P must be a Sylow p-subgroup of G. The intrinsic returns a PC group Q isomorphic to P, and also an isomorphism from P to Q and an isomorphism from Q to P.

Example H65E18

We construct a Sylow 7-subgroup P of $G = Sp(28, 17^2)$, take a random conjugate S of P and then find a conjugating element g that takes P to S.

```
> SetSeed(1);
> G := Sp(28,17^2);
> time P := ClassicalSylow(G,7);
Time: 0.080
> S := P^Random(G);
> time g := ClassicalSylowConjugation(G,P,S);
Time: 0.400
```

We next compute the normaliser of P in G.

```
> time N := ClassicalSylowNormaliser(G,P);
Time: 0.310
> // and a PC presentation of P
> time Pc, PtoPc, PctoP := ClassicalSylowToPC(G,P);
Time: 0.200
> Pc;
GrpPC : Pc of order 2401 = 7^4
PC-Relations:
    Pc.1^7 = Id(Pc),
    Pc.2^7 = Id(Pc),
    Pc.3^7 = Id(Pc),
    Pc.4^7 = Id(Pc)
> // We get inverse isomorphisms PtoPc and PctoP
> g := Random(P);
> PctoP(PtoPc(g)) eq g;
true
> x := Random(Pc);
> PtoPc(PctoP(x)) eq x;
true
```

65.4.4 Sylow Subgroups of Exceptional Groups

The flags `SuzukiSylow` and `ReeSylow` may be used to produce verbose output.

`SuzukiSylow(G, p)`

If G has been constructively recognised as a Suzuki group, and if p is a prime number, return a random Sylow p-subgroup S of G.

Also returns a list of `GrpSLPElt` from the word group of G, of the generators of S. If p does not divide $|G|$, then the trivial subgroup is returned.

`SuzukiSylowConjugacy(G, R, S, p)`

If G has been constructively recognised as a Suzuki group, if p is a prime number and if R and S are Sylow p-subgroups of G, then return an element g of G that conjugates R to S. A `GrpSLPElt` from the word group of G, that evaluates to g, is also returned.

Example H65E19

We demonstrate finding a conjugating element for Sylow subgroup in an example over a large field.

```
> q := 2^121;
> G := Sz(q);
> G ^:= Random(Generic(G));
> G := DerivedGroupMonteCarlo(G);
> NumberOfGenerators(G);
19
```

Non-constructive recognition is now a bit harder.

```
> time SuzukiRecognition(G);
true 26584559915698317458076141205606891 52
Time: 0.190
> time flag, iso, inv, g2slp, slp2g := RecogniseSz(G);
Time: 22.810
```

However, after this, each call to constructive membership testing is then easy.

```
> R := RandomProcess(G);
> g := Random(R);
> time w := Function(g2slp)(g);
Time: 0.060
> // evaluating SLPs always takes some time
> time slp2g(w) eq g;
true
Time: 1.250
```

We now create some Sylow subgroups and find conjugating elements.

```
> p := Random([x[1] : x in Factorization(q - 1)]);
> time R := SuzukiSylow(G, p);
Time: 1.370
> time S := SuzukiSylow(G, p);
Time: 1.310
> // that was easy, as is conjugating them
> time g, slp := SuzukiSylowConjugacy(G, R, S, p);
Time: 1.340
> slp2g(slp) eq g;
true
> #R, NumberOfGenerators(R);
23 1
```

```
> time R := SuzukiSylow(G, 2);
Time: 164.020
> time S := SuzukiSylow(G, 2);
Time: 171.740
> NumberOfGenerators(R), #R;
121 706738825911353731833319000297167406330993558750247583248642480517047910 4
> time g, slp := SuzukiSylowConjugacy(G, R, S, 2);
Time: 1.650
```

Creating the Sylow 2-subgroup is hard since they have so many generators. One the other hand, finding a conjugating element is relatively easy.

`ReeSylow(G, p)`

If G has been constructively recognised as a Ree group, and if p is a prime number, return a random Sylow p-subgroup S of G.

Also returns a list of `GrpSLPElt` from the word group of G, of the generators of S. If p does not divide $|G|$, then the trivial subgroup is returned.

`ReeSylowConjugacy(G, R, S, p)`

If G has been constructively recognised as $\mathrm{Ree}(q)$, if p is a prime number and if R and S are Sylow p-subgroups of G, then return an element g of G that conjugates R to S, and a `GrpSLPElt` from the word group of G, that evaluates to g, is also returned.

Currently, this is not implemented for odd p that divide $q^3 + 1$.

`LargeReeSylow(G, p)`

If G has been constructively recognised as a Large Ree group, and if p is a prime number, return a random Sylow p-subgroup S of G.

Also returns a list of `GrpSLPElt` from the word group of G, of the generators of S. If p does not divide $|G|$, then the trivial subgroup is returned.

Currently, this is not implemented for p that divide $q + 1$.

Example H65E20

Starting with the Ree group over the field $\mathbf{F}_{3^{3_1}}$, we construct Sylow p-subgroups for different primes p.

```
> m := 7;
> F := GF(3, 2 * m + 1);
> q := #F;
> q;
14348907
> G := ReeGroup(F);
> G ^:= Random(Generic(G));
> G := DerivedGroupMonteCarlo(G);
> NumberOfGenerators(G);
```

```
19
> ReeRecognition(G);
true 14348907
> flag, iso, inv, g2slp, slp2g := RecogniseRee(G);
> R := RandomProcess(G);
> g := Random(R);
> w := Function(g2slp)(g);
> slp2g(w) eq g;
true
```

We first create two Sylow p-subgroups of prime order and find a conjugating element. Note that 4561 divides the order of G exactly once and also divides $q - 1$.

```
> p := 4561;
> R := ReeSylow(G, p);
> S := ReeSylow(G, p);
> g, slp := ReeSylowConjugacy(G, R, S, p);
```

Thus $R^g = S$. In this case we also automatically get an SLP for the conjugating element.

```
> slp2g(slp) eq g;
true
> #R, NumberOfGenerators(R);
4561 1
```

Sylow 3-subgroups are harder: they have order 3^{45} and hence a considerable number of generators.

```
> time R := ReeSylow(G, 3);
Time: 3.730
> S := ReeSylow(G, 3);
> NumberOfGenerators(R), #R;
15 2954312706550833698643
> time g, slp := ReeSylowConjugacy(G, R, S, 3);
Time: 0.300
```

65.4.5 Conjugacy of Subgroups of the Classical Groups

`IsGLConjugate(H, K)`

Given H and K, both subgroups of the same general linear group $G = GL(n, q)$, return the value `true` if H and K are conjugate in G. The function returns a second value in the event that the subgroups are conjugate: an element z which conjugates H into K. The algorithm is described in Roney-Dougal [RD04].

65.4.6 Conjugacy of Elements of the Exceptional Groups

The flags `SuzukiElements` and `ReeElements` may be used to produce verbose output.

`SzConjugacyClasses(G)`

If G has been constructively recognised as a Suzuki group, return a list of conjugacy classes, using the same format as the `ConjugacyClasses` intrinsic.

`SzClassRepresentative(G, g)`

If G has been constructively recognised as a Suzuki group, and g is an element of G, return the conjugacy class representative h of g, such that h is in the list returned by `SzConjugacyClasses`. Also returns c in G such that $g^c = h$.

`SzIsConjugate(G, g, h)`

If G has been constructively recognised as a Suzuki group, and g and h are elements of G, determine if g is conjugate to h. If so, return `true` and an element c such that $g^c = h$, otherwise return `false`.

`SzClassMap(G)`

If G has been constructively recognised as a Suzuki group, return its class map, as in the `ClassMap` intrinsic.

`ReeConjugacyClasses(G)`

If G has been constructively recognised as a Ree group, return a list of conjugacy classes, using the same format as the `ConjugacyClasses` intrinsic.

65.4.7 Irreducible Subgroups of the General Linear Group

`IrreducibleSubgroups(n, q)`

Return the list of conjugacy classes of irreducible subgroups of $GL(n, q)$ where q is a prime power. At present, the dimension n is restricted to 2. The list is complete for characteristic at least 5. The algorithm is based on the classification of Flannery and O'Brien [FO05].

`IrreducibleSolubleSubgroups(n, q)`

Return the list of conjugacy classes of soluble irreducible subgroups of $GL(n, q)$ where q is a prime power. At present, the dimension n is restricted to 2 or 3. The list is complete for characteristic at least 5. The algorithm is based on the classification of Flannery and O'Brien [FO05].

Example H65E21

```
> L := IrreducibleSubgroups(2, 19^5);
> #[x : x in L | IsAbelian (x)];
552
```

```
> L := IrreducibleSolubleSubgroups(2, 97^2);
> #L;
10617
> L[7];
MatrixGroup(3, GF(97^2))
Generators:
    [$.1^8775 $.1^2037 $.1^6016]
    [$.1^6017 $.1^6705 $.1^7812]
    [$.1^7813 $.1^2817   $.1^33]
```

65.5 Atlas Data for the Sporadic Groups

Most of the functions described here use data derived from the Web Atlas. The data has been prepared for inclusion in MAGMA by Michael Downward and Eamonn O'Brien. It maintains Atlas names, conventions and orderings.

All of these functions, except `GoodBasePoints`, accept as input matrix or permutation groups. The algorithm underpinning `GoodBasePoints` due to O'Brien & Wilson [OW05].

`StandardGenerators(G, str : parameters)`

`Projective`	BOOLELT	*Default :* `false`
`AutomorphismGroup`	BOOLELT	*Default :* `false`

Construct standard generators for small quasisimple or sporadic group G having name *str*; words in SLP group defined on the defining generators of G are also obtained for the standard generators.

If G is sporadic and `AutomorphismGroup` is `true`, assume G is automorphism group of group having name *str*.

If standard generators found, return `true` and sequences of generators and corresponding words, else `false`.

Note: A return value of `false` only means that the algorithm's random search for standard generators did not succeed within the number of tries allowed. If the user is sure the group G matches the name *str*, then they should try the function again.

If G is absolutely irreducible matrix group and `Projective` is `true`, then construct standard generators possibly modulo centre of G.

This function currently works for all sporadic simple groups and all quasisimple groups for which the simple quotient has order at most 2×10^8. If you call it with an invalid value of *str*, then it will print out a list of all valid values.

`IsomorphismToStandardCopy(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`AutomorphismGroup`	BoolElt	*Default :* `false`

Use the StandardGenerators function to construct a (possibly projective) isomorphism from G to a standard copy of G. Options as for StandardGenerators. The first returned value indicates whether the call of StandardGenerators was successful.

`StandardPresentation(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`Generators`	SeqEnum	*Default :* []
`AutomorphismGroup`	BoolElt	*Default :* `false`

Return `true` if standard presentation is satisfied by generators of sporadic group G having name *str*, else `false`.

If `AutomorphismGroup` is `true`, assume G is automorphism group of sporadic group having name *str*.

Standard generators may be supplied as `Generators`, otherwise defining generators are assumed to be standard.

If G is absolutely irreducible matrix group and `Projective` is `true`, then verify presentation modulo centre of G.

`MaximalSubgroups(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`Generators`	SeqEnum	*Default :* []
`AutomorphismGroup`	BoolElt	*Default :* `false`

Construct **some** maximal subgroups for sporadic group G having name *str*. If `AutomorphismGroup` is `true`, assume G is automorphism group of sporadic group having name *str* and construct **some** of its maximal subgroups.

If standard generators supplied as `Generators` or found for G then return `true` and list of subgroups, else return `false`.

If G is absolutely irreducible matrix group and `Projective` is `true`, then construct standard generators and so subgroups possibly modulo centre of G.

`Subgroups(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`Generators`	SeqEnum	*Default :* []

Construct certain subgroups for sporadic group G having name *str*. If standard generators supplied as `Generators` or found for G then return `true` and list of subgroups, else return `false`.

If G is absolutely irreducible matrix group and `Projective` is `true`, then construct standard generators possibly modulo centre of G.

`GoodBasePoints(G, str : parameters)`

`Projective`	BoolElt	*Default :* `false`
`Generators`	SeqEnum	*Default :* []

If standard generators supplied as `Generators` or found for sporadic group G having name *str*, then return `true` and list of base points for G, else return `false`.

If G is absolutely irreducible and `Projective` is `true`, then standard generators are possibly modulo centre of G, and base points are correspondingly adjusted.

`SubgroupsData(str)`

Display stored subgroup data for sporadic group having name *str*.

`MaximalSubgroupsData (str : parameters)`

`AutomorphismGroup`	BoolElt	*Default :* `false`

Display stored data for some maximal subgroups of sporadic group having name *str*. If `AutomorphismGroup` is `true`, then display stored data for some maximal subgroups of automorphism group of sporadic group.

Example H65E22

The machinery is illustrated in the case of the sporadic Janko group J_1.

```
> G :=
> MatrixGroup<7, GF(11) |
> [ 9, 1, 1, 3, 1, 3, 3, 1, 1, 3, 1, 3, 3, 9, 1, 3, 1, 3, 3, 9, 1, 3, 1, 3,
>  3, 9, 1, 1, 1, 3, 3, 9, 1, 1, 3, 3, 3, 9, 1, 1, 3, 1, 3, 9, 1, 1, 3, 1, 3 ],
> [ 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0,
> 0, 1, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 10, 10, 0, 0, 0, 0, 0,0] >;
> flag, S := StandardGenerators (G, "J1");
> flag;
true
> StandardPresentation (G, "J1": Generators := S);
true
> flag, M:= MaximalSubgroups (G, "J1": Generators := S);
> #M;
7
> M[4];
rec<recformat<name: MonStgElt, parent: MonStgElt, generators: SeqEnum,
 group: Grp, order: RngIntElt, index: RngIntElt> |
     name := 19:6,
     parent := J1,
     group := MatrixGroup(7, GF(11))
     Generators:
         [ 0  1  4  3  3  4  7]
         [ 1  2  8  3  6  2  9]
         [ 4  8 10  1  6  0  9]
         [ 3  3  1  8  9  1 10]
```

```
    [ 3  6  6  9  1  3  7]
    [ 4  2  0  1  3  0  9]
    [ 7  9  9 10  7  9  0]
    [ 4  6  2  3  8  1  6]
    [ 8  1  3 10  2  7  4]
    [ 3  6  1  0  6  9  6]
    [ 2  3  6  9  0  3  7]
    [ 7  8  5  2  4  6  4]
    [10  4  5  2  8  6  8]
    [10  9  0  1  9  8  9],
order := 114,
index := 1540
>
```

65.6 Bibliography

[**Asc84**] M. Aschbacher. On the maximal subgroups of the finite classical groups. *Invent. Math*, 76:469–514, 1984.

[**Bää05**] H. Bäärnhielm. Tensor decomposition of the Suzuki groups. submitted, 2005.

[**Bää06a**] H. Bäärnhielm. Recognising the Suzuki groups in their natural representations. *J. Algebra*, 300(1):171–198, 2006.

[**Bää06b**] Henrik Bäärnhielm. Constructive recognition of the Ree groups. preprint, 2006.

[**BKPS02**] L. Babai, W. M. Kantor, P. P. Pálfy, and Á. Seress. Black-box recognition of finite simple groups of Lie type by statistics of element orders. *J. Group Theory*, 5:383–401, 2002.

[**BLGN$^+$03**] R. Beals, C. R. Leedham-Green, A. C. Niemeyer, C. E. Praeger, and A. Seress. A black-box algorithm for recognising finite symmetric and alternating groups, I. *Trans. Amer. Math. Soc.*, 2003. To appear.

[**BP00**] Sergey Bratus and Igor Pak. Fast constructive recognition of a black box group isomorphic to S_n or A_n using Goldbach's conjecture. *J. Symbolic Comp.*, 29:33–57, 2000.

[**Car72**] R. Carter. *Simple Groups of Lie Type.* John Wiley & Sons, London, New York, Sydney, Toronto, 1972.

[**CF64**] R. Carter and P. Fong. The Sylow 2-subgroups of the finite classical groups. *Journal of Algebra*, 1:139–151, 1964.

[**CLG97a**] Frank Celler and Charles R. Leedham-Green. Calculating the Order of an Invertible Matrix. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 55–60. AMS, 1997.

[**CLG97b**] Frank Celler and C.R. Leedham-Green. A non-constructive recognition algorithm for the special linear and other classical groups. In *Groups and computation II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 61–67. Amer. Math. Soc., 1997.

[**CLGM$^+$95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

[**CLGO06**] M.D.E. Conder, C.R. Leedham-Green, and E.A. O'Brien. Constructive recognition for $PSL(2, q)$. *Trans. Amer. Math. Soc.*, 358:1203–1221, 2006.

[**FO05**] D.L. Flannery and E.A. O'Brien. Linear groups of small degree over finite fields. *Internat. J. Algebra and Comput.*, 15:467–502, 2005.

[**HM01**] G. Hiß and G. Malle. Low-dimensional representations of quasi-simple groups. *LMS J. Comput. Math.*, 4:22–63, 2001.

[**HM02**] G. Hiß and G. Malle. Corrigenda: Low-dimensional representations of quasi-simple groups. *LMS J. Comput. Math.*, 5:95–126, 2002.

[**HRT01**] R. B. Howlett, L. J. Rylands, and D. E. Taylor. Matrix generators for exceptional groups of Lie type. *J. Symbolic Comput.*, 31(4):429–445, 2001.

[**KL90**] Peter Kleidman and Martin Liebeck. *The Subgroup Structure of the Finite Classical Groups*, volume 129 of *London Math. Soc. Lecture Note Ser.* CUP, Cambridge, 1990.

[**LÖ1**] F. Lübeck. Small degree representations of finite Chevalley groups in defining ch aracteristic. *LMS J. Comput. Math.*, 4:135–169, 2001.

[**LMO07**] F Lübeck, K Magaard, and E.A. O'Brien. Constructive recognition of $\mathrm{SL}_3(q)$. *J. Algebra*, 2007:617–633, 2007.

[**LO07**] Martin Liebeck and E.A. O'Brien. Finding the characteristic of a group of Lie type. *J. London Math. Soc.*, 2007.

[**MOAS08**] Kay Magaard, E. A. O'Brien, and Ákos Seress. Recognition of small dimensional representations of general linear groups. *J. Austral. Math. Soc.*, 85: 229–250, 2008.

[**NP92**] Peter M. Neumann and Cheryl E. Praeger. A Recognition Algorithm for Classical Groups. *Proc. London Math. Soc.*, 65(3):555–603, 1992.

[**NP97**] Alice C. Niemeyer and Cheryl E. Praeger. Implementing a Recognition Algorithm for Classical Groups. In *Groups and computation II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 273–296. Amer. Math. Soc., 1997.

[**NP98**] Alice C. Niemeyer and Cheryl E. Praeger. A Recognition Algorithm for Classical Groups over Finite Fields. *Proc. London Math. Soc.*, 77(3):117–169, 1998.

[**NP99**] Alice C. Niemeyer and Cheryl E. Praeger. A Recognition Algorithm for Non-Generic Classical Groups over Finite Fields. *J. Austral. Math. Soc. Ser. A*, 67:223–253, 1999.

[**OW05**] E.A. O'Brien and R.A. Wilson. Subgroup chains in matrix groups. *preprint*, 2005.

[**Pra99**] Cheryl E. Praeger. Primitive prime divisor elements in finite classical groups. In *Proc. of Groups St. Andrews 1997 in Bath II*, number 261 in London Math. Soc. Lecture Notes Series, pages 605–623. Cambridge Univ. Press, 1999.

[**RD04**] Colva M. Roney-Dougal. Conjugacy of subgroups of the general linear group. *Experiment. Math.*, 13:151–163, 2004.

[**R.R57**] R.Ree. On some simple groups defined by Chevalley. *Trans. Am. Math. Soc.*, 84:392–400, 1957.

[**RT98**] L.J. Rylands and D.E. Taylor. Matrix generators for the orthogonal groups. *J. Symbolic Comp.*, 25:351–360, 1998.

[**Sta**] M. Stather. Constructive Sylow Theorems for the Classical Groups. to appear in Journal of Algebra.

[**Tay87**] Don Taylor. Pairs of Generators for Matrix Groups. I. *Cayley Bulletin 3*, 1987.

[**Wei55**] A. Weir. Sylow p-subgroups of the classical groups over finite fields with characteristic prime to p. *Proc. Am. Math. Soc*, 6:529–533, 1955.

66 DATABASES OF GROUPS

Chapter 66
DATABASES OF GROUPS

66.1 Introduction

This chapter describes the use of the various databases of groups that form part of MAGMA. The available databases are as follows:

Small Groups: This database is constructed by Hans Ulrich Besche, Bettina Eick and Eamonn O'Brien [BE99a, BEO01, BE99b, O'B90, BE01, O'B91, MNVL04, OVL05, DE05], contains the following groups:

- All groups of order up to 2000, excluding the groups of order 1024.
- The groups whose order is the product of at most 3 primes.
- The groups of order dividing p^6 for p a prime.
- The groups of order $q^n p$, where q^n is a prime-power dividing 2^8, 3^6, 5^5 or 7^4 and p is a prime different to q.
- The groups of square-free order. For a different mechanism for accessing the p-groups in this collection, see Section 66.3, specifically the functions `SearchPGroups` and `CountPGroups`. These functions also access groups of order p^7.

p-groups: MAGMA contains the means to construct all p-groups of order p^n where $n \leq 7$. The data used in the constructions was supplied by Hans Ulrich Besche, Bettina Eick, Eamonn O'Brien, Mike Newman and Michael Vaughan-Lee [BE99a, BEO01, BE99b, O'B90, BE01, O'B91, MNVL04, OVL05].

Metacyclic p-groups: MAGMA is able to construct all metacyclic groups of order p^n. This machinery was developed by Mike Newman, Eamonn O'Brien, and Michael Vaughan-Lee.

Perfect Groups: This database contains all perfect groups up to order 50000, and many classes of perfect groups up to order one million. Each group is defined by means of a finite presentation. Further information is also provided which allows the construction of permutation representations. This database was constructed by Derek Holt and Willem Plesken [HP89].

Almost Simple Groups: This database contains information about every group G, where $S \leq G \leq \mathrm{Aut}(S)$ and S is a simple group of order less than 16000000, or S is one of M_{24}, HS, J_3, McL, $Sz(32)$ or $L_6(2)$.

Transitive Permutation Groups: This database is a MAGMA version of the database of transitive permutation groups constructed by Alexander Hulpke [Hul05] (for degree up to 30) and Cannon and Holt [CH08]. It contains all transitive permutation groups having degree up to 32.

Primitive Permutation Groups: This is a database containing all primitive permutation groups having degree less than 4095 as determined by Sims (for degree ≤ 50), Roney-Dougal and Unger [RDU03] (for degree < 1000), Roney-Dougal [RD05] (for degree < 2500), and Coutts, Quick and Roney-Dougal [CQRD11] (for degree < 4096).

Rational Maximal Matrix Groups: This contains the rational maximal finite matrix groups and their invariant forms, for small dimensions (up to 31) as determined by Gabi Nebe and Willem Plesken [NP95, Neb96]. Each entry can be accessed either as a matrix group or as a lattice.

Quaternionic Matrix Groups: A database of the finite absolutely irreducible subgroups of $\mathrm{GL}_n(\mathcal{D})$ where $\mathcal{D}$ is a definite quaternion algebra whose centre has degree d over $\mathbf{Q}$ and $nd \leq 10$. Each entry can be accessed either as a matrix group or as a lattice. The database was constructed by Gabi Nebe [Neb98].

Irreducible Matrix Groups: A database of the irreducible subgroups of $\mathrm{GL}_n(p)$, p prime, $n \geq 1$ and $p^n < 2500$. The groups were determined by Colva Roney-Dougal and William Unger [RDU03] (for $p^n < 1000$) and Roney-Dougal [RD05].

Soluble Irreducible Groups: This database contains one representative of each conjugacy class of irreducible soluble subgroups of $\mathrm{GL}(n, p)$, p prime, for $n > 1$ and $p^n < 256$. It was constructed by Mark Short [Sho92].

ATLAS Groups: This database contains representations of nearly simple groups, as in the Birmingham ATLAS of Finite Group Representations. The data was supplied by Rob Wilson.

66.2 Database of Small Groups

Magma includes the Small Groups Library prepared by Besche, Eick and O'Brien. For a description of the algorithms used to generate these groups, details on the data structures used and applications we refer to [BE99a, BEO01, BE99b, O'B90, BE01, O'B91, MNVL04] and the references therein.

The Small Groups Library contains the following groups.

- All groups of order up to 2000, excluding the groups of order 1024.
- The groups whose order is a product of at most 3 primes.
- The groups of order dividing p^6 for p a prime.
- The groups of order $q^n p$, where q^n is a prime-power dividing 2^8, 3^6, 5^5 or 7^4 and p is a prime different to q.

The descriptions of the groups of order p^4, p^5, p^6 for $p > 3$ were contributed by Boris Girnat, Robert McKibbin, M.F. Newman, E.A. O'Brien, and M.R. Vaughan-Lee.

The Magma version of this library uses the same internal data format as the implementation available in GAP. In particular, the numbering of the groups of a given order in both packages is the same.

For a different mechanism for accessing the p-groups in this collection, see the 66.3 section, specifically the functions `SearchPGroups` and `CountPGroups`. These functions also access the groups of order p^7 (contributed by O'Brien and Vaughan-Lee).

66.2.1 Basic Small Group Functions

Many of the functions in this section have an optional parameter `Search`. It can be used to limit the small group search to soluble (`Search := "Soluble"`) or insoluble (`Search := "Insoluble"`) groups. The default is `Search := "All"`, which allows all groups to be considered.

When a group is extracted from the database, it is returned as a `GrpPC` if it is soluble, or as a `GrpPerm` if it is insoluble.

When using the small groups database for an extended search, it is advisable to open the database using the function `SmallGroupDatabase`, which opens the database and returns a reference to it. This reference can then be passed as first argument to most of the functions described below, and will save that function from opening and closing the database for itself. Doing so will reduce the number of file operations when a lot of use is made of the database. When the database is no longer needed, it can be closed using the `delete` statement.

```
SmallGroupDatabase()
OpenSmallGroupDatabase()
```

Open the small groups database (for extended search) and return a reference to it. This reference may be passed to other functions so that they do fewer file operations.

```
delete D
```

Close the small groups database `D` and free the resources associated with its use.

```
SmallGroupDatabaseLimit()
SmallGroupDatabaseLimit(D)
```

The limiting order up to which all groups (except those of order 1024) are stored in the database of small groups, that is, currently 2000.

```
IsInSmallGroupDatabase(o)
IsInSmallGroupDatabase(D, o)
```

Return `true` if the groups of order o are contained in the database and `false` otherwise. This function can be used to check whether o is a legitimate argument for other functions described in this section, avoiding runtime errors in user written loops or functions.

```
NumberOfSmallGroups(o)
NumberOfSmallGroups(D, o)
```

Given a positive integer o, return the number of groups of order o in the database. If the groups of order o are not contained in the database, 0 is returned. This function can be used to check whether a pair o, n defines a group contained in the small groups database, that is, whether it is a legitimate argument for other functions described in this section, avoiding runtime errors in user written loops or functions.

`SmallGroup(o, n)`
`SmallGroup(D, o, n)`
`Group(D, o, n)`

Given a positive integer o, such that the groups of order o are contained in the small groups library, and a positive integer n, return the n-th group of order o in the database. If the groups of order o are not contained in the database or if n exceeds the number of groups of order o in the database, an error is reported. The function `NumberOfSmallGroups` can be used to check whether the arguments are valid.

`SmallGroup(o:` *parameters*`)`
`SmallGroup(D, o:` *parameters*`)`

`Search` MonStgElt *Default : "All"*

Given a positive integer o, such that the groups of order o are contained in the small groups library, return the first group of order o in the database meeting the search criterion set by the parameter `Search`. If the groups of order o are not contained in the database, an error is reported. The function `IsInSmallGroupDatabase` can be used to check whether o is a valid argument for this function.

`SmallGroup(o, f:` *parameters*`)`
`SmallGroup(D, o, f:` *parameters*`)`

`Search` MonStgElt *Default : "All"*

Given a positive integer o such that the groups of order o are contained in the small groups library and a predicate f (as a function or intrinsic), return the first group of order o in the database meeting the search criterion set by the parameter `Search`, which satisfies f.

`IsSoluble(D, o, n)`
`IsSolvable(D, o, n)`
`SmallGroupIsSoluble(o, n)`
`SmallGroupIsSoluble(D, o, n)`
`SmallGroupIsSolvable(o, n)`
`SmallGroupIsSolvable(D, o, n)`

Return `true` iff `SmallGroup(o, n)` is soluble. This function does not load the group. If the group specified by the arguments does not exist in the database, an error is reported. The function `NumberOfSmallGroups` can be used to check whether the arguments are valid.

`SmallGroupIsInsoluble(o, n)`
`SmallGroupIsInsoluble(D, o, n)`
`SmallGroupIsInsolvable(o, n)`
`SmallGroupIsInsolvable(D, o, n)`

Return `true` iff `SmallGroup(o, n)` is insoluble. This function does not load the group. If the group specified by the arguments does not exist in the database, an error is reported. The function `NumberOfSmallGroups` can be used to check whether the arguments are valid.

`SmallGroup(o, f:` *parameters*`)`
`SmallGroup(S, f:` *parameters*`)`
`SmallGroup(D, o, f:` *parameters*`)`
`SmallGroup(D, S, f:` *parameters*`)`

`Search` MonStgElt *Default :* "*All*"

Given a sequence S of orders or a single order o contained in the database and a predicate f (as a function or intrinsic), return the first group with order in S or equal to o, respectively, which meets the search criterion set by the parameter `Search` and satisfies f.

`SmallGroups(o:` *parameters*`)`
`SmallGroups(D, o:` *parameters*`)`

`Search` MonStgElt *Default :* "*All*"
`Warning` BoolElt *Default :* `true`

Given an order o contained in the database, return a list of all groups of order o, meeting the search criterion set by the parameter `Search`. Some orders will produce a very large sequence of groups – in such cases a warning will be printed unless the user specifies `Warning := false`.

`SmallGroups(S:` *parameters*`)`
`SmallGroups(D, S:` *parameters*`)`

`Search` MonStgElt *Default :* "*All*"
`Warning` BoolElt *Default :* `true`

Given a sequence S of orders contained in the database, return a list of all groups with order in S, meeting the search criterion set by the parameter `Search`. The resulting sequence may be very long – in such cases a warning will be printed unless the user specifies `Warning := false`.

`SmallGroups(o, f: parameters)`

`SmallGroups(D, o, f: parameters)`

`Search` MonStgElt *Default : "All"*

Given an order o contained in the database and a predicate (function or intrinsic) f, return a list containing all groups G of order o, meeting the search criterion set by the parameter `Search` and satisfying `f(G) eq true`.

`SmallGroups(S, f: parameters)`

`SmallGroups(D, S, f: parameters)`

`Search` MonStgElt *Default : "All"*

Given a sequence S of orders contained in the database and a predicate (function or intrinsic) f, return a list containing all groups G with order in S, meeting the search criterion set by the parameter `Search` and satisfying `f(G) eq true`.

Example H66E1

(1) We find the non-abelian groups of order 27.

```
> list := SmallGroups(27, func<x|not IsAbelian(x)> );
> list;
[*
GrpPC of order 27 = 3^3
PC-Relations:
  $.2^$.1 = $.2 * $.3,

GrpPC of order 27 = 3^3
PC-Relations:
  $.1^3 = $.3,
  $.2^$.1 = $.2 * $.3
*]
```

(2) We get the first group in the database with derived length greater than 2.

```
> G := SmallGroup([1..100], func<x|DerivedLength(x) gt 2>);
> G;
GrpPC of order 24 = 2^3 * 3
PC-Relations:
  G.1^3 = Id(G),
  G.2^2 = G.4,
  G.3^2 = G.4,
  G.4^2 = Id(G),
  G.2^G.1 = G.3,
  G.3^G.1 = G.2 * G.3,
```

```
  G.3^G.2 = G.3 * G.4
```

(3) Now for a list of the insoluble groups of order 240. The insoluble groups in the database are returned as permutation groups.

```
> list := SmallGroups(240:Search:="Insoluble");
> #list;
8
> list[7];
Permutation group acting on a set of cardinality 7
  (1, 2, 3, 4)
  (1, 5, 2, 4, 3)(6, 7)
```

(4) The groups of order $2432 = 2^7 \cdot 19$ should be contained in the small groups database. We check this using the function IsInSmallGroupDatabase...

```
> IsInSmallGroupDatabase(2432);
true
```

... and determine the number of groups of order 2432.

```
> NumberOfSmallGroups(2432);
19324
```

(5) We find all groups of order 7^6 with cyclic centre of order 7^2.

```
> f := function (G)
>   Z := Centre (G);
>   return IsCyclic (Z) and #Z eq 7^2;
> end function;
> P := SmallGroups(7^6, f);
> #P;
30
> NumberOfSmallGroups(7^6);
860
```

66.2.2 Processes

A small group process enables iteration over all groups of specified orders satisfying a given predicate, without having to create and store all such groups together.

A small group process is created via the function `SmallGroupProcess` (in various forms). The standard process functions `IsEmpty`, `Current`, `CurrentLabel` and `Advance` can then be applied to the process.

The functions used to create a small group process all have a parameter `Search` attached to them. It can be used to limit the small group search to soluble (`Search := "Soluble"`) or insoluble (`Search := "Insoluble"`) groups. The default is `Search := "All"`, which allows all groups to be considered.

The `Process` functions described below do not have a variant with the database as first argument, as each process opens the database for an extended search automatically.

`SmallGroupProcess(o: parameters)`

`Search` MonStgElt *Default :* "*All*"

Given an order o contained in the small groups database, return a small group process which will iterate though all groups of order o meeting the search criterion set by the parameter `Search`.

`SmallGroupProcess(S: parameters)`

`Search` MonStgElt *Default :* "*All*"

Given a sequence S of orders contained in the small groups database, return a small group process which will iterate though all groups with order in the sequence S meeting the search criterion set by the parameter `Search`.

`SmallGroupProcess(o, f: parameters)`

`Search` MonStgElt *Default :* "*All*"

Given an order o contained in the small groups database and a predicate f (as function or intrinsic), return a small group process which will iterate though all groups of order o, which meet the search criterion set by the parameter `Search` and satisfy the predicate f.

`SmallGroupProcess(S, f: parameters)`

`Search` MonStgElt *Default :* "*All*"

Given a sequence S of orders contained in the small groups database and a predicate f (as function or intrinsic), return a small group process which will iterate though all groups with order in the sequence S, which meet the search criterion set by the parameter `Search` and satisfy the predicate f.

`IsEmpty(p)`

Returns `true` if the process p has passed its last group.

`Current(p)`

Return the current group of the process p.

`CurrentLabel(p)`

Return the label of the current group of the process p. That is, return o and n such that the current group is `SmallGroup(o, n)`.

`Advance(~p)`

Move the process p to its next group.

Example H66E2

We use a small group process to look at all the groups of order 128. We find the nilpotency class of each of them.

```
> P := SmallGroupProcess(128);
> count := {* *};
> repeat
>   G := Current(P);
>   Include(~count, NilpotencyClass(G));
>   Advance(~P);
> until IsEmpty(P);
> count;
{* 1^^15, 2^^947, 3^^1137, 4^^197, 5^^29, 6^^3 *}
```

66.2.3 Small Group Identification

The following functions perform the inverse operation to the small group functions described earlier. Given a group G such that a group isomorphic to G is in the database and identification of groups of order $|G|$ is supported, the identification functions return a pair $\langle o, n \rangle$ so that `SmallGroup(o, n)` is isomorphic to G.

Note that identifying a finitely presented group involves the construction of a permutation representation of this group, which may fail. We refer to the description of `IdentifyGroup` in Chapter 70 for details.

`IdentifyGroup(G)`

Locate the pair of integers $\langle o, n \rangle$ so that `SmallGroup(o, n)` is isomorphic to G. If there is no such group in the database or if identification of groups of order $|G|$ is not supported, then an error will result. The function `CanIdentifyGroup` can be used to test whether groups of a certain order can be identified; this may be useful for avoiding runtime errors in user written loops or functions.

`CanIdentifyGroup(o)`

Return `true` if identification of groups of order o in the database is supported. This function can be used to check whether a group is a legitimate argument for the functions `IdentifyGroup` described above, avoiding runtime errors in user written loops or functions.

Example H66E3

We identify a permutation group in the small group database, and get an isomorphic group from the database.

```
> G := DihedralGroup(10);
> G;
Permutation group G acting on a set of cardinality 10
Order = 20 = 2^2 * 5
  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  (1, 10)(2, 9)(3, 8)(4, 7)(5, 6)
> IdentifyGroup(G);
<20, 4>
> H := SmallGroup(20, 4);
> H;
GrpPC : H of order 20 = 2^2 * 5
PC-Relations:
    H.1^2 = Id(H),
    H.2^2 = Id(H),
    H.3^5 = Id(H),
    H.3^H.1 = H.3^4
```

66.2.4 Accessing Internal Data

The following functions provide access to data used internally by the Small Groups Library for representing groups. They are included just for completeness and are intended to be used by experts only. In particular, we do not give a detailed explanation of the (complicated) data format in this manual.

`Data(D, o, n)`

Returns the data from which the group number n of order o in D is constructed. The format and the meaning of the items in the returned list depends on the group indicated by the pair (o, n).

`SmallGroupEncoding(G)`

Given a finite solvable group G in the category GrpPC, return two integers c and o encoding the power conjugate presentation of G.

The second return value is the order of G. The first return value is an integer specifying the power conjugate relations in the presentation of G.

`SmallGroupDecoding(c, o)`

Given two integers c and o encoding a power conjugate presentation G, return G as a group in the category GrpPC.

The second argument is the order of G. The first return value is an integer specifying the power conjugate relations in the presentation G.

Example H66E4

(1) We extract a power conjugate presentation from the database and compute its encoding.

```
> D := SmallGroupDatabase();
> G := SmallGroup(D,1053,51);
> Category(G);
GrpPC
>  SmallGroupEncoding(G);
1002867124873971659396781 73 1053
```

(2) The second group of order 525 in the small groups database is stored as encoded power conjugate presentation.

```
> Data(D,525,2);
[* code, 666501 *]
```

We can create the corresponding group by decoding this information.

```
> G := SmallGroupDecoding(666501, 525);
> G;
GrpPC : G of order 525 = 3 * 5^2 * 7
PC-Relations:
    G.1^3 = Id(G),
    G.2^5 = G.4,
    G.3^7 = Id(G),
    G.4^5 = Id(G)
```

This gives the same presentation as accessing the database in the "usual" way.

```
> SmallGroup(D,525,2);
GrpPC of order 525 = 3 * 5^2 * 7
PC-Relations:
    $.1^3 = Id($),
    $.2^5 = $.4,
    $.3^7 = Id($),
    $.4^5 = Id($)
```

66.3 The p-groups of Order Dividing p^7

Magma contains the means to construct all p-groups of order p^n where $n \leq 7$. This section describes the functions for accessing these constructions. The data used in the constructions was supplied by Hans Ulrich Besche, Bettina Eick, Eamonn O'Brien, Mike Newman and Michael Vaughan-Lee [BE99a, BEO01, BE99b, O'B90, BE01, O'B91, MNVL04, OVL05].

`SearchPGroups(p, n: parameters)`

Produce a sequence of groups of order p^n satisfying the conditions specified by the following parameters. The restrictions on the order are $n \leq 7$ or $p = 2$ and $n \leq 9$.

`Rank` SetEnum *Default :* $\{1, \ldots n\}$

All groups returned will have Frattini quotient rank in `Rank`. This parameter may also be set to a single integer.

`Class` SetEnum *Default :* $\{1, \ldots n\}$

All groups returned will have p-class in `Class`. This parameter may also be set to a single integer.

`Select` Program *Default :* `true`

The parameter must be set to a program returning either `true` or `false` when given a p-group satisfying the above conditions. All groups G returned will then satisfy `Select(G) eq true`.

`Limit` RngIntElt *Default :* 0

If `Limit` is set to a positive number n, then the program may end its search and return when there are at least n groups found.

`CountPGroups(p, n: parameters)`

Count the number of groups of order p^n satisfying the conditions specified by the parameters. The parameters are the same as for `SearchPGroups`, except that the `Limit` parameter is ignored.

Example H66E5

We search the groups of order 19^7 for specific examples. There are, in total, 9380741 groups with this order. We start with a search for those of rank 5, class 3, and exponent 19. Since we do not set the `Limit` parameter, we will get a sequence containing all the examples.

```
> time Q := SearchPGroups(19, 7:Rank := 5, Class := 3,
>    Select := func<G|IsPrime(Exponent(G))> );
Time: 0.050
> #Q;
4
> Q[1];
GrpPC of order 893871739 = 19^7
PC-Relations:
  $.2^$.1 = $.2 * $.6,
```

```
  $.6^$.1 = $.6 * $.7
```

This time we limit the number returned.

```
> time    Q := SearchPGroups(19, 7:Rank := 4, Class := {3,4},
>    Select := func<G|IsPrime(Exponent(G))>, Limit := 5);
Time: 13.090
> #Q;
5
> [pClass(G):G in Q];
[ 3, 3, 3, 3, 3 ]
> time    Q4 := SearchPGroups(19, 7:Rank := 4, Class := 4,
>   Select := func<G|IsPrime(Exponent(G))>, Limit := 5);
Time: 0.150
> #Q4;
6
```

Note that the limit is not always adhered to exactly. We can also count the number of groups with our property.

```
> time CountPGroups(19, 7:Rank := 4, Class := {3,4},
>   Select := func<G|IsPrime(Exponent(G))>);
Time: 334.720
43
> time CountPGroups(19, 7:Rank := 4, Class := 4,
>    Select := func<G|IsPrime(Exponent(G))>);
10
Time: 0.310
```

66.4 Metacyclic p-groups

Magma contains functions for constructing all metacyclic groups of order p^n. It can also decide if a given p-group is metacyclic, construct invariants which distinguish this metacyclic group from all others of this order, and construct a standard presentation for the group.

This section describes the functions for accessing these algorithms. The functions were developed by Mike Newman, Eamonn O'Brien, and Michael Vaughan-Lee.

`MetacyclicPGroups(p, n: parameters)`

Return a list of the metacyclic groups of order p^m, where p is a prime and n is a positive integer.

`PCGroups` BoolElt *Default :* `true`

If `true`, the groups returned are in category GrpPC, otherwise they are in category GrpFP – this will be faster if the groups have large class.

`IsMetacyclicPGroup (P)`

P is a p-group, either pc- or matrix or permutation group; if P is metacyclic, then return `true`, else `false`.

`InvariantsMetacyclicPGroup (P)`

P is a metacyclic p-group, either pc- or matrix or permutation group; return tuple of invariants which uniquely identify metacyclic p-group P. This tuple which contains at least four terms, $\langle r, s, t, n\rangle$ has the following meaning: P has order p^{n+s}; its derived quotient is $C_{p^r} \times C_{p^s}$; its derived group is cyclic of order p^{n-r}; it has exponent p^{n+s-t}.

If $p = 2$, then additional invariants are needed to distinguish among the groups. We record the abelian invariants of the centre of P. If $s = 1$ and the centre of P has order 2, then the 2-group is maximal class and we record whether it is dihedral, quaternion or semidihedral. If $s > 1$ then the group has two cyclic central normal subgroups of order 2^{s-1} whose central quotients are both semidihedral, or dihedral and quaternion. The invariant tuple has length at most 6.

`StandardMetacyclicPGroup (P)`

P is a metacyclic p-group, either pc- or matrix or permutation group; return metacyclic p-group having a canonical pc-presentation which is isomorphic to P. If two metacyclic p-groups have the same canonical presentation, then they are isomorphic.

`NumberOfMetacyclicPGroups (p, n)`

Return number of metacyclic groups of order p^n.

`HasAllPQuotientsMetacyclic (G)`

`HasAllPQuotientsMetacyclic (G, p)`

Return `true` if for all primes p all p-quotients of the finitely-presented group G are metacyclic; otherwise return `false` and a description of the set of primes for which G has non-metacyclic p-quotient.

If a prime p is supplied as a second argument, then the function returns `true` if all p-quotients of G are metacyclic; otherwise it returns `false`.

Example H66E6

```
> X := MetacyclicPGroups (3, 6);
> #X;
11
> X[4];
GrpPC of order 729 = 3^6
PC-Relations:
    $.1^3 = $.3,
    $.2^3 = $.4,
    $.3^3 = $.6,
```

```
    $.4^3 = $.5,
    $.5^3 = $.6,
    $.2^$.1 = $.2 * $.6^2
> H := SmallGroup (729, 59);
> IsMetacyclicPGroup (H);
true
> I := InvariantsMetacyclicPGroup(H);
> I;
<2, 2, 2, 4, [], , >
> S := StandardMetacyclicPGroup (H);
GrpPC : S of order 729 = 3^6
PC-Relations:
    S.1^3 = S.3,
    S.2^3 = S.4,
    S.3^3 = S.6,
    S.4^3 = S.5,
    S.5^3 = S.6,
    S.2^S.1 = S.2 * S.6^2
> /* find this group in list */
> [IsIdenticalPresentation (S, X[i]): i in [1..#X]];
[ false, false, false, true, false, false, false, false, false, false, false ]
> /* so this group is #4 in list */
> NumberOfMetacyclicPGroups (19, 7);
14
> Q := FreeGroup (4);
> G := quo < Q |  Q.2^2, Q.4^3, Q.2 * Q.3 * Q.2 * Q.3^-1, Q.1^9>;
> /* are all p-quotients of G metacyclic? */
> HasAllPQuotientsMetacyclic (G);
false [ 3 ]
> /* the 3-quotient is not metacyclic */
```

66.5 Database of Perfect Groups

MAGMA includes a database of finite perfect groups. This database includes all perfect groups up to order 50000, and many classes of perfect groups up to order one million. Each group is defined by means of a finite presentation. Further information is also provided which allows the construction of permutation representations.

66.5.1 Specifying an Entry of the Database

There are three ways to key a particular entry of the database. Firstly, a single integer i simply denotes the i-th entry of the database. There is no particular ordering in the correspondence.

Secondly, the database stores information to quickly locate perfect groups of a particular order; thus the pair o, i represents the i-th entry of order o.

The third method corresponds to the notation used in Chapter 5.3 of [HP89]. In this book, the expression $Q\#p$ denotes the class of groups that are isomorphic to perfect extensions of p-groups by Q, where p is a prime and Q is a fixed finite perfect group in which the largest normal p-subgroup is assumed to be trivial. Within a class $Q\#p$, an isomorphism type of groups is denoted by an ordered pair of integers $\langle r, n\rangle$, where $r \geq 0$ and $n \geq 0$.

To specify a particular group Q without extension, a (somewhat descriptive) string is given. The set of possible values can be accessed using the function `TopQuotients`. Among these strings, full or partial covering groups of G are named G`C`n where n is the index of G in G`C`n. Also, there are five classes of 3-extensions of groups which are also defined in the database. The names of these base groups are `A5#2<`r, n`>` where (r, n) are $(4, 2), (5, 5), (5, 6), (5, 7)$ or $(6, 7)$. Furthermore, there are some extensions of direct products: these have names of the form G`x`H. The remainder are names of simple groups. The convention with these names is that if they are elements of a family of simple groups with two parameters then the name will be *fam*`(`p_1`, `p_2`)`, while one parameter families will just be the concatenation of the family name and the parameter.

To illustrate the naming conventions, here are some examples: `A5`, `A5C2`, `A5#2<`$5, 5$`>`, `L(2, 59)C2`, `A5xL(2, 11)`. Notice that there is never a space before the `C` denoting a covering group or on either side of the `x` denoting direct product. However, there is always a space after a comma.

To specify a particular group $Q\#p\langle r, n\rangle$ the four values Q, p, r and n should be given. However, it should be noted that in three cases (`A5#2< 5, 1>`, `L(3, 2)#2<3, 1>`, `L(3, 2)#2<3, 2>`), there are *two* versions of `Q#p<`r, n`>` stored in the database. Strictly speaking, then, there is a fifth key value v required in this third method. However, it can be specified by an optional parameter `Variant :=` v (if necessary), and can normally be ignored. The variant forms are isomorphic to the original forms, and are included for compatibility with Holt & Plesken's tables.

66.5.2 Creating the Database

`PerfectGroupDatabase()`

This function returns a database object which contains information about the database. It is required as first argument to the other access functions.

66.5.3 Accessing the Database

`Group(D, i)`
`Group(D, o, i)`
`Group(D, Q)`
`Group(D, Q, p, r, n:` *parameters*`)`

`Variant` RngIntElt *Default :* 1

Returns the specified entry from the database D as a finitely presented group. In addition, it returns a sequence of pairs $\langle [i_1, \ldots, i_n], [H_1, \ldots, H_n] \rangle$, each of which affords an isomorphism onto a permutation group of degree $\sum_{j=1}^{n} i_j$. The subgroup H_j has index i_j in the defined group, and the sum of the permutation representations of the group on the cosets of the H_j's is faithful. For the meanings of the arguments, see Subsection 66.5.1 above.

`IdentificationNumber(D, i)`
`IdentificationNumber(D, o, i)`
`IdentificationNumber(D, Q)`
`IdentificationNumber(D, Q, p, r, n:` *parameters*`)`

`Variant` RngIntElt *Default :* 1

Returns a number which can be used to access the specified entry from the database D using method one. (See Subsection 66.5.1 above).

`NumberOfRepresentations(D, i)`
`NumberOfRepresentations(D, o, i)`
`NumberOfRepresentations(D, Q)`
`NumberOfRepresentations(D, Q, p, r, n:` *parameters*`)`

`Variant` RngIntElt *Default :* 1

Returns the number of ways stored in the database for building a permutation group representation of the specified entry. (See Subsection 66.5.1 above).

`PermutationRepresentation(D, i:` *parameters*`)`
`PermutationRepresentation(D, o, i:` *parameters*`)`
`PermutationRepresentation(D, Q:` *parameters*`)`
`PermutationRepresentation(D, Q, p, r, n:` *parameters*`)`

`Variant` RngIntElt *Default :* 1

Returns the isomorphism from the finitely presented group G specified to a permutation group representation H as well as the groups G and H. (See Subsection 66.5.1 above).

`Representation` RngIntElt *Default :* 1

Selects which of the stored methods of constructing the permutation representation should be used.

`PermutationGroup(D, i: parameters)`

`PermutationGroup(D, o, i: parameters)`

`PermutationGroup(D, Q: parameters)`

`PermutationGroup(D, Q, p, r, n: parameters)`

`Variant` RngIntElt *Default :* 1

Returns the specified entry from the database D as a permutation group. (See Subsection 66.5.1 above).

`Representation` RngIntElt *Default :* 1

Selects which of the stored methods of constructing the permutation representation should be used.

66.5.4 Finding Legal Keys

`#D`

`NumberOfGroups(D)`

Returns the number of entries stored in the database. (See Subsection 66.5.1, method 1, above).

`NumberOfGroups(D, o)`

Returns the number of entries stored in the database of order o. (See Subsection 66.5.1, method 2, above).

`TopQuotients(D)`

Returns the set of strings denoting the fixed perfect groups Q. (See Subsection 66.5.1, method 3, above).

`ExtensionPrimes(D, Q)`

Returns the set of primes p for which a non-trivial p-extension of the group denoted by Q lies in the database. (See Subsection 66.5.1, method 3, above).

`ExtensionExponents(D, Q, p)`

Returns the set of exponents r such that a non-trivial extension of the group denoted by Q by p^r lies in the database. (See Subsection 66.5.1, method 3, above).

`ExtensionNumbers(D, Q, p, r)`

Returns the set of numbers n such that there is a group $Q\#p\langle r, n\rangle$ in the database. (See Subsection 66.5.1, method 3, above).

`ExtensionClasses(D, Q)`

Returns the set of triples $\langle p, r, n\rangle$ such that there is a group $Q\#p\langle r, n\rangle$ in the database. (See Subsection 66.5.1, method 3, above).

Example H66E7

We hunt through the various levels of key-finding functions available to find an extension of L(3,4) in the database.

```
> DB := PerfectGroupDatabase();
> "L(3, 4)" in TopQuotients(DB);
true
> ExtensionPrimes(DB, "L(3, 4)");
{ 2 }
> ExtensionExponents(DB, "L(3, 4)", 2);
{ 1, 2, 3, 4 }
> ExtensionNumbers(DB, "L(3, 4)", 2, 2);
{ 1, 2, 3 }
```

The database contains extensions of L(3,4) by groups of order 2^1, 2^2, 2^3 and 2^4. We will look at one of the 3 extensions by a group of order 4.

```
> G := Group(DB, "L(3, 4)", 2, 2, 3);
> G;
Finitely presented group G on 3 generators
Relations
  a^2 = Id(G)
  b^4 * e^-2 = Id(G)
  a * b * a * b * a * b * a * b * a * b * a * b * a * b * e
  = Id(G)
  a * b^2 * a * b^2 * a * b^2 * a * b^2 * a * b^2 * e^-1 =
  Id(G)
  a^-1 * b^-1 * a * b * a^-1 * b^-1 * a * b * a^-1 * b^-1 *
  a * b * a^-1 * b^-1 * a * b * a^-1 * b^-1 * a * b * e^-2 =
  Id(G)
  a * b * a * b * a * b^3 * a * b * a * b * a * b^3 * a * b
  * a * b * a * b^3 * a * b * a * b * a * b^3 * a * b * a *
  b * a * b^3 * e^-2 = Id(G)
  (a * b * a * b * a * b^2 * a * b^-1)^5 = Id(G)
  (a, e^-1) = Id(G)
  (b, e^-1) = Id(G)
> P := PermutationGroup(DB, "L(3, 4)", 2, 2, 3);
> P;
Permutation group P acting on a set of cardinality 224
Order = 80640 = 2^8 * 3^2 * 5 * 7
> ChiefFactors(P);
    G
    |  A(2, 4)                = L(3, 4)
    *
```

```
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
> #Radical(P);
4
> IsCyclic(Radical(P));
true
> IsCentral(P, Radical(P));
true
```

66.6 Database of Almost-Simple Groups

MAGMA includes a database containing information about almost simple groups G, where $S \leq G \leq \mathrm{Aut}(S)$ and S is a simple group of small order. The G that are included in the database are those associated with S such that $|S|$ is less than 16000000, as well as M_{24}, HS, J_3, McL, $Sz(32)$ and $L_6(2)$.

The information stored here is primarily for use in computing maximal subgroups and automorphism groups. The database was originally conceived by Derek Holt, with a major extension by Volker Gebhardt and sporadic additions by Bill Unger. The implementation is by Bruce Cox.

It is possible to request the i-th entry of the database. Alternatively, and more usefully, one can supply three integers: the order $o1$ of S, the order $o2$ of G and the sum k of the orders of the class representatives of G. The last of these can be expensive to compute; however, knowledge of the classes is helpful to benefit from the information stored in the entry. Of course, if the entry is beyond the range of the database, then it is a wasted computation—the intrinsics `ExistsGroupData` and `NumberOfGroups` are provided to determine from the orders whether this is the case.

66.6.1 The Record Fields

The result returned by the `GroupData` function is a record with a number of fields containing information about the almost simple group and its socle. This information includes information used to compute automorphisms and maximal subgroups of the almost simple group by these MAGMA functions. The following describes these fields. The groups G and S are as above the almost simple group G and its socle, the simple group S. Let A be the full automorphism group of S, and let $F\langle x, y\rangle$ be a free group on two generators, called x and y.

First comes information about S. Each S is two generated, by x and y as above, say.

Field `resname`: A string giving a name to the simple group S. (S is the soluble residual of G).

Field `resorder`: The order of S as an integer.

Field `geninfo`: Information on where to find x and y in S. This is a sequence of two tuples, each with 3 entries. The first gives a generator order, the second the length of its conjugacy class, and the third the probability of picking the right generator given the previous information. The first tuple's order/length information always uniquely defines one conjugacy class of the group S and has probability 1, so x is easy to find.

Field `rels`: A sequence of words in F which, taken together with the generator orders from `geninfo`, form a presentation for S on x and y.

Field `permrep`: A permutation representation of the full automorphism group of S. The first two generators are x and y, followed by outer generators. In what follows the outer generators are called t, u, v.

Field `outimages` A sequence of sequences of words in F. These give the images of x and y under the generators of the outer automorphism group of S.

Field `order`: An integer, the order of G.

Field `inv`: The invariant used to separate non-isomorphic $\langle |S|, |G| \rangle$ possibilities. An integer, it is the sum over the classes of G of the order of the elements in each class.

Field `name`: A name for G as a string.

Field `conjelts`: If G is not normal in the automorphism group, these words are coset representatives of the normaliser of G in A as words in t, u, v.

Field `subgens`: Words in t, u, v that, together with x and y, generate G.

Field `subpres`: A presentation of G/S on `subgens`. Note: If $G = S$ then `subgens` will be the empty sequence, but `subpres` will be the trivial FP-group with one generator.

Field `normgens`: Words in t, u, v that generate the outer automorphism group of G.

Field `normpres`: A presentation of the outer automorphism group of G on `normgens`. Again, if $A = G$ then `normgens` will be the empty sequence, but `normpres` will be the trivial FP-group with one generator.

Field `maxsubints`: A sequence of records describing the intersections of the maximal subgroups of G that do not contain S with S. Each record gives the order of the intersection, its class length in S, generators as words in x and y, and a presentation on these generators.

66.6.2 Creating the Database

`AlmostSimpleGroupDatabase()`

This function returns a database object which contains information about the database.

66.6.3 Accessing the Database

`#D`

Returns the number of entries stored in the database.

`GroupData(D, i)`

`GroupData(D, o1, o2, k)`

Returns the specified entry from the database D as a record. The first form gives the ith entry of the database. The second form gives information on an almost simple group of order $o2$, with socle of order $o1$. The value of k should be as explained in `inv` above.

`ExistsGroupData(D, o1, o2)`

`ExistsGroupData(D, o1, o2, i)`

Returns whether any record exists for a simple group of order $o1$ and a supergroup G of order $o2$ lying within its automorphism group. In the second form an invariant, i as described above, is also supplied, and the result is true if there is a record in the database with the given orders and invariant, and false otherwise. When the result is true, the corresponding record is also returned.

`NumberOfGroups(D, o1, o2)`

Returns the number of records in the database D corresponding to a simple group of order $o1$ and a supergroup G of order $o2$ lying within its automorphism group. The second return value gives the index of the first such record, if there is one. (This is most useful when the first return value is 1.) If the return values are d and f, with $d > 0$, then the corresponding database entries are numbered $f, f+1, \ldots, f+d-1$.

`IdentifyAlmostSimpleGroup(G)`

`IdentifyAlmostSimpleGroup(G)`

Use the information in the database to construct a monomorphism f from the almost simple group G into A, the permutation representation of the full automorphism group of its socle stored in the database. This function will also cope with groups isomorphic to the alternating and symmetric groups of degree up to 50, which are not actually in the database. Note that the conjugacy class of the image of f in A determines G up to isomorphism. The algorithms used to deal with the alternating and symmetric groups not in the database are by Derek Holt, starting from the paper of Bratus & Pak [BP00].

Example H66E8

We query the database on about an almost simple group of order 720.

```
> G := PermutationGroup<10 |
>  [ 7, 9, 5, 2, 1, 8, 10, 4, 6, 3 ],
>  [ 6, 3, 10, 7, 2, 4, 1, 8, 9, 5 ]>;
> #G;
720
> CompositionFactors(G);
    G
    |  Cyclic(2)
    *
    |  Alternating(6)
    1
> #Radical(G);
1
```

There are 3 such groups, so we really do need k to tell them apart.

```
> S := SolubleResidual(G);
> k := &+[c[1]: c in Classes(G)];
> D := AlmostSimpleGroupDatabase();
> R := GroupData(D, #S, #G, k);
> R`name;
M_10
```

The group is identified. Let's see some of the other information in R.

```
> P := R`permrep;
> P;
Permutation group P acting on a set of cardinality 10
  (1, 6)(2, 9)(3, 10)(4, 8)
  (1, 7, 9, 4)(2, 8, 5, 10)
  (3, 5, 9, 6, 7, 4, 8, 10)
  (3, 7)(4, 6)(5, 10)
> #P;
1440
> R`subgens;
[ t * u ]
> SS := sub<P|P.1, P.2>;
> #SS;
360
> GG := sub<P|SS, P.3*P.4>;
> #GG;
720
>R`normgens;
[ t ]
```

The full automorphism group of S has order 1440. The group P is a representation of this automorphism group with first two generators generating a faithful image of S. The image of S,

together with the product of P.3 and P.4, generate a faithful image of G. The outer automorphism group of G is generated by P.3 modulo GG. We can also get a constructive identification as follows.

```
> f, A := IdentifyAlmostSimpleGroup(G);
> f;
Homomorphism of GrpPerm: $, Degree 10, Order 2^4 * 3^2 * 5
into GrpPerm: A, Degree 10 induced by
  (1, 10)(3, 8)(5, 9)(6, 7) |--> (1, 6)(2, 9)(3, 10)(4, 8)
  (1, 7, 2, 8)(3, 10, 4, 5) |--> (1, 7, 9, 4)(2, 8, 5, 10)
  (1, 10, 4, 8)(2, 6, 9, 5) |--> (3, 10, 7, 6)(4, 8, 5, 9)
> A;
Permutation group A acting on a set of cardinality 10
  (1, 6)(2, 9)(3, 10)(4, 8)
  (1, 7, 9, 4)(2, 8, 5, 10)
  (3, 5, 9, 6, 7, 4, 8, 10)
  (3, 7)(4, 6)(5, 10)
```

66.7 Database of Transitive Groups

MAGMA has a database containing all transitive permutation groups having degree up to 32, and one containing all primitive permutation groups with degree less than 4096.

The transitive groups up to degree 15 were determined by Greg Butler and John McKay, the groups having degree in the range 16 to 30 were determined by Alexander Hulpke [Hul05]. John Cannon and Derek Holt [CH08] have determined the transitive groups of degree 32.

66.7.1 Accessing the Databases

`TransitiveGroupDatabaseLimit()`

The limiting degree of the database of transitive groups.

`NumberOfTransitiveGroups(d)`

Given a degree d in the required range, return the number of transitive groups of degree d.

`TransitiveGroup(d, n)`

Given a degree d in the required range and a positive integer n, return the n-th transitive group of degree d. Also returns a string giving a description of the group.

`TransitiveGroupDescription(d, n)`

A string giving a description of the n-th transitive group of degree d.

`TransitiveGroupDescription(G)`

A string giving a description of the transitive group G.

`TransitiveGroup(d)`

Given a degree d in the required range, return the first transitive group of degree d. Also returns a string giving a description of the group.

`TransitiveGroup(d, f)`

Given a degree d in the required range and a predicate f (as a function or intrinsic), return the first transitive group of degree d which satisfies f. Also returns a string giving a description of the group.

`TransitiveGroup(S, f)`

Given a sequence S of degrees and a predicate f (as a function or intrinsic), return the first transitive group with degree in S which satisfies f. Also returns a string giving a description of the group.

`TransitiveGroups(d:` *parameters*`)`

`Warning` BoolElt *Default :* `true`

Return a sequence of all transitive groups of degree d. Some degrees will produce a very large sequence of groups – in such cases a warning will be printed unless the user specifies `Warning := false`.

`TransitiveGroups(S:` *parameters*`)`

`Warning` BoolElt *Default :* `true`

Given a sequence S of degrees, return a sequence of all transitive groups with degree in S. The resulting sequence may be very long – in such cases a warning will be printed unless the user specifies `Warning := false`.

`TransitiveGroups(d, f)`

Given an integer d and a predicate (function or intrinsic) f, return a sequence containing all transitive groups G of degree d satisfying `f(G) eq true`.

`TransitiveGroups(S, f)`

Given a sequence S of degrees and a predicate (function or intrinsic) f, return a sequence containing all transitive groups G with degree in S satisfying `f(G) eq true`.

Example H66E9

We apply some of these functions to the degree 8 case.

```
> NumberOfTransitiveGroups(8);
50
> TransitiveGroup(8, 3);
Permutation group acting on a set of cardinality 8
    (1, 2)(3, 4)(5, 6)(7, 8)
    (1, 4)(2, 3)(5, 8)(6, 7)
    (1, 8)(2, 7)(3, 6)(4, 5)
E(8) = 2[x]2[x]2
> S := TransitiveGroups(8, IsPrimitive);
> #S;
7
> S;
[
    Permutation group acting on a set of cardinality 8
        (1, 8)(2, 3)(4, 5)(6, 7)
        (1, 3)(2, 8)(4, 6)(5, 7)
        (1, 5)(2, 6)(3, 7)(4, 8)
        (1, 2, 6, 3, 4, 5, 7),
    Permutation group acting on a set of cardinality 8
        (1, 8)(2, 3)(4, 5)(6, 7)
        (1, 3)(2, 8)(4, 6)(5, 7)
        (1, 5)(2, 6)(3, 7)(4, 8)
        (1, 2, 6, 3, 4, 5, 7)
        (1, 2, 3)(4, 6, 5),
    Permutation group acting on a set of cardinality 8
        (1, 2, 3, 4, 5, 6, 8)
        (1, 2, 4)(3, 6, 5)
        (1, 6)(2, 3)(4, 5)(7, 8),
    Permutation group acting on a set of cardinality 8
        (1, 2, 3, 4, 5, 6, 8)
        (1, 3, 2, 6, 4, 5)
        (1, 6)(2, 3)(4, 5)(7, 8),
    Permutation group acting on a set of cardinality 8
        (1, 8)(2, 3)(4, 5)(6, 7)
        (1, 3)(2, 8)(4, 6)(5, 7)
        (1, 5)(2, 6)(3, 7)(4, 8)
        (1, 2, 6, 3, 4, 5, 7)
        (1, 2, 3)(4, 6, 5)
        (1, 2)(5, 6),
    Permutation group acting on a set of cardinality 8
        (1, 2)(3, 4, 5, 6, 7, 8)
        (1, 2, 3),
    Permutation group acting on a set of cardinality 8
        (1, 2, 3, 4, 5, 6, 7, 8)
        (1, 2)
```

```
]
```

66.7.2 Processes

A transitive group process enables iteration over all transitive groups of specified degrees satisfying a given predicate, without having to create and store all such groups together.

The intrinsic function `TransitiveGroupProcess` may be used to create a transitive group process in MAGMA. The standard process functions `IsEmpty`, `Current`, `CurrentLabel` and `Advance` can then be applied to the process.

`TransitiveGroupProcess(d)`

Return a group process which will iterate though all transitive groups of degree d.

`TransitiveGroupProcess(S)`

Return a process which will iterate though all transitive groups with degree in the sequence S.

`TransitiveGroupProcess(d, f)`

Return a process which will iterate though all transitive groups with degree d which satisfy the predicate f.

`TransitiveGroupProcess(S, f)`

Return a process which will iterate though all transitive groups with degree in the sequence S which satisfy the predicate f.

`IsEmpty(p)`

Returns `true` if the process p has passed its last group.

`Current(p)`

Return the current group of the process p, as well as a description of the group.

`CurrentLabel(p)`

Return the label of the current group of the process p. That is, return d and n such that the current group is `TransitiveGroup(d, n)`.

`Advance(~p)`

Move the process p to its next group.

Example H66E10

The use of processes is illustrated by the following code, in which the orders of all transitive groups of degree 5 are listed.

```
> p := TransitiveGroupProcess(5);
> while not IsEmpty(p) do
>     CurrentLabel(p), #Current(p);
>     Advance(~p);
> end while;
5 1 5
5 2 10
5 3 20
5 4 60
5 5 120
```

66.7.3 Transitive Group Identification

Given a transitive group G whose degree is at most 30, it is possible to obtain the number of the group in the transitive groups database which is isomorphic to G.

`TransitiveGroupIdentification(G)`

`Raw` BoolElt *Default :* `true`

The number (and degree) of the group in the transitive groups database which is isomorphic to the transitive group G.

If the optional parameter `Raw` is set to `false`, a third value is returned. In this case, the third value is a permutation conjugating the given group to the copy in the library.

Example H66E11

We get a transitive permutation group from the small groups database and identify it as a transitive group.

```
> G := SmallGroup(336, IsTransitive: Search:="Insoluble");
> G;
Permutation group G acting on a set of cardinality 16
  (1, 14, 6, 2, 12, 8, 13, 7)(3, 15, 10, 5, 16, 9, 4, 11)
  (2, 5, 6)(3, 10, 9)(4, 15, 16)(7, 11, 13)
> TransitiveGroupIdentification(G : Raw := false);
715 16 (1, 16, 3, 4, 2, 11, 5, 6, 9, 8, 13, 10)(7, 12, 15)
> n, d, p := $1;
> G^p eq TransitiveGroup(d, n);
true
```

We found it to be group 715 of degree 16.

66.8 Database of Primitive Groups

MAGMA has a database containing all primitive permutation groups with degree less than 2500.

The list of primitive groups up to degree 50 was prepared by C. C. Sims (see [Sim70] for the early part of the list). The list up to degree 999 was determined by Roney-Dougal and Unger. See [RDU03] for details of the methods used. The list was extended to degree 2499 by Roney-Dougal, as described in [RD05], and was further extended to degree 4095 by Coutts, Quick and Roney-Dougal [CQRD11].

Within the database the groups are stored by degree. Within each degree they are stored by O'Nan-Scott class in the order soluble affine, insoluble affine, diagonal action, product action, almost simple. Within each class groups are ordered by increasing size. (It follows that the alternating and symmetric groups come last at each degree.)

The basic access function takes two parameters, degree and number, and returns the corresponding primitive group. Functions with name prefixed by `NumberOfPrimitive` tell how many groups of each class there are stored. We recommend the use of the `PrimitiveGroupProcess` or `PrimitiveGroups` functions, with appropriate `Filter` value, to access all primitive groups in a specific class.

66.8.1 Accessing the Databases

`PrimitiveGroupDatabaseLimit()`

The limiting degree of the database of primitive groups.

`NumberOfPrimitiveGroups(d)`

`NumberOfPrimitiveSolubleGroups(d)`

`NumberOfPrimitiveAffineGroups(d)`

`NumberOfPrimitiveDiagonalGroups(d)`

`NumberOfPrimitiveProductGroups(d)`

`NumberOfPrimitiveAlmostSimpleGroups(d)`

Given a degree d in the required range, `NumberOfPrimitiveGroups` returns the number of primitive groups of degree d. The other functions return the number of groups of each class at that degree.

`PrimitiveGroup(d, n)`

Given a degree d in the required range and a positive integer n, return the n-th primitive group of degree d. Also returns a string (possibly empty) giving a description of the group and a string giving the group's O'Nan-Scott type.

`PrimitiveGroupDescription(d, n)`

A string giving a description of the n-th primitive group of degree d.

`PrimitiveGroup(d)`

Given a degree d in the required range, return the first primitive group of degree d. Also returns a string giving a description of the group and a string giving the group's O'Nan-Scott type.

`PrimitiveGroup(d, f)`

Given a degree d in the required range and a predicate f (as a function or intrinsic), return the first transitive (primitive) group of degree d which satisfies f.

`PrimitiveGroup(S, f)`

Given a sequence S of degrees and a predicate f (as a function or intrinsic), return the first transitive (primitive) group with degree in S which satisfies f.

`PrimitiveGroups(d:` *parameters*`)`

`Filter` MonStgElt *Default :* "*All*"

Return a sequence of all primitive groups of degree d, modified by the value assigned to `Filter`. The possible values for the parameter are the strings `All`, `Soluble`, `Affine`, `Diagonal`, `Product`, `AlmostSimple`, `Simple` and `SimpleNA`. Generally these values restrict the list to groups in the appropriate O'Nan-Scott type, with the exceptions being `All` giving no restriction, `Simple` restricting to a lsit of all simple groups in the database, and `SimpleNA` being as for `Simple` but omitting all alternating groups in their natural representations.

`PrimitiveGroups(S:` *parameters*`)`

`PrimitiveGroups( :` *parameters*`)`

`Filter` MonStgElt *Default :* "*All*"

Given a sequence S of degrees, return a sequence of all primitive groups with degree in S. The result is modified by `Filter` with values as above. Omitting the sequence of degrees gives the same result as specifying all legal degrees.

`PrimitiveGroups(d, f:` *parameters*`)`

`PrimitiveGroups(S, f)`

`PrimitiveGroups(f)`

`Filter` MonStgElt *Default :* "*All*"

Given an integer d and a predicate (function or intrinsic) f, return a sequence containing all primitive groups G of degree d passing the filter satisfying `f(G) eq true`. Note that the filter will be generally much quicker in rejecting candidates than the predicate will be, and only groups passing the filter have `f(G)` evaluated.

Instead of giving a single degree, a sequence of degrees may be given. Omitting the degree is the same as specifying the sequence of all legal degrees.

Example H66E12

We apply some of these functions to the degree 625 case.

```
> NumberOfPrimitiveGroups(625);
698
> NumberOfPrimitiveAffineGroups(625);
647
> NumberOfPrimitiveSolubleGroups(625);
509
> NumberOfPrimitiveDiagonalGroups(625);
0
> NumberOfPrimitiveProductGroups(625);
49
> NumberOfPrimitiveAlmostSimpleGroups(625);
2
> PrimitiveGroup(625, 511);
Permutation group acting on a set of cardinality 625
Order = 150000 = 2^4 * 3 * 5^5
5^4:SL(2, 5).2 Affine
> PrimitiveGroup(625,690);
Permutation group acting on a set of cardinality 625
Order = 2^14 * 3^5 * 5^4
Alt(5)^4:Q_8:Sym(4) ProductAction
> Q := PrimitiveGroups(625, func<G|#G eq 3*10^4>
>    : Filter := "Affine");
> #Q;
26
```

66.8.2 Processes

A primitive group process enables iteration over all primitive groups of specified degrees satisfying a given predicate, without having to create and store a list of all such groups.

The intrinsic function `PrimitiveGroupProcess` may be used to create a primitive group process. The standard process functions `IsEmpty`, `Current`, `CurrentLabel` and `Advance` can then be applied to the process.

`PrimitiveGroupProcess(d:` *parameters*`)`

`PrimitiveGroupProcess(S:` *parameters*`)`

`PrimitiveGroupProcess(:` *parameters*`)`

`Filter` MonStgElt *Default :* "*All*"

Return a group process which will iterate though all primitive groups of degree d that pass the filter as described above. A sequence of degrees may be given instead of a single degree. In this case the process will iterate though the groups of all the degrees in S. Omitting any degree information is the same as specifying the sequence of all legal degrees.

`PrimitiveGroupProcess(d, f: parameters)`

`PrimitiveGroupProcess(S, f: parameters)`

`PrimitiveGroupProcess(f: parameters)`

`Filter` MonStgElt *Default : "All"*

Return a process which will iterate though all primitive groups with degree d which pass the filter and satisfy the predicate f. A sequence of degrees may be given instead of a single degree. In this case the process will iterate though the groups of all the degrees in S. Omitting any degree information is the same as specifying the sequence of all legal degrees.

`IsEmpty(p)`

Returns `true` if the process p has passed its last group.

`Current(p)`

Return the current group of the process p, as well as a description of the group.

`CurrentLabel(p)`

Return the label of the current group of the process p. That is, return d and n such that the current group is `TransitiveGroup(d, n)` (or `PrimitiveGroup(d, n)`).

`Advance(~p)`

Move the process p to its next group.

Example H66E13

The use of processes is illustrated by the following code, in which the orders of all primitive groups with degree 60 of diagonal type are listed. We also compute the orbit structures of their Sylow 2-subgroups, which demonstrates that they are non-conjugate.

```
> p := PrimitiveGroupProcess(60:Filter:="Diagonal");
> while not IsEmpty(p) do
>     G := Current(p);
>     CurrentLabel(p), #G,
>     [t[1]:t in OrbitRepresentatives(Sylow(G,2))];
>     Advance(~p);
> end while;
60 1 3600 [ 4, 4, 4, 16, 16, 16 ]
60 2 7200 [ 4, 4, 4, 16, 32 ]
60 3 7200 [ 4, 8, 16, 32 ]
60 4 7200 [ 4, 8, 16, 16, 16 ]
60 5 14400 [ 4, 8, 16, 32 ]
```

66.8.3 Primitive Group Identification

Given a primitive group G whose degree is at most 2499, it is possible to obtain the number of the group in the primitive groups database which is permutation isomorphic to G.

`PrimitiveGroupIdentification(G)`

The number (and degree) of the group in the primitive groups database which is permutation isomorphic to the primitive group G.

Example H66E14

We construct a permutation group of affine type and identify it as a primitive group.

```
> M := WreathProduct(SL(2,5), Sym(2));
> Q := Getvecs(M);
> G := Semidir(M, Q);
> G;
Permutation group G acting on a set of cardinality 625
> PrimitiveGroupIdentification(G);
595 625
```

We found it to be group 595 of degree 625.

66.9 Database of Rational Maximal Finite Matrix Groups

MAGMA includes a database of rational maximal finite matrix groups and their invariant forms, for small dimensions (up to 31 at V2.8 and above). This section defines the interface to that database. See the articles of Nebe & Plesken [NP95] and Nebe [Neb96].

A particular entry of the database can be specified in one of two ways. Firstly, a number in the range 1 to the size of the database can be given. Alternatively, the desired dimension can be provided, together with a number in the range 1 to the number of entries of that dimension.

Each entry can be accessed either as a matrix group or as a lattice. If accessed as a matrix group, the order and base are set on return. If as a lattice, the automorphism group is set.

`RationalMatrixGroupDatabase()`

This function returns a database object which contains information about the database.

`LargestDimension(D)`

Returns the largest dimension of any entry stored in the database. It is an error to refer to larger dimensions in the database.

`#D`

`NumberOfGroups(D)`

`NumberOfLattices(D)`

Returns the number of entries stored in the database.

`NumberOfGroups(D, d)`

`NumberOfLattices(D, d)`

Returns the number of entries stored in the database of dimension d.

`Group(D, i)`

Returns the i-th entry from the database D as a matrix group.

`Lattice(D, i)`

Returns the i-th entry from the database D as a lattice.

`Group(D, d, i)`

Returns the i-th entry of dimension d from the database D as a matrix group.

`Lattice(D, d, i)`

Returns the i-th entry of dimension d from the database D as a lattice.

Example H66E15

```
> D := RationalMatrixGroupDatabase();
> #D;
354
> maxdim := LargestDimension(D);
> maxdim;
31
> &+[ NumberOfGroups(D, d) : d in [ 1 .. maxdim ] ];
354
```

These numbers agree (which is nice). The dimension in that range with the most curves is 24.

```
> S := [ NumberOfGroups(D, d) : d in [ 1 .. maxdim ] ];
> Max(S);
65 24
```

The groups have known order, so it is easy to find the group with smallest order and dimension 24.

```
> time orders := [#Group(D, 24, i): i in [1 .. NumberOfGroups(D, 24)]];
Time: 0.480
> Min(orders);
1872 53
```

66.10 Database of Integral Maximal Finite Matrix Groups

MAGMA includes a database of representatives of the $\mathrm{GL}(n, \mathbf{Z})$-conjugacy classes of irreducible maximal finite subgroups of $\mathrm{GL}(n, \mathbf{Z})$ for $n <= 11$ and $n \in \{13, 17, 19, 23\}$. This section defines the interface to that database.

For $n < 10$ the groups have been described in [PP77, PP80]. The groups of dimension 10 can be found in [Sou94]. In the cases $n > 10$ prime, the representatives have been constructed using the descriptions given in [Ple85].

A particular entry of the database can be specified in one of two ways. Firstly, a number in the range 1 to the size of the database can be given. Alternatively, the desired dimension can be provided, together with a number in the range 1 to the number of entries of that dimension.

Each entry can be accessed either as a matrix group or as a lattice. If accessed as a matrix group, the order and base are set on return. If as a lattice, the automorphism group is set.

`IntegralMatrixGroupDatabase()`

This function returns a database object which contains information about the database.

`LargestDimension(D)`

Returns the largest dimension of any entry stored in the database. It is an error to refer to larger dimensions in the database.

`#D`
`NumberOfGroups(D)`
`NumberOfLattices(D)`

Returns the number of entries stored in the database.

`NumberOfGroups(D, d)`
`NumberOfLattices(D, d)`

Returns the number of entries stored in the database of dimension d.

`Group(D, i)`

Returns the i-th entry from the database D as a matrix group.

`Lattice(D, i)`

Returns a lattice L and sequence of additional forms F fixed by the i-th group in the database D.

`Construction(D, i)`

Returns a string S which describes the construction of the i-th group G in the database D.

If the G-invariant lattice is well known, S equals the name of this lattice. If the Degree d of G is a prime, G usually can be chosen to fix the form $a_0 I_d + a_1(z + z^{-1}) + ... + a_k(z^k + z^{-1})$ with $k = (d-1)/2$ and some $a_i \in \mathbf{Z}$ where z denotes the permutation matrix of some d-cycle in $\mathbf{Z}^{d\times d}$ (see [Ple85]). In this case, S equals $[a_0, a_1, a_2, ...]$. In all other cases, S describes the isomorphism type of G.

The second return value gives the numbers of all groups of degree d in the `Rational Matrix Group Database` which contain a $\mathrm{GL}(d, \mathbf{Q})$-conjugate copy of G.

`Group(D, d, i)`

Returns the i-th entry of dimension d in the database D as a matrix group.

`Lattice(D, d, i)`

Returns a lattice L and sequence of additional forms F fixed by the i-th group of dimension d in the database D.

`Construction(D, d, i)`

Returns a string and integer which describe the construction of the i-th entry of dimension d in the database D.

Example H66E16

```
> D:= IntegralMatrixGroupDatabase();
> #D;
222
> G:= Group(D, 8, 7); Construction(D, 8, 7);
A8* [ 3 ]
```

So G is the automorphism group of the dual of the root lattice A_8 and it is conjugate to a subgroup of the third entry of dimension 8 in the `RationalMatrixgroupDatabase`. We find an explicit embedding T of G into that group.

```
> DQ:= RationalMatrixGroupDatabase();
> H:= Group(DQ, 8, 3); L:= Lattice(DQ, 8, 3);
> F:= PositiveDefiniteForm(G);
> for s in Sublattices(G) do
>   B:= BasisMatrix(s);
>   FF:= B * F * Transpose(B);
>   ok, T:= IsIsometric(LatticeWithGram(FF div GCD(Eltseq(FF))), L);
>   if ok then break; end if;
> end for;
> assert ok;
> T:= Matrix(Rationals(), T*B);
> [Matrix(Integers(), T*Matrix(G.i)*T^-1) in H : i in [1..Ngens(G)]];
[ true, true ]
```

66.11 Database of Finite Quaternionic Matrix Groups

MAGMA includes a database of the finite absolutely irreducible subgroups of $\mathrm{GL}_n(\mathcal{D})$ where $\mathcal{D}$ is a definite quaternion algebra whose centre has degree d over $\mathbf{Q}$ and $nd \leq 10$. This collection is due to Gabriele Nebe [Neb98]. This section defines the interface to that database.

A particular entry of the database can be specified in one of two ways. Firstly, a number in the range 1 to the size of the database can be given. Alternatively, the desired dimension can be provided, together with a number in the range 1 to the number of entries of that dimension.

Each entry can be accessed either as a matrix group or as a lattice. If accessed as a matrix group, the order and base are set on return.

`QuaternionicMatrixGroupDatabase()`

This function returns a database object which contains information about the database.

`LargestDimension(D)`

Returns the largest dimension of any entry stored in the database. It is an error to refer to larger dimensions in the database.

`#D`
`NumberOfGroups(D)`
`NumberOfLattices(D)`

Returns the number of entries stored in the database.

`NumberOfGroups(D, d)`
`NumberOfLattices(D, d)`

Returns the number of entries stored in the database of dimension d.

`Group(D, i)`

Returns the i-th entry from the database D as a matrix group.

`Lattice(D, i)`

Returns a lattice L and sequence of forms F corresponding to the i-th entry of the database D.

`Construction(D, i)`

Returns a string and integer which describe the construction of the i-th entry of the database D.

`Group(D, d, i)`

Returns the i-th entry of dimension d in the database D as a matrix group.

`Lattice(D, d, i)`

Returns a lattice L and sequence of forms F corresponding to the i-th entry of dimension d in the database D.

`Construction(D, d, i)`

Returns a string and integer which describe the construction of the i-th entry of dimension d in the database D.

Example H66E17

We illustrate accessing the quaternionic matrix groups database with a group and lattice of dimension 36.

```
> DB := QuaternionicMatrixGroupDatabase();
> LargestDimension(DB);
40
> NumberOfGroups(DB, 36);
10
> G := Group(DB, 36, 8);
> G : Minimal;
MatrixGroup(36, Integer Ring) of order 43545600 = 2^10 * 3^5
* 5^2 * 7
> #pCore(G, 2);
2
> L, forms := Lattice(DB, 36, 8);
> Determinant(L);
3874204890000
> IsSquare($1);
true 1968300
```

66.12 Database of Finite Symplectic Matrix Groups

MAGMA includes a database of the maximal finite irreducible subgroups of $\mathrm{Sp}_{2n}(\mathbf{Q})$ for $1 \leq i \leq 11$ up to conjugacy in $\mathrm{GL}_{2n}(\mathbf{Q})$. This collection is due to Markus Kirschmer [Kir09]. This section defines the interface to that database.

To avoid non-integral entries, the stored matrix groups do not fix the standard skewsymmetric form but some other nondegenerate skewsymmetric form. The example below illustrates how to construct a conjugate matrix group which fixes the standard skewsymmetric form.

A particular entry of the database can be specified in one of two ways. Firstly, a number in the range 1 to the size of the database can be given. Alternatively, the desired dimension can be provided, together with a number in the range 1 to the number of entries of that dimension.

Each entry can be accessed either as a matrix group or as a lattice with a pair of forms. If accessed as a matrix group, the order and base are set on return.

`SymplecticMatrixGroupDatabase()`

This function returns a database object which contains information about the database.

`LargestDimension(D)`

Returns the largest dimension of any entry stored in the database. It is an error to refer to larger dimensions in the database.

`#D`

`NumberOfGroups(D)`

`NumberOfLattices(D)`

Returns the number of entries stored in the database.

`NumberOfGroups(D, d)`

`NumberOfLattices(D, d)`

Returns the number of entries stored in the database of dimension d.

`Group(D, i)`

Returns the i-th entry from the database D as a matrix group.

`Lattice(D, i)`

Returns a lattice L and a sequence S of two integral forms such that the automorphism group of L with respect to S equals `Group(DB, i)`. The first form in S is the gram matrix of L and the second form is skewsymmetric. The sequence S is normalized as described in the appendix of [Kir09] to simplify the recognition of the matrix group.

`Construction(D, i)`

Returns a string which describes the construction of the i-th entry of the database D.

`Group(D, d, i)`

Returns the i-th entry of dimension d in the database D as a matrix group.

`Lattice(D, d, i)`

Returns a lattice L and a sequence S of forms corresponding to the i-th entry of dimension d in the database D.

`Construction(D, d, i)`

Returns a string which describes the construction of the i-th entry of dimension d in the database D.

Example H66E18

We illustrate accessing the symplectic matrix group database with a group of dimension 16.

```
> DB := SymplecticMatrixGroupDatabase();
> NumberOfGroups(DB, 16);
91
> G := Group(DB, 16, 1);
> G : Minimal;
MatrixGroup(16, Integer Ring) of order 2^21 * 3^4 * 5^2
```

The group G does not fix the standard skewsymmetic form. But it can be conjugated to do so.

```
> _, S := Lattice(DB, 16, 1);
> T := TransformForm(Matrix(Rationals(), S[2]), "symplectic");
> H := ChangeRing(G, Rationals())^(GL(16,Rationals()) ! T);
> J := SymplecticForm(16, Rationals());
> forall{h: h in Generators(H) | h * J * Transpose(h) eq J};
true
```

66.13 Database of Irreducible Matrix Groups

MAGMA has a database containing all irreducible subgroups of $\mathrm{GL}_k(p)$, for p prime, $k \geq 1$ and $p^k < 2500$. One representative of each conjugacy class of subgroups is stored.

The data used is the same as that used to store the affine primitive permutation groups. See the Primitive Groups Database section for the provenance of the data.

Within the database the groups are stored according to p^k. First are the soluble groups, followed by the insoluble. Within each subdivision, the groups are stored by increasing order. (It follows that $\mathrm{GL}_k(p)$ is the last in each list.)

The basic access function takes three parameters, k, p and number, and returns the corresponding group. Functions with name prefixed by `NumberOf` tell how many groups of each class there are stored.

66.13.1 Accessing the Database

`NumberOfIrreducibleMatrixGroups(k, p)`

`NumberOfSolubleIrreducibleMatrixGroups(k, p)`

Given k and p, p prime, $k \geq 1$ and $p^k < 2500$, `NumberOfIrreducibleMatrixGroups` returns the number of subgroups of $\mathrm{GL}_k(p)$ stored. The other function returns the number of soluble subgroups stored.

`IrreducibleMatrixGroup(k, p, n)`

Given k and p p prime, $k \geq 1$ and $p^k < 2500$, and a positive integer n, return the n-th subgroup of $\mathrm{GL}_k(p)$ stored.

Example H66E19

We apply some of these functions to the $\mathrm{GL}_4(5)$ case.

```
> NumberOfIrreducibleMatrixGroups(4, 5);
647
> NumberOfSolubleIrreducibleMatrixGroups(4, 5);
509
> G := IrreducibleMatrixGroup(4, 5, 511);
> ChiefFactors(G);
    G
    |  Cyclic(2)
    *
    |  Alternating(5)
    *
    |  Cyclic(2)
    1
> IsIrreducible(G);
true
> IsAbsolutelyIrreducible(G);
false
```

66.14 Database of Quasisimple Matrix Groups

MAGMA has a database containing characteristic 0 representations of some finite quasisimple groups.

`QuasisimpleMatrixGroup(N, d, p : parameters)`

`OverZ`	BOOLELT	*Default :* `true` $\Leftrightarrow p = 0$
`Automorphisms`	BOOLELT	*Default :* `false`
`RepNo`	RNGINTELT	*Default :* 1

Return an absolutely irreducible matrix group in characteristic p, which may be a prime number or 0, derived from the reduction modulo p of of an absolutely irreducible representation in charateristic 0 and dimension d of the quasisimple group G with name N. The generators of G used are its standard generators. For those quasisimple groups in the ATLAS-database (Section 66.16), the same names are used as there. Other quasisimple groups are named according to the same conventions.

If there is more than one representation of G in dimension d in the database, then the first such is used by default, and the others can be accessed by using the `RepNo` option.

If the reduction modulo p of the representation is not irreducible, then a random non-trivial irreducible constituent is used. (This behaviour may change in the future.)

For those representations that are not realisable over $\mathbf{Z}$ in dimension d, a representation in dimension d over a minimal extension of the rationals and also an irreducible representation in a higher dimension over $\mathbf{Z}$ are both stored in the database. The representation used is the one over $\mathbf{Z}$ if the parameter `OverZ` is `true`, and the one over the number field otherwise. Reduction modulo p is generally faster using the integral representation, so that is the default when $p > 0$.

If the parameter `Automorphisms` is set, then extra generators inducing those outer automorphisms of G that stabilise the representation are included in the group returned. This may result in extra scalars being present in the group returned, and when $p = 0$ this scalar subgroup can sometimes be infinite.

`QuasisimpleMatrixGroups()`

Returns a list of tuples specifying the names of the groups in the quasisimple matrix group database, together with the dimension and the number of stored representations of the group in that dimension.

66.15 Database of Soluble Irreducible Groups

This database contains one representative of each conjugacy class of irreducible soluble subgroups of $\mathrm{GL}(n, p)$, p prime, They may be accessed through specifying a group by its label in the database, as described in the section on basic functions, or through searching using predicates, or through a process. The database was constructed by Mark Short [Sho92].

66.15.1 Basic Functions

The basic access functions for the database are described in this section. The label of a group in the database is three integers, d, p, i. The first, $d \geq 2$, is the degree of the matrix group. The second, a prime p, specifies the base field of the group. The third is the number of the group in this degree/field set.

`IsolGroupDatabase()`

Open the database and return a reference to it. This reference may be passed to other functions so that they do fewer file operations.

`IsolGroup(n, p, i)`

`Group(D, n, p, i)`

Given a positive integer $o \leq 1000$ (with $o \neq 512$ or 768) and a positive integer n, return the n-th group of order o.

`IsolNumberOfDegreeField(n, p)`

The number of groups in the database of degree n over $\mathbf{F}_p$.

`IsolInfo(n, p, i)`

This function returns a string which gives some information about a group in the database given its label. In particular, it contains the order and primitivity information about the group.

`IsolOrder(n, p, i)`

This function returns the order of a group given its label.

`IsolMinBlockSize(n, p, i)`

This function returns the minimal block size of a group given its label. If it is primitive, it returns 0.

`IsolIsPrimitive(n, p, i)`

This function returns whether a group is primitive given its label.

`IsolGuardian(n, p, i)`

This function returns the "guardian" of a group given its label, i.e., the maximal subgroup of $\mathrm{GL}(n, p)$ of which the group is a subgroup.

Example H66E20

We find a group of degree 3 and its guardian.

```
> IsolNumberOfDegreeField(3, 5);
22
> G := IsolGroup(3, 5, 10);
> #G;
62
> GG := IsolGuardian(3, 5, 10);
> #GG;
372
> G;
MatrixGroup(3, GF(5)) of order 62 = 2 * 31
Generators:
  [0 0 1]
  [3 0 4]
  [2 3 1]
> GG;
MatrixGroup(3, GF(5)) of order 372 = 2^2 * 3 * 31
Generators:
  [1 0 0]
  [3 2 2]
  [1 4 2]

  [0 1 0]
  [0 0 1]
  [3 0 4]
```

66.15.2 Searching with Predicates

We may search the database for a group satisfying some predicate. A predicate for a group in this database is one of the following:

- A function f (which may either be an intrinsic function or a user defined function) which takes a matrix group and returns a boolean value.
- A tuple of one function $\langle g \rangle$, where g takes a label and returns a boolean value. Again g is either intrinsic or user defined.
- A tuple of two functions $\langle g, f \rangle$ where g, f are as above. In this case, the tested predicate will be g first, then f. This form is introduced to avoid expanding the group from its label until absolutely necessary.

`IsolGroupSatisfying(f)`

Given a predicate f, return a group satisfying it. This function runs through all the stored groups and applies the predicate until it finds a suitable one. If no group is found, an error message is printed.

`IsolGroupOfDegreeSatisfying(d, f)`

As `IsolGroupSatisfying(f)`, except it only runs through the groups of degree d.

`IsolGroupOfDegreeFieldSatisfying(d, p, f)`

As `IsolGroupSatisfying(f)`, except it only runs through the groups of degree d and defined over $\mathbf{F}_p$.

`IsolGroupsSatisfying(f)`

As `IsolGroupSatisfying(f)`, except a sequence of all such groups is returned.

`IsolGroupsOfDegreeSatisfying(d, f)`

As `IsolGroupOfDegreeSatisfying(d, f)`, except a sequence of all such groups is returned.

`IsolGroupsOfDegreeFieldSatisfying(d, p, f)`

As `IsolGroupOfDegreeFieldSatisfying(d, p, f)`, except a sequence of all such groups is returned.

66.15.3 Associated Functions

Associated with this database are two functions useful for constructing semidirect product of a finite vector space and an irreducible matrix group. Thus for constructing soluble affine permutation groups.

`Getvecs(G)`

This function takes a matrix group G over a finite prime field and returns a sequence, Q say, containing all the vectors of the natural module for G. The ordering of Q does *not* depend on G, but only on its natural module.

`Semidir(G, Q)`

Given an irreducible matrix group G of degree n and over a finite prime field of size p and the sequence Q obtained from `Getvecs`, this function returns the permutation group H of degree p^n that is the semidirect product of G with its natural module. H acts on the set $\{1 \dots p^n\}$ and G is isomorphic to each of the point stabilizers. It is well known that H is primitive, and that every primitive permutation group with soluble socle arises in this way. Note that if `Semidir` is to be called more than once for subgroups of the same general linear group, then `Getvecs` need only be called on the first occasion, since the ordering of Q depends only on n and p. This is why the call to `Getvecs` is not made by `Semidir` itself.

66.15.4 Processes

A small group process enables iteration over all groups with specified degrees and fields, without having to create and store all such groups together.

A process is created via the function `IsolProcess` and its variants. The standard process functions `IsEmpty`, `Current`, `CurrentLabel` and `Advance` can then be applied to the process.

A specifier for degree or field is one of a valid degree (field size), or a tuple $\langle l, h \rangle$, of valid degrees (field sizes) which is interpreted to mean all degrees (prime field sizes) in $[l, h]$.

`IsolProcess()`

Return a process which will iterate though all groups in the database.

`IsolProcessOfDegree(d)`

Return a process which will iterate though all groups in the database of degree d.

`IsolProcessOfField(p)`

Return a process which will iterate though all groups in the database over the specified field.

`IsolProcessOfDegreeField(d, p)`

Return a process for iterating over all the stored groups with degree specifier d and field specifier p. Initially it points to the first such group (the principal key is the degree).

`IsEmpty(p)`

Returns `true` if the process p has passed its last group.

`Current(p)`

Return the current group of the process p.

`CurrentLabel(p)`

Return the label of the current group of the process p. That is, return d, n and i such that the current group is `IsolGroup(d, n, i)`.

`Advance(~p)`

Move the process p to its next group.

Example H66E21

We use a small group process to look at all the groups of degree 3.

```
> P := IsolProcessOfDegree(3);
> ords := {* *};
> repeat
>   Include(~ords, #Current(P));
>   Advance(~P);
> until IsEmpty(P);
> ords;
{* 31, 62, 93, 7, 124, 96^^4, 39, 12^^2, 186, 13, 192^^2, 48^^4,
21, 24^^6, 26 *}
```

66.16 Database of ATLAS Groups

MAGMA includes representations of nearly simple groups from the ATLAS of Finite Group Representations http://web.mat.bham.ac.uk/atlas/v2.0. The data was supplied by Robert Wilson.

Groups in the database are accessed by name. The intrinsic ATLASGroupNames gives a list of the names that may currently be used to access the database. The names are based on ATLAS names for simple groups, with some exceptions (usually caused by an aversion to subscripting automorphisms). Classical group names take precedence over their Lie-type names. Within a name, the letter "T" denotes a twisted group of Lie type. (The two sorts of twisting of D_4 are distinguished by one being "O8m" and the other "TD4".) An initial number on the name denotes a central element, a "d" is used to separate the simple group name from an automorphism (when there is no other letter there), and an "i" denotes an isoclinic variant.

Example H66E22

The list of names in V2.11 is printed as follows.

```
> ATLASGroupNames();
{@ A5, 2A5, 2S5, 2S5i, S5, A6, 2A6, 2S6, 3A6, 3S6, 6A6,
6S6, A6V4, M10, PGL29, S6, A7, A8, 2A8, S8, A9, 2A9,
S9, A10, 2A10, S10, A11, 2A11, 2S11, S11, A12, 2A12,
S12, A13, 2A13, S13, A14, 2A14, 2S14, 2S14i, S14, O93,
2O93, 2O93d2, O93d2, O10m2, O10m2d2, O73, 2O73, 2O73d2,
3O73, 3O73d2, O73d2, O8m2, O8m2d2, O8m3, 2O8m3,
2O8m3d2a, O8m3D8, O8m3V4, O8m3d2a, O8p2, S102, S44,
S44d2, S44d4, S45, 2S45, S45d2, S47, 2S47, 2S47d2,
S47d2, S62, 2S62, S63, 2S63, 2S63d2, S63d2, S82, U311,
3U311, 3U311d2, U311d2, U33, U33d2, U42, 2U42, 2U42d2,
U42d2, U43, U52, U52d2, U53, U62, 12U62, 2U62, 3U62,
4U62, 6U62, U62S3, U62d2, U72, E74, E85, E82, E72, E62,
TF42, TF42d2, G25, TE62, 2TE62, 2TE62d2, 3TE62,
3TE62S3, 3TE62d2, 3TE62d3, 4TE62, TE62S3, TE62d2,
TE62d3, E64, 3E64, 3E64d2, TD42, TD42d3, G23, 3G23,
3G23d2, G23d2, G24, 2G24, 2G24d2, 2G24d2i, G24d2, F42,
2F42, 2F42d2, 2F42d4i, F42d2, R27, R27d3, Sz8, 2Sz8,
4Sz8d3, Sz8d3, Sz32, Sz32d5, TD43, L27, L28, L28d3,
L211, 2L211, L211d2, L213, 2L213, 2L213d2, L213d2,
L216, L216d2, L216d4, L217, 2L217, 2L217d2, L217d2,
L219, 2L219, 2L219d2i, L219d2, L223, 2L223, 2L223d2i,
L223d2, L227, L229, 2L229, L231, 2L231, L231d2, L232,
L232d5, L249, 2L249, L33, L33d2, L34, 12aL34, 12bL34,
2L34, 3L34, 4aL34, 4bL34, 6L34, L35, L35d2, L37, 3L37,
3L37d2, L37d2, L311, L52, L52d2, L62, L62d2, L72,
L72d2, B, Co1, 2Co1, Co2, Co3, F22, 2F22, 2F22d2, 3F22,
3F22d2, F22d2, F23, F24, 3F24, 3F24d2, F24d2, HN, HNd2,
HS, 2HS, 2HSd2, HSd2, He, Hed2, J1, J2, 2J2, 2J2d2,
```

```
J2d2, J3, 3J3, 3J3d2, J3d2, J4, Ly, ON, 3ON, 3ONd2,
ONd2, ONd4, Ru, 2Ru, Suz, 2Suz, 2Suzd2, 3Suz, 3Suzd2,
6Suz, 6Suzd2, Suzd2, Th, M, M11, M12, 2M12, 2M12d2,
M12d2, M22, 12M22, 2M22, 2M22d2, 3M22, 3M22d2, 4M22,
4M22d2, 6M22, 6M22d2, M22d2, M23, M24, McL, 3McL,
3McLd2, McLd2, S7 @}
```

The basic access function takes a name and returns a special type of group, an ATLAS group, with MAGMA type `GrpAtlas`. Access to the information stored about the named group are then done through this ATLAS group.

66.16.1 Accessing the Database

`ATLASGroupNames()`

The names of the groups that have representations stored in the database.

`ATLASGroup(N)`

The ATLAS group stored in the database that has name N.

66.16.2 Accessing the ATLAS Groups

Once an ATLAS group has been extracted from the database, the following intrinsics give access to the information stored with it.

`Order(A)`
`#G`

The order of A.

`Multiplier(A)`

The order of the multiplier of A, when A is simple.

`MatRepKeys(A)`

The sequence of keys to the matrix representations of A stored in the database. This will be the empty sequence if no matrix representations are stored.

`MatRepDegrees(A)`

The set of degrees of the matrix representations stored for A.

`MatRepFieldSizes(A)`

The set of sizes of the fields for which a matrix representation of A is available.

`MatRepCharacteristics(A)`

The set of characteristics of the fields for which a matrix representation of A is available.

`PermRepKeys(A)`

The sequence of keys to the permutation representations of A stored in the database. This will be the empty sequence if no permutation representations are stored.

`PermRepDegrees(A)`

The set of degrees of the permutation representations stored for A.

66.16.3 Representations of the ATLAS Groups

The intrinsics described below construct concrete representations of the ATLAS groups from the data in the database. Each representation is accessed by its key, sequences of which are produced by the intrinsics `MatRepKeys` and `PermRepKeys` described above. The intrinsics described in this section take a key and produce a concrete representation.

`MatrixGroup(K)`

Given a key to a matrix representation of an ATLAS group, construct and return the corresponding matrix group.

`MatRep(K)`

The generators of the matrix group designated by database key K.

`PermutationGroup(K)`

Given a key to a permutation representation of an ATLAS group, construct and return the corresponding permutation group.

`PermRep(K)`

The generators of the permutation group designated by database key K.

Example H66E23

We get a representation of $2.J_2.2$ from the database.

```
> A := ATLASGroup("2J2d2");
> PermRepKeys(A);
[]
> mrk := MatRepKeys(A);
> mrk;
[
  Matrix rep of degree 12 over GF(3),
  Matrix rep of degree 6 over GF(25) named a,
  Matrix rep of degree 12 over GF(7)
]
```

The database has no permutation representations and three matrix representations. We construct the first of the matrix groups. It is small enough to check its composition factors.

```
> K := mrk[1];
> M := MatrixGroup(K);
```

```
> M`Order := #A;
> RandomSchreier(M);
> CompositionFactors(M);
    G
    |  Cyclic(2)
    *
    |  J2
    *
    |  Cyclic(2)
    1
```

For efficiency, we asserted the order of the matrix group to be the order of the ATLAS group and constructed a BSGS by the random schreier.

66.17 Fundamental Groups of 3-Manifolds

The database consists of the fundamental groups of the 10,986 small-volume closed hyperbolic manifolds in the Hodgson-Weeks census. The presentations included were generated by Jeffrey Weeks' program *SnapPea* `http://www.geometrygames.org/SnapPea/`. Information about finite-index subgroups with homology was generated by Dunfield and Thurston in [DT03].

66.17.1 Basic Functions

The basic access functions for the database are described in this section.

The result returned by the `Manifold` function is a record with a number of fields containing information about the manifold and its fundamental group. The fields of the records are as follows:

Field `Name`: A string giving a name to the manifold M.

Field `Volume`: The volume of M as a floating point number.

Field `Homology`: A sequence of integers describing the first homology group of M.

Field `Group`; The fundamental group of M as a finitely presented group.

Field `GoodCoverImage`: A possibly empty sequence of permutations or integers 1 representing the identity permutation. These permutations define a homomorphism from the fundamental group to S_n, such that the kernel of the homomorphism has infinite abelianization.

Field `GoodCover`: A list describing the construction of the good cover.

Field `Degree`: A positive integer, the degree of the `GoodCoverImage` permutation representation.

Field `KnownPosBettiCover`: A boolean value, always true in the current database.

Field `KnownWeakPosBettiCover`: A boolean value, always true in the current database.

Field `Reason`: A string, one of `"AbelianInvariants"`, `"RationalReconstruction"` or `"MAGMA"`.

Field `Rank`: A positive integer.

Field `GoodCoverImageU`: A possibly empty sequence of permutations or integers 1 representing the identity permutation.

`ManifoldDatabase()`

Open the database and return a reference to it.

`Manifold(D, i)`

Extract the ith record from the database of fundamental groups of 3-dimensional manifolds. The current limits on i are $1 \leq i \leq 11126$.

66.17.2 Accessing the Data

The intrinsic `Manifold` is one way to access the data in the database. It may be more convenient to iterate over the database object returned by `ManifoldDatabase`. The following examples show how this may be done.

Example H66E24

We extract a record from the database.

```
> D := ManifoldDatabase();
> r := Manifold(D, 100);
> r`Name;
m019(1,4)
> r`Homology;
[ 2, 31 ]
> r`Group;
Finitely presented group on 2 generators
Relations
  $.1 * $.2^3 * $.1 * $.2 * $.1^4 * $.2 * $.1 * $.2 * $.1^4
    * $.2 = Id($)
  $.1 * $.2 * $.1 * $.2^2 * $.1^-3 * $.2^2 = Id($)
> r`GoodCoverImage;
[
  (1, 2, 4, 6, 5, 8, 7, 9, 3),
  (1, 3, 5, 4, 7, 6, 9, 8, 2)
]
```

In [DT03], Dunfield and Thurston note that they found 132 manifolds with positive Betti number. We find them in the database as those records where the `Degree` is 1. We then search the database for one of these, but by name. Both searches use the facility to iterate over the database that was mentioned above.

```
> D := ManifoldDatabase();
> pos_betti := {r`Name:r in D|r`Degree eq 1};
```

```
> #pos_betti;
132
> Random(pos_betti);
s527(-5,1)
> exists(r){r:r in D|r`Name eq "s527(-5,1)"};
true
> F := r`Group; F;
Finitely presented group F on 2 generators
Relations
  F.1^2 * F.2^2 * F.1^2 * F.2^-1 * F.1^2 * F.2^2 * F.1^2 *
  F.2^2 * F.1^-1 * F.2^2 = Id(F)
  F.1^2 * F.2^2 * F.1^2 * F.2 * F.1^2 * F.2^2 * F.1^2 * F.2
  * F.1^2 * F.2^2 * F.1^2 * F.2 * F.1^2 * F.2^2 * F.1^2 *
  F.2 * F.1^2 * F.2^2 * F.1^2 * F.2 * F.1^2 * F.2^2 * F.1^2
  * F.2^2 * F.1^-3 * F.2^2 = Id(F)
> AbelianQuotientInvariants(F);
[ 7, 0 ]
> r`Homology;
[ 0, 7 ]
```

As expected, we see that the fundamental group has infinite abelianization.

66.18 Bibliography

[**BE99a**] Hans Ulrich Besche and Bettina Eick. Construction of finite groups. *J. Symbolic Comput.*, 27(4):387–404, 1999.

[**BE99b**] Hans Ulrich Besche and Bettina Eick. The groups of order at most 1000 except 512 and 768. *J. Symbolic Comput.*, 27(4):405–413, 1999.

[**BE01**] Hans Ulrich Besche and Bettina Eick. The groups of order $q^n \cdot p$. *Comm. Algebra*, 29(4):1759–1772, 2001.

[**BEO01**] Hans Ulrich Besche, Bettina Eick, and E. A. O'Brien. The groups of order at most 2000. *Electron. Res. Announc. Amer. Math. Soc.*, 7:1–4 (electronic), 2001.

[**BP00**] Sergey Bratus and Igor Pak. Fast constructive recognition of a black box group isomorphic to S_n or A_n using Goldbach's conjecture. *J. Symbolic Comp.*, 29:33–57, 2000.

[**CH08**] J.J. Cannon and D.F. Holt. The transitive permutation groups of degree 32. *Experiment. Math.*, 17:307–314, 2008.

[**CQRD11**] Hannah J. Coutts, Martyn Quick, and Colva M. Roney-Dougal. The primitive permutation groups of degree less than 4096. *Communications in Algebra*, 39:10: 3526–3546, 2011.

[**DE05**] Heiko Dietrich and Bettina Eick. On the groups of cubefree order. *J. Algebra*, 292:122–137, 2005.

[**DT03**] Nathan M. Dunfield and William P. Thurston. The virtual Haken conjecture; experiments and examples. *Geometry & Topology*, 7:399–441, 2003.

[**HP89**] D.F. Holt and W. Plesken. *Perfect Groups*. Oxford University Press, 1989.

[**Hul05**] Alexander Hulpke. Constructing transitive permutation groups. *J. Symbolic Comput.*, 39(1):1–30, 2005.

[**Kir09**] M. Kirschmer. *Finite symplectic matrix groups*. Dissertation, RWTH Aachen, 2009. available at URL:http://www.math.rwth-aachen.de/ Markus.Kirschmer/symplectic/thesis.pdf.

[**MNVL04**] E.A. O'Brien M.F. Newman and M.R. Vaughan-Lee. Groups and nilpotent Lie rings whose order is the sixth power of a prime. *J. Algebra*, 278:383–401, 2004.

[**Neb96**] G. Nebe. Finite subgroups of $\mathrm{GL}_n(\mathbf{Q})$ for $25 \leq n \leq 31$. *Comm. Algebra*, 24(7):2341–2397, 1996.

[**Neb98**] G. Nebe. Finite quaternionic matrix groups. *Represent. Theory*, 2:106–223, 1998.

[**NP95**] G. Nebe and W. Plesken. Finite rational matrix groups. *Mem. Amer. Math. Soc.*, 116(556), 1995.

[**O'B90**] E.A. O'Brien. The p-group generation algorithm. *J. Symbolic Comput.*, 9:677–698, 1990.

[**O'B91**] E.A. O'Brien. The Groups of Order 256. *J. Algebra*, 143:219–235, 1991.

[**OVL05**] E.A. O'Brien and M.R. Vaughan-Lee. The groups with order p^7 for odd prime p. *J. Algebra*, 2005.

[**Ple85**] Wilhelm Plesken. Finite unimodular groups of prime degree and circulants. *J. Algebra*, 97:286–312, 1985.

[**PP77**] Wilhelm Plesken and Michael Pohst. On maximal finite irreducible subgroups of GL(n,Z). Parts I and II. *Math. Comp.*, 31:536–576, 1977.

[**PP80**] Wilhelm Plesken and Michael Pohst. On maximal finite irreducible subgroups of GL(n,Z). Parts III-V. *Math. Comp.*, 34(149):245–301, 1980.

[**RD05**] Colva M. Roney-Dougal. The primitive permutation groups of degree less than 2500. *J. Algebra*, 292(1):154–183, 2005.

[**RDU03**] Colva M. Roney-Dougal and William R. Unger. The affine primitive permutation groups of degree less than 1000. *J. Symbolic Comp.*, 35:421–439, 2003.

[**Sho92**] Mark W. Short. *The Primitive Soluble Permutation Groups of Degree less than 256*, volume 1519 of *Lecture Notes in Math.* Springer, Berlin and Heidelberg, 1992.

[**Sim70**] C.C. Sims. Computational methods in the study of permutation groups. In J. Leech, editor, *Computational problems in abstract algebra*, pages 169–183. Oxford - Pergamon, 1970.

[**Sou94**] Bernd Souvignier. Irreducible finite integral matrix groups of degree 8 and 10. *Math. Comp.*, 63:335–350, 1994.

67 AUTOMORPHISM GROUPS

Chapter 67
AUTOMORPHISM GROUPS

67.1 Introduction

MAGMA provides facilities for constructing and working with automorphism groups of various objects. In this chapter we describe the machinery provided in MAGMA for groups of automorphisms in the case of groups.

An *automorphism* of a group G is a bijective homomorphism from G to itself. The set of all automorphisms of G forms a group U known as the *automorphism group* of G. A subgroup A of U will be referred to as a *group of automorphisms* of G. The group G is called the *base group* of a group of automorphisms A and we say that A acts on G. Each MAGMA automorphism group A stores, as part of its data structure, a generating set for its base group, and each automorphism of A is described by its action on these generators.

The full group of automorphisms may be found using an algorithm that proceeds as follows: A series of characteristic subgroups

$$1 = N_r < N_{r-1} < \ldots < N_1 = L < G$$

is constructed for the given group G, such that each N_i/N_{i+1} is elementary abelian and such that G/L has no non-trivial soluble normal subgroup. The automorphism group is found for each of the associated factor groups of G, starting with the top factor G/L and lifting through each layer N_i/N_{i+1} in turn, until we finally have the automorphism group for G itself. The general algorithm for a non-soluble group is described in Cannon and Holt [CH03]. More specialised versions are described by Eick, Leedham-Green and O'Brien [ELGO02] (p-groups) and Smith [Smi94] (soluble groups). This general class of algorithms will be referred to collectively as *lifting algorithms*.

When G is a non-soluble permutation or matrix group, the algorithm relies on a database of automorphism groups for the non-cyclic simple factors of G, hence the non-abelian composition factors of G must belong to a restricted list. In V2.11 this list includes all simple groups of order at most 1.6×10^7, the alternating groups of degree at most 1000, all groups from several generic families, including $PSL(2, q)$, $PSL(3, q)$, $PSL(4, p)$, $PSL(5, p)$, $PSU(3, p)$ and $PSp(4, p)$ and the sporadic groups M_{11}, M_{12}, M_{22}, M_{23}, M_{24}, J_1, J_2, J_3, HS, McL, $Co3$, He and others. The list is being extended regularly.

An automorphism group A of G is represented as a set of homomorphisms of G into itself. We shall refer to this as the mappings representation of A. The full automorphism group is also returned as a finitely presented group and, in addition, it is also possible to construct a permutation representation of the automorphism.

The family of all groups of automorphisms forms a category. The objects are the automorphism groups and the morphisms are group homomorphisms. The MAGMA designation for this category of automorphism groups is `GrpAuto`.

67.2 Creation of Automorphism Groups

An automorphism group of the finite group G may be created in one of two ways. Firstly, the full automorphism group of G may be constructed by invoking an appropriate lifting algorithm. Secondly, an arbitrary group of automorphisms A of G may be created by giving a set of generators for A defined in terms of their action on a set of generators for G.

`AutomorphismGroup(G)`

Given a finite group G, construct the full automorphism group F of G. The group G may be a permutation group, a (finite) matrix group or a finite soluble group given by a pc-presentation. The function returns the full automorphism group of G as a group of mappings (i.e., as a group of type `GrpAuto`). If G is a permutation or matrix group, then the automorphism group F is also computed as a finitely presented group and can be accessed via the function `FPGroup(F)`. A function `PermutationRepresentation` is provided that when applied to F attempts to construct a faithful permutation representation of reasonable degree (see below).

`SmallOuterAutGroup` RngIntElt *Default :* 20000

`SmallOuterAutGroup := t`: Specify the strategy for the backtrack search when testing an automorphism for lifting to the next layer. If the outer automorphism group O at the previous level has order at most t, then the regular representation of O is used, otherwise the program tries to find a smaller degree permutation representation of O.

`Print` RngIntElt *Default :* 0

The level of verbose printing. The possible values are 0, 1, 2 or 3.

`PrintSearchCount` RngIntElt *Default :* 1000

`PrintSearchCount := s`: If `Print := 3`, then a message is printed at each s-th iteration during the backtrack search for lifting automorphisms.

In the case of a non-soluble group, the algorithm described in Cannon and Holt [CH03] is used. If G is a p-group of type `GrpPC` the algorithm described in Eick, Leedham-Green and O'Brien [ELGO02] is used. For more details see Section 63.12.2. If G is of type `GrpPC` but is not a p-group, the algorithm of Smith [Smi94], as extended by Smith and Slattery, is used. For more details see Section 63.12.

When G is a non-soluble permutation or matrix group, the algorithm relies on a database of automorphism groups for the non-cyclic simple factors of G, hence the non-abelian composition factors of G must belong to a restricted list. In V2.11 this list includes all simple groups of order at most 1.6×10^7, the alternating groups of degree at most 1000, all groups from several generic families, including $PSL(2,q)$, $PSL(3,q)$, $PSL(4,p)$, $PSL(5,p)$, $PSU(3,p)$ and $PSp(4,p)$ and the sporadic groups M_{11}, M_{12}, M_{22}, M_{23}, M_{24}, J_1, J_2, J_3, HS, McL, $Co3$, He and others. The list is being extended regularly.

Example H67E1

We create a non-soluble group G of 4×4 matrices defined over the field of 8-th roots of unity and construct its automorphism group.

```
> L<zeta_8> := CyclotomicField(8);
> i := -zeta_8^2;
> t := zeta_8^3;
> G := MatrixGroup< 4, L |
>             [ 1/2, 1/2, 1/2, 1/2,
>               1/2,-1/2, 1/2,-1/2,
>               1/2, 1/2,-1/2,-1/2,
>               1/2,-1/2,-1/2, 1/2 ],
>             DiagonalMatrix( [1,1,1,-1] ),
>             DiagonalMatrix( [1,i,1,i] ),
>             DiagonalMatrix( [t,t,t,t] ) >;
> Order(G);
92160
> CompositionFactors(G);
    G
    |  Cyclic(2)
    *
    |  Alternating(6)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
> A := AutomorphismGroup(G);
> Order(A);
92160
```

`AutomorphismGroup(G, Q, I)`

Let G be a finite group and let Q be a sequence of elements which generate G. Let $\phi_1, \ldots, \phi_r$ be a sequence of automorphisms of G that generate the group of automorphisms A. The group A is specified by a sequence I of length r where the i-th term of I defines ϕ_i in terms of a sequence containing the images of the elements of Q under the action of ϕ_i. The function returns the group of automorphisms A of G.

67.3 Access Functions

The functions described here provide access to basic information stored for an automorphism group A.

`Group(A)`

Given a group of automorphisms A of the group G, return the base group G on which A acts.

`NumberOfGenerators(A)`

`Ngens(A)`

Given a group of automorphisms A of the group G, return the number of defining generators for A.

`NumberOfPCGenerators(A)`

`NPCGenerators(A)`

`NPCgens(A)`

Given a group of automorphisms A of the group G, where a pc-representation has been created for A (and attribute `PCGenerators` is set on A), return the number of pc-generators for A.

`Generators(A)`

Given a group of automorphisms A of the group G, return a set containing the defining generators of A.

`PCGenerators(A)`

Given a group of automorphisms A of the group G, where a pc-representation has been created for A (and attribute `PCGenerators` is set on A), return an indexed set containing the pc-generators of A.

`InnerGenerators(A)`

Given the full group of automorphisms A of the group G, return a sequence of generators for the inner automorphism group of the base group of A (attribute `InnerGenerators`), if this attribute has been set.

`CharacteristicSeries(A)`

Given a group of automorphisms A of the group G, return the value of the characteristic series of G used to compute A, if this attribute has been set.

`IsSoluble(A)`
`IsSolvable(A)`

Given a group of automorphisms A of the group G, return the value of A's attribute `Soluble`, if this attribute has been set.

`IsSolubleAutomorphismGroupPGroup(A)`
`IsSolvableAutomorphismGroupPGroup(A)`

Given a group of automorphisms A of a p-group G constructed using the intrinsic `AutomorphismGroup(G)` or any equivalent alias, determine if A is soluble and return the result. This function also sets the `Soluble` attribute on A.

67.4 Order Functions

Unless the order is already known, each of the functions in this family will create a faithful permutation representation of the group of automorphisms in order to compute the order.

`Order(A)`
`#A`

The order of the group of automorphisms A, returned as an integer. If not already known, this function will create a permutation representation for A.

`FactoredOrder(A)`

The factored order of the group of automorphisms A. If not already known, this function will create a permutation representation for A.

`OuterOrder(A)`

The order of the outer automorphism group associated with the group of automorphisms A.

Example H67E2

We create the non-soluble group $G = PGL(2, 9)$ and examine the properties of its automorphism group.

```
> G := PGL(2, 9);
> A := AutomorphismGroup(G);
> A;
A group of automorphisms of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5
Generators:
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 7, 3, 5, 4, 2, 10, 9)
```

```
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 6, 8)(2, 7, 10)(3, 9, 5)
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 4, 6, 10, 7, 8, 5, 9)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 9, 10)(2, 6, 3)(4, 8, 7)
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (3, 5, 9, 6, 7, 4, 8, 10)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 10, 2)(3, 4, 7)(5, 8, 9)
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 10, 3, 5, 2, 4, 7, 6)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 6, 2)(3, 7, 5)(4, 9, 8)
> #A;
1440
> FactoredOrder(A);
[ <2, 5>, <3, 2>, <5, 1> ]
> OuterOrder(A);
2
> InnerGenerators(A);
[
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 7, 3, 5, 4, 2, 10, 9)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 6, 8)(2, 7, 10)(3, 9, 5),
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (1, 4, 6, 10, 7, 8, 5, 9)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 9, 10)(2, 6, 3)(4, 8, 7),
    Automorphism of GrpPerm: G, Degree 10, Order 2^4 * 3^2 * 5 which maps:
        (3, 5, 9, 6, 7, 4, 8, 10) |--> (3, 5, 9, 6, 7, 4, 8, 10)
        (1, 8, 2)(3, 4, 5)(6, 10, 7) |--> (1, 10, 2)(3, 4, 7)(5, 8, 9)
]
> CharacteristicSeries(A);
[
    Permutation group G acting on a set of cardinality 10
    Order = 720 = 2^4 * 3^2 * 5
        (3, 5, 9, 6, 7, 4, 8, 10)
        (1, 8, 2)(3, 4, 5)(6, 10, 7),
    Permutation group acting on a set of cardinality 10
    Order = 1
]
```

67.5 Representations of an Automorphism Group

To compute with automorphism groups, MAGMA uses various concrete representations of the group. These are summarised in this section.

`PermutationRepresentation(A)`

Construct a permutation representation of the group of automorphisms A. The function finds a union of conjugacy classes of the base group G which is closed under the action of A and with G-normal closure equal to G. The permutation action of A on such a set is faithful. The results returned are the representation of A as a homomorphism $A \to P$, the image of this homomorphism as a permutation group with standard support, and the set of elements of G used.

`PermutationGroup(A)`

Given a group of automorphisms A of a group G, this function returns a permutation group isomorphic to A as defined in the description of the function `PermutationRepresentation`.

`PermutationSupport(A)`

Given a group of automorphisms A of a group G, this function returns the set of elements of G (i.e., a union of conjugacy classes) used as the support of the permutation group constructed by the `PermutationRepresentation` function.

`PCGroupAutomorphismGroupPGroup(A)`

Attempt to directly construct a pc-representation for the group of automorphisms A of a conditioned p-group G. A must have been constructing using the `AutomorphismGroup` intrinsic, or any equivalent alias. The results returned are a boolean value indicating the solubility of A, and if soluble, a representation of A as a homomorphism $A \to P$ and the image of this homomorphism as a pc-group.

`FPGroup(A)`

A presentation for the group of automorphisms A on the generators of A. The isomorphism from the finitely presented group to the group of automorphisms A is also returned.

`OuterFPGroup(A)`

Suppose that A is the full group of automorphisms of a group G. This function returns a finitely presented group O isomorphic to the outer automorphism group of the base group G. The natural homomorphism from `FPGroup(A)` onto O is also returned.

Example H67E3

We calculate a permutation representation and presentation for the group of automorphisms of $PSL(2, 9)$.

```
> G := PGL(2, 9);
> A := AutomorphismGroup(G);
> PermutationGroup(A);
Permutation group acting on a set of cardinality 36
Order = 1440 = 2^5 * 3^2 * 5
    (1, 30)(3, 27)(5, 17)(6, 24)(8, 9)(10, 14)(11, 13)(12, 32) (15, 34)(16, 21)
       (18, 25)(19, 28)(22, 29)(23, 31)(26, 33)(35, 36)
    (1, 32, 19, 22)(2, 34)(3, 18, 7, 17)(4, 25, 30, 31) (5, 23, 33, 24)
       (6, 15, 26, 21)(8, 16, 29, 20)(9, 35, 14, 27) (10, 13, 11, 12)(28, 36)
    (1, 2, 3, 5, 8, 13, 22, 31)(4, 7, 9, 15, 24, 26, 34, 29)
       (6, 10, 17, 23, 32, 33, 28, 35)(11, 19, 27, 16, 25, 21, 30, 36)
       (12, 20, 14, 18)
    (1, 32, 33, 12, 21, 6)(2, 34, 26, 35, 17, 16)(3, 28, 22, 7, 18, 13)
       (4, 31, 20, 29, 24, 11)(5, 19, 25, 10, 15, 8)(9, 30, 36, 14, 23, 27)
> F<x, y, z, t> := FPGroup(A);
> F;
Finitely presented group F on 4 generators
Relations
    x^2 = Id(F)
    y^4 = Id(F)
    (x * y^-1)^5 = Id(F)
    y^-2 * x * y^-2 * x * y^-2 * x * y^2 * x * y^2 * x = Id(F)
    z^-1 * x * z * y^-1 * x^-1 * y^-2 * x^-1 * y * x^-1 *
       y^-2 * x^-1 * y * x^-1 * y^-1 * x^-1 = Id(F)
    z^-1 * y * z * y * x^-1 * y * x^-1 * y^-1 * x^-1 = Id(F)
    z^2 * y^-1 * x^-1 * y * x^-1 * y^-1 * x^-1 = Id(F)
    x^t = y * x * y^-1
    y^t = y^-1 * x * y * x * y
    z^t = z * x * y^-1 * x
    t^2 = x * y^2 * x * y^-1 * x * y
```

Example H67E4

We illustrate the process of finding a low degree permutation representation of an automorphism group using the above functions. We start with the Higman-Sims sporadic simple group, construct its automorphism group, and then use the function `PermutationGroup` to obtain a permutation representation.

```
> load hs100;
Loading "/home/magma/libs/pergps/hs100"
The simple group of Higman-Sims represented as a
permutation group of degree 100.
Order: 44 352 000 = 2^9 * 3^2 * 5^3 * 7 * 11.
Base: 1, 2, 3, 4, 5, 6.
```

```
Group: G
> aut := AutomorphismGroup(G);
> P := PermutationGroup(aut);
> P;
Permutation group P acting on a set of cardinality 5775
Order = 88704000 = 2^10 * 3^2 * 5^3 * 7 * 11
```

We've got a permutation representation on 5775 letters. Now we want to get it on 100 letters, so we need to find the subgroup of index 100.

```
> lix := LowIndexSubgroups(P, 100);
> [ Index(P, H) : H in lix];
[ 1, 2, 100 ]
```

There it is, so we can compute the corresponding permutation representation.

```
> H := CosetImage(P, lix[3]);
> H;
Permutation group H acting on a set of cardinality 100
> CompositionFactors(H);
    G
    |  Cyclic(2)
    *
    |  HS
    1
```

67.6 Automorphisms

The elements of a group of automorphisms are automorphisms of the base group, so MAGMA treats them as both homomorphisms and group elements. Thus they may be applied to elements and subgroups of the base group as a homomorphism, or they may be multiplied and have inverses taken as group elements. Of course, these last two operations are also homomorphism operations, being composition and the usual inverse of a bijection. Elements of a group of automorphisms are of type `GrpAutoElt`.

`A . i`

Let A be a group of automorphisms of a group G and let i be an integer such that $-n \leq i \leq n$, where n is the number of generators of A. This operator returns the i-th generator for A. A negative subscript indicates that the inverse of the generator is to be created. Finally, $A.0$ denotes the identity of A.

`Identity(A)`

`Id(A)`

`A ! 1`

The identity element of the group of automorphisms A.

`A ! f`

Let A be a group of automorphisms of a group G. Given an automorphism f of G, represented as a MAGMA map, this function returns the element of A corresponding to f. An error will result if f is not in the group generated by the generators of A. This uses the permutation representation of A to test for membership.

`Order(f)`

The order of the group automorphism f.

`f * g`

The product of the group automorphisms f and g. If f and g are regarded as maps, this function returns their composite: first apply f, then apply g.

`f ^ n`

The nth power of the group automorphism f. The integer n may be positive or negative.

`(g1, ..., gr)`

The left-normed commutator of the group automorphisms $g_1, \ldots, g_r$. Each of $g_1, \ldots, g_r$ must belong to a common automorphism group.

`g eq h`

Given group automorphisms g and h belonging to the same automorphism group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given group automorphisms g and h belonging to the same automorphism group, return `false` if g and h are the same element, `true` otherwise.

`IsInner(f)`

Returns `true` if the group automorphism f is an inner automorphism of the base group, `false` otherwise. If f is inner, then an element of the base group with conjugation action equal to the action of f is also returned.

Example H67E5

We illustrate some arithmetic operations with elements of the full group of automorphisms of a group of order 81.

```
> G := SmallGroup(81, 10);
> G;
GrpPC : G of order 81 = 3^4
PC-Relations:
  G.1^3 = G.4,
  G.2^3 = G.4^2,
  G.2^G.1 = G.2 * G.3,
  G.3^G.1 = G.3 * G.4
> A := AutomorphismGroup(G);
> #A;
486
> Ngens(A);
5
> IsInner(A.3);
false
> Order(A.3);
3
> A.3;
Automorphism of GrpPC : G of order 3 which maps:
  G.1 |--> G.1
  G.2 |--> G.2 * G.4^2
  G.3 |--> G.3
  G.4 |--> G.4
> A.3*A.4;
Automorphism of GrpPC : G which maps:
  G.1 |--> G.1
  G.2 |--> G.2 * G.3 * G.4^2
  G.3 |--> G.3 * G.4
  G.4 |--> G.4
> (A.3*A.4)^3;
Automorphism of GrpPC : G which maps:
  G.1 |--> G.1
  G.2 |--> G.2
  G.3 |--> G.3
  G.4 |--> G.4
> $1 eq Id(A);
true
```

Example H67E6

We can use the automorphism group machinery to determine the characteristic subgroups of a group.

```
> CharacteristicSubgroups := function(G)
```

```
>    local A, outers, NS, CS;
>    A := AutomorphismGroup(G);
>    outers := [ a : a in Generators(A) | not IsInner(a) ];
>    NS := NormalSubgroups(G);
>    CS := [n : n in NS | forall{a: a in outers| a(n`subgroup) eq n`subgroup }];
>    return CS;
> end function;
>
> CS := CharacteristicSubgroups(DirectProduct(Alt(4),Alt(4)));
> [c`order: c in CS];
[ 1, 16, 144 ]
> G := SmallGroup(512,298);
> #NormalSubgroups(G);
42
> #CharacteristicSubgroups(G);
28
```

67.7 Stored Attributes of an Automorphism Group

Groups of automorphisms have several attributes that may be stored as part of their data structure. The function `HasAttribute` is used to test if an attribute is stored and to retrieve its value, while the function `AssertAttribute` is used to set attribute values. The user is warned that when using `AssertAttribute` the data given is *not* checked for validity, apart from some simple type checks. *Setting attributes incorrectly will result in errors.*

`HasAttribute(A, s)`

`AssertAttribute(A, s, v)`

The `HasAttribute` function returns whether the group of automorphisms A has the attribute named by the string s defined and, if so, also returns the value of the attribute.

The `AssertAttribute` procedure sets the attribute of the group of automorphisms group named by string s to have value v. The possible names are:
`Group`: The base group of the automorphism group. This is always set.
`Order`: The order of the automorphism group. It is an integer and may be set by giving either an integer or a factored integer.
`OuterOrder`: The order of the outer automorphism group associated with A. It is an integer and may be set by giving either an integer or a factored integer.
`Soluble`: (also `Solvable`) A boolean value telling whether or not the automorphism group is soluble.
`InnerGenerators`: A sequence of generators of A known to be inner automorphisms.
`InnerMap`: A homomorphism from the base group to the automorphism group taking each base group element to its corresponding inner automorphism.
`ClassAction`: Stores the result of the `PermutationRepresentation` function call.

`ClassImage`: Stores the result of the `PermutationGroup` function call.
`ClassUnion`: Stores the result of the `ClassUnion` function call.
`FpGroup`: Stores the result of the `FPGroup` function call. The attribute is a pair $\langle F, m\rangle$ where F is an fp-group and m is an isomorphism $m : F \to A$. F and m are the two return values of `FPGroup(A)`.
`OuterFpGroup`: Stores the result of the `OuterFPGroup` function call. The attribute is a pair $\langle O, f\rangle$ where O is an fp-group and f is an epimorphism $f : F \to O$, where F is the first component of the `FpGroup` attribute. The kernel of f is the inner automorphism group. O and f are the two return values of `OuterFPGroup(A)`.
`GenWeights`: `WeightSubgroupOrders`: See the section on automorphism groups in the chapter on soluble groups for details.

Example H67E7

We select a group of order 904 from the small groups database and compute its group of automorphisms.

```
> G := SmallGroup(904, 4);
> FactoredOrder(G);
[ <2, 3>, <113, 1> ]
> FactoredOrder(Centre(G));
[ <2, 1> ]
> A := AutomorphismGroup(G);
> FactoredOrder(A);
[ <2, 7>, <7, 1>, <113, 1> ]
> HasAttribute(A, "FpGroup");
false
> HasAttribute(A, "OuterFpGroup");
false
> F,m := FPGroup(A);
> AssertAttribute(A, "FpGroup", <F,m>);
```

Note that values for some attributes, such as `FpGroup`, have not been calculated by the original `AutomorphismGroup` call, but they may be calculated and set later if desired. The outer automorphism group has order $2^5 \times 7$. We find the characteristic subgroups of `G`.

```
> n := NormalSubgroups(G);
> [x`order : x in n];
[ 1, 2, 113, 4, 226, 452, 452, 452, 904 ]
> characteristics := [s : x in n |
>   forall{f: f in Generators(A) | s@f eq s}
>   where s is x`subgroup];
> [#s : s in characteristics];
[ 1, 2, 113, 4, 226, 452, 904 ]
```

Note that two of the normal subgroups of order 452 are not characteristic.

Example H67E8

```
> G := AlternatingGroup(6);
> A := AutomorphismGroup(G);
> HasAttribute(A, "OuterFpGroup");
true
> F, f := FPGroup(A);
> O, g := OuterFPGroup(A);
> O;
Finitely presented group O on 2 generators
Relations
  O.1^2 = Id(O)
  O.2^2 = Id(O)
  (O.1 * O.2)^2 = Id(O)
> A`OuterOrder;
4
```

We find the outer automorphism group is elementary abelian of order 4. The direct product of `G` with itself has maximal subgroups isomorphic to `G`, in the form of diagonal subgroups. We can construct four non-conjugate examples using the outer automorphism group. The first example can be constructed without using an outer automorphism.

```
> GG, ins := DirectProduct(G, G);
> M := sub<GG|[(x@ins[1])*(x@ins[2]):x in Generators(G)]>;
> IsMaximal(GG, M);
true
```

The subgroup `M` is the first, the obvious diagonal, constructed using just the embeddings returned by the direct product function. We get others by twisting the second factor using an outer automorphism. First we get (representatives of) the outer automorphisms explicitly.

```
> outers := {x @@ g @ f : x in [O.1, O.2, O.1*O.2]};
> Representative(outers);
Automorphism of GrpPerm: G, Degree 6, Order 2^3 * 3^2 * 5 which maps:
  (1, 2)(3, 4, 5, 6) |--> (1, 3, 6, 2)(4, 5)
  (1, 2, 3) |--> (1, 4, 2)(3, 5, 6)
```

The set `outers` now contains three distinct outer automorphisms of `G`. We use them to get three more diagonal subgroups.

```
> diags := [M] cat
> [sub<GG|[(x @ ins[1])*(x @ f @ ins[2]):x in Generators(G)]>:
>   f in outers];
> [IsMaximal(GG, m) : m in diags];
[ true, true, true, true ]
> IsConjugate(GG, diags[2], diags[4]);
false
```

The other five tests for conjugacy will give similarly negative results.

67.8 Holomorphs

Given a group G and the full group of automorphisms A of G then the holomorph of G is the semidirect product $G \times_\theta A$, where $\theta : A \to Aut(G)$ is the identity map.

`Holomorph(G)`

`Holomorph(GrpFP, G)`

Given a finite permutation, matrix or pc-group G with full group of automorphisms A, this function returns the semidirect product E of G by A. The group E is returned as a permutation group (or a finitely presented group if `GrpFP` is specified) of degree $|G|$ in which G is a regular normal subgroup, and A is the stabilizer of the point 1. The embedding map $G \to E$, and the natural epimorphism $E \to A$ are also returned. In the returned group E, the generators of G appear first, followed by those of A.

`Holomorph(G, A)`

`Holomorph(GrpFP, G, A)`

Given a finite permutation, matrix or pc-group G and a group of automorphisms A, this function returns the semidirect product E of G by A. The group E is returned as a permutation group (or a finitely presented group if `GrpFP` is specified) of degree $|G|$ in which G is a regular normal subgroup, and A is the stabilizer of the point 1. The embedding map $G \to E$, and the natural epimorphism $E \to A$ are also returned. In the returned group E, the generators of G appear first, followed by those of A.

Example H67E9

We construct the holomorph of the group $G = PGL(2, 9)$.

```
> G := PGL(2, 9);
> E := Holomorph(G); E;
Permutation group E acting on a set of cardinality 720
> #E;
1036800
> CompositionFactors(E);
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Alternating(6)
    *
    |  Alternating(6)
    1
```

67.9 Bibliography

[**CH03**] J.J. Cannon and D.F. Holt. Automorphism group computation and isomorphism testing in finite groups. *J. Symbolic Comp.*, 35(3):241–267, 2003.

[**ELGO02**] Bettina Eick, C.R. Leedham-Green, and E.A. O'Brien. Constructing automorphism groups of a p-groups. *Comm. Algebra*, 30:2271–2295, 2002.

[**Smi94**] Michael J. Smith. *Computing automorphisms of finite soluble groups.* PhD thesis, Australian National University, 1994.

68 COHOMOLOGY AND EXTENSIONS

Chapter 68
COHOMOLOGY AND EXTENSIONS

68.1 Introduction

The following collection of cohomology functions is designed to provide a flexible set of tools for computing with first and second cohomology groups of any type of finite group acting on any reasonable module, including a module defined by an action on an arbitrary finitely generated abelian group. First (but not second) cohomology groups can also be calculated for infinite groups defined by a finite presentation.

Zero-cocycles, one-cocycles and two-cocycles may be computed and identified. Extensions of modules by groups can be constructed as finitely presented groups, or as PC-groups when the acting group is a PC-group. It is also possible to compute a representative set of extensions of the module by the group each of which is distinct up to a group isomorphism fixing the module. These functions complement, but do not completely supplant, an older collection of functions pertaining to cohomology groups, Schur multiplicators and covering groups which apply to permutation groups (see Chapter 58 on Permutation Groups).

The first cohomology group $H^1(G, M)$ is calculated as the nullspace of a certain matrix. The details can be found in Section 5 of [CCH01]. This immediately allows manipulation and identification of one-cocycles. The second cohomology group $H^2(G, M)$ is more difficult to compute. While it can also be found as the nullspace of a suitable matrix, this matrix can be uncomfortably large in big examples. For soluble groups defined by a PC-presentation, the matrix corresponds to solving the consistency equations for a PC-presentation of a general extension of the module by the group, which depends on the number of group generators rather than its order, and is manageable for quite large groups. For permutation and matrix groups G, the size of the matrix for which the nullspace is required is much larger, but can often be reduced to a reasonable size by using a base and strong generating set for G. In the case where only the dimension of $H^2(G, M)$ is required, and M is a module over a finite field of prime order p, then the calculation of this dimension can be reduced to the determination of $H^2(Q, M)$ for a suitable collection of p-subgroups Q of G. The latter calculation can be carried out efficiently using the PC-presentation approach (see [Hol85b] for details).

To use the new functions, the user must initially invoke the function `CohomologyModule`, which creates a special object for the group action corresponding to the module, and all subsequent (new) cohomology functions take this object as their first argument.

In the case of a finite permutation or matrix group G acting on a module M over a prime field, the dimension of $H^2(G, M)$ may be found much more quickly by executing `CohomologicalDimension(CM, 2)`, where CM is the cohomology module for the action of G on M, rather than by invoking `Dimension(CohomologyGroup(CM, 2))`. However, the

former call does not allow the possibility of subsequent calculations with two-cocycles or extensions.

The equivalent older function, `CohomologicalDimension(G, M, 2);` (for a permutation group G) is often faster still for small examples, but the new function will succeed on much larger examples than the old. For the convenience of the reader, some of these older functions are described in this section of the Handbook. For complete details about the older functions, see the section on cohomology in the chapter on Permutation Groups.

68.2 Creation of a Cohomology Module

In order to compute the cohomology of a group with respect to a G-module M, it is first necessary to construct a data structure known as a *cohomology module*.

`CohomologyModule(G, M)`

Given a group G and a G-module M with acting group G this function returns a cohomology module for the action of G. The group G may be a finite permutation group, a finite matrix group, a PC-group, or any finitely presented group. For the PC-group case, however, the PC-presentation of G must be conditioned. This can be achieved by first executing the statement `G := ConditionedGroup(G);`

`CohomologyModule(G, Q, T)`

Let G be a group which acts on a finitely-generated abelian group with invariants given by the sequence Q, and action described by T. The action T is given in the form of a sequence of $d \times d$ matrices over the integers, where d is the length of T, and `T[i]` defines the action of the i-th generator of G on the abelian group. The function returns a cohomology module for the action of G. The group G may be a finite permutation group, a finite matrix group, a PC-group or any finitely presented group. For the PC-group case, however, the PC-presentation of G must be conditioned. This can be achieved by first executing the statement `G := ConditionedGroup(G);`

Example H68E1

We construct the cohomology module for $PSL(3,2)$ acting on a module of dimension 3 over $GF(2)$. We first need to find a module of dimension 3.

```
> G := PSL(3, 2);
> Irrs := AbsolutelyIrreducibleModules(G, GF(2));
> Irrs;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 3 over GF(2),
    GModule of dimension 3 over GF(2),
    GModule of dimension 8 over GF(2)
]
> M := Irrs[2];
> CM := CohomologyModule(G, M);
```

```
> CM;
Cohomology Module
```

`CohomologyModule(G, A, M)`

For a permutation group G acting on some abelian group A through M, compute the cohomology module. M has to be either a map from G into the endomorphisms of A, or a sequence of endomorphisms of A, one for each of the generators of G.

68.3 Accessing Properties of the Cohomology Module

The functions described in this section merely return data used to define the cohomology module. In each case, the argument CM must be a cohomology module returned by a call to `CohomologyModule`.

`Module(CM)`

The $K[G]$-module used to define the cohomology module CM. An error occurs if CM was defined by an action on a finitely generated abelian group.

`Invariants(CM)`

Given a cohomology module CM that was defined by an action on a finitely generated abelian group A, return the invariants of A. If CM was not defined by an action on an abelian group, an error results.

`Dimension(CM)`

Let CM be a cohomology module. If CM was defined by the action of a group on an R-module M, return the dimension of M. In the case in which CM was defined by the action of a group on a finitely generated abelian group A, the rank of A is returned.

`Ring(CM)`

The ring over which the module used to define the cohomology module CM is defined. If CM is defined in terms of an action on a finitely generated abelian group A, then the ring will be the integers if A is infinite, and the integers modulo the exponent of A if A is finite.

`Group(CM)`

The group used to define action on the cohomology module CM.

`FPGroup(CM)`

Given a cohomology module CM with associated group G, return a finitely presented group F isomorphic to G and the isomorphism from F to G. This presentation is on a strong generating set if G is a permutation or matrix group. It is used in the construction of presentations of extensions returned by the function `Extension`.

`MatrixOfElement(CM, g)`

The matrix representing the action of the element g in the group of CM on the module of CM.

68.4 Calculating Cohomology

`CohomologyGroup(CM, n)`

Given a cohomology module CM for the group G acting on the module M and a non-negative integer n taking one of the values 0, 1 or 2, this function returns the cohomology group $H^n(G, M)$. For modules defined over the ring of integers only, n may also be equal to 3. (In this case, $H^3(G, M)$ is computed as the second cohomology group of M regarded as a module over Q/Z.) If the group used to define CM was a finitely presented group, then n may only be equal to 0 or 1. Note that CM must be a module returned by invoking `CohomologyModule`.

`CohomologicalDimension(CM, n)`

Given a cohomology module CM for the group G acting on the module M defined over a finite field K and a non-negative integer n taking one of the values 0, 1 or 2, this function returns the dimension of $H^n(G, M)$ over K. Note that this function may only be applied to the module returned by a call to `CohomologyModule(G, M)`, where M is a module over a finite field K. When $n = 2$, this function is faster and may be applied to much larger examples than `CohomologyGroup(CM, n)` but, unlike that function, it does not enable the user to compute with explicit extensions and two-cocycles.

Note that there are some alternative functions for performing these calculations described in other manual chapters.

`CohomologicalDimension(M, n)`

For $K[G]$-module M (with K a finite field and G a finite group), compute and return the K-dimension of the cohomology group $H^n(G, M)$ for $n \geq 0$. For $n = 0$ and 1, this is carried out by using the function `CohomologicalDimension(CM,n)` just described. For $n \geq 2$, it is done recursively using projective covers and dimension shifting to reduce to the case $n = 1$.

`CohomologicalDimensions(M, n)`

For $K[G]$-module M (with K a finite field and G a finite group), compute and return the sequence of K-dimensions of the cohomology groups $H^k(G, M)$ for $1 \leq k \leq n$. On account of the recursive method used, this is quicker than computing them all individually.

`CohomologicalDimension(G, M, n)`

Given the permutation group G, the $K[G]$-module M and an integer n (equal to 1 or 2), return the dimension of the n-th cohomology group of G acting on M. *Note that K must be a finite field of prime order.* This function invokes Derek Holt's original C cohomology code (see [Hol85b]). In some cases it will be faster than the function that uses the cohomology module data structure.

Example H68E2

We examine the first and second cohomology groups of the group A_8.

```
> G := Alt(8);
> M := PermutationModule(G, GF(3));
```

We first calculate the dimensions of $H^1(G, M)$ and $H^2(G, M)$ using the old functions.

```
> time CohomologicalDimension(G, M, 1);
0
Time: 0.020
> time CohomologicalDimension(G, M, 2);
1
Time: 0.020
```

We now recalculate the dimensions of $H^1(G, M)$ and $H^2(G, M)$ using the new functions.

```
> X := CohomologyModule(G, M);
> time CohomologicalDimension(X, 1);
0
Time: 0.020
> time CohomologicalDimension(X, 2);
1
Time: 0.920
> X := CohomologyModule(G, M);
> time C:=CohomologyGroup(X, 2);
Time: 4.070
> C;
Full Vector space of degree 1 over GF(3)
```

Example H68E3

In the case of $\Omega^-(8, 3)$ acting on its natural module, the new function succeeds, but the old function does not.

```
> G := OmegaMinus(8, 3);
> M := GModule(G);
> X := CohomologyModule(G, M);
> time CohomologicalDimension(X, 2);
2
Time: 290.280
> phi, P := PermutationRepresentation(G);
```

```
> MM := GModule(P, [ActionGenerator(M, i): i in [1..Ngens(G)]] );
>  time CohomologicalDimension(P, MM, 2);
Out of space.
>> time CohomologicalDimension(P, MM, 2);
                                ^
Runtime error in 'CohomologicalDimension': Cohomology failed
```

68.5 Cocycles

Before invoking the functions in this section, it is necessary to first invoke the function `CohomologyGroup(CM, n)` for the appropriate n.

For $n = 0, 1$ or 2, an *n-cocycle* is a function from G^n to the module M, where elements of G^n are represented as an n-tuple $\langle g_1, \ldots, g_n \rangle$ of group elements, for which a certain relation is satisfied. These relations are consistent with the MAGMA convention of the use of right actions, and so they are slightly different from those encountered in many textbooks, where left actions are more common.

0-, 1- and 2-cocycles z, o and t, respectively, satisfy the following relations for all $g, h, \in G$.

$$z(\langle\rangle)^g = z(\langle\rangle);$$

$$o(\langle gh \rangle) = o(\langle g \rangle)^h + o(\langle h \rangle);$$

$$t(\langle gh, k \rangle) + t(\langle g, h \rangle)^k = t(\langle g, hk \rangle) + t(\langle h, k \rangle).$$

`ZeroCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and an element s of the cohomology group $H^0(G, M)$ associated with CM, this function returns the corresponding zero-cocycle. The zero-cocycle is returned as a function of the 0-tuple $\langle\rangle$, of which the image is an element of the fixed point submodule of M. The argument s may either be given as an element of $H^0(G, M)$ or as a sequence of integers defining such an element.

`IdentifyZeroCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a zero-cocycle given as a function of the 0-tuple $\langle\rangle$, of which the image is an element of the fixed point submodule of M, this function returns the corresponding element of $H^0(G, M)$. Hence this function is the inverse function to `ZeroCocycle`.

`OneCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and an element s of the cohomology group $H^1(G, M)$ associated with CM, the function returns a corresponding one-cocycle. The one-cocycle is returned as a function from G to the module M, where elements g of G are represented as 1-tuples $\langle g \rangle$. The argument s may either be given as an element of $H^1(G, M)$ or as a sequence of integers defining such an element.

`IdentifyOneCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a one-cycle s for CM, specified as a function from G to the module M (where elements g of G are represented as 1-tuples $\langle g \rangle$), this function returns the corresponding element of $H^1(G, M)$. Thus, the function is the inverse to `OneCocycle`.

`IsOneCoboundary(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a one-cycle s for CM, specified as a function from G to the module M (where elements g of G are represented as 1-tuples $\langle g \rangle$), this function determines whether the cocyle is a 1-coboundary; that is, whether it corresponds to the zero element of $H^1(G, M)$. If so, then it also returns a corresponding 0-cochain $t(\langle\rangle)$ that satisfies $s(\langle g \rangle) = t(\langle\rangle) - t(\langle\rangle)^g$ for all $g \in G$.

`TwoCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and an element s of the cohomology group $H^2(G, M)$ associated with CM, the function returns a corresponding two-cocycle. The two-cocycle is returned as a function from $G \times G$ to the module M, where elements of $G \times G$ are represented as 2-tuples $\langle g_1, g_2 \rangle$. The argument s may either be given as an element of $H^2(G, M)$ or as a sequence of integers defining such an element.

`IdentifyTwoCocycle(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a two-cycle s for CM, specified as a function from $G \times G$ to the module M (where elements of $G \times G$ are represented as 2-tuples $\langle g_1, g_2 \rangle$), this function returns the corresponding element of $H^2(G, M)$. Thus, the function is the inverse to `TwoCocycle`.

`IsTwoCoboundary(CM, s)`

Given a cohomology module CM constructed from the $K[G]$-module M and a two-cycle s for CM, specified as a function from G to the module M (where elements of $G \times G$ are represented as 2-tuples $\langle g_1, g_2 \rangle$), this function determines whether the cocyle is a 2-coboundary; that is, whether it corresponds to the zero element of $H^2(G, M)$. If so, then it also returns a corresponding 1-cochain $t(\langle g \rangle)$ that satisfies $s(\langle g, h \rangle) = t(\langle g \rangle)^h + t(\langle h \rangle) - t(\langle gh \rangle)$ for all $g, h \in G$.

Example H68E4

An easy example where the module is an abelian group defined by its invariant factors.

```
> G := PermutationGroup< 4 | (1,2,3,4) >;
> invar:=[2,4,4];
> mats := [ Matrix(Integers(),3,3,[1,2,0,0,0,1,0,1,2]) ];
> X := CohomologyModule(G,invar,mats);
> C := CohomologyGroup(X,0);
> C;
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 4 ]
> ZeroCocycle(X,[3]);
function(tp) ... end function
> IdentifyZeroCocycle(X,func<x|-$1(<>)>);
(1)
> C := CohomologyGroup(X,1);
> C;
Full Quotient RSpace of degree 2 over Integer Ring
Column moduli:
[ 2, 2 ]
> z1 := OneCocycle(X,[1,0]);
> z2 := OneCocycle(X,[0,1]);
> z1(<G.1>);
(1 0 0)
> z := func< x | z1(x)+z2(x) >;
> IdentifyOneCocycle(X,z);
(1 1)
> C := CohomologyGroup(X,2);
> C;
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 4 ]
> z1 := TwoCocycle(X,[1]);
> z1(<G.1,G.1^2>);
(1 1 3)
> z := func< xy | z1(xy)+z1(xy) >;
> IdentifyTwoCocycle(X,z);
(2)
```

68.6 The Restriction to a Subgroup

`Restriction(CM, H)`

Given a cohomology module for a group G and a subgroup H of G, form the restriction of the input cohomology module to H.

Note that, denoting this restriction by CMH, we can define the restriction maps on the first and second cohomology groups of CM by

```
>  res1 := hom<CohomologyGroup(CM, 1) -> CohomologyGroup(CMH, 1) |
>                          x:->IdentifyOneCocycle(CMH,OneCocycle(CM,x)) >;
>  res2 := hom<CohomologyGroup(CM, 2) -> CohomologyGroup(CMH, 2) |
>                          x:->IdentifyTwoCocycle(CMH,TwoCocycle(CM,x)) >;
```

Example H68E5

In this example we define G to be the group $GL(3,2)$ and H to be the Sylow 2-subgroup of G. We illustrate how to calculate the restriction mappings of $H^n(G,M)$ to $H^n(G,MH)$, where MH is the restriction of M to H.

```
> G := GL(3, 2);
> M := GModule(G);
> H := Sylow(G, 2);
> CG := CohomologyModule(G, M);
> CH := Restriction(CG, H);
```

We first consider $H^1(G,M)$.

```
> H1G := CohomologyGroup(CG, 1); H1G;
Full Vector space of degree 1 over GF(2)
> H1H := CohomologyGroup(CH, 1); H1H;
Full Vector space of degree 2 over GF(2)
> res1 := hom<H1G -> H1H | x:->IdentifyOneCocycle(CH,OneCocycle(CG,x)) >;
> res1(H1G.1);
(1 1)
```

We now consider $H^2(G,M)$.

```
> H2G := CohomologyGroup(CG, 2); H2G;
Full Vector space of degree 1 over GF(2)
> H2H := CohomologyGroup(CH, 2); H2H;
Full Vector space of degree 3 over GF(2)
> res2 := hom<H2G -> H2H | x:-> IdentifyTwoCocycle(CH,TwoCocycle(CG,x)) >;
> res2(H2G.1);
(0 0 1)
```

In the case of a zero restriction, we can find a corresponding coboundary.

```
> H:=sub< G | G.2, G.2^(G.1*G.2*G.1) >;
> #H;
21
```

```
> CH := Restriction(CG, H);
> CohomologyGroup(CH, 1); CohomologyGroup(CH, 2);
Full Vector space of degree 0 over GF(2)
Full Vector space of degree 0 over GF(2)
> t:=TwoCocycle(CG,[1]);
> isc, o := IsTwoCoboundary(CH, t);
> isc;
true
> forall{ <h,k> : h in H, k in H | t(<h,k>) eq
>            o(<h>)*MatrixOfElement(CH,k) + o(<k>) - o(<h*k>) };
true
```

68.7 Other Operations on Cohomology Modules

`CorestrictionMapImage(G, C, c, i)`

`CorestrictCocycle(G, C, c, i)`

Given an i-cochain c for the cohomology module C which has to be defined wrt. to some subgroup U of G, return the corestriction of c to $H^i(G, \ldots)$.

`InflationMapImage(M, c)`

`LiftCocycle(M, c)`

`NewCodomain`	ANY	*Default :* `false`
`Level`	RNGINTELT	*Default :* `false`

Given a cochain $c : G^i \to X$ and a (transversal) map $H \to G$, return the inflation (lift) of c to H, ie. a cochain $d : H^i \to X$ defined by $d(h) := c(M(h))$. If `Level` is given c is assumed to be in the cohomology group of that level, ie. i :=`Level`. If `Level` is not specified, MAGMA tries its best to guess the correct level.

If `NewCodomain` is given, the values of d are coerced into this structure.

`CoboundaryMapImage(M, i, c)`

For a cohomology module M, a level i and a i-cochain c (as a user program), return a $i+1$-coboundary as obtained from the cohomological coboundary operator.

68.8 Constructing Extensions

`Extension(CM, s)`

Given the cohomology module CM for the group G acting on the module M and an element s of $H^2(G, M)$, this function returns the corresponding extension E of the module M by G as a finitely presented group. The generators of E are chosen so that the generators of the acting group G (or rather strong generators for G when G is a permutation or matrix group) come first, and the generators of M come last. The argument s should be either an element of $H^2(G, M)$ or a sequence of integers defining such an element.

The projection from E to G and the injection from an abelian group isomorphic to M to E are also returned.

This function may only be applied when the module M used to define CM is defined over a finite field of prime order, the integers, or as an abelian group in a call of `CohomologyModule(G, Q, T)`.

`SplitExtension(CM)`

Given the cohomology module CM for the group G acting on the module M, this function returns the split extension E of the module M by G as a finitely presented group. The generators of E are chosen so that the generators of the acting group G (or strong generators for G when G is a permutation or matrix group) come first, and the generators of M come last. The extension returned is the same as for `Extension(CM, s)` with s taken as the zero element of $H^2(G, M)$, but `SplitExtension` is much faster, and does not require $H^2(G, M)$ to be calculated first. This function will also work when the group used to define CM was a finitely presented group.

The projection from E to G and the injection from an abelian group isomorphic to M to E are also returned.

This function may only be applied when the module M used to define CM is defined over a finite field of prime order, the integers, or as an abelian group in a call of `CohomologyModule(G, Q, T)`.

`pMultiplicator(G, p)`

Given the permutation group G and a prime p dividing the order of G, return the invariant factors of the p-part of the Schur multiplicator of G. This function calls Derek Holt's original cohomology code (see [Hol84]).

`pCover(G, F, p)`

Given the permutation group G and the finitely presented group F such that G is an epimorphic image of F in the sense described below, and a prime p, return a presentation for the p-cover of G, constructed as an extension of the p-multiplier by F. Note that the epimorphism of F onto G must satisfy the conditions that, firstly, the generators of F are in one-to-one correspondence with those of G and, secondly, the relations of F are satisfied by the generators of G. In other words, the mapping

taking the i-th generator of F to the i-th generator of G must be an epimorphism. Usually this mapping will be an isomorphism, although this is not mandatory. This function calls Derek Holt's original cohomology code (see [Hol85a]).

Example H68E6

We apply the machinery to construct a non-split extension of the elementary abelian group of order 3^8 by A_8.

```
> G := Alt(8);
> M := PermutationModule(G,GF(3));
> X := CohomologyModule(G,M);
> C := CohomologyGroup(X,2);
> C;
Full Vector space of degree 1 over GF(3)
```

The function `Extension` is used to construct a non-split extension E of the module M by the group G.

```
> E := Extension(X,[1]);
```

The object E is a finitely presented group, in which the 8 module generators come last. We now construct a (rather large-degree) faithful permutation representation of E.

```
> n := Ngens(E);
> D := sub< E | [E.i : i in [n-7..n-1]] >;
> ct := CosetTable(E,D:CosetLimit:=10^6,Hard:=true);
> P := CosetTableToPermutationGroup(E,ct);
> Degree(P);
60480
> #P eq 3^8 * #G;
true
```

We extract the normal subgroup of order 3^8 of the extension E, and verify that the extension is non-split.

```
> Q := sub<P | [P.i : i in [n-7..n]] >;
> #Q eq 3^8;
true
> IsNormal(P,Q);
true
> Complements(P,Q);
[]
```

Example H68E7

We investigate the cohomology of the permutation module for A_5 taken over the integers.

```
> G := Alt(5);
> M := PermutationModule(G,Integers());
> X := CohomologyModule(G,M);
```

```
> CohomologyGroup(X,0);
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 0 ]
> CohomologyGroup(X,1);
Full Quotient RSpace of degree 0 over Integer Ring
Column moduli:
[ ]
> CohomologyGroup(X,2);
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 3 ]
```

While we can form extensions of M in this case, we are unable to determine the distinct extensions.

```
> E := Extension(X,[1]);
> E;
Finitely presented group E on 8 generators
Relations
    (E.4, E.5) = Id(E)
    (E.4, E.6) = Id(E)
    (E.4, E.7) = Id(E)
    (E.4, E.8) = Id(E)
    (E.5, E.6) = Id(E)
    (E.5, E.7) = Id(E)
    (E.5, E.8) = Id(E)
    (E.6, E.7) = Id(E)
    (E.6, E.8) = Id(E)
    (E.7, E.8) = Id(E)
    (E.1, E.4^-1) = Id(E)
    (E.1, E.5^-1) = Id(E)
    E.1^-1 * E.6 * E.1 * E.7^-1 = Id(E)
    E.1^-1 * E.7 * E.1 * E.8^-1 = Id(E)
    E.1^-1 * E.8 * E.1 * E.6^-1 = Id(E)
    E.2^-1 * E.4 * E.2 * E.5^-1 = Id(E)
    E.2^-1 * E.5 * E.2 * E.6^-1 = Id(E)
    E.2^-1 * E.6 * E.2 * E.4^-1 = Id(E)
    (E.2, E.7^-1) = Id(E)
    (E.2, E.8^-1) = Id(E)
    (E.3, E.4^-1) = Id(E)
    E.3^-1 * E.5 * E.3 * E.6^-1 = Id(E)
    E.3^-1 * E.6 * E.3 * E.7^-1 = Id(E)
    E.3^-1 * E.7 * E.3 * E.5^-1 = Id(E)
    (E.3, E.8^-1) = Id(E)
    E.1^-3 * E.4^-1 * E.5^-2 = Id(E)
    (E.1^-1 * E.3^-1)^2 = Id(E)
    E.3^-3 * E.4 * E.8^2 = Id(E)
    E.2^-1 * E.1 * E.3^-1 * E.2 * E.1^-1 * E.4^-1 * E.8^2 = Id(E)
    E.2 * E.3 * E.2 * E.3 * E.8^-4 = Id(E)
```

```
    E.2^-1 * E.3^-1 * E.2^2 * E.3^-1 * E.4 * E.5 * E.6^-2 * E.7 = Id(E)
> DE := DistinctExtensions(X);
Sorry, can only compute distinct extensions over prime field or finite abelian
group
```

68.9 Constructing Distinct Extensions

The functions below compute the distinct extensions of one group by another.

`DistinctExtensions(CM)`

Given the cohomology module CM for the group G acting on the module M, this function returns a sequence containing all of the distinct extensions of the module M by G, each in the form returned by `Extension(CM, s)`. Two such extensions E_1, E_2 are regarded as being distinct if there is no group isomorphism from one to the other that maps the subgroup of E_1 corresponding to M to the subgroup of E_2 corresponding to M.

This function may only be applied when the module M used to define CM is defined over a finite field of prime order, the integers, or as an abelian group in a call of `CohomologyModule(G, Q, T)`.

Example H68E8

We consider the extensions of the trivial module over $GF(2)$ by the group $Z_2 \times Z_2$.

```
> G := DirectProduct(CyclicGroup(2),CyclicGroup(2));
> M := TrivialModule(G,GF(2));
> C := CohomologyModule(G,M);
> CohomologicalDimension(C,2);
3
> D := DistinctExtensions(C);
> #D;
4
```

So there are $2^3 = 8$ equivalence classes of extensions. But only four are distinct up to an isomorphism fixing the module. To examine them, we form permutation representations:

```
> DP := [ CosetImage(g,sub<g|>) : g in D ];
> [IsAbelian(d): d in DP];
[ true, true, false, false ]
// the first two are abelian
> [IsIsomorphic(d,DihedralGroup(4)) : d in DP];
[ false, false, true, false ]
// The third one is dihedral
> #[g : g in DP[4] | Order(g) eq 4];
6
```

So the fourth group must be the quaternion group.

`ExtensionsOfElementaryAbelianGroup(p, d, G)`

Given a prime p, a positive integer d, and a permutation group G, this function returns a list of finitely presented groups which are isomorphic to the distinct extensions of an elementary abelian group N of order p^d by G. Two such extensions E_1 and E_2 with normal subgroups N_1 and N_2 isomorphic to N are considered to be distinct if there is no group isomorphism $G_1 \to G_2$ that maps N_1 to N_2. Each extension E is defined on $d+r$ generators, where r is the number of generators of G. The last d of these generators generate the normal subgroup N, and the quotient of E by N is a presentation of G on its own generators.

Example H68E9

We form the distinct extensions of the elementary abelian group $Z_2 \times Z_2$ by the alternating group A_4.

```
> E := ExtensionsOfElementaryAbelianGroup(2,2,Alt(4));
> #E;
4
```

So there are four distinct extensions of an elementary group of order 4 by A_4

```
> EP := [ CosetImage(g,sub<g|>) : g in E ];
> [#Centre(e): e in EP];
[ 1, 1, 4, 4 ]
```

The first two have nontrivial action on the module. The module generators in the extensions come last, so these will be $e.3$ and $e.4$. We can use this to test which of the extensions are non-split.

```
> [ Complements(e,sub<e|e.3,e.4>) eq [] : e in EP];
[ false, true, false, true ]
> AbelianInvariants(Sylow(EP[2],2));
[ 4, 4 ]
```

So the first and fourth extensions split and the second and third do not. $EP[2]$ has a normal abelian subgroup of type $[4, 4]$.

`ExtensionsOfSolubleGroup(H, G)`

Given permutation groups G and H, where H is soluble, this function returns a sequence of finitely presented groups, the terms of which are isomorphic to the distinct extensions of H by G. Two such extensions E_1 and E_2 with normal subgroups H_1 and H_2 isomorphic to H are considered to be distinct if there is no group isomorphism $G_1 \to G_2$ that maps H_1 to H_2. Each extension E is defined on $d + r$ generators, where the last d generators generate the normal subgroup H, and the quotient of E by H is a presentation for G on its own generators. (The last d generators of E do not correspond to the original generators of H, but to a PC-generating sequence for H.)

Example H68E10

How many extensions are there of a dihedral group of order 8 by itself? This calculation is currently rather slow.

```
> D4 := DihedralGroup(4);
> time S := ExtensionsOfSolubleGroup(D4, D4);
Time: 120.210
> #S;
20
> ES := [CosetImage(g,sub<g|>) : g in S ];
> [#Centre(g): g in ES];
[ 4, 2, 4, 2, 4, 2, 2, 4, 2, 4, 4, 2, 4, 2, 4, 2, 4, 2, 4, 2 ]
> [NilpotencyClass(g) : g in ES ];
[ 2, 3, 2, 3, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
> [Exponent(g): g in ES];
[ 4, 8, 4, 8, 4, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8 ]
```

Example H68E11

We determine the distinct extensions of the abelian group with invariants $[2, 4, 4]$ by the cyclic group of order 4.

```
> Z := Integers();
> G := PermutationGroup<4 | (1,2,4,3)>;
> Q := [2, 4, 4];
> T := [ Matrix(Z,3,3,[1,2,0,0,0,1,0,1,2]) ];
> CM := CohomologyModule(G, Q, T);
> extns := DistinctExtensions(CM);
> extns;
[
    Finitely presented group on 4 generators
    Relations
        $.2^2 = Id($)
        $.3^4 = Id($)
        $.4^4 = Id($)
        ($.2, $.3) = Id($)
        ($.2, $.4) = Id($)
        ($.3, $.4) = Id($)
        $.1^-1 * $.2 * $.1 * $.3^-2 * $.2^-1 = Id($)
        $.1^-1 * $.3 * $.1 * $.4^-1 = Id($)
        $.1^-1 * $.4 * $.1 * $.4^-2 * $.3^-1 = Id($)
        $.1^4 = Id($),
    Finitely presented group on 4 generators
    Relations
        $.2^2 = Id($)
        $.3^4 = Id($)
        $.4^4 = Id($)
        ($.2, $.3) = Id($)
```

```
        ($.2, $.4) = Id($)
        ($.3, $.4) = Id($)
        $.1^-1 * $.2 * $.1 * $.3^-2 * $.2^-1 = Id($)
        $.1^-1 * $.3 * $.1 * $.4^-1 = Id($)
        $.1^-1 * $.4 * $.1 * $.4^-2 * $.3^-1 = Id($)
        $.1^4 * $.2^-1 * $.3^-1 * $.4^-3 = Id($),
    Finitely presented group on 4 generators
    Relations
        $.2^2 = Id($)
        $.3^4 = Id($)
        $.4^4 = Id($)
        ($.2, $.3) = Id($)
        ($.2, $.4) = Id($)
        ($.3, $.4) = Id($)
        $.1^-1 * $.2 * $.1 * $.3^-2 * $.2^-1 = Id($)
        $.1^-1 * $.3 * $.1 * $.4^-1 = Id($)
        $.1^-1 * $.4 * $.1 * $.4^-2 * $.3^-1 = Id($)
        $.1^4 * $.3^-2 * $.4^-2 = Id($)
]
```

Since the extensions are soluble groups, we construct pc-presentations of each and verify that no two of the groups are isomorphic.

```
> E1 := SolubleQuotient(extns[1]);
> E2 := SolubleQuotient(extns[2]);
> E3 := SolubleQuotient(extns[3]);
> IsIsomorphic(E1, E2);
false
> IsIsomorphic(E1, E3);
false
> IsIsomorphic(E2, E3);
false
```

`IsExtensionOf(G)`

`Degree`	RngInt	*Default :* 0
`MaxId`	RngInt	*Default :* 15
`DegreeBound`	RngInt	*Default :* ∞

For a given permutation group G, find normal abelian subgroup $A < G$ such that G can be obtained by extending G/A by A. The function returns a sequence of tuples T containing

- the cohomology module of G/A acting on A
- the 2-cocyle as an element in $H^2(G/A, A)$ corresponding to G
- the actual 2-cocyle as a user defined function

- a pair $\langle a, b\rangle$ giving the degree a of the transitive group G/A and the number b identifying the group in the data base. If b is larger than 20 (or `MaxId`) the hash value of the group is returned instead.
- the abelian invariants of A
- a set containing all pairs $\langle a, b\rangle$ such that ${}_aT_b$ can be obtained through this extension process.

If `DegreeBound` is given, only subgroups A are considered such that G/A has less than `DegreeBound` many elements. The list considered contains only sugroups that are maximal under the restrictions. If `Degree` is given, G/A must have exactly `Degree` many elements.

`IsExtensionOf(L)`

`Degree`	RngInt	*Default :* 0
`MaxId`	RngInt	*Default :* 15
`DegreeBound`	RngInt	*Default :* ∞

For all groups G in L, `IsExtensionOf` is called. The first sequence returned contains tuples as in `IsExtensionOf` above. The sequence is minimal such that all groups in L can be generated using the cohomology modules in the sequence. The second return value contains a set of pairs $\langle a, b\rangle$ describing all transitive groups that can be obtained through the processes.

68.10 Finite Group Cohomology

This section describes Magma functions for computing the first cohomology group of a finite group with coefficient in a finite (not necessarily abelian) group. These functions are based on [Hal05].

Let Γ be a group. A group A on which Γ acts by group automorphisms from the right, is called a Γ-*group*. Given a Γ-group A, define

$$H^0(\Gamma, A) := \{a \in A \mid a^\sigma = a \text{for all } \sigma \in \Gamma\}.$$

A 1-*cocycle of* Γ *on* A is a map

$$\alpha : \Gamma \to A, \quad \sigma \mapsto \alpha_\sigma,$$

such that

$$\alpha_{\sigma\tau} = (\alpha_\sigma)^\tau \alpha_\tau \quad \text{for all } \sigma, \tau \in \Gamma.$$

Two cocycles α, β on A are called *cohomologous* (with respect to a) if there exists $a \in A$, such that $\beta_\sigma = a^{-\sigma} \cdot \alpha_\sigma \cdot a$ for all $\sigma \in \Gamma$. Note that being cohomologous is an equivalence relation.

We denote by $Z^1(\Gamma, A)$ the set of all 1-cocycles of Γ on A. We denote by $[\alpha]$ the equivalence class of α and by $H^1(\Gamma, A)$ the set of equivalence classes of 1-cocycles.

$Z^1(\Gamma, A)$ and $H^1(\Gamma, A)$ are *pointed sets*.

The constant map $t : \sigma \mapsto 1$ is the distinguished element of $Z^1(\Gamma, A)$, called the *trivial* 1-*cocycle*. Its cohomology class is the distinguished element of $H^1(\Gamma, A)$.

A twisted form A_β of A by the cocycle $\beta \in Z^1(\Gamma, A)$ is the same group A but with a different action of Γ on it, given by

$$a * \sigma := a^{\sigma\alpha_\sigma} \quad \text{for } \sigma \in \Gamma \text{ and } a \in A.$$

68.10.1 Creation of Gamma-groups

This section describes intrinsics dealing with cocycles and the first cohomology.

`GammaGroup(Gamma, A, action)`

Given a group A and a group Γ acting on it by the map *action*, return the object of type `GGrp`, which is the Group A together with this particular action of Γ. The map *action* must be a homomorphism from Γ to the automorphism group of A.

If B is a normal subgroup of A and normalised by the action of Γ on A (thus a Γ-group itself), then the action of Γ on A induces in the natural way to A/B. It is possible to create such a group:

`InducedGammaGroup(A, B)`

Given a Γ-group A and a normal subgroup B normalised by the action of Γ, return the induced Γ-group A/B.

Example H68E12

Let Γ act on A by conjugation:

```
> A := SymmetricGroup(4);
> Gamma := sub<A|(1,2,3), (1,2)>;
> action := hom< Gamma -> Aut(A) |
>             g :-> iso< A -> A | a :-> a^g, a :-> a^(g^-1) > >;
> A := GammaGroup( Gamma, A, action );
> A;
Gamma-group:  Symmetric group acting on a set of cardinality 4
Order = 24 = 2^3 * 3
(1, 2, 3, 4)
(1, 2)
Gamma-action: Mapping from: GrpPerm: $, Degree 4 to
Set of all automorphisms of GrpPerm: $, Degree 4, Order 2^3 * 3
given by a rule [no inverse]
Gamma:        Permutation group acting on a set of cardinality 4
(1, 2, 3)
(1, 2)
>
```

and B be a normal subgroup of A:

```
> B := AlternatingGroup(4);
```

```
> AmodB := InducedGammaGroup( A, B );
> AmodB;
Gamma-group:  Symmetric group acting on a set of cardinality 2
Order = 2
(1, 2)
(1, 2)
Gamma-action: Mapping from: GrpPerm: $, Degree 4, Order 2 * 3 to
Set of all automorphisms of GrpPerm: $, Degree 2, Order 2
given by a rule [no inverse]
Gamma:        Permutation group acting on a set of cardinality 4
Order = 6 = 2 * 3
(1, 2, 3)
(1, 2)
Induced from another Gamma-group
```

`IsNormalised(B, action)`

Returns `true` if the group B is normalised by the action *action*, where *action* is as above.

`IsInduced(AmodB)`

Returns `true` iff the Γ-group *AmodB* was created as an induced Γ-group. If it is, then the Γ-groups A, B, the projection and representative maps are returned as well.

68.10.2 Accessing Information

`Group(A)`

Returns the group A as a `Grp` object to be used in MAGMA.

`GammaAction(A)`

Returns the action of Γ on A as a map.

`ActingGroup(A)`

Returns the group Γ acting on A.

68.10.3 One Cocycles

`OneCocycle(A, imgs)`

`OneCocycle(A, alpha)`

`Check` BoolElt *Default :* `true`

If the map $\alpha : \Gamma \to A$ or the sequence *imgs* of images of the generators $\Gamma.1, ..., \Gamma.n$ defines a 1-cocycle, return the 1-cocycle. By default, the map is checked to define a 1-cocycle. If it doesn't, `OneCocycle` will abort with an error. This check can be disabled by setting the optional argument `Check` to `false`.

`TrivialOneCocycle(A)`

Return the trivial 1-cocycle.

`IsOneCocycle(A, imgs)`

`IsOneCocycle(A, alpha)`

Return true if the map $\alpha : \Gamma \to A$ or the sequence *imgs* of images of the generators $\Gamma.1, ..., \Gamma.n$ defines a 1-cocycle and `false` otherwise. If `true`, return the cocycle as the second argument.

Note that `IsOneCocycle` does not abort with an error in contrast to `OneCocycle` if the map does not define a cocycle.

`AreCohomologous(alpha, beta)`

Return `true` if and only if the 1-cocycles α and β are cohomologous. If they are, return the intertwining element as the second return value.

`CohomologyClass(alpha)`

Return the cohomology class of the 1-cocycle α.

`InducedOneCocycle(AmodB, alpha)`

`InducedOneCocycle(A, B, alpha)`

Given a 1-cocycle on A, return the induced 1-cocycle on $AmodB$. The second version will generate the induced Γ-group A/B first.

`ExtendedOneCocycle(alpha)`

`OnlyOne` BoolElt *Default :* `false`

Given a 1-cocycle on an induced Γ-group A/B, return the set of all non-cohomologous 1-cocycles on A, which induce to α. If the optional argument `OnlyOne` is `true`, the set will contain at most one 1-cocycle. If α is not extendible, the returned set is empty.

`ExtendedCohomologyClass(alpha)`

Given a 1-cocycle on an induced Γ-group A/B, return the the set of all non-cohomologous 1-cocycles on A, which induce to a cocycle in the cohomology class of α. If no such cocycles on A exist, the returned set is empty.

`GammaGroup(alpha)`

Return the Γ-group on which α is defined.

`CocycleMap(alpha)`

Return the `Map` object corresponding to α.

68.10.4 Group Cohomology

`Cohomology(A, n)`

Given a finite group A and an integer n (currently restricted to being 1) return the n-th cohomology group $H^n(\Gamma, A)$. Since the group A is not assumed to be abelian, only $n = 0, 1$ can be used. Currently, only $n = 1$ implemented. (The zero cohomology of A is the subgroup of A centralised by Γ and can be constructed using group theoretical methods available in Magma.)

`OneCohomology(A)`

Return the first cohomology $H^1(\Gamma, A)$. as a set of representatives of all cohomology classes. If the group A is abelian, existing code by Derek Holt is used (see Chapter 68). Otherwise use [Hal05].

`TwistedGroup(A, alpha)`

Given the Γ-group A and a 1-cocycle α on it, return the twisted group A_α.

Example H68E13

First, we create the group $A = D_8$. The returned group is the usual permutation group on the octagon. Γ is the Normaliser of A in S_8 and is acting by conjugation.

```
> A := DihedralGroup(8);
> Gamma := sub< Sym(8) | (1, 2, 3, 4, 5, 6, 7, 8),
>    (1, 8)(2, 7)(3, 6)(4, 5), (2, 4)(3, 7)(6, 8) >;
> A^Gamma eq A;
true
> Gamma;
Permutation group Gamma acting on a set of cardinality 8
Order = 32 = 2^5
    (1, 2, 3, 4, 5, 6, 7, 8)
    (1, 8)(2, 7)(3, 6)(4, 5)
    (2, 4)(3, 7)(6, 8)
> action := hom< Gamma -> Aut(A) |
>             g :-> iso< A -> A | a :-> a^g, a :-> a^(g^-1) > >;
```

```
> A := GammaGroup( Gamma, A, action );
```

Now let B be the center of A and create the induced Γ-group A/B:

```
> B := Center(Group(A));
> AmodB := InducedGammaGroup(A, B);
```

Create the trivial 1-cocycle on A/B and compute its cohomology class:

```
> triv := TrivialOneCocycle(AmodB);
> CohomologyClass( triv );
{@
    One-Cocycle
    defined by [
    Id($),
    Id($),
    Id($)
    ],
    One-Cocycle
    defined by [
    Id($),
    (1, 4)(2, 7)(3, 8)(5, 6),
    (1, 4)(2, 7)(3, 8)(5, 6)
    ],
    One-Cocycle
    defined by [
    (1, 4)(2, 7)(3, 8)(5, 6),
    Id($),
    (1, 4)(2, 7)(3, 8)(5, 6)
    ],
    One-Cocycle
    defined by [
    (1, 4)(2, 7)(3, 8)(5, 6),
    (1, 4)(2, 7)(3, 8)(5, 6),
    Id($)
    ]
@}
```

Pick one of the cocycles in this class and compute the intertwining element:

```
> alpha := Random($1);alpha;
One-Cocycle
defined by [
(1, 4)(2, 7)(3, 8)(5, 6),
(1, 4)(2, 7)(3, 8)(5, 6),
Id($)
]
> bo, a := AreCohomologous(alpha,triv);
> bo; a;
true
```

```
(1, 5)(2, 8)(3, 7)(4, 6)
```

Now create another cocycle on A/B and extend it to A:

```
> alpha := OneCocycle( AmodB,
>                [Group(AmodB)| (1, 7, 4, 2)(3, 5, 8, 6),
>                               (1, 2, 4, 7)(3, 6, 8, 5),
>                               1 ] );
> ExtendedOneCocycle(alpha);
{
    One-Cocycle
    defined by [
    (1, 4, 7, 2, 5, 8, 3, 6),
    (1, 2, 3, 4, 5, 6, 7, 8),
    Id($)
    ],
    One-Cocycle
    defined by [
    (1, 8, 7, 6, 5, 4, 3, 2),
    (1, 6, 3, 8, 5, 2, 7, 4),
    Id($)
    ]
}
```

Pick a cocycle β in this set and check if it really induces to α:

```
> beta := Rep($1);
> InducedOneCocycle(AmodB, beta) eq alpha;
true
```

Finally, create the twisted group A_β:

```
> A_beta := TwistedGroup(A, beta);
> A_beta;
Gamma-group:  Permutation group acting on a set of cardinality 8
Order = 16 = 2^4
(1, 2, 3, 4, 5, 6, 7, 8)
(1, 8)(2, 7)(3, 6)(4, 5)
Gamma-action: Mapping from: GrpPerm: $, Degree 8, Order 2^5 to
Set of all automorphisms of GrpPerm: $, Degree 8, Order 2^4
given by a rule [no inverse]
Gamma:        Permutation group acting on a set of cardinality 8
Order = 32 = 2^5
(1, 2, 3, 4, 5, 6, 7, 8)
(1, 8)(2, 7)(3, 6)(4, 5)
(2, 4)(3, 7)(6, 8)
>
```

68.11 Bibliography

[**CCH01**] J.J. Cannon, B. Cox, and D.F. Holt. Computing the subgroup lattice of a permutation group. *J. Symbolic Comp.*, 31:149–161, 2001.

[**Hal05**] Sergei Haller. *Computing Galois Cohomology and Forms of Linear Algebraic Groups*. Phd thesis, Technical University of Eindhoven, 2005.

[**Hol84**] D.F. Holt. The calculation of the Schur multiplier of a permutation group. In *Computational group theory (Durham, 1982)*, pages 307–319. Academic Press, London, 1984.

[**Hol85a**] D.F. Holt. A computer program for the calculation of a covering group of a finite group. *J. Pure Appl. Algebra*, 35(3):287–295, 1985.

[**Hol85b**] D.F. Holt. The mechanical computation of first and second cohomology groups. *J. Symbolic Comp.*, 1(4):351–361, 1985.

PART X
FINITELY-PRESENTED GROUPS

69 ABELIAN GROUPS

Chapter 69
ABELIAN GROUPS

69.1 Introduction

This Chapter contains a description of the MAGMA machinery provided for computing with abstract abelian groups. Such a group may be described either in terms of defining relations (category `GrpAb`) or by defining the group operations on some finite set (category `GrpAbGen`). In the case of finitely presented groups, the abelian groups may be finite and infinite, the only restriction being that the group be finitely generated.

69.2 Construction of a Finitely Presented Abelian Group and its Elements

69.2.1 The Free Abelian Group

`FreeAbelianGroup(n)`

Construct the free abelian group F on n generators, where n is a positive integer. The i-th generator may be referenced by the expression `F.i`, $i = 1, \ldots, n$. Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of F. In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the *left hand side* of the assignment statement.

Example H69E1

The statement

```
> F := FreeAbelianGroup(2);
```

creates the free abelian group on two generators. Here the generators may be referenced using the standard names, $F.1$ and $F.2$.
The statement

```
> F<x, y> := FreeAbelianGroup(2);
```

defines F to be the free abelian group on two generators and assigns the names x and y to the generators.

69.2.2 Relations

`w1 = w2`

Given words w_1 and w_2 over the generators of an abelian group A, create the relation $w_1 = w_2$. Note that this relation is not automatically added to the existing set of defining relations R for S. It may be added to R, for example, through use of the `quo`-constructor (see below).

`r[1]`
`LHS(r)`

Given a relation r over the generators of A, return the left hand side of the relation r. The object returned is a word over the generators of A.

`r[2]`
`RHS(r)`

Given a relation r over the generators of A, return the right hand side of the relation r. The object returned is a word over the generators of A.

`Parent(r)`

Group over which the relation r is taken.

Example H69E2

We may define a group and a set of relations as follows:

```
> F<x, y> := FreeAbelianGroup(2);
> rels := { 2*x = 3*y, 4*x + 4*y = Id(F)} ;
```

To replace one side of a relation, the easiest way is to reassign the relation. So for example, to replace the relation $2x = 3y$ by $2x = 4y$:

```
> r := 2*x = 3*y;
> r := LHS(r) = 4*y;
```

69.2.3 Specification of a Presentation

An abelian group with non-trivial relations is constructed as a quotient of an existing abelian group, possibly a free abelian group.

```
AbelianGroup<  X | R  >
```

Given a list X of variables $x_1, \ldots, x_r$, and a list of relations R over these generators, let F be the free abelian group on the generators $x_1, \ldots, x_r$. Then construct (a) an abelian group A isomorphic to the quotient of F by the subgroup of F defined by R, and (b) the natural homomorphism $\phi : F \to A$.

Each term of the list R is either a *word*, a *relation*, a *relation list* or a *subgroup* of F.

- A *relation* consists of a pair of words, separated by '='.
- A *word* w is interpreted as a *relator*, that is, it is equivalent to the relation $w = 0$. (See above).
- A *relation list* consists of a list of words, where each pair of adjacent words is separated by '=': $w_1 = w_2 = \cdots = w_r$. This is interpreted as the set of relations $w_1 = w_r, \ldots, w_{r-1} = w_r$. Note that the relation list construct is only meaningful in the context of the `quo`-constructor.

A *subgroup* H appearing in the list R contributes its generators to the relation set for A, i.e., each generator of H is interpreted as a relator for A.

The group F may be referred to by the special symbol `$` in any word appearing to the right of the '`|`' symbol in the `quo`-constructor. Also, in the context of the `quo`-constructor, the identity element (empty word) may be represented by the digit 0. The function returns:

(a) The quotient group A;

(b) The natural homomorphism $\phi : F \to A$.

Example H69E3

We create the abelian group defined by the presentation $< a, b, c \mid 7a + 4b + c, 8a + 5b + 2c, 9a + 6b + 3c >$.

```
> F<a, b, c> := FreeAbelianGroup(3);
> A := quo< F | 7*a + 4*b + c, 8*a + 5*b + 2*c, 9*a + 6*b + 3*c >;
> A;
AbelianGroup isomorphic to Z_3 + Z
Defined on 2 generators
Relations:
    3*A.1 = 0
```

A simple way of specifying an abelian group is as a product of cyclic groups.

`AbelianGroup([n1,...,nr])`

Construct the abelian group defined by the sequence $[n_1, \ldots, n_r]$ of non-negative integers as an abelian group. The function returns the direct product of cyclic groups $C_{n_1} \times C_{n_2} \times \cdots \times C_{n_r}$, where C_0 is interpreted as an infinite cyclic group.

Example H69E4

We create the abelian group $Z_2 \times Z_3 \times Z_4 \times Z_5 \times Z_6 \times Z \times Z$.

```
> G<[x]> := AbelianGroup([2,3,4,5,6,0,0]);
> G;
Abelian Group isomorphic to Z/2 + Z/6 + Z/60 + Z + Z
Defined on 7 generators
Relations:
    2*G.1 = 0
    3*G.2 = 0
    4*G.3 = 0
    5*G.4 = 0
    6*G.5 = 0
```

69.2.4 Accessing the Defining Generators and Relations

The functions described here provide access to basic information stored for an abelian group A.

`A . i`

The i-th defining generator for A.

`Generators(A)`

A set containing the generators for A.

`NumberOfGenerators(A)`

`Ngens(A)`

The number of generators for A.

`Parent(u)`

The parent group A of the word u.

`Relations(A)`

A sequence containing the defining relations for A.

`RelationMatrix(A)`

A matrix where each row corresponds to one of the defining relations of A.

69.3 Construction of a Generic Abelian Group

The term generic abelian group refers to the situation in which a group A is described by giving a set X together with functions acting on X that perform the fundamental group operations for A. Specifically, the user must provide functions which compute the product, the inverse and the identity. For efficiency, the user may also optionally specify the order and a generating set for the group. This is made explicit below.

Going from such a definition of an abelian group A to a presentation for A will often be extremely expensive and so a small number of operations are implemented so as to not require this information. The two key operations are computing the order of an element and computing the discrete logarithm of an element to a given base. For many abelian group operations, knowledge of a presentation is required and if such an operation is invoked, the structure of A (and hence a presentation) will be automatically constructed.

There are two possible ways of computing the structure of the group. If the order of the group is known (or can be computed) then it is relatively easy to construct each of the p-Sylow subgroups. If the order is not available, the structure is computed from a set of generators supplied by the user.

69.3.1 Specification of a Generic Abelian Group

`GenericAbelianGroup(U: parameters)`

`IdIntrinsic`	MonStg	*Default :*
`AddIntrinsic`	MonStg	*Default :*
`InverseIntrinsic`	MonStg	*Default :*
`UseRepresentation`	Bool	*Default :* `false`
`Order`	RngIntElt	*Default :*
`UserGenerators`	SetEnum	*Default :*
`ProperSubset`	Bool	*Default :* `false`
`RandomIntrinsic`	MonStg	*Default :*
`ComputeStructure`	Bool	*Default :* `false`

Construct the generic abelian group A over the domain U. The domain U can be an aggregate (set, indexed set or sequence) of elements or it can be any structure, for example, an elliptic curve, a jacobian, a magma of quadratic forms.

If the parameters `IdIntrinsic`, `AddIntrinsic` and `InverseIntrinsic` are not set, then the identity, the composition and the inverse function of A are taken to be the identity, the composition and the inverse function of U or of `Universe(`U`)` if U is an aggregate. If the parameters `IdIntrinsic`, `AddIntrinsic` and `InverseIntrinsic` are set, they define the identity, the composition and the inverse function of A.

The parameter `IdIntrinsic` must be a function name whose sole argument is U or `Universe(`U`)` if U is an aggregate. `AddIntrinsic` must be a function name whose only two arguments are elements of U or of `Universe(`U`)` if U is

an aggregate. `InverseIntrinsic` must be a function name whose only two arguments are elements of U or of `Universe(`U`)` if U is an aggregate. That is, it is required that `InverseIntrinsic` be a binary operation (see the example below where `InverseIntrinsic := "/"` is indeed binary). Defining any of the three above parameters implies that the other two must be defined as well.

Setting the parameter `UseRepresentation` to true implies that all elements of A will be internally represented as linear combinations of the generators of A obtained while computing the structure of A. This can be a costly procedure, since such a representation is akin to solving the discrete logarithm problem. The advantage of such a representation is that arithmetic for elements of A as well as computing the order of elements of A are then essentially trivial operations.

The parameter `Order` can be set to the order of the group, if known. Knowledge of the order may save a considerable amount of time for operations such as computing a Sylow subgroup, determining the group structure or solve a discrete logarithm problem. More importantly, if A is a proper subset of U, or of `Universe(`U`)` if U is an aggregate, then one of `Order` or `UserGenerators` must be set.

One can set `UserGenerators` to some set of elements of U, or of `Universe(`U`)` if U is an aggregate, which generate the group A. This can be useful when A is a subset of U (`Universe(`U`)`), or more generally when the computation of the group structure of A is made from a set of generators. Finally, setting `UserGenerators` may be an alternative to setting `RandomIntrinsic`.

The parameter `ProperSubset` must be set when A is a proper subset of U (`Universe(`U`)`).

The parameter `RandomIntrinsic` indicates the random function to use. If it is not set, the random function (if it is required) is taken to be the random function applying to U (`Universe(`U`)`).

The parameter `RandomIntrinsic` must be the name of a function taking as its sole argument U (`Universe(`U`)`) and returning an element of U (`Universe(`U`)`) which is also an element of the group A, which is important when A is a proper subset of U (`Universe(`U`)`. Therefore, if A is a proper subset of U (`Universe(`U`)`), then `RandomIntrinsic` must be set, unless the parameter `UserGenerators` is set.

The parameter `Structure` indicates that the group structure should be determined at the time of creation. If this parameter is set then the user may also want to set the following parameters:

`UseUserGenerators`	BOOL	*Default :* `false`
`PollardRhoRParam`	RNGINT	*Default :* 20
`PollardRhoTParam`	RNGINT	*Default :* 8
`PollardRhoVParam`	RNGINT	*Default :* 3

Their meaning is detailed in Section 69.3.3.

Example H69E5

In our first example we create the subset U of $\mathbf{Z}/34384\mathbf{Z}$ corresponding to its unit group as a

generic abelian group G. Note that U is an indexed set.

```
> m :=  34384;
> Zm := Integers(m);
> U := {@ x : x in Zm | GCD(x, m) eq 1 @};
> G := GenericAbelianGroup(U :
>        IdIntrinsic := "Id",
>        AddIntrinsic := "*",
>        InverseIntrinsic := "/");
> G;
Generic Abelian Group over
Residue class ring of integers modulo 34384
```

In our next example we construct unique representatives for the classes of binary quadratic forms corresponding to elements of a subgroup of the class group of the imaginary quadratic field of discriminant -4000004.

```
> n := 6;
> Dk := -4*(10^n + 1);
> Q := QuadraticForms(Dk);
>
> p := 2;
> S := { };
> while #S lt 10 do
>    if KroneckerSymbol(Dk,p) eq 1 then
>       I := Reduction(PrimeForm(Q,p));
>       Include(~S, I);
>    end if;
>    p := NextPrime(p);
> end while;
>
> QF := GenericAbelianGroup(Q : UserGenerators := S,
>                               ComputeStructure := true,
>                               UseUserGenerators := true);
> QF;
Generic Abelian Group over
Binary quadratic forms of discriminant -4000004
Abelian Group isomorphic to Z/2 + Z/516
Defined on 2 generators
Relations:
  2*$.1 = 0
  516*$.2 = 0
```

So the structure of the subgroup is $Z_2 \times Z_{516}$

69.3.2 Accessing Generators

These functions give access to generating sets for a generic group A. If a generating set is requested and none has been supplied by the user then this will require the determination of the group structure which could be very expensive. Note the distinction between `Generators` and `UserGenerators`. From now on, unless otherwise specified, whenever mention is made of the generators of A, we refer to the generators as given by the `Generators` function.

`Universe(A)`

The universe over which the generic abelian group A is defined.

`A . i`

The i-th generator for the generic abelian group A.

`Generators(A)`

A sequence containing a generating set for the generic abelian group A as elements of A. The set of generators returned for A is a reduced set of generators as constructed during the structure computation. Therefore, if the group structure of A has been computed from a user-supplied set of generators, it is unlikely that `Generators(`A`)` and `UserGenerators(`A`)` will be the same.

`UserGenerators(A)`

A sequence containing the user-supplied generators for the generic abelian group A as elements of A.

`NumberOfGenerators(A)`

`Ngens(A)`

The number of generators for the generic abelian group A.

69.3.3 Computing Abelian Group Structure

If the order of a generic abelian group A can be obtained then the structure of A is found by constructing each p-Sylow subgroup of A. The p-Sylow subgroups are built from random elements of the underlying set X of A. Recall that U (`Universe(`U`)`) is the domain of A. Random elements are obtained using either a random function attached to X or using a user-supplied function (the `RandomIntrinsic` parameter), or using ser-supplied generators (the `UserGenerators` parameter). It is therefore vital that user-supplied generators truly generate the group one wishes to construct. A drawback of this method of obtaining the structure of A is the memory needed to store all the elements of a specific p-Sylow subgroup while under construction. This algorithm is mostly based on work by Michael Stoll.

The second approach computes the group structure from a set of generators as supplied by the user, removing the need to compute the order of A. This can be particularly useful when computing this order is expensive. Note that computing the structure of a group from a set of generators is akin to solving a number of the discrete logarithm problems.

This second algorithm uses a variant of the Pollard–Rho algorithm and is due to Edlyn Teske [Tes98a].

If A is a subgroup of a generic abelian group, G say, then the structure of G is computed first. The rationale is that once the structure of G is known, computing the structure of A is almost always cheap.

`AbelianGroup(A: parameters)`

`UseUserGenerators`	BOOL	*Default :* `false`
`PollardRhoRParam`	RNGINT	*Default :* 20
`PollardRhoTParam`	RNGINT	*Default :* 8
`PollardRhoVParam`	RNGINT	*Default :* 3

Compute the group structure of the generic abelian group A, which may be a subgroup as created by the subgroup constructor or the Sylow function. The two values returned are the abstract abelian group and the invertible map from the latter into A.

If `UseUserGenerators` is false, then the group structure computation is made via the construction of each p-Sylow subgroup, using the factorization of the order of A.

If `UseUserGenerators` is set to `true`, the group structure computation uses the user-supplied set of generators for A. In this case, the additional parameters `PollardRhoRParam`, `PollardRhoTParam`, and `PollardRhoVParam` can be supplied. Their values will be passed to the Pollard–Rho algorithm used in the group structure computation: `PollardRhoRParam` sets the size of the r-adding walks, `PollardRhoTParam` sets the size of the internal storage of elements, and `PollardRhoVParam` is used for an efficient finding of the periodic segment. It is conjectured that the default values as given above are 'best' (see [Tes98b]), therefore there should be no need to set these parameters in general.

Example H69E6

The following statement computes the structure of the unit group of $\mathbf{Z}_{34384}$.

```
> G := AbelianGroup(G); G;
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2 + Z/6 + Z/612
Defined on 4 generators
Relations:
  2*G.1 = 0
  2*G.2 = 0
  6*G.3 = 0
  612*G.4 = 0
```

69.4 Elements

69.4.1 Construction of Elements

Unless otherwise stated, the operations in this section apply to fp-abelian groups and generic abelian groups.

`A ! [a`$_1$`, ... ,a`$_n$`]`

Given an abelian group A with generators $e_1, \ldots, e_r$ and a sequence $Q = [a_1, \cdots, a_r]$ of integers, construct the element $a_1e_1 + \cdots + a_re_r$ of A.

`A ! e`

Given a generic abelian group A and an element e of the domain over which it is defined, return e as an element of A. If A is a proper subset of its underlying domain, then e must be a linear combination of the generators (which may be user-supplied) of A.

`A ! g`

Given a generic abelian group A and an element g of the underlying set X of A, return g as an element of A.

`A ! n`

Given an abelian group A with exactly one generator x, construct the element nx.

`Random(A)`

Given either a finite fp-abelian group or a generic abelian group A, return a random element of A.

`Identity(A)`

`Id(A)`

`A ! 0`

Construct the identity element (empty word) for the abelian group A.

Let A be a generic abelian group defined in the universe U of A. If g is an element of A, then $U!g$ is an element of U.

69.4.2 Representation of an Element

An element g of an abelian group A can be represented as a linear combination with respect to a given generating sequence. The coefficients appearing in this linear combination provide an alternative representation for g. If A is a fp-group, the generating set will be the one on which the group was defined. In the case of a generic group, the generating set can either be that obtained when constructing a presentation for A or a user-supplied generating set.

`Representation(g)`

`ElementToSequence(g)`

`Eltseq(g)`

Let A be an abelian group with generating set $e_1, \ldots, e_n$ and suppose g is an element of A, where $g = a_1e_1 + \ldots + a_ne_n$. These functions return the sequence Q of n integers defined by $Q[i] = a_i$, for $i = 1, \ldots, n$. Moreover, each a_i, $i = 1, \ldots, n$, is the integer residue modulus the order of the ith generator.

`UserRepresentation(g)`

Let A be a generic abelian group with a user-supplied set of generators $u_1, \ldots, u_n$ and suppose g is an element of A, where $g = a_1u_1 + \ldots + a_nu_n$. This function returns the sequence Q of n integers defined by $Q[i] = a_i$, for $i = 1, \ldots, n$. Moreover, each a_i, $i = 1, \ldots, n$, is the integer residue modulus the order of the ith generator.

`Representation(S, g)`

Let A be a generic abelian group and let $S = [s_1, \ldots, s_m]$ be any sequence of elements of A. Assume g is an element of A such that $bg = a_1s_1 + \ldots + a_ms_m$. This function returns as its first value the sequence Q of m integers defined by $Q[i] = a_i$, for $i = 1, \ldots, m$. The second value returned is the coefficient b of g. Note that b might not be 1.

Example H69E7

We use the quadratic forms example considered above to illustrate these functions.

```
> Generators(QF);
[ <2,2,500001>, <206,-102,4867> ]
> g := QF ! [5, 6];
> g;
<837,-766,1370>
> Representation(g);
[ 1, 6 ]
>
> g := Random(QF);
> Representation(g);
[ 1, 270 ]
>
```

```
> UserRepresentation(g);
[ 377, 0, 515, 0, 0, 0, 0, 0, 0, 0 ]
>
> S := [];
> for i in [1..3] do
>     d := Random(QF);
>     Include(~S, d);
> end for;
> seq, coeff := Representation(S, g);
> seq; coeff;
[ -170, -3, 0 ]
1
```

69.4.3 Arithmetic with Elements

If the generic abelian group A has been constructed with the flag `UseRepresentation` set true, then arithmetic with elements of A is trivial.

`u + v`

Given elements u and v belonging to the same abelian group A, return the sum of u and v.

`-u`

The inverse of element u.

`u - v`

Given elements u and v belonging to the same abelian group A, return the sum of u and the inverse of v.

`m * u`
`u * m`

Given an integer m, return the element $w+w+\cdots w$ ($|m|$ summands), where $w = u$, if m is positive and $w = -u$ if m is negative.

69.5 Construction of Subgroups and Quotient Groups

The operations in this section apply to both, free abelian groups and arbitrary abelian groups.

69.5.1 Construction of Subgroups

`sub< A | L >`

Construct the subgroup B of the abelian group A generated by the elements specified by the terms of the *generator list* L. A term $L[i]$ of the generator list may consist of any of the following objects:

(a) An element liftable to A;

(b) A sequence of integers representing an element of A;

(c) A subgroup of A;

(d) A set or sequence of type (a), (b), or (c).

The collection of words and groups specified by the list must all belong to the group A and the group B will be constructed as a subgroup of A.

Example H69E8

We create a subgroup of the group $A = Z_2 + Z_3 + Z_4 + Z_5 + Z_6 + Z + Z$.

```
> A<[x]> := AbelianGroup([2,3,4,5,6,0,0]);
> A;
Abelian Group isomorphic to Z/2 + Z/6 + Z/60 + Z + Z
Defined on 7 generators
Relations:
    2*x[1] = 0
    3*x[2] = 0
    4*x[3] = 0
    5*x[4] = 0
    6*x[5] = 0
> B<[y]> := sub< A | x[1], x[3], x[5], x[7] >;
> B;
Abelian Group isomorphic to Z/2 + Z/2 + Z/12 + Z
Defined on 4 generators in supergroup A:
    y[1] = 2*x[3] + 3*x[5]
    y[2] = x[1]
    y[3] = 3*x[3] + x[5]
    y[4] = x[7]
Relations:
    2*y[1] = 0
    2*y[2] = 0
    12*y[3] = 0
```

In the case of subgroups of generic groups, a number of parameters are provided.

`sub< A | L: parameters >`

Construct the subgroup of the generic abelian group A generated by the elements specified by the terms of the *generator list* L. A term $L[i]$ of the generator list may consist of any of the following objects:

(a) An element liftable into A;

(b) A sequence of integers representing an element of A;

(c) A set or sequence whose elements may be of either of the above types.

An element liftable into A may be an element of A itself, or it may be an element of U (`Universe(U)`), U being as usual the domain over which A is defined. For consistency with the construction, the values of the following parameters may also be passed to the subgroup constructor:

`Order`	RngInt	*Default :*
`RandomIntrinsic`	MonStg	*Default :*
`ComputeStructure`	Bool	*Default :* `false`
`UseUserGenerators`	Bool	*Default :* `false`
`PollardRhoRParam`	RngInt	*Default :* 20
`PollardRhoTParam`	RngInt	*Default :* 8
`PollardRhoVParam`	RngInt	*Default :* 3

In particular, it is possible to construct a subgroup by giving its order and a random function generating elements of the *subgroup*. In this case, the list L would be empty since calculation of the subgroup structure would result in the construction of the p-Sylow subgroups from random elements of the subgroup. Further, when the structure of A is already known and if the subgroup is defined in terms of a set of generators in L then the subgroup structure is computed at the time of creation.

Example H69E9

We create a subgroup of the quadratic forms group considered above.

```
> S := [];
> for j in [1..2] do
>     P := Random(QF);
>     Include(~S, P);
> end for;
> S;
[ <45,26,22226>, <937,-930,1298> ]
> QF1 := sub< QF | S>;
> QF1;
Generic Abelian Group over
Binary quadratic forms of discriminant -4000004
Abelian Group isomorphic to Z/2 + Z/258
Defined on 2 generators in supergroup A:
```

```
  QF1.1 = QF.1
  QF1.2 = 2*QF.2
Relations:
  2*QF1.1 = 0
  258*QF1.2 = 0
>
```

69.5.2 Construction of Quotient Groups

`quo< F | R >`

Given an abelian group F, and a set of relations R in the generators of F, construct (a) an abelian group A isomorphic to the quotient of F by the subgroup of F defined by R, and (b) the natural homomorphism $\phi : F \to A$.

The expression defining F may be either simply the name of a previously constructed group, or an expression defining an abelian group. The possibilities for the relation list R are the same as for the `AbelianGroup` construction.
The function returns:

(a) The quotient group A;

(b) The natural homomorphism $\phi : F \to A$.

`A / B`

Given a subgroup B of the abelian group A, construct the quotient of A by B.

69.6 Standard Constructions and Conversions

`AbelianGroup(GrpAb, Q)`

`AbelianGroup(Q)`

Let $Q = [a_1, \ldots, a_r]$ be a sequence of non-negative integers. This function creates the abelian group $\mathbf{Z}_1 + \cdots + \mathbf{Z}_r$, where Z_i is the cyclic group of order $|a_i|$ if $a_i \neq 0$ or the infinite cyclic group $\mathbf{Z}$ otherwise, $i = 1, \ldots, r$.

`AbelianGroup(G)`

Given an abelian permutation, matrix or polycyclic group G, represent it as an abelian group A. The function also returns the isomorphism $\phi : G \to A$ as its second value.

`AbelianQuotient(G)`

Given a finitely presented, permutation, matrix or polycyclic group G, return the maximal abelian quotient A of G. The function returns the natural homomorphism $\phi : G \to A$ as its second value.

`DirectSum(A, B)`

The direct sum of abelian groups A and B.

`PCGroup(A)`

A pc-group representation G of A. The isomorphism $\phi : A \to G$ is also returned.

`PermutationGroup(A)`

A permutation group representation of A. The particular group G is generated by disjoint cycles whose lengths are the abelian invariants of A. The isomorphism $\phi : G \to A$ is also returned.

`FPGroup(A)`

A fp-group group representation of A. The particular group G is generated by commuting generators whose orders are the abelian invariants of A. The isomorphism $\phi : G \to A$ is also returned.

`CommutatorSubgroup(G)`
`DerivedSubgroup(G)`

The derived subgroup of G, that is the trivial group, since G is abelian.

`CommutatorSubgroup(H, K)`
`CommutatorSubgroup(G, H, K)`

The commutator subgroup of groups H and K in their common overgroup G.

`Centralizer(G, a)`
`Centraliser(G, a)`

The centraliser of a in G.

`Core(G, H)`

The maximal normal subgroup of G that is contained in the subgroup H of G. Since G is abelian, this is H itself.

`Centre(G)`
`Center(G)`

The center of G, ie. G itself.

69.7 Operations on Elements

69.7.1 Order of an Element

`Order(x)`

Order of the element x belonging to an fp-abelian group. If the element has infinite order, the value zero is returned.

Example H69E10

We compute the orders of some elements in the group $Z_2 \times Z_3 \times Z_4 \times Z_5 \times Z$.

```
> G<[x]> := AbelianGroup([2,3,4,5,0]);
> Order( x[1] + 2*x[2] + 3*x[4]);
30
> Order( x[1] + x[5] );
0
```

It is possible to determine the order of an element of a generic group with first calculating the structure of the group. The order function involves the use of one of several algorithms:

– an improved baby-step giant-step algorithm, due to J. Buchmann, M.J. Jacobson, E. Teske [BJT97],

– the Pollard–Rho method based algorithm, described above [Tes98a].

When computing the order of an element whose lower and upper bounds are known, or where lower and upper bounds for the group order are known, the following two algorithms have been shown to be significantly faster than the two algorithms mentioned above.

– the standard baby-step giant-step Shanks algorithm,

– another variant of the Pollard–Rho method which is due to P. Gaudry and R. Harley [GH00].

To avoid confusion we will distinguish the algorithms due to Teske *et al* and name them the `T` baby-step giant-step algorithm and the `T` Pollard–Rho algorithm respectively. The Pollard–Rho algorithm has smaller space requirements than the baby-step giant-step algorithm, so it is recommended for use in very large groups.

In all cases, if the group order is known beforehand, the element order is computed using this knowledge. This is a trivial operation.

`Order(g: parameters)`

`ComputeGroupOrder`	Bool	*Default :* `true`
`BSGSLowerBound`	RngIntElt	*Default :* 0
`BSGSStepWidth`	RngIntElt	*Default :* 0

Assume that g is an element of the generic abelian group A. This functions returns the order of the element g. Note that if `UseRepresentation` is set to true for A, then this is a trivial operation.

If the parameter `ComputeGroupOrder` is true, the order of A is computed first (unless it is already known). The order of g is then computed using this knowledge; this last computation is trivial.

If `ComputeGroupOrder` is false, the order of g is computed using the `T` baby-step giant-step algorithm.

`BSGSLowerBound` and `BSGSStepWidth` are parameters which can be passed to the baby-step giant-step algorithm.

`BSGSLowerBound` sets a lowerbound on the order of the element g, and `BSGSStepWidth` sets the step-width in the algorithm.

`Order(g, l, u:` *parameters*`)`

`Alg`	MonStg	*Default :* "*Shanks*"
`UseInversion`	BoolElt	*Default :* `false`

Assume that g is an element of the generic abelian group A such that the order of g or the order of A is bounded by u and l. This function returns the order of the element g.

If the parameter `Alg` is set to `"Shanks"` then the generic Shanks algorithm is used, and when `Alg` is set to `"PollardRho"`, the Gaudry–Harley Pollard–Rho variant is used. Setting `UseInversion` halves the search space. To be effective element inversion must be fast.

`Order(g, l, u, n, m:` *parameters*`)`

`Alg`	MonStg	*Default :* "*Shanks*"
`UseInversion`	BoolElt	*Default :* `false`

Assume that g is an element of the generic abelian group A such that the order of g or the order of A is bounded by u and l. Assume also that `Order(`g`)` $\equiv n$ `(mod` m`)` or that $\#A \equiv$ `(mod` m`)` This function returns the order of the element g. The two parameters `Alg` and `UseInversion` have the same meaning as for the previous `Order` function.

69.7.2 Discrete Logarithm

`Log(g, d:` *parameters*`)`

`ComputeGroupOrder`	Bool	*Default :* `true`
`AlInPohligHellmanLoop`	MonStg	*Default :* "*BSGS*"
`BSGSStepWidth`	RngIntElt	*Default :* 0
`PollardRhoRParam`	RngInt	*Default :* 20
`PollardRhoTParam`	RngInt	*Default :* 8
`PollardRhoVParam`	RngInt	*Default :* 3

Assume that g and d are elements of the generic abelian group A. This function returns the discrete logarithm of d to the base g. If `ComputeGroupOrder` is true, then the group order is computed first (if not already known) and from this the order of the base g is computed. The discrete logarithm problem is then solved using a Pohlig–Hellman loop calling either the `T` baby-step giant-step algorithm (`AlInPohligHellmanLoop := "BSGS"`) or the `T` Pollard–Rho algorithm (`AlInPohligHellmanLoop := "PollardRho"`).

The parameter `BSGSStepWidth` has the same meaning as for the Order function. Parameters `PollardRhoRParam`, `PollardRhoTParam`, and `PollardRhoVParam` have the same meaning as they do for the determination of structure (function AbelianGroup). If `ComputeGroupOrder` is false then the discrete logarithm problem is solved using the `T` baby-step giant-step algorithm. Here again the parameter `BSGSStepWidth` applies.

Example H69E11

It is assumed that the structure of the groups QF has already been computed. We illustrate the computation of the discrete logarithm relative to a given base.

```
> n := 38716;
> Ip := Reduction(PrimeForm(Q,11));
> g := GA_qf!Ip;
> d := n * g;
>
> l1 := Log(g, d : BSGSStepWidth := Floor((-Dk)^(1/4)/2));
> l2 := Log(g, d : AlInPohligHellmanLoop := "PollardRho");
> l3 := Log(g, d : ComputeGroupOrder := false);
> l4 := Log(g, d: ComputeGroupOrder := false,
>                 BSGSStepWidth := Floor((-Dk)^(1/4)/2));
> assert l1 eq l2 and l2 eq l3 and l3 eq l4;
> assert IsDivisibleBy(n - l1, Order(g));
```

69.7.3 Equality and Comparison

`u eq v`

Returns `true` if the elements u and v are identical (as elements of the appropriate free abelian group), `false` otherwise.

`u ne v`

Returns `true` if the elements u and v are not identical (as elements of the appropriate free abelian group), `false` otherwise.

`IsIdentity(u)`

`IsId(u)`

Returns `true` if the element u, belonging to the abelian group A, is the identity element (zero), `false` otherwise.

69.8 Invariants of an Abelian Group

`ElementaryAbelianQuotient(G, p)`

The maximal p-elementary abelian quotient of the group G as `GrpAb`. The natural epimorphism is returned as second value.

`FreeAbelianQuotient(G)`

The maximal free abelian quotient of the group G as `GrpAb`. The natural epimorphism is returned as second value.

`Invariants(A)`

The invariants of the abelian group G. Each infinite cyclic factor is represented by zero.

`TorsionFreeRank(A)`

The torsion-free rank of the abelian group G.

`TorsionInvariants(A)`

The torsion invariants of the abelian group G.

`PrimaryInvariants(A)`

The primary invariants of the abelian group G.

`pPrimaryInvariants(A, p)`

The p-primary invariants of the abelian group G.

69.9 Canonical Decomposition

`TorsionFreeSubgroup(A)`

The torsion-free subgroup of the abelian group G.

`TorsionSubgroup(A)`

The torsion subgroup of the abelian group G.

`pPrimaryComponent(A, p)`

The p-primary component of the abelian group G.

69.10 Set-Theoretic Operations

69.10.1 Functions Relating to Group Order

`Order(G)`
`#G`

The order of the group G, returned as an ordinary integer. If G is an infinite group, the value zero is returned. Note that if G is a generic group then determining the order will require the structure of G to be determined.

`FactoredOrder(G)`

The factored order of the group G, returned as a sequence of prime-exponent pairs. If G is an infinite group, the empty sequence is returned. Note that if G is a generic group then determining the order will require the structure of G to be determined.

`Exponent(G)`

The exponent of the group G. If the group is infinite, the value zero is returned. Note that if G is a generic group then determining the exponent will require the structure of G to be determined.

`IsFinite(G)`

Return `true` if the group G is finite.

`IsInfinite(G)`

Return `true` if G , `false` otherwise.

69.10.2 Membership and Equality

`g in G`

Given an element g and a group G, return `true` if g is an element of G, `false` otherwise.

`g notin G`

Given an element g and a group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G`

Given an group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G`

Given a group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G`

Given groups G and H, subgroups of some common overgroup, return `true` if H is a subgroup of G, and `false` otherwise.

`H notsubset G`

Given groups G and H, subgroups of some common overgroup, return `true` if H is not a subgroup of G, and `false` otherwise.

`G eq H`

Given groups G and H, subgroups of some common overgroup, return `true` if G and H are identical, and `false` otherwise.

`G ne H`

Given groups G and H, subgroups of some common overgroup, return `true` if G and H are distinct groups, and `false` otherwise.

69.10.3 Set Operations

`NumberingMap(G)`

A bijective mapping from the finite group G onto the set of integers $\{1 \dots |G|\}$. The actual mapping depends upon choice of standard generators for G.

`RandomProcess(G)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 100

Create a process to generate randomly chosen elements from the finite group G. The process is based on the product-replacement algorithm of [CLGM$^+$95], modified by the use of an accumulator. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G)+1$. Initially, these are just the generators of G. As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses one of the elements in the slots and adds it into the accumulator. The element in that slot is replaced by the sum of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting `Scramble := m` causes m such sum-replacement operations to be performed before the process is returned. Note that this algorithm cannot produce well-distributed random elements of an infinite group.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the finite abelian group G, return the next random element of G defined by the process.

`Random(G)`

An element chosen at random from the finite group G.

`Rep(G)`

A representative element of G.

69.11 Coset Spaces

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a group G and a subgroup H of G, this function returns:

(a) An indexed set of elements T of G forming a right transversal for G over H; and,

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = \{t_1, \ldots, t_r\}$ and g in G, ϕ is defined by $\phi(g) = t_i$, where $g \in Ht_i$.

69.11.1 Coercions Between Groups and Subgroups

`G ! g`

Given an element g belonging to the subgroup H of the group G, rewrite g as an element of G.

`H ! g`

Given an element g belonging to the group G, and given a subgroup H of G containing g, rewrite g as an element of H.

`K ! g`

Given an element g belonging to the group H, and a group K, such that H and K are subgroups of G, and both H and K contain g, rewrite g as an element of K.

`Morphism(H, G)`

The integer matrix defining the inclusion monomorphism from the subgroup H of G into G.

69.12 Subgroup Constructions

Although, in the case of an abelian group, many of the standard subgroup constructors are trivial, they are all implemented for the sake of uniformity. Here we document only those which are meaningful in the conrext of abelian groups.

`H meet K`

Given subgroups H and K of some group G, construct their intersection.

`H meet:= K`

Replace H with the intersection of groups H and K.

`H + K`

Given subgroups H and K of some group G, construct the smallest subgroup containing both.

`n * G`

For an integer n and some abelian group G, construct the subgroup nG. The second return value is the map $G \to G$ sending $g \to ng$.

`FrattiniSubgroup(G)`

The Frattini subgroup of the finite abelian group G.

`SylowSubgroup(G, p : parameters)`

`Sylow(G, p : parameters)`

`Structure` BOOL *Default :* `false`

The Sylow p-subgroup for the group G. If G is a generic group and the parameter `Structure` is true, or if the group structure of A is known, then the group structure of the Sylow subgroup is computed.

Example H69E12

In the following example, we construct the Sylow 2-subgroup of $G = Z_{34384}$.

```
> m := 34384;
> Zm := Integers(m);
> U := {@ x : x in Zm | GCD(x, m) eq 1 @};
> G := GenericAbelianGroup(U : IdIntrinsic := "Id",
>    AddIntrinsic := "*", InverseIntrinsic := "/");
> _ := AbelianGroup(G);
> Factorization(#G);
> Sylow(G, 2);
2-Sylow subgroup: Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4
Defined on 4 generators in supergroup G:
  GAp.1 = G.1
  GAp.2 = G.2
```

```
  GAp.3 = 3*G.3
  GAp.4 = 153*G.4
Relations:
  2*GAp.1 = 0
  2*GAp.2 = 0
  2*GAp.3 = 0
  4*GAp.4 = 0
```

69.13 Subgroup Chains

`CompositionSeries(G)`

A composition series for the finite abelian group G returned as a sequence of subgroups.

`Agemo(G, i)`

Given a finite p-group G, return the characteristic subgroup of G generated by the elements x^{p^i}, $x \in G$, where i is a positive integer.

`Omega(G, i)`

Given a finite p-group G, return the characteristic subgroup of G generated by the elements of order dividing p^i, where i is a positive integer.

69.14 General Group Properties

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise.

`IsFree(G)`

Returns `true` if G is free, `false` otherwise.

`IsMixed(G)`

Returns `true` if G is a mixed group, `false` otherwise. An abelian group is mixed if it is neither a torsion group nor free.

`IspGroup(G)`

Returns `true` if the finite group G is a p-group, ie. if all elements have order a power of p.

69.14.1 Properties of Subgroups

`IsMaximal(G, H)`

Returns `true` if the subgroup H of the finite group G is a maximal subgroup of G, `false` otherwise.

`Index(G, H)`

The index of the subgroup H in the group G, returned as an ordinary integer. If H has infinite index in G, the value zero is returned.

`FactoredIndex(G, H)`

The factored index of the subgroup H in the group G, returned as a sequence of prime-exponent pairs. If H has infinite index in G, the empty sequence is returned.

`IsPure(G, H)`

Returns `true` if the subgroup H of the finite group G is pure, ie. if for all n we have $nG \cap H = nH$.

`IsNeat(G, H)`

Returns `true` if the subgroup H of the finite group G is neat, i.e., if for all primes p we have $pG \cap H = pH$.

69.14.2 Enumeration of Subgroups

`MaximalSubgroups(G)`

The maximal subgroups of the finite group G returned as a sequence of subgroups.

`Subgroups(G:`*parameters*`)`

The subgroups of the finite group G are returned as a sequence of records. The record fields are `subgroup`, storing the actual group; `order`, storing the group order; and `length`, storing the length of the conjugacy class, which is always 1 for abelian groups.

`Sub` [RngIntElt] *Default :* []

If parameter `Sub` is set, only subgroups with invariants equal to the given sequence are found. The given sequence should contain positive integers, such that each divides the following.

`Quot` [RngIntElt] *Default :* []

If parameter `Quot` is set, only subgroups such that the quotient group has invariants equal to the given sequence are found. The given sequence should contain positive integers, such that each divides the following.

`NumberOfSubgroupsAbelianPGroup (A)`

Return the number of subgroups of each non-trivial order in the abelian p-group G where $A = [a_1, a_2, \ldots]$ and $G = C_{a_1} \times C_{a_2} \times \ldots$. The m-th entry in the sequence returned is the number of subgroups of order p^m.

`HasComplement(G, U)`

For a finite abelian group G and a subgroup U decide if there exist some other subgroup V such that $G = U + V$ and $U \cap V = \{0\}$. In case such a V exists, it is returned as the second value.

Example H69E13

We look at subgroups of an abelian group of order 12.

```
> G := AbelianGroup([2,6]);
> s := Subgroups(G); #s;
10
> s[7];
rec<recformat<order, length, subgroup, presentation> |
   order := 3, length := 1,
   subgroup := Abelian Group isomorphic to Z/3
Defined on 1 generator in supergroup G:
  $.1 = 2*G.2
Relations:
  3*$.1 = 0>
> [x`order:x in s];
[ 12, 6, 4, 2, 6, 6, 3, 2, 2, 1 ]
```

Now we find the elementary abelian subgroup of order 4.

```
> s22 := Subgroups(G:Sub := [2,2]); #s22;
1
> s22;
Conjugacy classes of subgroups
------------------------------
 [1]     Order 4            Length 1
        Abelian Group isomorphic to Z/2 + Z/2
        Defined on 2 generators in supergroup G:
          $.1 = G.1
          $.2 = 3*G.2
        Relations:
          2*$.1 = 0
          2*$.2 = 0
```

There is more than one subgroup of index 2 in G.

```
> q2 := Subgroups(G:Quot := [2]); #q2;
3
> q2[3]`subgroup;
```

```
Abelian Group isomorphic to Z/6
Defined on 1 generator in supergroup G:
  $.1 = G.1 + G.2
Relations:
  6*$.1 = 0
```

69.15 Representation Theory

`CharacterTable(G)`

The table of irreducible characters for the group G.

69.16 The Hom Functor

`Hom(G, H)`

Given finite abelian groups G and H, return an abelian group A isomorphic to $\mathrm{Hom}(G, H)$, and a transfer map t such that, given an element a of A, $t(a)$ yields the corresponding (MAGMA `Map` type) homomorphism from G to H. The structure of $\mathrm{Hom}(G, H)$ may thus be analyzed by examining A.

`HomGenerators(G, H)`

Given finite abelian groups G and H, return a sequence of (Z-module) generators of $\mathrm{Hom}(G, H)$. The generators are returned as actual (MAGMA `Map` type) homomorphisms. Note that $\mathrm{Hom}(G, H)$ is usually not free, so it is difficult to generate all homomorphisms uniquely using the generators alone (use `Hom` or `Homomorphisms` if that is desired).

`Homomorphisms(G, H)`

Given finite abelian groups G and H, return a sequence containing all elements of $\mathrm{Hom}(G, H)$. The elements are returned as actual (MAGMA `Map` type) homomorphisms. Note that this function simply uses `Hom`, transferring each element of the returned group to the actual MAGMA `Map` type homomorphism.

Example H69E14

We examine $A = \mathrm{Hom}(G, H)$, for certain abelian groups G and H.

```
> G := AbelianGroup([2, 3]);
> H := AbelianGroup([4, 6]);
> A, t := Hom(G, H);
> #A;
12
> A;
Abelian Group isomorphic to Z/2 + Z/6
Defined on 2 generators
Relations:
    2*A.1 = 0
    6*A.2 = 0
> h := t(A.1);
> h;
Mapping from: GrpAb: G to GrpAb: H
> h(G.1);
3*H.2
> h(G.2);
0
```

We now enumerate all elements of A and examine the images of each generator of G under each homomorphism. We note that each possible list of images occurs only once.

```
> I := [<h(G.1), h(G.2)> where h is t(x): x in A];
> I;
[
    <0, 0>,
    <3*H.2, 0>,
    <2*H.1, 2*H.2>,
    <2*H.1 + 3*H.2, 2*H.2>,
    <0, 4*H.2>,
    <3*H.2, 4*H.2>,
    <2*H.1, 0>,
    <2*H.1 + 3*H.2, 0>,
    <0, 2*H.2>,
    <3*H.2, 2*H.2>,
    <2*H.1, 4*H.2>,
    <2*H.1 + 3*H.2, 4*H.2>
]
> #I;
12
> #Set(I);
12
```

69.17 Automorphism Groups

The full automorphism group of the abelian group G.

69.18 Cohomology

`Dual(G)`

Computes the dual group G^* of G and a map M from $G \times G^* \to \mathbf{Z}/m\mathbf{Z}$ for m the exponent of G that allows G^* to act on G. G must be finite.

`H2_G_QmodZ(G)`

Computes $H := H^2(G, \mathbf{Q}/\mathbf{Z})$ and a map $f : H \to (G \times G \to \mathbf{Z}/m\mathbf{Z})$ that will give the cocycles as maps from $G \times G \to \mathbf{Z}/m\mathbf{Z}$. $m := \#G$.

`Res_H2_G_QmodZ(U, H2)`

For a subgroup U of G and $H2 = H^2(G, \mathbf{Q}/\mathbf{Z})$ computes $H^2(U, \mathbf{Q}/\mathbf{Z})$ in a compatible way together with the restriction map into $H2$.

$H2$ must be the result of `H2_G_QmodZ` as this function relies on the attributes stored in there.

69.19 Homomorphisms

Two functions are provided to construct homomorphisms or isomorphisms from one group into another, where either of the groups, or both, may be generic abelian groups.

`hom< A -> B | L >`

Given groups A and B, construct a homomorphism from A to B as defined by the extension L. If one or both of A and B are generic abelian groups this works as usual, with one minor difference as explained below. Suppose that the generators of A are $g_1, \ldots, g_n$, and that $\phi(g_i) = h_i$ for each i, where ϕ is the homomorphism one wishes to construct. The list L as required by the constructors must be one of the following:

(a) a list of the n 2-tuples $< g_i, h_i >$ (order not important);

(b) a list of the n arrow-pairs $g_i \to h_i$ (order not important);

(c) $h_1, \ldots, h_n$ (order is important).

If A is a generic abelian group this rule is relaxed somewhat in the following sense: If L is a list of n 2-tuples or of n arrow-pairs, the elements g_i need not be the defining generators of A. The only requirement is that the set $\{g_1, \ldots, g_n\}$ does actually generate the whole of A.

`Homomorphism(A, B, X, Y)`

Creates a homomorphism from A into B as given by the mapping of X into Y. The arguments A and B may be any type of group, including of course, generic abelian groups.

The function `Homomorphism` does not require the elements of the argument X to be generators of A as given by `Generators(`A`)`, so it allows more freedom when creating a homomorphism. If, however, these elements fail to generate the whole of A then the subsequent map application will fail.

`iso< A -> B | L >`

Given groups A and B, construct an isomorphism from A to B as defined by the extension L. If one or both of A and B are generic abelian groups this works as usual, with one minor difference as explained below. Suppose that the generators of A are $g_1, \ldots, g_n$, and that $\phi(g_i) = h_i$ for each i, where ϕ is the isomorphism one wishes to construct. The list L as required by the constructors must be one of the following:

(a) a list of the n 2-tuples $< g_i, h_i >$ (order not important);

(b) a list of the n arrow-pairs $g_i \to h_i$ (order not important);

(c) $h_1, \ldots, h_n$ (order is important).

If A is a generic abelian group this rule is relaxed somewhat in the following sense: If L is a list of n 2-tuples or of n arrow-pairs, the elements g_i may not necessarily be generators of A as given by the function `Generators(`A`)`. The only requirement is that the set $\{h_1, \ldots, h_n\}$ does actually generate the whole of B.

`Isomorphism(A, B, X, Y)`

Creates a isomorphism from A into B as given by the mapping of X into Y. The arguments A and B can be any type of group, including of course, generic abelian groups.

The function `Isomorphism` does not require the elements of the argument X to be generators of A as given by `Generators(`A`)`, so it allows more freedom when creating an isomorphism. If, however, these elements fail to generate the whole of A then the subsequent map application will fail.

Example H69E15

Recall that we defined the subgroups $GH1_{Zm}$ and $GH2_{Zm}$ of G as:

```
> GH1_Zm;
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/6 + Z/612
Defined on 2 generators in supergroup G:
  GH1_Zm.1 = G.2 + G.3
  GH1_Zm.2 = G.4
```

```
Relations:
  6*GH1_Zm.1 = 0
  612*GH1_Zm.2 = 0
> GH2_Zm;
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2 + Z/6 + Z/612
Defined on 4 generators in supergroup G:
  GH2_Zm.1 = G.1
  GH2_Zm.2 = G.2
  GH2_Zm.3 = G.3
  GH2_Zm.4 = G
Relations:
  2*GH2_Zm.1 = 0
  2*GH2_Zm.2 = 0
  6*GH2_Zm.3 = 0
  612*GH2_Zm.4 = 0
```

We construct the homomorphism

```
> h := hom<GH1_Zm -> GH2_Zm | GH2_Zm.1, GH2_Zm.2 >;
> h(GH1_Zm);
Generic Abelian Group over
Residue class ring of integers modulo 34384
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators in supergroup GH2_Zm:
  $.1 = GH2_Zm.2
  $.2 = GH2_Zm.1
Relations:
  2*$.1 = 0
  2*$.2 = 0
```

but we cannot construct the isomorphism

```
>  i := iso<GH1_Zm -> GH2_Zm | GH2_Zm.1, GH2_Zm.2 >;
>> i := iso<GH1_Zm -> GH2_Zm | GH2_Zm.1, GH2_Zm.2 >;
           ^
Runtime error in map< ... >: Images do not generate the (whole) codomain
```

An alternative way of creating the homomorphism h would be

```
> h := Homomorphism(GH1_Zm, GH2_Zm, Generators(GH1_Zm), [GH2_Zm.1, GH2_Zm.2]);
```

69.20 Bibliography

[**BJT97**] J. Buchmann, M. J. Jacobson, Jr., and E. Teske. On Some Computational Problems in Finite Abelian Groups. *Mathematics of Computation*, 66:1663–1687, 1997.

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**CLGM+95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

[**GH00**] P. Gaudry and R. Harley. Counting Points on Hyperelliptic Curves over Finite Fields. In Bosma [Bos00], pages 313–332.

[**Tes98a**] E. Teske. A Space Efficient Algorithm for Group Structure Computation. *Mathematics of Computation*, 67:1637–1663, 1998.

[**Tes98b**] E. Teske. Better Random Walks for Pollard's Rho Method. 1998.

70 FINITELY PRESENTED GROUPS

Chapter 70
FINITELY PRESENTED GROUPS

70.1 Introduction

This Chapter presents the functions designed for computing with finitely-presented groups (fp-groups for short). The name of the corresponding MAGMA category is `GrpFP`. The functions considered here are designed for doing what is sometimes referred to as *combinatorial group theory*.

70.1.1 Overview of Facilities

The facilities provided for fp-groups fall into a number of natural groupings:

- The construction of fp-groups in terms of generators and relations;
- The construction of particular types of quotient groups: abelian quotient, p-quotient, nilpotent quotient and soluble quotient;
- Index determination and subgroup building based on the Todd-Coxeter procedure;
- Calculations with subgroups having finite index in a group, where the subgroups are represented by coset tables;
- The construction of all subgroups having index less than some (small) specified bound;
- The construction of representations of an fp-group corresponding to actions on coset spaces and elementary abelian sections;
- The use of a rewriting process for constructing presentations of subgroups;
- The simplification of words with respect to a given set of relations.

For a description of fundamental algorithms for finitely presented groups, we refer the reader to [Sim94].

70.1.2 The Construction of Finitely Presented Groups

The construction of fp-groups utilises the fact that every group is a quotient of some free group. Thus, two general fp-group constructors are provided: `FreeGroup(n)` which constructs a free group of rank n, and `quo< F | R >` which constructs the quotient of group F by the normal subgroup defined by the relations R.

The naming of generators presents special difficulties since they are not always used in a consistent manner in the mathematical literature. A generator name is used in two distinct ways. Firstly, it plays the role of a *variable* having as its value a designated generator of G. Secondly, it appears as the *symbol* designating the specified generator whenever elements of the group are output. These two uses are separated in the MAGMA semantics.

In MAGMA, a standard indexing notation is provided for referencing the generators of any fp-group G. Thus, `G.i` denotes the i-th generator of G. However, users may give

individual names to the generators by means of the *generator*-assignment. Suppose that the group G is defined on r generators. Then if the right hand side of the following statement creates a group, the special assignment

```
> G< v_1, ..., v_r> := construction;
```

is equivalent to the statements

```
> G := construction;
>    v_1 := G.1;
>    ...
>    v_r := G.r;
```

It should be noted that when the fp-group G is created as the quotient of the group F, any names that the user may have associated with the generators of F will not be associated with the corresponding generators of G. If this were allowed, then it would violate the fundamental principle that every object is viewed as belonging to a *unique* structure.

70.2 Free Groups and Words

70.2.1 Construction of a Free Group

`FreeGroup(n)`

Construct the free group F of rank n, where n is a positive integer.

The i-th generator of F may be referenced by the expression `F.i`, $i = 1, \ldots, n$. Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of F. In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the *left hand side* of the assignment statement: `F< v`$_1$`, ..., v`$_n$` > := FreeGroup(n);`

Example H70E1

The statement

```
> F := FreeGroup(2);
```

creates the free group of rank 2. Here the generators may be referenced using the standard names, `F.1` and `F.2`.
The statement

```
> F<x, y> := FreeGroup(2);
```

defines F to be the free group of rank 2 and assigns the names x and y to the two generators.

70.2.2 Construction of Words

The operations in this section apply to both, free groups and arbitrary fp-groups.

`G ! [ i1, ..., is ]`

Given a group G defined on r generators and a sequence $[i_1, \cdots, i_s]$ of integers lying in the range $[-r, r]$, excluding 0, construct the word

$$G.|i_1|^{\epsilon_1} * G.|i_2|^{\epsilon_2} * \cdots * G.|i_s|^{\epsilon_s}$$

where ϵ_j is $+1$ if i_j is positive, and -1 if i_j is negative.

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity element, represented as the empty word, for the fp-group G. For a sample application of this function, see Example H70E3.

`Random(G, m, n)`

A random word of length l in the generators of the group G, where $m \leq l \leq n$. For a sample application of this function, see Example H70E3.

70.2.3 Access Functions for Words

This section describes some basic access functions for words. These operations apply to both, free groups and arbitrary fp-groups.

`#w`

The length of the word w.

`ElementToSequence(w)`

`Eltseq(w)`

The sequence Q obtained by decomposing the word w into its constituent generators and generator inverses. Suppose w is a word in the group G. Then, if $w = G.i_1^{e_1} \cdots G.i_m^{e_m}$, with each $e_i = \pm 1$, then $Q[j] = i_j$ if $e_j = +1$ and $Q[j] = -i_j$ if $e_j = -1$, for $j = 1, \ldots, m$.

`ExponentSum(w, x)`

`Weight(w, x)`

Given a word w, and the name of a generator x of a group G, compute the sum of the exponents of the generator x in the word w. For a sample application of this function, see Example H70E3.

`GeneratorNumber(w)`

Suppose w is a word belonging to a group G. Assume x is the name of the i-th generator of G. Then

(i) if $w =$ `Identity(G)`, `GeneratorNumber(w)` is 0;

(ii) if $w = x * w'$, w' a word in G, `GeneratorNumber(w)` is i;

(iii) if $w = x^{-1} * w'$, w' a word in G, `GeneratorNumber(w)` is $-i$.

`LeadingGenerator(w)`

Suppose w is a word belonging to a group G. If $w = x^{\epsilon} * w'$, w' a word in G, x a generator of G and $\epsilon \in \{-1, +1\}$, the functions returns x^{ϵ}. If $w =$ `Identity(G)`, it returns `Identity(G)`. For a sample application of this function, see Example H70E3.

`Parent(w)`

The parent group G of the word w.

Example H70E2

Consider the free group F on 6 generators

```
> F<u,v,w,x,y,z> := FreeGroup(6);
```

and the sequence of words

```
> rels := [ (u*v)^42, (v,x), (x*z^2)^4,
>           v^2*y^3, (v*z)^3, y^4, (x*z)^3 ];
```

The abelianised relation matrix of the quotient

$$\langle u, v, w, x, y, z \mid (uv)^{42}, (v, x), (xz^2)^4, v^2y^3, (vz)^3, y^4, (xz)^3 \rangle$$

can be obtained using the following construction

```
> R := Matrix(Integers(),
>        [ [ Weight(r, F.j) : j in [1..6] ] : r in rels ]);
> R;
[42 42  0  0  0  0]
[ 0  0  0  0  0  0]
[ 0  0  0  4  0  8]
[ 0  2  0  0  3  0]
[ 0  3  0  0  0  3]
[ 0  0  0  0  4  0]
[ 0  0  0  3  0  3]
```

(The function `Matrix` constructs a matrix from a sequence of row vectors.)

70.2.4 Arithmetic Operators for Words

Suppose G is an fp-group for which generators have already been defined. This subsection defines the elementary arithmetic operations on words that are derived from the multiplication and inversion operators. The availability of the operators defined here enables the user to construct an element (*word*) of G in terms of the generators as follows:

(i) A generator is a word;

(ii) The expression (u) is a word, where u is a word;

(iii) The product $u * v$ of the words u and v is a word;

(iv) The conjugate u^v of the word u by the word v, is a word (u^v expands into the word $v^{-1} * u * v$);

(v) The power of a word, u^n, where u is a word and n is an integer, is a word;

(vi) The commutator (u, v) of the words u and v is a word ((u, v) expands into the word $u^{-1} * v^{-1} * u * v$). Note that (u, v, w) is equivalent to $((u, v), w)$, i.e. commutators are *left-normed*.

The word operations defined here may be applied either to the words of a free group or the words of a group with non-trivial relations. If such an operator is applied to a group possessing non-trivial relations, only free reduction will be applied to the resulting words.

`u * v`

Given words u and v belonging to the same fp-group G, return the product of u and v.

`u ^ n`

The n-th power of the word u, where n is an integer. When invoked with $n = -1$, the function computes the inverse of u. When invoked with $n = 0$, the function returns the identity element.

`u ^ v`

Given words u and v belonging to the same fp-group G, return the conjugate $v^{-1} * u * v$ of the word u by the word v.

`(u, v)`

Given words u and v belonging to the same fp-group G, return the commutator $u^{-1}v^{-1}uv$ of the words u and v.

`(u1, ..., un)`

Given the n words $u_1, \ldots, u_n$ belonging to the same fp-group G, return their commutator. Commutators are *left-normed*, so that they are evaluated from left to right.

70.2.5 Comparison of Words

Words in an fp-group may be compared both for equality and for their relationship with respect to a natural lexicographic ordering. It should be noted that even when a pair of words belong to a group defined by non-trivial relations, only the free reductions of the words are compared. Thus, a pair of words belonging to a group G may be declared to be distinct even though they may represent the same element of G.

The words of an fp-group G are ordered first by length and then lexicographically. The lexicographic ordering is determined by the following ordering on the generators and their inverses:

$$G.1 < G.1^{-1} < G.2 < G.2^{-1} < \cdots$$

Here, u and v are words belonging to some common fp-group.

`u eq v`

Return `true` if the free reductions of the words u and v are identical.

`u ne v`

Return `true` if the free reductions of the words u and v are not identical.

`u lt v`

Return `true` if the word u precedes the word v, with respect to the ordering defined above for elements of an fp-group. The words u and v are freely reduced before the comparison is made.

`u le v`

Return `true` if the word u either precedes, or is equal to, the word v, with respect to the ordering defined above for elements of an fp-group. The words u and v are freely reduced before the comparison is made.

`u ge v`

Return `true` if the word u either follows, or is equal to, the word v, with respect to the ordering defined above for elements of an fp-group. The words u and v are freely reduced before the comparison is made.

`u gt v`

Return `true` if the word u follows the word v, with respect to the ordering defined above for elements of an fp-group. The words u and v are freely reduced before the comparison is made.

Example H70E3

We construct the free group on three generators and generate a random word w of length between 4 and 6.

```
> F<a,b,c> := FreeGroup(3);
>
> w := Random(F, 4, 6);
> w;
b^-1 * a^-1 * b^2 * a^-1
```

We print the length of w and the weight (the exponent sum) of generator a in w.

```
> #w;
5
>
> Weight(w, a);
-2
```

We now strip the generators from w one by one, using arithmetic and comparison operators for words and the access function `LeadingGenerator`.

```
> while w ne Identity(F) do
>     g := LeadingGenerator(w);
>     print g;
>     w := g^-1 * w;
> end while;
b^-1
a^-1
b
b
a^-1
```

70.2.6 Relations

A *relation* is an equality between two words in a fp-group. To facilitate working with relations, a *relation type* is provided.

`w1 = w2`

Given words w_1 and w_2 over the generators of an fp-group G, create the relation $w_1 = w_2$. Note that this relation is not automatically added to the existing set of defining relations R for G. It may be added to R, for example, through use of the `quo`-constructor (see below).

`r[1]`

`LHS(r)`

Given a relation r over the generators of G, return the left hand side of the relation r. The object returned is a word over the generators of G.

`r[2]`

`RHS(r)`

Given a relation r over the generators of G, return the right hand side of the relation r. The object returned is a word over the generators of G.

`r[1] := w`

Redefine the left hand side of the relation r to be the word w.

`r[2] := w`

Redefine the right hand side of the relation r to be the word w.

`f(r)`

Given a homomorphism of the group G for which r is a relation, return the image of r under f.

`Parent(r)`

Group over which the relation r is taken.

Example H70E4

We may define a group and a set of relations as follows:

```
> F<x, y> := FreeGroup(2);
> rels := { x^2 = y^3, (x*y)^4 = Id(F)} ;
```

To replace one side of a relation, the easiest way is to reassign the relation. So for example, to replace the relation $x^2 = y^3$ by $x^2 = y^4$, we go:

```
> r := x^2 = y^3;
> r := LHS(r) = y^4;
```

70.3 Construction of an FP-Group

An fp-group is normally constructed either by giving a presentation in terms of generators and relations or it is defined to be a subgroup or quotient group of an existing fp-group. However, fp-groups may be also created from finite permutation and matrix groups. Finally, Magma has separate types for a number of families of fp-groups that satisfy a condition such as being polycyclic. Again functions are provided to convert a member of one of these families into a general fp-group.

70.3.1 The Quotient Group Constructor

A group with non-trivial relations is constructed as a quotient of an existing group, usually a free group. For convenience, the necessary free group may be constructed in-line.

`quo< F | R >`

Given an fp-group F, and a set of relations R in the generators of F, construct the quotient G of F by the normal subgroup of F defined by R. The group G is defined by means of a presentation which consists of the relations for F (if any), together with the additional relations defined by the list R.

The expression defining F may be either simply the name of a previously constructed group, or an expression defining an fp-group.

If R is a list then each term of the list is either a *word*, a *relation*, a *relation list* or a *subgroup* of F.

A *word* is interpreted as a relator.

A *relation* consists of a pair of words, separated by '='. (See above.)

A *relation list* consists of a list of words, where each pair of adjacent words is separated by '=': $w_1 = w_2 = \cdots = w_r$. This is interpreted as the set of relations $w_1 = w_r, \ldots, w_{r-1} = w_r$. Note that the relation list construct is only meaningful in the context of the `quo`-constructor.

A *subgroup* H appearing in the list R contributes its generators to the relation set for G, i.e. each generator of H is interpreted as a relator for G.

The group F may be referred to by the special symbol $ in any word appearing to the right of the '`|`' symbol in the `quo`-constructor. Also, in the context of the `quo`-constructor, the identity element (empty word) may be represented by the digit `1`.

The function returns:

(a) The quotient group G;

(b) The natural homomorphism $\phi : F \to G$.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

`G / H`

Given a subgroup H of the group G, construct the quotient of G by the normal closure N of H. The quotient is formed by taking the presentation for G and including the generating words of H as additional relators.

Example H70E5

The symmetric group of degree 4 may be represented as a two generator group with presentation $< a, b \mid a^2, b^3, (ab)^4 >$. Giving the relations as a list of relators, the presentation would be specified as:

```
> F<a, b> := FreeGroup(2);
> G<x, y>, phi := quo< F | a^2, b^3, (a*b)^4 >;
```

Alternatively, giving the relations as a relations list, the presentation would be specified as:

```
> F<a, b> := FreeGroup(2);
> G<x, y>, phi :=  quo< F | a^2 = b^3 = (a*b)^4 = 1>;
```

Finally, giving the relations in the form of a set of relations, this presentation would be specified as:

```
> F<a, b> := FreeGroup(2);
> rels := { a^2, b^3, (a*b)^4 };
> G<x, y>, phi :=  quo< F | rels >;
```

Example H70E6

A group may be defined using the quo-constructor without first assigning a free group. The $ symbol is used to reference the group whose quotient is being formed.

```
> S4<x, y> := quo< FreeGroup(2) | $.1^2, $.2^3, ($.1*$.2)^4 >;
> S4;
Finitely presented group S4 on 2 generators
Relations
      x^2 = Id(S4)
      y^3 = Id(S4)
      (x * y)^4 = Id(S4)
```

Example H70E7

We illustrate the use of the quo-constructor in defining the quotient of a group other than a free group.

```
> F<x, y> := Group< x, y | x^2 = y^3 = (x*y)^7 = 1 >;
> F;
Finitely presented group F on 2 generators
Relations
      x^2 = Id(F)
      y^3 = Id(F)
```

```
      (x * y)^7 = Id(F)
> G<a, b> := quo< F | (x, y)^8 >;
> G;
Finitely presented group G on 2 generators
Relations
      a^2 = Id(G)
      b^3 = Id(G)
      (a * b)^7 = Id(G)
      (a, b)^8 = Id(G)
> Order(G);
10752
```

70.3.2 The FP-Group Constructor

For convenience, a constructor is provided which allows the user to define an fp-group in a single step.

```
Group<  X | R  >
```

Given a list X of variables $x_1, \ldots, x_r$, and a list of relations R over these generators, first construct the free group F on the generators $x_1, \cdots, x_r$ and then construct the quotient of F corresponding to the normal subgroup of F defined by the relations R.

The syntax for the relations R is the same as for the quo-constructor. The function returns:

(a) The quotient group G;

(b) The natural homomorphism $\phi : F \to G$.

Example H70E8

We illustrate the Group-constructor by defining the binary tetrahedral group in terms of the presentation $< r, s \mid r^3 = s^3 = (rs)^2 >$:

```
> G<r, s> := Group< r, s | r^3 = s^3 = (r*s)^2 >;
```

Example H70E9

Again, using the Group-constructor, the group $< r, s, t \mid r^2, s^2, t^2, rst = str = trs >$ would be specified as:

```
> G<r, s, t> := Group<r, s, t | r^2, s^2, t^2, r*s*t = s*t*r = t*r*s>;
```

Example H70E10

In our final example we illustrate the use of functions to represent parametrised families of groups. In the notation of Coxeter, the symbol $(l, m|n, k)$ denotes the family of groups having presentation

$$< a, b \mid a^l, b^m, (a * b)^n, (a * b^{-1})^k > .$$

```
> Glmnk := func< l, m, n, k | Group< a, b | a^l, b^m, (a*b)^n, (a*b^-1)^k > >;
> G<a, b> := Glmnk(3, 3, 4, 4);
> G;
Finitely presented group G on 2 generators
Relations
      a^3 = Id(G)
      b^3 = Id(G)
      (a * b)^4 = Id(G)
      (a * b^-1)^4 = Id(G)
> Order(G);
168
> G<a, b> := Glmnk(2, 3, 4, 5);
> G;
Finitely presented group G on 2 generators
Relations
    a^2 = Id(G)
    b^3 = Id(G)
    (a * b)^4 = Id(G)
    (a * b^-1)^5 = Id(G)
> Order(G);
1
```

Thus $(2, 3 \mid 4, 5)$ is the trivial group.

70.3.3 Construction from a Finite Permutation or Matrix Group

`FPGroup(G)`

Given a finite group G in category GrpPerm or GrpMat, this function returns a finitely presented group F, isomorphic to G, together with the isomorphism $\phi : F \to G$. The generators of F correspond to the generators of G, so this function can be used to obtain a set of defining relations for the given generating set of G.

It should be noted that this function is only practical for groups of order at most a few million. In the case of much larger permutation groups, an isomorphic fp-group can be constructed using the function FPGroupStrong.

FPGroupStrong(G)

Given a finite group G in category GrpPerm or GrpMat, this function returns a finitely presented group F, isomorphic to G, together with the isomorphism $\phi : F \to G$. The generators of F correspond to a set of strong generators of G. If no strong generating set is known for G, one will be constructed.

For a detailed description of this function, in particular for a list of available parameters, we refer to Chapter 58 and Chapter 59, respectively.

FPGroupStrong(G, N)

Given a permutation group G and a normal subgroup N of G, this function returns a finitely presented group F, isomorphic to G/N, together with a homomorphism $\phi : G \to F$.

For a detailed description of this function, we refer to Chapter 58.

Example H70E11

We start with defining the alternating group $G \simeq A_5$ as a permutation group.

```
> G := AlternatingGroup(5);
> G;
Permutation group G acting on a set of cardinality 5
Order = 60 = 2^2 * 3 * 5
    (3, 4, 5)
    (1, 2, 3)
```

Now we create an fp-group F isomorphic to G, using the function FPGroup. The presentation is constructed by computing a set of defining relations for the generators of G, i.e. the generators of the returned fp-group correspond to the generators of G. This defines a homomorphism from F to G, which the function FPGroup returns as second return value.

```
> F<x,y>, f := FPGroup(G);
> F;
Finitely presented group F on 2 generators
Relations
    x^3 = Id(F)
    y^3 = Id(F)
    (x^-1 * y * x * y)^2 = Id(F)
    (x * y^-1 * x * y)^2 = Id(F)
> f;
Mapping from: GrpFP: F to GrpPerm: G
> f(x);
(3, 4, 5)
> f(y);
(1, 2, 3)
```

Using the function FPGroupStrong, we now create another fp-group Fs, isomorphic to G, whose generators correspond to a set of strong generators of G.

```
> Fs<[z]>, fs := FPGroupStrong(G);
```

```
> Fs;
Finitely presented group Fs on 3 generators
Relations
    z[1]^-3 = Id(Fs)
    (z[1]^-1 * z[3]^-1)^2 = Id(Fs)
    z[3]^-3 = Id(Fs)
    z[1]^-1 * z[2]^-1 * z[1] * z[3]^-1 * z[2] = Id(Fs)
    z[2]^-3 = Id(Fs)
    (z[3]^-1 * z[2]^-1)^2 = Id(Fs)
> fs;
Mapping from: GrpFP: Fs to GrpPerm: G
```

Applying the isomorphism fs, we have a look at the strong generating set constructed for G.

```
> [ fs(z[i]) : i in [1..#z] ];
[
    (3, 4, 5),
    (1, 2, 3),
    (2, 3, 4)
]
```

70.3.4 Construction of the Standard Presentation for a Coxeter Group

There is a special MAGMA category GrpPermCox, a subcategory of GrpPerm, for finite Coxeter groups. Here, we describe a function to create from a Coxeter group W a finitely presented group F, isomorphic to W, which is given by the standard Coxeter group presentation. We refer to Chapter 98.

`CoxeterGroup(GrpFP, W)`

Given a finite Coxeter group W in the category GrpFPCox or GrpPermCox, construct a finitely presented group F isomorphic to W, given by a standard Coxeter presentation. The isomorphism from W to F is returned as second return value. The first argument to this function must be the category GrpFP.

`Local` BOOLELT *Default :* `false`

If the parameter `Local` is set to `true`, F is the appropriate subgroup of the FP version of the overgroup of W.

Example H70E12

We construct a Coxeter group W of Cartan type $C5$ and create an isomorphic fp-group F. We can use the isomorphism from W to F to map words in the generators of F to permutation group elements and vice versa.

```
> W := CoxeterGroup("C5");
> F<[s]>, h := CoxeterGroup(GrpFP, W);
> F;
```

```
Finitely presented group F on 5 generators
Relations
    s[3]^2 = Id(F)
    s[2] * s[4] = s[4] * s[2]
    s[1]^2 = Id(F)
    s[1] * s[2] * s[1] = s[2] * s[1] * s[2]
    s[2] * s[5] = s[5] * s[2]
    s[4]^2 = Id(F)
    s[1] * s[3] = s[3] * s[1]
    s[3] * s[4] * s[3] = s[4] * s[3] * s[4]
    s[2]^2 = Id(F)
    s[1] * s[4] = s[4] * s[1]
    s[5]^2 = Id(F)
    (s[4] * s[5])^2 = (s[5] * s[4])^2
    s[3] * s[5] = s[5] * s[3]
    s[1] * s[5] = s[5] * s[1]
    s[2] * s[3] * s[2] = s[3] * s[2] * s[3]
> h;
Mapping from: GrpCox: W to GrpFP: F given by a rule
> h(W.1*W.2);
s[1] * s[2]
> (s[1]*s[2]*s[3]*s[4]) @@ h;
(1, 39, 4, 3, 2)(5, 13, 19, 23, 25)(6, 36, 35, 8, 7)(9, 16, 21,
    24, 17)(10, 33, 32, 31, 11)(12, 18, 22, 15, 20)(14, 29, 28,
    27, 26)(30, 38, 44, 48, 50)(34, 41, 46, 49, 42)(37, 43, 47,
    40, 45)
```

70.3.5 Conversion from a Special Form of FP-Group

Groups that satisfy certain properties, such as being abelian or polycyclic, are known to possess presentations with respect to which the word problem is soluble. Specialised categories have been constructed in Magma for several of these, e.g. the categories GrpGPC, GrpPC and GrpAb. The functions described in this section allow a group created in one of the special presentation categories to be recast as an fp-group.

`FPGroup(G)`

Given a group G, defined either by a polycyclic group presentation (types GrpPC and GrpGPC) or an abelian group presentation (type GrpAb), return a group H isomorphic to G, together with the isomorphism $\phi : G \to H$. The generators for H will correspond to the generators of G. The effect of this function is to convert a presentation in a special form into a general fp-group presentation.

Example H70E13

We illustrate the cast from special forms of fp-groups to the category `GrpFP` by converting a polycyclic group.

```
> G := DihedralGroup(GrpGPC, 0);
> G;
GrpGPC : G of infinite order on 2 PC-generators
PC-Relations:
    G.1^2 = Id(G),
    G.2^G.1 = G.-2
> F := FPGroup(G);
> F;
Finitely presented group F on 2 generators
Relations
    F.1^2 = Id(F)
    F.2^F.1 = F.2^-1
    F.1^-1 * F.2^-1 * F.1 = F.2
```

70.3.6 Construction of a Standard Group

A number of functions are provided which construct presentations for various standard groups.

`AbelianGroup(GrpFP, [n1,...,nr])`

Construct the abelian group defined by the sequence $[n_1, \ldots, n_r]$ of non-negative integers as an fp-group. The function returns the direct product of cyclic groups $C_{n_1} \times C_{n_2} \times \cdots \times C_{n_r}$, where C_0 is interpreted as an infinite cyclic group.

`AlternatingGroup(GrpFP, n)`

`Alt(GrpFP, n)`

Construct the alternating group of degree n as an fp-group, where the generators correspond to the permutations $(3, 4, \ldots, n)$ and $(1, 2, 3)$, for n odd, or $(1, 2)(3, 4, \ldots, n)$ and $(1, 2, 3)$, for n even.

`BraidGroup(GrpFP, n)`

Construct the braid group on n strings ($n - 1$ Artin generators) as an fp-group.

`CoxeterGroup(GrpFP, t)`

Construct the Coxeter group of Cartan type t as a finitely presented group, given by the standard Coxeter presentation. The Cartan type t is passed to this function as a string; we refer to Chapter 97 for details.

`CyclicGroup(GrpFP, n)`

Construct the cyclic group of order n as an fp-group.

`DihedralGroup(GrpFP, n)`

For $n > 2$, return the dihedral group of order $2n$ as an fp-group. For $n = 0$, return the infinite dihedral group as an fp-group.

`ExtraSpecialGroup(GrpFP, p, n : parameters)`

Given a small prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1} in the category `GrpFP`. The isomorphism type of G can be selected using the parameter `Type`.

`Type` MonStgElt *Default :* "+"

Possible values for this parameter are "+" (default) and "−".

If `Type` is set to "+", the function returns for $p = 2$ the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If `Type` is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

`SymmetricGroup(GrpFP, n)`

`Sym(GrpFP, n)`

Construct the symmetric group of degree n as an fp-group, where the generators correspond to the permutations $(1, 2, \ldots, n)$ and $(1, 2)$.

Example H70E14

We create the symmetric group Sym(8) as an fp-group:

```
> S8 := SymmetricGroup(GrpFP, 8);
> S8;
Finitely presented group S8 on 2 generators
Relations
      S8.1^8 = Id(S8)
      S8.2^2 = Id(S8)
      (S8.1 * S8.2)^7 = Id(S8)
      (S8.1^-1 * S8.2 * S8.1 * S8.2)^3 = Id(S8)
      (S8.2 * S8.1^-2 * S8.2 * S8.1^2)^2 = Id(S8)
      (S8.2 * S8.1^-3 * S8.2 * S8.1^3)^2 = Id(S8)
      (S8.2 * S8.1^-4 * S8.2 * S8.1^4)^2 = Id(S8)
```

We create the Coxeter group of Cartan type $F4$ as an fp-group:

```
> F := CoxeterGroup(GrpFP, "F4");
> F;
```

```
Finitely presented group F on 4 generators
Relations
    (F.2 * F.3)^2 = (F.3 * F.2)^2
    F.1^2 = Id(F)
    F.1 * F.3 = F.3 * F.1
    F.2 * F.4 = F.4 * F.2
    F.1 * F.2 * F.1 = F.2 * F.1 * F.2
    F.2^2 = Id(F)
    F.3^2 = Id(F)
    F.3 * F.4 * F.3 = F.4 * F.3 * F.4
    F.4^2 = Id(F)
    F.1 * F.4 = F.4 * F.1
```

70.3.7 Construction of Extensions

`Darstellungsgruppe(G)`

Given an fp-group G, construct a maximal central extension $\tilde{G}$ of G. The group $\tilde{G}$ is created as an fp-group.

`DirectProduct(G, H)`

Given two fp-groups G and H, construct the direct product of G and H.

`DirectProduct(Q)`

Given a sequence Q of r fp-groups, construct the direct product $Q[1] \times \ldots \times Q[r]$.

`FreeProduct(G, H)`

Given two fp-groups G and H, construct the free product of G and H.

`FreeProduct(Q)`

Given a sequence Q of r fp-groups, construct the free product of the groups $Q[1], \ldots, Q[r]$.

Example H70E15

We construct a maximal central extension of the following group of order 36.

```
> G<x1, x2> := Group<x1, x2 | x1^4,(x1*x2^-1)^2,x2^4,(x1*x2)^3>;
> G;
Finitely presented group G on 2 generators
Relations
       x1^4 = Id(G)
       (x1 * x2^-1)^2 = Id(G)
       x2^4 = Id(G)
       (x1 * x2)^3 = Id(G)
> D := Darstellungsgruppe(G);
> D;
Finitely presented group D on 4 generators
Relations
       D.1^4 * D.3^-1 * D.4^2 = Id(D)
       D.1 * D.2^-1 * D.1 * D.2^-1 * D.4 = Id(D)
       D.2^4 = Id(D)
       D.1 * D.2 * D.1 * D.2 * D.1 * D.2 * D.4 = Id(D)
       (D.1, D.3) = Id(D)
       (D.2, D.3) = Id(D)
       (D.1, D.4) = Id(D)
       (D.2, D.4) = Id(D)
       (D.3, D.4) = Id(D)
> Index(D, sub< D | >);
108
```

Thus, a maximal central extension of G has order 108

Example H70E16

We create the direct product of the alternating group of degree 5 and the cyclic group of order 2.

```
> A5 := Group<a, b | a^2, b^3, (a*b)^5 >;
> Z2 := quo< FreeGroup(1) | $.1^2 >;
> G := DirectProduct(A5, Z2);
> G;
Finitely presented group G on 3 generators
Relations
      G.1^2 = Id(G)
      G.2^3 = Id(G)
      (G.1 * G.2)^5 = Id(G)
      G.3^2 = Id(G)
      G.1 * G.3 = G.3 * G.1
      G.2 * G.3 = G.3 * G.2
```

70.3.8 Accessing the Defining Generators and Relations

The functions in this group provide access to basic information stored for a finitely-presented group G.

`G . i`

The i-th defining generator for G. A negative subscript indicates that the inverse of the generator is to be created. $G.0$ is `Identity(G)`, the empty word in G.

`Generators(G)`

A set containing the generators for the group G.

`NumberOfGenerators(G)`
`Ngens(G)`

The number of generators for the group G.

`PresentationLength(G)`

The total length of the relators for G.

`Relations(G)`

A sequence containing the defining relations for G.

70.4 Homomorphisms

For a general description of homomorphisms, we refer to Chapter 16. This section describes some special aspects of homomorphisms the domain of which is a finitely presented group.

70.4.1 General Remarks

The kernel of a homomorphism with a domain of type `GrpFP` can be computed using the function `Kernel`, if the codomain is of one of the types `GrpGPC`, `GrpPC` (cf. Chapter 63), `GrpAb` (cf. Chapter 69), `GrpPerm` (cf. Chapter 58), `GrpMat` (cf. Chapter 59), `ModAlg` or `ModGrp` (cf. Chapter 89), if the image is finite and its order sufficiently small. In this case, a regular permutation representation of the image is constructed and the kernel is created as a subgroup of the domain, defined by a coset table.

The kernel may also be computable, if the codomain is of the type `GrpFP`, the image is sufficiently small and a presentation for the image is known.

If the kernel of a map can be computed successfully, forming preimages of substructures is possible. An attempt to compute the kernel of a map will be made automatically, if the preimage of a substructure of the codomain is to be computed.

Note, that trying to compute the kernel may be very time and memory consuming; use this feature with care.

70.4.2 Construction of Homomorphisms

```
hom< P -> G | S >
```

Returns the homomorphism from the fp-group P to the group G defined by the assignment S. S can be the one of the following:

(i) A list, sequence or indexed set containing the images of the n generators $P.1, \ldots, P.n$ of P. Here, the i-th element of S is interpreted as the image of $P.i$, i.e. the order of the elements in S is important.

(ii) A list, sequence, enumerated set or indexed set, containing n tuples $< x_i, y_i >$ or arrow pairs $x_i -> y_i$, where x_i is a generator of P and $y_i \in G$ $(i = 1, \ldots, n)$ and the set $\{x_1, \ldots, x_n\}$ is the full set of generators of P. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

Note, that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of P. It is the user's responsibility to ensure that the arguments passed to the `hom`-constructor actually yield a well-defined homomorphism. For certain codomain categories, this may be checked using the function `IsSatisfied` described below.

```
IsSatisfied(U, E)
```

U is a set or sequence of either words belonging to an n-generator fp-group H or relations over H. E is a sequence of n elements $[e_1, \ldots, e_n]$ belonging to a group G for which both, multiplication and comparison of elements are possible. Using the mapping $H.i \to e_i$ $(i = 1, \ldots, n)$, we evaluate the relations given by U. If U is a set or sequence of relations, the left and right hand sides of each relation are evaluated and compared for equality. Otherwise, each word in U is evaluated and compared to the identity. If all relations are satisfied, `IsSatisfied` returns the Boolean value `true`. On the other hand, if any relation is not satisfied, `IsSatisfied` returns the value `false`.

This function may be used to verify the correctness of the definition of a homomorphism from an fp-group to a group in a category for which both, multiplication and comparison of elements are possible.

70.4.3 Accessing Homomorphisms

```
w @ f
```

```
f(w)
```

Given a homomorphism whose domain is an fp-group G and an element w of G, return the image of w under f as an element of the codomain of f.

```
H @ f
```

```
f(H)
```

Given a homomorphism whose domain is an fp-group G and a subgroup H of G, return the image of H under f as a subgroup of the codomain of f.

Some maps do not support images of subgroups.

`g @@ f`

Given a homomorphism whose domain is an fp-group G and an element g of the image of f, return the preimage of g under f as an element of G.

Some maps do not support inverse images.

`H @@ f`

Given a homomorphism whose domain is an fp-group G and a subgroup H of the image of f, return the preimage of H under f as a subgroup of G.

Some maps do not support inverse images. The inverse image of a subgroup of the codomain can only be computed if the kernel of the homomorphism can be computed, i.e. if the kernel has moderate index in the domain.

`Domain(f)`

The domain of the homomorphism f.

`Codomain(f)`

The codomain of the homomorphism f.

`Image(f)`

The image or range of the homomorphism f as a subgroup of the codomain of f.

Some maps do not support this function.

`Kernel(f)`

The kernel of the homomorphism f as a (normal) subgroup of the domain of f, represented by a coset table.

Some maps do not support this function. The kernel of a homomorphism can only be computed, if it has moderate index in the domain.

Example H70E17

For arbitrary $n > 0$, the symmetric group of degree $n + 1$ is an epimorphic image of the braid group on n generators. In this example, we exhibit this relationship for $n = 4$.
We start with creating the braid group B on 5 strings, i.e. 4 Artin generators.

```
> B := BraidGroup(GrpFP, 5);
> B;
Finitely presented group B on 4 generators
Relations
    B.1 * B.2 * B.1 = B.2 * B.1 * B.2
    B.1 * B.3 = B.3 * B.1
    B.1 * B.4 = B.4 * B.1
    B.2 * B.3 * B.2 = B.3 * B.2 * B.3
    B.2 * B.4 = B.4 * B.2
```

```
    B.3 * B.4 * B.3 = B.4 * B.3 * B.4
```

In the symmetric group of degree 5, we define 4 transpositions which will be the images of the generators of B.

```
> S := SymmetricGroup(5);
> imgs := [ S!(1,2), S!(2,3), S!(3,4), S!(4,5) ];
```

In order to verify that this assignment actually gives rise to a well defined homomorphism, we check whether the potential images satisfy the defining relations of B.

```
> rels := Relations(B);
> rels;
[ B.1 * B.2 * B.1 = B.2 * B.1 * B.2, B.1 * B.3 = B.3 * B.1,
  B.1 * B.4 = B.4 * B.1, B.2 * B.3 * B.2 = B.3 * B.2 * B.3,
  B.2 * B.4 = B.4 * B.2, B.3 * B.4 * B.3 = B.4 * B.3 * B.4 ]
> IsSatisfied(rels, imgs);
true
```

They do. So we can define the homomorphism from B to S.

```
> f := hom< B->S | imgs >;
```

We see that f is surjective, i.e. S is an epimorphic image of B as claimed above.

```
> f(B) eq S;
true
```

We now check the kernel of f.

```
> Kernel(f);
Finitely presented group
Index in group B is 120 = 2^3 * 3 * 5
Subgroup of group B defined by coset table
```

Using the function GeneratingWords described later, we can obtain a set of generators of $\ker(f)$ as a subgroup of B.

```
> GeneratingWords(B, Kernel(f));
{ B.2^-2, (B.1 * B.2 * B.3^-1 * B.2^-1 * B.1^-1)^2, B.1^-2,
  (B.3 * B.4^-1 * B.3^-1)^2, (B.2 * B.3^-1 * B.2^-1)^2,
  (B.2 * B.3 * B.4^-1 * B.3^-1 * B.2^-1)^2, B.4^-2,
  (B.1 * B.2^-1 * B.1^-1)^2, B.3^-2,
  (B.1 * B.2 * B.3 * B.4^-1 * B.3^-1 * B.2^-1 * B.1^-1)^2 }
```

It is easy to see that all generators of $\ker(f)$ are conjugates of words of the form $g^{\pm 2}$, where g is a generator of B. We check this, using the normal closure constructor ncl described later.

```
> Kernel(f) eq ncl< B | B.1^2, B.2^2, B.3^2, B.4^2 >;
true
```

Thus, the braid relations together with the relations $B.1^2, B.2^2, B.3^2, B.4^2$ are a set of defining relations for S.

70.4.4 Computing Homomorphisms to Finite Groups

This section describes functions for computing representatives of the classes of homomorphisms from a finitely presented group F to a finite group G modulo a group A of automorphisms of G.

`Homomorphisms(F, G, A` : *parameters*`)`

`Homomorphisms(F, G` : *parameters*`)`

Given a finitely presented group F and two permutation groups G and A with $G \triangleleft A$, return a sequence containing representatives of the classes of homomorphisms from F to G modulo automorphisms of G induced by elements of A. (That is, two homomorphisms f_1, $f_2 : F \to G$ are considered equivalent if there exists an element $a \in A$ such that $f_1(x) = f_2(x)^a$ for all $x \in F$.) The call `Homomorphisms(F, G)` is equivalent to `Homomorphisms(F, G, G)`.

The function uses a backtrack algorithm testing certain conjugates of representatives of the A-classes in G as possible images of the generators of F.

The following parameters are available for this function.

`Surjective` BoolElt *Default :* `true`

If this parameter is set to `true` (default), only epimorphisms are considered.

`Limit` RngIntElt *Default :* 0 (no limit)

If this parameter is set to n, the function terminates after n classes of homomorphisms satisfying the specified conditions have been found. A value of 0 (default) means no limit.

`TimeLimit` RngIntElt *Default :* 0 (no limit)

A limit in seconds for the amount of time spent in the backtrack search for homomorphisms. The time spent in initial coset enumerations is *not* counted towards this limit. A value of 0 (default) means no limit.

`CosetEnumeration` BoolElt *Default :* `true`

If this parameter is set to `true` (default), a number of short coset enumerations are performed for each generator of F in order to check whether some A-classes in G can be ruled out as possible images for this generator. If a class can be ruled out, the complexity of the backtrack search may be reduced significantly.

In situations where it seems unlikely that classes can be ruled out as possible generator images, experienced users may wish to turn this feature off in order to save the time spent on the coset enumerations.

`CacheCosetAction` BoolElt *Default :* `true`

The value of this parameter indicates whether the actions of A on the cosets of the centralisers of representatives of the A-classes in G are cached during the backtrack search.

Setting this parameter to `true` (default) results in faster computations and is recommended for normal applications. When computing homomorphisms to large

groups with many conjugate classes, this parameter can be set to `false` in order to reduce memory requirements at the expense of increased computing time.

Example H70E18

Consider the finitely presented group

$$F := < a, b, c \mid ac^{-1}bc^{-1}aba^{-1}b, abab^{-1}c^2b^{-1}, a^2b^{-1}(ca)^4cb^{-1} > .$$

```
> F := Group< a,b,c | a*c^-1*b*c^-1*a*b*a^-1*b,
>                     a*b*a*b^-1*c^2*b^-1,
>                     a^2*b^-1*c*a*c*a*c*a*c*a*c*b^-1 >;
```

We use the function Homomorphisms to prove that F maps onto A_5.

```
> G := Alt(5);
> homs := Homomorphisms(F, G : Limit := 1);
> #homs gt 0;
true
```

`Homomorphisms(F, G, A : parameters)`

`Homomorphisms(F, G : parameters)`

Given a finitely presented group F and two finite polycyclic groups G and A with $G \triangleleft A$, return a sequence containing representatives of the classes of homomorphisms from F to G modulo automorphisms of G induced by elements of A. (That is, two homomorphisms f_1, $f_2 : F \to G$ are considered equivalent if there exists an element $a \in A$ such that $f_1(x) = f_2(x)^a$ for all $x \in F$.) The call `Homomorphisms(F, G)` is equivalent to `Homomorphisms(F, G, G)`.

The following parameters are available for this function.

`Surjective` BoolElt *Default :* `true`

If this parameter is set to `true` (default), only epimorphisms are considered.

`Limit` RngIntElt *Default :* 0 (no limit)

If this parameter is set to n, the function terminates after n classes of homomorphisms satisfying the specified conditions have been found. A value of 0 (default) means no limit.

70.4.4.1 Computing Homomorphisms to Permutation Groups Interactively

A process version of the algorithm used by Homomorphisms is available for computing homomorphisms one at a time. The functions relevant for this interactive version are described in this section.

`HomomorphismsProcess(F, G, A : parameters)`

`HomomorphismsProcess(F, G : parameters)`

Given a finitely presented group F and two permutation groups G and A with $G \triangleleft A$, return a process P for computing representatives of the classes of homomorphisms from F to G modulo automorphisms of G induced by elements of A. (That is, two homomorphisms $f_1, f_2 : F \to G$ are considered equivalent if there exists an element $a \in A$ such that $f_1(x) = f_2(x)^a$ for all $x \in F$.) `HomomorphismsProcess(F, G)` is equivalent to `HomomorphismsProcess(F, G, G)`.

After constructing the process, the search for homomorphisms is started and runs until the first homomorphism is found, the time limit is reached (in which case P is marked as *invalid*) or the search is completed without finding a homomorphism (in which case P is marked as *empty*).

The parameters have the same meaning as for the function Homomorphisms.

`Surjective`	BoolElt	*Default :* `true`
`Limit`	RngIntElt	*Default :* 0 (no limit)
`TimeLimit`	RngIntElt	*Default :* 0 (no limit)
`CosetEnumeration`	BoolElt	*Default :* `true`
`CacheCosetAction`	BoolElt	*Default :* `true`

Setting a time limit for a process P limits the total amount of time spent in the backtrack search for homomorphisms during the construction of P and in subsequent calls to NextElement and Complete. The time spent in initial coset enumerations is *not* counted towards this limit.

If the time limit or the limit on the number of homomorphisms set for a process P is reached, P becomes *invalid*. Calling NextElement or Complete for an invalid process or for a process which is *empty*, that is, which has found all possible classes of homomorphisms, will cause a runtime error. The functions IsValid and IsEmpty can be used to check whether a process is valid or empty, respectively. The use of these functions is recommended to avoid runtime errors in loops or user written functions.

`NextElement(~P)`

Given a valid and non-empty process P, continue the backtrack search until a new class of homomorphisms is found. If the search completes without a new representative being found, P is marked as empty. If a limit set for P is reached, P is marked as invalid.

`Complete(~P)`

Given a valid and non-empty process P, continue the search for homomorphisms until all classes of homomorphisms have been found or a limit set for P is reached. If the search for homomorphisms completes, P is marked as empty. If a limit set for P is reached, P is marked as invalid.

`IsEmpty(P)`

Returns whether P is empty, that is, whether all possible classes of homomorphisms have been found.

`IsValid(P)`

Returns `false` if a limit set for P has been reached and `true` otherwise. Note that the return value of `IsValid` is not related to whether or not P currently defines a homomorphism.

`DefinesHomomorphism(P)`

Returns whether P currently defines a homomorphism which can be extracted using the function `Homomorphism`.

`Homomorphism(P)`

Given a process P which defines a homomorphism, return this homomorphism, that is, the homomorphism most recently found by P. If P does not define a homomorphism, a runtime error will result. The function `DefinesHomomorphism` can be used to test whether a call to `Homomorphism` is legal for a process.

`#P`

Return the number of homomorphisms that have been found by the process P.

`Homomorphisms(P)`

Return a sequence containing all homomorphisms that have been found by the process P. Note that the sequence will only contain a complete set of representatives of the classes of homomorphisms if P is empty, that is, if the backtrack search for P has been completed.

Example H70E19

Consider the braid group B on 4 strings. We show how the interactive computation of homomorphisms can be used to determine a homomorphism $f : B \to \mathrm{PSL}(2, 16)$ whose image is a maximal subgroup of $\mathrm{PSL}(2, 16)$.

Note how the functions `IsValid`, `IsEmpty` and `DefinesHomomorphism` are used to avoid runtime errors in the `while` loop.

```
> B := BraidGroup(GrpFP, 4);
> G := PSL(2,16);
> P := HomomorphismsProcess(B, G : Surjective := false,
```

```
>                                  TimeLimit := 10);
> while not IsEmpty(P) do
>    if DefinesHomomorphism(P) then
>       f := Homomorphism(P);
>       img := Image(f);
>       if IsMaximal(G,img) then
>          print "found image which is maximal subgroup";
>          break;
>       end if;
>    end if;
>    if IsValid(P) then
>       NextElement(~P);
>     else
>       print "Limit has been reached";
>       break;
>    end if;
> end while;
found image which is maximal subgroup
>
> f;
Homomorphism of GrpFP: B into GrpPerm: G, induced by
  B.1 |--> (1, 4, 3, 6, 12)(2, 11, 9, 16, 7)(5, 17, 8, 15, 13)
  B.2 |--> (1, 4, 2, 10, 16)(3, 11, 9, 12, 14)(5, 15, 8, 13, 17)
  B.3 |--> (1, 4, 3, 6, 12)(2, 11, 9, 16, 7)(5, 17, 8, 15, 13)
```

Example H70E20

This example sketches how the interactive version of the homomorphism algorithm could be used as part of a function trying to prove that a group is infinite.

Note again how the functions IsValid, IsEmpty and DefinesHomomorphism are used to avoid runtime errors.

```
> function MyIsInfinite(F)
>
>  // ...
>
>  // quotient approach: check whether an obviously infinite
>  //    normal subgroup can be found in reasonable time.
>  S := [ Alt(5), PSL(2,7), PSL(2,9), PSL(2,11) ];
>  for G in S do
>     P := HomomorphismsProcess(F, G : Surjective := false,
>                                      TimeLimit := 5);
>     while IsValid(P) and not IsEmpty(P) do
>       if DefinesHomomorphism(P) then
>          f := Homomorphism(P);
>          if 0 in AQInvariants(Kernel(f)) then
>             print "found infinite normal subgroup";
```

```
>             print "Hence group is infinite";
>             return true;
>          end if;
>       end if;
>       if IsValid(P) then
>          NextElement(~P);
>       end if;
>    end while;
>  end for;
>  print "quotient approach failed; trying other strategies";
>
>  // ...
>
> end function;
```

We try the code fragment on the group

$$< a, b \mid ab^{-1}a^{-1}ba^{-1}b^{-1}abb, ab^{-1}a^{-1}baaba^{-1}b^{-1}ab^{-1}a^{-1}baba^{-1}b^{-1} > .$$

```
> F := Group< a,b |
>    a*b^-1*a^-1*b*a^-1*b^-1*a*b*b,
>    a*b^-1*a^-1*b*a*a*b*a^-1*b^-1*a*b^-1*a^-1*b*a*b*a^-1*b^-1 >;
> MyIsInfinite(F);
found infinite normal subgroup
Hence group is infinite
true
```

70.4.4.2 Finding Homomorphisms onto Simple Groups

We describe utilities for finding homomorphisms onto simple groups. As in the previous example, this may be useful when the presentation defines a perfect group. The methods used are similar to the example, with a list of simple groups to try, and using the function Homomorphisms.

The list of simple groups supplied in V2.10 contains all non-abelian simple groups with order $\leq 10^9$. Such a list is dominated by $PSL(2, q)$'s with q odd. In this implementation these $PSL(2, q)$'s are treated as an infinite family rather than stored individually, and so continue beyond the above limit.

`SimpleQuotients(F, deg1, deg2, ord1, ord2:` *parameters*`)`

`SimpleQuotients(F, ord1, ord2:` *parameters*`)`

`SimpleQuotients(F, ord2:` *parameters*`)`

`Family`	ANY	*Default :* "*All*"
`Limit`	RNGINTELT	*Default :* 1
`HomLimit`	RNGINTELT	*Default :* 0

Uses Homomorphisms to find epimorphisms from F onto simple groups in a fixed list. The arguments $deg1$ and $deg2$ are respectively lower and upper bounds for the degree of the image group. If the degree arguments are not present then bounds of 5 and 10^7 are used. The arguments $ord1$ and $ord2$ are respectively lower and upper bounds for the orders of the image group. (Setting $ord2$ low enough is particularly important if a quick search is wanted.) If $ord1$ is not given then it defaults to 1.

The return value is a list of sequences of epimorphisms found. Each sequence contains epimorphisms onto one simple group. The parameter `Limit` limits the number of successful searches to be carried out by Homomorphisms. The default value is 1, so by default the search terminates with the first simple group found to be a homomorphic image of F.

The parameter `HomLimit` limits the number of homomorphisms that will be searched for by any particular call to Homomorphisms. It defaults to zero, so that all homomorphisms for any group found will be returned.

The parameter `Family` selects sublists of the main list to search. Possible values of this parameter are `"All"`, `"PSL"`, `"PSL2"`, `"Mathieu"`, `"Alt"`, `"PSp"`, `"PSU"`, `"Other"`, and `"notPSL2"`; sets of these strings are also allowed, which searches on the union of the appropriate sublists.

`SimpleQuotientProcess(F, deg1, deg2, ord1, ord2: parameters)`

`Family` ANY *Default : "All"*

Produce a record that defines a process for searching for simple quotients of F as SimpleQuotients does. Calling this function sets up the record and conducts the initial search until a quotient is found. Continuing the search for another quotient is done by calling NextSimpleQuotient. Extracting the epimorphisms found is achieved using SimpleEpimorphisms, and testing if the process has expired is the task of IsEmptySimpleQuotientProcess.

`NextSimpleQuotient(~P)`

When P is a record returned by SimpleQuotientProcess, advance the search to the next simple group which is a homomorphic image of the finitely presented group. Does nothing if the process has expired.

`IsEmptySimpleQuotientProcess(P)`

When P is a record returned by SimpleQuotientProcess, test whether or not P has expired.

`SimpleEpimorphisms(P)`

When P is a record returned by SimpleQuotientProcess, extract the most recently found epimorphisms onto a simple group, plus a tuple describing the image group. This is a valid operation when IsEmptySimpleQuotientProcess returns `false`.

Example H70E21

We take a perfect finitely presented group and search for simple quotients.

```
> F := Group<a,b,c|a^13,b^3,c^2,a = b*c>;
> IsPerfect(F);
true
> L := SimpleQuotients(F,1, 100, 2, 10^5:Limit := 2);
> #L;
2
> for x in L do CompositionFactors(Image(x[1])); end for;
    G
    |  A(1, 13)               = L(2, 13)
    1
    G
    |  A(2, 3)                = L(3, 3)
    1
> L[2,1];
Homomorphism of GrpFP: F into GrpPerm: $, Degree 13, Order
2^4 * 3^3 * 13 induced by
F.1 |--> (1, 10, 4, 5, 11, 8, 3, 6, 7, 12, 9, 13, 2)
F.2 |--> (2, 10, 4)(3, 6, 7)(5, 11, 13)(8, 12, 9)
F.3 |--> (1, 10)(2, 5)(3, 12)(8, 13)
> #L[2];
2
```

We've found L(2,13) and L(3,3) as images, with 2 inequivalent homomorphisms onto the second. We'll try a process looking through a smaller family.

```
> P := SimpleQuotientProcess(F,1, 100, 2, 10^6:Family:="PSU");
> IsEmptySimpleQuotientProcess(P);
false
> eps, info := SimpleEpimorphisms(P);
> info;
<65, 62400, PSU(3, 4)>
```

We've found PSU(3,4) of order 62400 and degree 65 as an image. We continue with this process.

```
> NextSimpleQuotient(~P);
> IsEmptySimpleQuotientProcess(P);
true
```

No, there are no more within the limits given.

70.4.5 The L_2-Quotient Algorithm

Given a finitely presented group G on two generators, the L_2-quotient algorithm of Plesken and Fabianska [PF09] computes all quotients of G which are isomorphic to some $\mathrm{PSL}(2,q) = L_2(q)$, simultaneously for any prime power q. It can handle the case of infinitely many quotients, and also works for very large prime powers.

Note that, at the moment, the algorithm does not return images onto the groups $\mathrm{PSL}(2,2)$, $\mathrm{PSL}(2,3)$, and $\mathrm{PSL}(2,4) = \mathrm{PSL}(2,5)$.

`L2Quotients(G)`

This is the main method. It takes as parameter a finitely presented group on two generators, and returns a list of prime ideals of $Z[x_1, x_2, x_{12}]$. These prime ideals contain all information about the L_2-quotients.

`L2Type(P)`

For a prime ideal P in $Z[x_1, x_2, x_{12}]$, this method returns a string describing the L_2-type which the ideal encodes. The possible types are "`reducible`", "`dihedral`", "`Alt(4)`", "`Sym(4)`", "`Alt(5)`", "`infinite (characteristic zero)`", "`infinite (characteristic p)`", "`PGL(2, q)`", and "`PSL(2, q)`". Note that for prime ideals which are returned by `L2Quotients` or `L2Ideals`, only the last four types can occur; the other types are eliminated in these methods.

`L2Generators(P)`

Given a maximal ideal M in $Z[x_1, x_2, x_{12}]$, compute two matrices with entries in $Z[x_1, x_2, x_{12}]/M$ corresponding to M.

`L2Ideals(I)`

Given an ideal I in $Z[x_1, x_2, x_{12}]$, compute the minimal associated primes of I which give rise to L_2-quotients. This is mainly used for ideals returned by `L2Quotients` which encode infinitely many L_2 images.

Example H70E22

The first few examples are taken from Conder, Havas and Newman [CHN11].

```
> Gamma< x, y > := Group< x, y | x^2, y^3 >;
> u := x*y; v := x*y^-1;
> G := quo< Gamma | u^10*v^2*u*v*u*v^2 >;
> quot := L2Quotients(G); quot;
[
Ideal of Polynomial ring of rank 3 over Integer Ring
Order: Lexicographical
Variables: x1, x2, x12
Inhomogeneous, Dimension 0
Groebner basis:
[
    x1,
```

```
    x2 + 1,
    x12^2 + 4*x12 + 2,
    5
]
]
```

`L2Quotients` returns only one ideal, so there is only one L_2 quotient. We can use `L2Type` to check what group this is.

```
> L2Type(quot[1]);
PSL( 2, 5^2 )
```

Finally, `L2Generators` returns matrices in $\mathrm{SL}(2, 25)$ which map onto generators in $\mathrm{PSL}(2, 25)$.

```
> L2Generators(quot[1]);
MatrixGroup(2, GF(5^2))
Generators:
    [     3 $.1^21]
    [     0      2]
    [     0      4]
    [     1      4]
Mapping from: MatrixGroup(2, GF(5^2)) to GL(2, ext<GF(5) | Polynomial(GF(5),
\[3, 3, 1])>)
```

There are other quotients of the modular group with only finitely many L_2 quotients:

```
> G := quo< Gamma | u^3*v*u^3*v*u^3*v^2*u*v^2 >;
> quot := L2Quotients(G);
> [L2Type(P) : P in quot];
[ PGL( 2, 13 ) ]
>
> G := quo< Gamma | u^3*v*u^3*v^2*u*v^3*u*v^2 >;
> quot := L2Quotients(G);
> [L2Type(P) : P in quot];
[]
```

This tells us that the first group has only one L_2 quotient, isomorphic to $\mathrm{PGL}(2, 13)$, and the second group has no L_2 quotients at all.

Example H70E23

Next, we look at Coxeter presentations of the form

$$(l, m|n, k) = \langle x, y | x^l, y^m, (xy)^n, (x^(- 1)y)^k \rangle$$

and

$$(l, m, n; q) = \langle x, y | x^l, y^m, (xy)^n, [x, y]^k \rangle$$

for various values of l, m, n, k, q.

```
> G := Group< x,y | x^8, y^9, (x*y)^5, (x^-1*y)^7 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 71 ), PSL( 2, 71 ), PSL( 2, 71 ), PSL( 2, 2521 ), PSL( 2, 239 ),
```

```
PSL( 2, 449 ), PSL( 2, 3876207679 ), PSL( 2, 1009 ), PSL( 2, 29113631 ),
PSL( 2, 41 ), PSL( 2, 3056201 ) ]
```

The group $\mathrm{PSL}(2,71)$ occurs three times. This means that there are three essentially different epimorphisms of G onto $\mathrm{PSL}(2,71)$, i.e., there is no automorphism of $\mathrm{PSL}(2,71)$ which transforms one epimorphism into another.
Here is another example, this time for the other family.

```
> G := Group< x, y | x^4, y^3, (x*y)^5, (x,y)^6 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 79 ), PSL( 2, 3^2 ), PSL( 2, 5^2 ) ]
```

There are three groups in the second family for which it is not known whether they are finite or infinite (Havas & Holt (2010) [HH10]). They are $(3,4,9;2)$, $(3,4,11;2)$, and $(3,5,6;2)$. Each of them has only one L_2 image.

```
> G := Group< x, y | x^3, y^4, (x*y)^9, (x,y)^2 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 89 ) ]
> G := Group< x, y | x^3, y^4, (x*y)^11, (x,y)^2 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 769 ) ]
> G := Group< x, y | x^3, y^5, (x*y)^6, (x,y)^2 >;
> [L2Type(P) : P in L2Quotients(G)];
[ PSL( 2, 61 ) ]
```

The algorithm currently only works for finitely presented groups on two generators. However, it is sometimes possible to reduce the number of generators for finitely presented groups on more than two generators.
The following examples are taken from Cavicchioli, O'Brien and Spaggiari [COS08], with code supplied by Eamonn O'Brien.

```
> CHR := function (X)
>    n := X[1]; m := X[2]; k := X[3];
>    F := FreeGroup (n);
>    R := [];
>    for i in [1..n] do
>       a := i + m;
>       if a gt n then repeat a := a - n; until a le n; end if;
>       b := i + k;
>       if b gt n then repeat b := b - n; until b le n; end if;
>       Append (~R, F.i * F.a = F.b);
>    end for;
>    Q := quo < F | R >;
>    return Q;
> end function;
> G := CHR([9, 1, 3]); G;
Finitely presented group G on 9 generators
Relations
    G.1 * G.2 = G.4
    G.2 * G.3 = G.5
```

```
    G.3 * G.4 = G.6
    G.4 * G.5 = G.7
    G.5 * G.6 = G.8
    G.6 * G.7 = G.9
    G.7 * G.8 = G.1
    G.8 * G.9 = G.2
    G.9 * G.1 = G.3
> H := ReduceGenerators(G); H;
Finitely presented group H on 2 generators
Generators as words in group G
    H.1 = G.2
    H.2 = G.5
Relations
    H.2^-2 * H.1^-2 * H.2^-1 * H.1^2 * H.2 * H.1 * H.2^-1 * H.1 = Id(H)
    H.2^-1 * H.1^-1 * H.2^-2 * H.1^-1 * H.2^-1 * H.1 * H.2^-2 * H.1^-1 * H.2
    * H.1^-1 = Id(H)
> L2Quotients(H);
[
    Ideal of Polynomial ring of rank 3 over Integer Ring
    Order: Lexicographical
    Variables: x1, x2, x12
    Inhomogeneous, Dimension 0
    Groebner basis:
    [
        x1 + x12,
        x2 + x12,
        x12^3 + x12 + 1,
        2
    ]
]
> [L2Type(Q) : Q in $1];
[ PSL( 2, 2^3 ) ]
>
> G := CHR([9, 1, 4]);
> H := ReduceGenerators(G); H;
Finitely presented group H on 2 generators
Generators as words in group G
    H.1 = G.1
    H.2 = G.5
Relations
    H.2 * H.1 * H.2^-1 * H.1 * H.2^-2 * H.1^2 * H.2^2 * H.1^-1 * H.2 = Id(H)
    H.1^-1 * H.2 * H.1^-1 * H.2^2 * H.1^-1 * H.2^3 * H.1^-1 * H.2 * H.1^-1 * H.2^2
    * H.1^-1 * H.2 * H.1^-1 = Id(H)
> [L2Type(Q) : Q in L2Quotients(H)];
[]
```

Example H70E24

The algorithm also handles the case in which there are infinitely many L2 quotients. This is the case if one of the prime ideals returned by `L2Quotients` is not maximal. They can be handled theoretically to get a precise list of all images. The functions in Magma can be used to get information for a specified ideal containing the prime ideal.
We start again with a quotient of the modular group.

```
> Gamma< x, y > := Group< x, y | x^2, y^3 >;
> u := x*y; v := x*y^-1;
> G := quo< Gamma | u^5*v*u*v*u*v^5*u*v*u*v >;
> quot := L2Quotients(G);
> [L2Type(P) : P in quot];
[ infinite (characteristic zero) ]
```

This means that there are infinitely many L_2 quotients of G (which already proves that G is infinite). Let's look at the prime ideal.

```
> P := quot[1]; P;
Ideal of Polynomial ring of rank 3 over Integer Ring
Order: Lexicographical
Variables: x1, x2, x12
Inhomogeneous, Dimension 0
Groebner basis:
[
    x1,
    x2 + 1,
    x12^8 - 5*x12^6 + 6*x12^4 - 1
]
```

The residue class ring of P is a finite algebraic extension S of Z, and the algorithm found a homomorphism of G into $\mathrm{PSL}(2, S)$. Since S has epimorphisms onto finite fields in every characteristic, this gives homomorphisms of G into $\mathrm{PSL}(2, p^k)$ for every prime p and suitable k. To check what images there are in characteristic p, we construct the ideal $I = \langle p \rangle + P$. This is no longer prime in general. We can use `L2Ideals` to compute the associated primes of I; this method also removes all ideals which don't give rise to epimorphisms.

```
> R := Generic(P);
> I := ideal< R | 2 > + P;
> quot := L2Ideals(I);
> [L2Type(Q) : Q in quot];
[ PSL( 2, 2^4 ) ]
```

So G has one quotient isomorphic to $\mathrm{PSL}(2, 16)$, and we can construct images of the generators of G.

```
> L2Generators(quot[1]);
MatrixGroup(2, GF(2^4))
Generators:
    [     1  $.1^6]
    [     0      1]
```

```
    [     0      1]
    [     1      1]
Mapping from: MatrixGroup(2, GF(2^4)) to GL(2, ext<GF(2) | Polynomial(GF(2),
\[1, 0, 0, 1, 1])>)
```

We can check the images in various other characteristics:

```
> for p in PrimesInInterval(3, 19) do
>     [L2Type(Q) : Q in L2Ideals(ideal< R | p > + P)];
> end for;
[ PGL( 2, 3^4 ) ]
[ PGL( 2, 5^2 ) ]
[ PSL( 2, 7^2 ), PSL( 2, 7^2 ), PGL( 2, 7^2 ) ]
[ PGL( 2, 11 ), PGL( 2, 11 ), PGL( 2, 11^2 ) ]
[ PSL( 2, 13^2 ), PSL( 2, 13^2 ), PSL( 2, 13^2 ), PSL( 2, 13^2 ) ]
[ PSL( 2, 17^4 ), PSL( 2, 17^4 ) ]
[ PSL( 2, 19 ), PSL( 2, 19 ), PSL( 2, 19^2 ), PSL( 2, 19^2 ), PGL( 2, 19 ) ]
```

There is virtually no limit on the size of the prime:

```
> p := RandomPrime(200);
> p;
307941320171307176726971038693755343299358400663182805777637
> I := ideal< R | p > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PGL( 2, 307941320171307176726971038693755343299358400663182805777637^2 ),
PGL( 2, 307941320171307176726971038693755343299358400663182805777637^2 ) ]
> p := NextPrime(p);
> I := ideal< R | p > + P;
> quot := L2Ideals(I);
> [L2Type(Q) : Q in quot];
[ PSL( 2, 307941320171307176726971038693755343299358400663182805777879 ),
PSL( 2, 307941320171307176726971038693755343299358400663182805777879 ),
PSL( 2, 307941320171307176726971038693755343299358400663182805777879^2 ),
PSL( 2, 307941320171307176726971038693755343299358400663182805777879^2 ),
PGL( 2, 307941320171307176726971038693755343299358400663182805777879 ) ]
> L2Generators(quot[1]);
MatrixGroup(2, GF(307941320171307176726971038693755343299358400663182805777879))
Generators:
    [0 1]
    [307941320171307176726971038693755343299358400663182805777878 0]
    [260117736370596621578446660105615721012604377822664417621373
        391706448932024045169858791210518451969529567969067994042 93]
    [124461947209331225807153039245860372991649922432590162143941
        478235838007105551485243785881396222867540228405183881565 05]
Mapping from: MatrixGroup(2, GF(307941320171307176726971038693755343299358400663182805777879)) to GL(2, GF(307941320171307176726971038693755343299358400663182805777879, 1))
```

Example H70E25

In the previous example, the group had infinitely many L_2 quotients, finitely many in every characteristic. The next example presents a group which has infinitely many L_2r quotients, but only in a single characteristic.

```
> G := Group< x, y | x*y*x^-1*y*x*y^4*x^-1*y^-4 >;
> quot := L2Quotients(G);
> [L2Type(Q) : Q in quot];
[ infinite (characteristic 41) ]
> P := quot[1]; P;
Ideal of Polynomial ring of rank 3 over Integer Ring
Order: Lexicographical
Variables: x1, x2, x12
Inhomogeneous, Dimension >0
Groebner basis:
[
    x1 + 30*x2*x12,
    x2^2 + 22,
    41
]
```

The residue class ring of P is $S = F_{41^2}[x_{12}]$, and the algorithm found a homomorphism into $\mathrm{PSL}(2, S)$. Since S has epimorphisms onto every finite extension of F_{41^2}, this gives homomorphisms of G into $\mathrm{PSL}(2, 41^{(}2 * k))$ for every k.
We specialize x_{12} to different elements of extensions of F_{41^2}. For example, for different specializations in F_{41^2} we get an image onto $\mathrm{PGL}(2, 41)$ or $\mathrm{PSL}(2, 41^2)$:

```
> R< x1, x2, x12 > := Generic(P);
>
> I := ideal< R | x12 - 1 > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PGL( 2, 41 ) ]
>
> I := ideal< R | x12 - (x2 + 1) > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PSL( 2, 41^2 ) ]
```

We can check the images for a random specialization:

```
> pol := &+[Random([0..40])*x12^i : i in [0..30]]; pol;
19*x12^30 + x12^29 + 16*x12^28 + 14*x12^27 + 38*x12^26 + 27*x12^25 + 20*x12^24 +
    31*x12^23 + 15*x12^22 + 22*x12^21 + 32*x12^20 + 26*x12^19 + 26*x12^18 +
    18*x12^17 + 16*x12^16 + 29*x12^15 + 18*x12^14 + 11*x12^13 + 3*x12^12 +
    11*x12^11 + 36*x12^10 + 13*x12^9 + 15*x12^8 + 33*x12^7 + 30*x12^6 + 8*x12^5 +
    12*x12^4 + 40*x12^3 + 12*x12^2 + 25*x12 + 8;
> I := ideal< R | pol > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PGL( 2, 41 ), PGL( 2, 41 ), PGL( 2, 41 ), PSL( 2, 41^2 ), PSL( 2, 41^2 ),
PSL( 2, 41^2 ), PSL( 2, 41^2 ), PSL( 2, 41^14 ), PSL( 2, 41^14 ),
```

```
PGL( 2, 41^9 ) ]
```

And again, this works for very large fields:

```
> pol := IrreduciblePolynomial(GF(41), 301);
> pol;
$.1^301 + $.1^2 + 6*$.1 + 30
> I := ideal< R | x12^301 + x12^2 + 6*x12 + 30 > + P;
> [L2Type(Q) : Q in L2Ideals(I)];
[ PGL( 2, 41^301 ) ]
```

70.4.6 Infinite L2 quotients

This section contains functions that use the L2-quotient algorithm of Plesken and Fabianska ([PF09]) to establish the existence or not of an infinite quotient of a finitely-presented group in $PSL_2(K)$ for K a field of characteristic zero. The algorithm is often used to find surjective quotients over finite fields but as a test for non-finiteness it is easier to work directly in characteristic zero. We make some comments on the relation to finite fields below.

The algorithm first constructs an affine scheme X over $\mathbf{Q}$ from the "trace ideal" of the group G, for which the algebraic points over a field K are in 1-1 correspondence with equivalence classes of representations of G into $SL_2(K)$. If $g_1, \ldots, g_n$ are the generators of G, then the affine coordinates of X correspond to the trace under the representation of various products of the G_i. For example, when $n = 2$, X lies in 3-dimensional affine space and the coordinates correspond to the traces of g_1, g_2 and g_1g_2. The ideal is actually generated by polynomials with coefficients in $\mathbf{Z}$ so X is naturally defined as a scheme over $Spec(\mathbf{Z})$, whose reduction mod p just gives the scheme corresponding to characteristic p representations of G for $p > 2$.

Homomorphisms into PSL_2 rather than SL_2 are dealt with by considering a number of X schemes for a particular G. Each one represents the homomorphisms from the free cover of G to SL_2 that takes each word in the set of relations defining G to either I or $-I$. Each of the 2^r choices of sign for the r defining relations gives an X (by the same procedure) and the totality cover all possibilities for maps to PSL_2.

The set of points of X corresponding to geometrically reducible, dihedral A_4, S_4 or A_5 images in PSL_2 is a closed subscheme Y. In fact the first two possibilities give closed subschemes with equations defined over $\mathbf{Z}$ and the last three give a dimension zero scheme over $\mathbf{Q}$. The overall equations defining Y reduce mod p to those defining the corresponding subscheme in characteristic p for almost all primes p. The explicit equations defing Y are determined by the algorithm as explained for $n = 2$ or 3 in the above reference or in more detail in Fabianska's MSc thesis ([Fab09]). Let U be the open complement of Y in X.

There are two possibilities.

1 A U is non-empty, so an algebraic point gives $\phi : G \to PSL_2(K)$, K a number field, with infinite image, so G is infinite. Further, for all but finitely many primes p, ϕ can be reduced mod v for any place v of characteristic p AND the reduction ϕ_v corresponds to

a point in the analogue of U over the finite field. Thus the reductions give surjections of G onto a $PSL_2(k)$ [or maybe a $PGL_2(k)$] for finite fields of any characteristic outside of a finite set.

2 All Us are empty. Here, all the homomorphisms of G into $PSL_2(K)$ in characteristic zero have geometrically reducible, dihedral, A_4, S_4 or A_5 images and the same holds for K a field of characteristic p for all p outside of a finite set of primes. G may or may not be infinite.

NB: In case 2), there may still be images of G which ARE infinite, but lie in a Cartan or Borel subgroup, so are geometrically reducible, or lie in the normaliser of a Cartan and have an infinite dihedral image. These infinite reducible/dihedral possibilities are not currently checked for.

`HasInfinitePSL2Quotient(G)`

`signs`	SeqEnum	*Default :* 0
`full`	BoolElt	*Default :* `false`
Verbose	`IsInfGrp`	*Maximum :* 1

Function to return whether the two-generator finitely-presented group G has an infinite quotient lying in a $PSL_2(K)$ with K a field of characteristic 0 as described above.

For the convenience of the user, we have provided some parameters to output more detailed information coming from the analysis of the X representation schemes as well as to control which sign variations are considered.

Let ws be the sequence of words giving the defining relations of G (if a defining relation is of the form $w = v$ where v is not the identity word, then the corresponding word is $w * v^{-1}$). Let $ws = [w_1, .., w_r]$.

As described in the introduction, for each of the 2^r combinations of signs attached to the w_i - let s be one - the program will consider the scheme X of homomorphisms of G into PSL_2 such that, when lifted to a homomorphism of the two-generator free cover F of G into SL_2, w_i maps to $s_i * I$.

In some cases, the user may realise that there is no point in considering certain choices of signs. For example, if g_1^2 is a relation in ws, $g_1^2 \mapsto I$ in $SL_2(K)$ means that $g_1 \mapsto 1$ in $PSL_2(K)$, so the $PSL_2(K)$ image would be (maybe infinitely) cyclic, and it is a waste of time to consider this possibility. Similarly, if g_1^r, r odd, is a relation then ,without loss of generality, in a PSL_2 to SL_2 lift, $g_1^r \mapsto I$, which could be specified.

The parameter `signs` allows the user to specify a restricted set of sign options to analyse. `signs` should be an $0, 1, -1$ or a sequence of length $\#ws$ of such integers. A single value is converted into the sequence containing that value $\#ws$ times. An entry e of 1 or -1 in position i means that the function will only consider sign sequences s with $s[i] = e$, ie homomorphisms where w_i map to eI in the lift to SL_2. If $e = 0$, then there is no condition at place i of the sign sequences considered. The default value for `signs` is the single value 0.

For each representation space X (corresponding to an allowable choice of signs s), after removing positive dimensional components of Y corresponding to geometrically reducible or dihedral type representations, there is often a zero-dimensional subscheme left. The existence of an infinite (non-cyclic or dihedral) image homomorphism to PSL_2 then comes down to examining the finite set of closed points remaining and finding one that is not geometrically reducible, dihedral, A_4, S_4 or A_5 type. Clearly, once one such is found the procedure can stop. However, it might be of interest for the user to see the types of ALL of the representations corresponding to closed points if the zero-dimensional analysis stage is performed.

If parameter `full` is set to `true` (the default is `false`), the program will continue analysing all of the representations in the 0-dimensional locus, even after one corresponding to an infinite image is found. Furthermore a sequence of (signs,types) pairs is also returned which gives, for each sign combination s of maps considered, the sequence of types corresponding to the 0-dimensional locus (or empty if we don't reduce to dimension 0). These types are given as strings: "infinite", "reducible", "dihedral", "A4", "S4", and "A5".

Setting the verbose flag `IsInfGrp` to `true` or 1 gives output of information on the various stages as the function progresses, including the analysis of dimension zero loci.

Example H70E26

We look at two examples of quotients of the two-generator free group with relations g_1^2, g_2^3 and one further word. The first is infinite, the second is not. As noted above, we may as well specify that g_1^2 maps to $-I$ and g_2^3 to I in SL_2 and this can be done with the `signs` parameter.

```
> F := FreeGroup(2);
> rel := (F.1 * F.2 * F.1 * F.2 * F.1 * F.2 * F.1 * F.2 * F.1 * F.2 *
>   F.1 * F.2 * F.1 * F.2 * F.1 * F.2^-1 * F.1 * F.2 * F.1 * F.2^-1)^2;
> G := quo<F | [F.1^2 ,F.2^3, rel]>;
> HasInfinitePSL2Quotient(G : full := true);
true
[ [*
    [ 1, 1, 1 ],
    []
*], [*
    [ 1, 1, -1 ],
    []
*], [*
    [ 1, -1, 1 ],
    []
*], [*
    [ 1, -1, -1 ],
    []
*], [*
    [ -1, 1, 1 ],
```

```
    [ A4, A4 ]
*], [*
    [ -1, 1, -1 ],
    [ A5, A5, infinite ]
*], [*
    [ -1, -1, 1 ],
    [ A4, A4 ]
*], [*
    [ -1, -1, -1 ],
    [ A5, A5, infinite ]
*] ]
```

Now try it with the sign restriction.

```
> HasInfinitePSL2Quotient(G : full := true, signs := [-1,1,0]);
true
[ [*
    [ -1, 1, 1 ],
    [ A4, A4 ]
*], [*
    [ -1, 1, -1 ],
    [ A5, A5, infinite ]
*] ]
```

The second example is just A_5.

```
> G := quo<F | [F.1^2 ,F.2^3, (F.1*F.2)^5]>;
> HasInfinitePSL2Quotient(G);
false
> HasInfinitePSL2Quotient(G : full := true, signs := [-1,1,0]);
false
[ [*
    [ -1, 1, 1 ],
    [ A5 ]
*], [*
    [ -1, 1, -1 ],
    [ A5 ]
*] ]
```

70.4.7 Searching for Isomorphisms

This section describes a function for searching for isomorphisms between two finitely presented groups.

`SearchForIsomorphism(F, G, m : parameters)`

Attempt to find an isomorphism from the finitely presented group F to the finitely presented group G. The search will be restricted to those homomorphisms for which the sum of the word-lengths of the images of the generators of F in G is at most m.

If an isomorphism ϕ is found, then the values `true`, ϕ, ϕ^-1 are returned. Otherwise, the values `false`, _, _ are returned; of course, that does not necessarily mean that the groups are not isomorphic.

An error will result if any of the generators of F turn out to be trivial.

By setting the verbose flag `"IsoSearch"` to 1, information about the progress of the search will be printed.

The parameters available for the function SearchForIsomorphism are:

`All` BoolElt *Default :* `false`

If `All` is `false` (default), then the function halts and returns as soon as a single isomorphism is found. If `All` is `true`, then the search continues through all possible images that satisfy the image length condition, and a list of pairs $\langle \phi, \phi^-1 \rangle$ for all isomorphisms $\phi : F \to G$ found is returned as the second return value.

`IsomsOnly` BoolElt *Default :* `true`

If `IsomsOnly` is set to `false`, then all homomorphisms $F \to G$ will be returned if `All` is `true`, and the first nontrivial homomorphism found will be returned if `All` is `false`.

`MaxRels` RngIntElt *Default :* $250 * m$

The value of the `MaxRelations` parameter used in runs of RWSGroup. It is hard to find a sensible default, because if the value is unnecessarily large then time can be wasted unnecessarily. This may need to be increased if the function is used with finite groups, for example (although it is usually much more efficient to use permutation or matrix representations when testing isomorphism of finite groups).

`CycConjTest` BoolElt *Default :* `true`

When `CycConjTest` is `true` and `All` is `false`, then images of the first generator which have a cyclic conjugate that comes earlier in the lexicographical order are rejected, because there would be a conjugate isomorphism in which the image was the cyclic conjugate. This nearly always results in faster run-times, but occasionally it can happen that the conjugate isomorphism has a larger sum of lengths of generator images, which is clearly bad. So the user has the option of not rejecting such images.

Example H70E27

John Hillman asked whether the following two groups are isomorphic.

```
> G1<s,t,u> := Group <s,t,u | s*u*s^-1=u^-1, t^2=u^2, t*s^2*t^-1=s^-2,
```

```
>                              u*(s*t)^2=(s*t)^2*u >;
> G2<x,y,z> := Group<x,y,z | x*y^2*x^-1=y^-2, y*x^2*y^-1=x^-2, x^2=z^2*(x*y)^2,
>                            y^2=(z^-1*x)^2, z*(x*y)^2=(x*y)^2*z >;
> isiso, f1, f2 := SearchForIsomorphism(G1,G2,7);
> isiso;
true
> f1;
Homomorphism of GrpFP: G1 into GrpFP: G2 induced by
    s |--> x * z^-1
    t |--> y * z
    u |--> x * y^-1 * z
> f2;
Homomorphism of GrpFP: G2 into GrpFP: G1 induced by
    x |--> s^2 * u * t^-1
    y |--> s^-1 * u^-1
    z |--> s * u * t^-1
```

The search for an isomorphism succeeeded and returns the isomorphisms explicitly.

Example H70E28

Walter Neumann asked whether the next two groups are isomorphic.

```
> G1<x,y,z> := Group< x,y,z | x^2*y^5, x^14*z^23, (x^2,y), (x^2,z), x*y*z>;
> G2<a,b> := Group<a,b | a*b^16*a*b^-7, a^4*b^7*a^-1*b^7>;
```

It is a good idea to minimize the number of generators of `G1` - since there is clearly a redundant generator here.

```
> G1s := Simplify(G1);
> Ngens(G1s);
2
> G1!G1s.1, G1!G1s.2;
x z
```

Now a direct call `SearchForIsomorphism(G1, G2, 15)` will succeed, but will take many hours, and also use a lot of memory.
In contrast to `G1`, it can sometimes help to introduce a new generator in `G2` if there are common substrings in relators.

```
> G2b<a,b,c> := Group< a,b,c | a*b^16*a*b^-7, a^4*b^7*a^-1*b^7, c=b^7>;
> isiso, f1, f2 := SearchForIsomorphism(G1s,G2b,4);
> isiso;
true
> f1;
Homomorphism of GrpFP: G1s into GrpFP: G2b induced by
    G1s.1 |--> a * c^-1
    G1s.2 |--> c
> f2;
Homomorphism of GrpFP: G2b into GrpFP: G1s induced by
```

```
a |--> G1s.1 * G1s.2
b |--> G1s.1^6 * G1s.2^10
c |--> G1s.2
```

70.5 Abelian, Nilpotent and Soluble Quotient

70.5.1 Abelian Quotient

The functions in this section compute information about the abelian quotient of an fp-group G. Some functions may require the computation of a coset table. Experienced users can control the behaviour of an implicit coset enumeration with a set of global parameters. These global parameters can be changed using the function SetGlobalTCParameters. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

The functions returning the abelian quotient or the abelian quotient invariants report an error if the abelian quotient cannot be computed, for example, because the relation matrix is too large. To avoid problems in user written programs or loops, the functions HasComputableAbelianQuotient and HasInfiniteComputableAbelianQuotient can be used.

`AbelianQuotient(G)`

The maximal abelian quotient G/G' of the group G as GrpAb (cf. Chapter 69). The natural epimorphism $\pi : G \to G/G'$ is returned as second value.

`ElementaryAbelianQuotient(G, p)`

The maximal p-elementary abelian quotient Q of the group G as GrpAb (cf. Chapter 69). The natural epimorphism $\pi : G \to Q$ is returned as second value.

`AbelianQuotientInvariants(G)`

`AQInvariants(G)`

Given a finitely presented group G, this function computes the elementary divisors of the derived quotient group G/G', by constructing the relation matrix for G and transforming it into Smith normal form. The algorithm used is an algorithm of Havas which does the reduction entirely over the ring of integers $\mathbf{Z}$ using clever heuristics to minimize the growth of coefficients.

The divisors are returned as a sequence of integers.

`AbelianQuotientInvariants(H)`

`AQInvariants(H)`

`AbelianQuotientInvariants(G, T)`

`AQInvariants(G, T)`

Given a subgroup H of the finitely presented group G, this function computes the elementary divisors of the derived quotient group of H. (The coset table T may be used to define H.) This is done by abelianising the Reidemeister-Schreier presentation for H and then proceeding as above. The divisors are returned as a sequence of integers.

`AbelianQuotientInvariants(G, n)`

`AQInvariants(G, n)`

Given a finitely presented group G, this function computes the elementary divisors of the quotient group G/N, where N is the normal subgroup of G generated by the derived group G' together with all n-th powers of elements of G. The algorithm constructs the relation matrix corresponding to the presentation of G and computes its Smith normal form over the ring Z/nZ. The calculation is particularly efficient when n is a small prime. The divisors are returned as a sequence of integers.

`AbelianQuotientInvariants(H, n)`

`AQInvariants(H, n)`

`AbelianQuotientInvariants(G, T, n)`

`AQInvariants(G, T, n)`

Given a subgroup H of the finitely presented group G, this function computes the elementary divisors of the quotient group H/N, where N is the normal subgroup of H generated by H' together with all n-th powers of elements of H. (The coset table T may be used to define H.) This is done by abelianising the Reidemeister-Schreier presentation for H and then proceeding as above. The divisors are returned as a sequence of integers.

`HasComputableAbelianQuotient(G)`

Given an fp-group G, this function tests whether the abelian quotient of G can be computed. If so, it returns the value `true`, the abelian quotient A of G and the natural epimorphism $\pi : G \to A$. If the abelian quotient of G cannot be computed, the value `false` is returned.

This function is especially useful to avoid runtime errors in user written loops or functions.

`HasInfiniteComputableAbelianQuotient(G)`

Given an fp-group G, this function tests whether the abelian quotient of G can be computed and is infinite. If so, it returns the value `true`, the abelian quotient A of G and the natural epimorphism $\pi : G \to A$. If the abelian quotient of G cannot be computed or if it is finite, the value `false` is returned.

The function first checks the modular abelian invariants for a set of small primes. If for one of these primes, the modular abelian quotient is trivial, A must be finite and the function returns without actually computing the abelian quotient. If one is interested only in infinite quotients, this heuristics may save time.

`IsPerfect(G)`

Given an fp-group G, this function tries to decide whether G is perfect by checking whether the abelian quotient of G is trivial.

`TorsionFreeRank(G)`

Given the finitely presented group G, return the torsion-free rank of the derived quotient group of G.

Example H70E29

The Fibonacci group $F(7)$ has the following 2-generator presentation:

$$< a, b \mid a^2b^{-2}a^{-1}b^{-2}(a^{-1}b^{-1})^2, abab^2abab^{-1}(ab^2)^2 > .$$

We proceed to investigate the structure of this group.

```
> F<a, b> := FreeGroup(2);
> F7<a, b> := quo< F |  a^2*b^-2*a^-1*b^-2*(a^-1*b^-1)^2,
>        a*b*a*b^2*a*b*a*b^-1*(a*b^2)^2 >;
> F7;
Finitely presented group F7 on 2 generators
Relations
       a^2 * b^-2 * a^-1 * b^-2 * a^-1 * b^-1 * a^-1 * b^-1 =Id(F7)
       a * b * a * b^2 * a * b * a * b^-1 * a * b^2 * a * b^2 = Id()7
```

We begin by determining the structure of the maximal abelian quotient of $F(7)$.

```
> AbelianQuotientInvariants(F7);
[ 29 ]
```

The maximal abelian quotient of $F(7)$ is cyclic of order 29. At this point there is no obvious way to proceed, so we attempt to determine the index of some subgroups.

```
> Index( F7, sub< F7 | a > );
1
```

We are in luck: $F(7)$ is generated by a and so must be cyclic. This fact coupled with the knowledge that its abelian quotient has order 29 tells us that the group is cyclic of order 29.

Example H70E30

The group $G = (8, 7 \mid 2, 3)$ is defined by the presentation

$$< a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 > .$$

We consider the subgroup H of G, generated by the words a^2 and $a^{-1}b$:

```
> G<a, b> := Group<a, b| a^8, b^7, (a * b)^2, (a^-1 * b)^3>;
> H<x, y> := sub< G | a^2, a^-1 * b >;
```

The fastest way to determine the order of the maximal 2-elementary abelian quotient of H is to use the function `AbelianQuotientInvariants`:

```
> #AbelianQuotientInvariants(H,2);
1
```

We see that the maximal 2-elementary abelian quotient of H has order 2^1.

70.5.2 p-Quotient

Let F be a finitely presented group, p a prime and c a positive integer. A p-quotient algorithm constructs a consistent power-conjugate presentation for the largest p-quotient of F having lower exponent-p class at most c. The p-quotient algorithm used by MAGMA is part of the ANU p-Quotient program. For details of the algorithm, see [NO96]. In MAGMA the result is returned as a group of type `GrpPC` (cf. Chapter 63).

Assume that the p-quotient has order p^n, Frattini rank d, and that its generators are $a_1, \ldots, a_n$. Then the power-conjugate presentation constructed has the following additional structure. The set $\{a_1, \ldots, a_d\}$ is a generating set for G. For each a_k in $\{a_{d+1}, \ldots, a_n\}$, there is at least one relation whose right hand side is a_k. One of these relations is taken as the *definition* of a_k. (The list of definitions is also returned by `pQuotient`.) The power-conjugate generators also have a *weight* associated with them: a generator is assigned a weight corresponding to the stage at which it is added and this weight is extended to all normal words in a natural way.

The p-quotient function and its associated commands allows the user to construct a power-conjugate presentation (pcp) for a p-group. Note that there is also a process version of the p-quotient algorithm, which gives the user complete control over its execution. For a description, we refer to Chapter 71.

70.5.3 The Construction of a p-Quotient

`pQuotient(F, p, c: parameters)`

Given an fp-group F, a prime p and a positive integer c, construct a pcp for the largest p-quotient G of F having lower exponent-p class at most c. If c is given as 0, then the limit 127 is placed on the class. The function also returns the natural homomorphism π from F to G, a sequence S describing the definitions of the pc-generators of G and a flag indicating whether G is the maximal p-quotient of F.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $G.k$ of G as follows.

- If $S[k] = [0, r]$, then $G.k$ is defined via the image of $F.r$ under π.
- If $S[k] = [r, 0]$, then $G.k$ is defined via the power relation for $G.r$.
- If $S[k] = [r, s]$, then $G.k$ is defined via the conjugate relation involving $G.r^{G.s}$.

There exist a number of parameters for controlling the behaviour of this function, which are described below.

Example H70E31

We construct the largest 2-quotient of class 6 for a two-generator, two-relator group.

```
> F<a,b> := FreeGroup(2);
> G := quo< F | (b, a, a) = 1, (a * b * a)^4 = 1 >;
> Q, fQ := pQuotient(G, 2, 6);
> Order(Q);
524288
> fQ;
Mapping from: GrpFP: G to GrpPC: Q
```

The parameters available for the function `pQuotient` are:

`Exponent` RngIntElt *Default :* 0

If `Exponent := m`, enforce the exponent law, $x^m = 1$, on the group.

`Metabelian` BoolElt *Default :* `false`

If `Metabelian := true`, then a consistent pcp is constructed for the largest metabelian p-quotient of F having lower exponent-p class at most c.

`Print` RngIntElt *Default :* 0

This parameter controls the volume of printing. By default its value is that returned by `GetVerbose("pQuotient")`, which is 0 unless it has been changed through use of `SetVerbose`. The effect is the following:

`Print := 0`: No output.

`Print := 1`: Report order of p-quotient at each class.

`Print := 2`: Report statistics and redundancy information about tails, consistency, collection of relations and exponent enforcement components of calculation.

`Print := 3`: Report in detail on the construction of each class.

Note that the presentation displayed is a *power-commutator* presentation (since this is the version stored by the p-quotient).

`Workspace` RngIntElt *Default :* 5000000

The amount of space requested for the p-quotient computation.

Example H70E32

We construct the largest 3-quotient of class 6 for a two-generator group of exponent 9.

```
> F<a,b> := FreeGroup(2);
> G := quo< F | a^3 = b^3 = 1 >;
> q := pQuotient(G, 3, 6: Print := 1, Exponent := 9);
Lower exponent-3 central series for G
Group: G to lower exponent-3 central class 1 has order 3^2
Group: G to lower exponent-3 central class 2 has order 3^3
Group: G to lower exponent-3 central class 3 has order 3^5
Group: G to lower exponent-3 central class 4 has order 3^7
Group: G to lower exponent-3 central class 5 has order 3^9
Group: G to lower exponent-3 central class 6 has order 3^11
```

Example H70E33

We use the metabelian parameter to construct a metabelian 5-quotient of the group

$$< a, b \mid a^{625} = b^{625} = 1, (b, a, b) = 1, (b, a, a, a, a) = (b, a)^5 > .$$

```
> F<a, b> := FreeGroup(2);
> G := quo< F | a^625 = b^625 = 1, (b, a, b) = 1,
>           (b, a, a, a, a) = (b, a)^5 >;
> q := pQuotient(G, 5, 20: Print := 1, Metabelian := true);
Lower exponent-5 central series for G
Group: G to lower exponent-5 central class 1 has order 5^2
Group: G to lower exponent-5 central class 2 has order 5^5
Group: G to lower exponent-5 central class 3 has order 5^8
Group: G to lower exponent-5 central class 4 has order 5^11
Group: G to lower exponent-5 central class 5 has order 5^12
Group: G to lower exponent-5 central class 6 has order 5^13
Group: G to lower exponent-5 central class 7 has order 5^14
Group: G to lower exponent-5 central class 8 has order 5^15
Group: G to lower exponent-5 central class 9 has order 5^16
Group: G to lower exponent-5 central class 10 has order 5^17
Group: G to lower exponent-5 central class 11 has order 5^18
Group: G to lower exponent-5 central class 12 has order 5^19
Group: G to lower exponent-5 central class 13 has order 5^20
Group completed. Lower exponent-5 central class = 13, order = 5^20
```

Example H70E34

In the final example, we construct the largest finite 2-generator group having exponent 5.

```
> F := FreeGroup(2);
> q := pQuotient (F, 5, 14: Print := 1, Exponent := 5);
Lower exponent-5 central series for F
Group: F to lower exponent-5 central class 1 has order 5^2
Group: F to lower exponent-5 central class 2 has order 5^3
Group: F to lower exponent-5 central class 3 has order 5^5
Group: F to lower exponent-5 central class 4 has order 5^8
Group: F to lower exponent-5 central class 5 has order 5^10
Group: F to lower exponent-5 central class 6 has order 5^14
Group: F to lower exponent-5 central class 7 has order 5^18
Group: F to lower exponent-5 central class 8 has order 5^22
Group: F to lower exponent-5 central class 9 has order 5^28
Group: F to lower exponent-5 central class 10 has order 5^31
Group: F to lower exponent-5 central class 11 has order 5^33
Group: F to lower exponent-5 central class 12 has order 5^34
Group completed. Lower exponent-5 central class = 12, order = 5^34
```

70.5.4 Nilpotent Quotient

A nilpotent quotient algorithm constructs, from a finite presentation of a group, a polycyclic presentation of a nilpotent quotient of the finitely presented group. The nilpotent quotient algorithm used by MAGMA is the Australian National University Nilpotent Quotient program, as described in [Nic96]. The version included in MAGMA is Version 2.2 of January 2007.

The lower central series $G_0, G_1, \ldots$ of a group G can be defined inductively as $G_0 = G$, $G_i = [G_(i-1), G]$. G is said to have nilpotency class c if c is the smallest non-zero integer such that $G_c = 1$. If N is a normal subgroup of G and G/N is nilpotent, then N contains G_i for some non-negative integer i. G has infinite nilpotent quotients if and only if G/G_1 (the maximal abelian quotient of G) is infinite and a prime p divides a finite factor of a nilpotent quotient if and only if p divides a cyclic factor of G/G_1. The i-th ($i > 1$) factor G_{i-1}/G_i of the lower central series is generated by the elements $[g, h]G_i$, where g runs through a set of representatives of G/G_1 and h runs through a set of representatives of G_{i-2}/G_{i-1}.

Any finitely generated nilpotent group is polycyclic and, therefore, has a subnormal series with cyclic factors. Such a subnormal series can be used to represent the group in terms of a polycyclic presentation. The ANU NQ computes successively the factor groups modulo the terms of the lower central series. Each factor group is represented by a special form of polycyclic presentation, a nilpotent presentation, that makes use of the nilpotent structure of the factor group.

The algorithm has highly efficient code for enforcing the n-Engel identity in nilpotent groups. When appropriate parameters are set, the algorithm computes the largest n-Engel quotient of G/G_c.

More generally, the algorithm has code for enforcing arbitrary identical relations. You can even enforce relations which combine generators of G with "free variables". (Werner Nickel calls these "identical generators".) For instance, the relation `(a,x) = 1`, where a is a group generator and x is a free variable, will force the construction of quotients where the image of a is central.

Mike Vaughan-Lee offers advice on when to use the nilpotent quotient algorithm. If you know nothing about the finitely presented group, it is probably a good idea to look at the abelian quotient first. If the abelian quotient is trivial then all nilpotent quotients will be trivial. Similarly, if the abelian quotient is cyclic, then all nilpotent quotients will be cyclic. Less trivially, if the abelian quotient is finite then all nilpotent quotients will be finite and so will be the direct product of finite p-groups. Moreover, the relevant primes p will all occur in the abelian quotient. Usually it will be more effective to use the p-quotient algorithm to study the direct factors. However, if you want to study 3-Engel quotients (say), then you are better using the nilpotent quotient even when the abelian quotient is finite.

`NilpotentQuotient(G, c: parameters)`

This function returns the class c nilpotent quotient of G as a group in category GrpGPC, together with the epimorphism π from G onto this quotient. When c is set to zero, the function attempts to compute the maximal nilpotent quotient of G. Using the parameters described below, the user can enforce certain conditions on the quotient found.

Example H70E35

Here is a finitely presented group. The abelian quotient is infinite, so we look at the class 2 nilpotent quotient.

```
> G := Group<x,y,z|(x*y*z^-1)^2, (x^-1*y^2*z)^2, (x*y^-2*x^-1)^2 >;
> AbelianQuotient(G);
Abelian Group isomorphic to Z/2 + Z/2 + Z
Defined on 3 generators
Relations:
  2*$.1 = 0
  2*$.2 = 0
> N := NilpotentQuotient(G,2); N;
GrpGPC : N of infinite order on 6 PC-generators
PC-Relations:
  N.1^2 = N.3^2 * N.5,
  N.2^2 = N.4 * N.6,
  N.4^2 = Id(N),
  N.5^2 = Id(N),
  N.6^2 = Id(N),
  N.2^N.1 = N.2 * N.4,
  N.3^N.1 = N.3 * N.5,
  N.3^N.2 = N.3 * N.6
```

Example H70E36

The free nilpotent group of rank r and class e is defined as $F/\gamma_{e+1}(F)$, where F is a free group of rank r and $\gamma_{e+1}(F)$ denotes the $(e+1)$ st term of the lower central series of F.
We construct the free nilpotent group N of rank 2 and class 3 as quotient of the free group F of rank 2 and the natural epimorphism from F onto N.

```
> F<a,b> := FreeGroup(2);
> N<[x]>, pi := NilpotentQuotient(F, 3);
> N;
GrpGPC : N of infinite order on 5 PC-generators
PC-Relations:
    x[2]^x[1] = x[2] * x[3],
    x[2]^(x[1]^-1) = x[2] * x[3]^-1 * x[4],
    x[3]^x[1] = x[3] * x[4],
    x[3]^(x[1]^-1) = x[3] * x[4]^-1,
    x[3]^x[2] = x[3] * x[5],
    x[3]^(x[2]^-1) = x[3] * x[5]^-1
```

Using the function NilpotencyClass described in Chapter 72, we check the nilpotency class of the quotient.

```
> NilpotencyClass(N);
3
```

Example H70E37

The Baumslag-Solitar groups

$$BS(p,q) = \langle < a, b | ab^p a^{-1} = b^q \rangle$$

form a fascinating class of 1-relator groups. We compute the nilpotent of class 4 quotient of two of them, and print out the resulting presentations, and the structure of the generators.

```
> G<a,b> := Group<a,b|a*b*a^-1=b^4>;
> N,f := NilpotentQuotient(G,4);
> N;
GrpGPC : N of infinite order on 5 PC-generators
PC-Relations:
  N.2^3 = N.3^2 * N.4^2 * N.5,
  N.3^3 = N.4^2 * N.5^2,
  N.4^3 = N.5^2,
  N.5^3 = Id(N),
  N.2^N.1 = N.2 * N.3,
  N.2^(N.1^-1) = N.2 * N.3^2 * N.4^2 * N.5,
  N.3^N.1 = N.3 * N.4,
  N.3^(N.1^-1) = N.3 * N.4^2 * N.5^2,
  N.4^N.1 = N.4 * N.5,
  N.4^(N.1^-1) = N.4 * N.5^2
> for i := 1 to Ngens(N) do
```

```
> N.i @@ f;
> end for;
a
b
(b, a)
(b, a, a)
(b, a, a, a)
> G<a,b> := Group<a,b|a*b^2*a^-1=b^4>;
> N,f := NilpotentQuotient(G,4);
> N;
GrpGPC : N of infinite order on 8 PC-generators
PC-Relations:
  N.2^2 = Id(N),
  N.3^2 = N.5 * N.8,
  N.4^2 = N.7,
  N.5^2 = N.8,
  N.6^2 = Id(N),
  N.7^2 = Id(N),
  N.8^2 = Id(N),
  N.2^N.1 = N.2 * N.3,
  N.2^(N.1^-1) = N.2 * N.3 * N.4 * N.5 * N.6,
  N.3^N.1 = N.3 * N.4,
  N.3^(N.1^-1) = N.3 * N.4 * N.6 * N.7,
  N.3^N.2 = N.3 * N.5,
  N.4^N.1 = N.4 * N.6,
  N.4^(N.1^-1) = N.4 * N.6,
  N.4^N.2 = N.4 * N.7,
  N.5^N.1 = N.5 * N.7,
  N.5^(N.1^-1) = N.5 * N.7,
  N.5^N.2 = N.5 * N.8
> for i := 1 to Ngens(N) do
> N.i @@ f;
> end for;
a
b
(b, a)
(b, a, a)
(b, a, b)
(b, a, a, a)
(b, a, b, a)
(b, a, b, b)
```

The parameters available for the function `NilpotentQuotient` are:

`NumberOfEngelGenerators`

RngIntElt *Default :* 1

Setting this parameter to k forces the first k generators to be left or right Engel elements, provided one (or both) of the parameters `LeftEngel` or `RightEngel` is positive. Otherwise it is ignored.

`LeftEngel` RngIntElt *Default :* 0

Setting this parameter to n forces the first k generators $g_1, \ldots, g_k$ of the quotient Q to be left n-Engel elements. That is, they satisfy $[x, \ldots, x, g_i] = 1$ (x appearing n times) for all x in Q. The value of k is determined by the parameter `NumberOfEngelGenerators`.

`RightEngel` RngIntElt *Default :* 0

This is the same as for `LeftEngel`, but here the generators are right n-Engel elements, so $[g_i, x, \ldots, x] = 1$.

`Engel` RngIntElt *Default :* 0

Setting this parameter to n enforces the nth Engel law on the quotient Q. That is, $[x, y, \ldots, y] = 1$ (y appearing n times) for all x, y in Q.

`SemigroupOnly` BoolElt *Default :* `true`

This option causes the program to check only semigroup words in the generating set of the nilpotent quotient when an Engel condition is enforced. If none of the Engel parameters are set, then it is ignored.

`SemigroupFirst` BoolElt *Default :* `false`

This option causes the program to check semigroup words in the generating set of the quotient first and then all other words, when an Engel condition is enforced. If `SemigroupOnly` is set, or no Engel condition is enforced, then it is ignored.

`ReverseOrder` BoolElt *Default :* `false`

In checking Engel identities, instances are processed in order of increasing weight. This flag reverses the order.

`ReverseEngel` BoolElt *Default :* `false`

This flag changes the Engel conditions from the first k generators, to the *last* k generators.

`CheckFewInstances` BoolElt *Default :* `false`

This option stops checking the Engel law at each class if all the checks of a certain weight did not yield any non-trivial instances of the law.

`Nickel` BoolElt *Default :* `false`

Enforce the identities x^8 and $[[x_1, x_2, x_3], [x_4, x_5, x_6]]$ on the nilpotent quotient.

`NumberOfFreeVariables` RngIntElt *Default :* 0

If this parameter is set to $n > 0$ then the last n variables of the group are treated as variables rather than generators of the group. In any relation they are treated as standing for all group elements, enabling the user to enforce identical relations on the quotient being computed. The value of n must be less than the number of generators

of the input group. When this facility is used, the domain of the epimorphism returned is the subgroup of G generated by the (first) non-free generators.

`PrintResult` BoolElt *Default :* `false`

If set to true, this parameter switches on the printing of results given by the stand alone version of NQ.

`SetVerbose("NilpotentQuotient", n)`

Turn on and off the printing of information as the nilpotent quotient algorithm proceeds. n may be set to any of 0, 1, 2 or 3. Setting n to be 0 turns printing off, while successively higher values for n print more and more information as the algorithm progresses. The default value of n is 0.

Example H70E38

We compute the maximal nilpotent quotient $N1$ of the group G given by the following presentation

$$\langle a, b, c, d, e \mid (b, a),\ (c, a),\ (d, a) = (c, b),\ (e, a) = (d, b),\ (e, b) = (d, c),\ (e, c),\ (e, d)\rangle.$$

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> :=
>          quo<F | (b,a), (c,a),
>          (d,a)=(c,b), (e,a)=(d,b), (e,b)=(d,c),
>          (e,c), (e,d)>;
> N1<[x]>, pi1 := NilpotentQuotient(G, 0);
```

Using the function `NilpotencyClass` described in Chapter 72, we check the nilpotency class of the quotient. It turns out to have nilpotency class 6.

```
> NilpotencyClass(N1);
6
```

Next we compute a metabelian quotient $N2$ of G, construct the natural epimorphism from $N1$ onto $N2$ and check its kernel. (The functions applied to the nilpotent quotients are described in Chapter 72.) To get a metabelian quotient we adjoin 4 variables and a relation to the presentation of G and use the "free variables" facility.

```
> M := Group<w,x,y,z|((w,x),(y,z))>; // metabelian identity
> D := FreeProduct(G,M); // adjoin to G
> N2, pi2 := NilpotentQuotient(D, 0: NumberOfFreeVariables := 4);
> NilpotencyClass(N2);
4
> DerivedLength(N2);
2
> f := hom< N1->N2 | [ pi1(G.i)->pi2(D.i) : i in [1..Ngens(G)] ] >;
> PCGenerators(Kernel(f), N1);
{@ x[14], x[15] * x[17], x[16], x[17]^2, x[18], x[19], x[20],
x[21], x[22], x[23], x[24], x[25], x[26], x[27], x[28], x[29],
```

```
x[30], x[31] @}
```

We compute a quotient $N3$ of G satisfying the 4 th Engel law, construct the natural epimorphism from $N1$ onto $N3$ and again check the kernel. (The functions applied to the nilpotent quotients are described in Chapter 72.)

```
> N3, pi3 := NilpotentQuotient(G, 0 : Engel := 4);
> NilpotencyClass(N3);
4
> DerivedLength(N3);
3
> h := hom< N1->N3 | [ pi1(g)->pi3(g) : g in Generators(G) ] >;
> PCGenerators(Kernel(h), N1);
{@ x[19], x[20], x[21], x[22], x[23], x[24], x[25], x[26], x[27],
x[28], x[29], x[30], x[31] @}
```

70.5.5 Soluble Quotient

A soluble quotient algorithm computes a consistent power-conjugate presentation of the largest finite soluble quotient of a finitely presented group, subject to certain algorithmic and user supplied restrictions. In this section we describe only the simplest use of such an algorithm within MAGMA. For more information the user is referred to Chapter 71.

`SolvableQuotient(G : parameters)`

`SolubleQuotient(G : parameters)`

Let G be a finitely presented group. The function constructs the largest finite soluble quotient of G.

Example H70E39

We compute the soluble quotient of the group

$$\langle a, b \mid a^2, b^4, ab^{-1}ab(abab^{-1})^5ab^2ab^{-2}\rangle.$$

```
> G<a,b> := Group< a, b | a^2, b^4,
>                     a*b^-1*a*b*(a*b*a*b^-1)^5*a*b^2*a*b^-2 >;
> Q := SolubleQuotient(G);
> Q;
GrpPC : Q of order 1920 = 2^7 * 3 * 5
PC-Relations:
    Q.1^2 = Q.4,
    Q.2^2 = Id(Q),
    Q.3^2 = Q.6,
    Q.4^2 = Id(Q),
    Q.5^2 = Q.7,
    Q.6^2 = Id(Q),
```

```
Q.7^2 = Id(Q),
Q.8^3 = Id(Q),
Q.9^5 = Id(Q),
Q.2^Q.1 = Q.2 * Q.3,
Q.3^Q.1 = Q.3 * Q.5,
Q.3^Q.2 = Q.3 * Q.6,
Q.4^Q.2 = Q.4 * Q.5 * Q.6 * Q.7,
Q.4^Q.3 = Q.4 * Q.6 * Q.7,
Q.5^Q.1 = Q.5 * Q.6,
Q.5^Q.2 = Q.5 * Q.7,
Q.5^Q.4 = Q.5 * Q.7,
Q.6^Q.1 = Q.6 * Q.7,
Q.8^Q.1 = Q.8^2,
Q.8^Q.2 = Q.8^2,
Q.9^Q.1 = Q.9^3,
Q.9^Q.2 = Q.9^4,
Q.9^Q.4 = Q.9^4
```

`SolvableQuotient(F, n : parameters)`

`SolubleQuotient(F, n : parameters)`

`SolvableQuotient(F, P : parameters)`

`SolubleQuotient(F, P : parameters)`

Find a soluble quotient G and the epimorphism $\pi : F \twoheadrightarrow G$ with a specified order. n must be a nonnegative integer. P must be a set of primes.

The three forms reflect possible information about the order of an expected soluble quotient. In the first form the order of G is given by n, if n is greater than zero. If n equals zero, nothing about the order is known and the relevant primes will be calculated completely.

The second form, with no n argument, is equivalent to the first with $n = 0$. This is a standard argument, and usually it is the most efficient way to calculate soluble quotients.

Note that, if $n > 0$ is not the order of the maximal finite soluble quotient, it may happen that no group of order n can be found, since an epimorphic image of size n may not be exhibited by the chosen series.

In the third form a set P of relevant primes is given. The algorithm calculates the biggest quotient such that the order has prime divisors only in P. P may have a zero as element, this is just for consistency reasons. It is equivalent to the first form with n equal zero.

For a description of the algorithm used and of the set of parameters available for this function, see Chapter 71. Other more specialised functions for computing soluble quotients and some examples can be found there as well.

Example H70E40

Consider the group G defined by the presentation

$$\langle x, y \mid x^3, y^8, [x, y^4], x^{-1}yx^{-1}y^{-1}xyxy^{-1}, (xy^{-2})^2(x^{-1}y^{-2})^2(xy^2)^2(x^{-1}y^2)^2, (x^{-1}y^{-2})^6(x^{-1}y^2)^6\rangle.$$

```
> G<x, y> := Group< x, y |
>     x^3, y^8, (x,y^4), x^-1*y*x^-1*y^-1*x*y*x*y^-1,
>     (x*y^-2)^2*(x^-1*y^-2)^2*(x*y^2)^2*(x^-1*y^2)^2,
>     (x^-1*y^-2)^6*(x^-1*y^2)^6 >;
```

We apply the soluble quotient algorithm to G and compute the order of the soluble quotient Q.

```
> time Q := SolubleQuotient(G);
Time: 116.920
> Order(Q);
165888
```

Note that $165888 = 2^{11} \cdot 3^4$. If we knew the possible primes in advance the soluble quotient can be computed much more quickly by using this knowledge.

```
> time Q := SolubleQuotient(G, {2, 3});
Time: 39.400
```

Note that this reduces the execution time by a factor of almost three.

We now assume that G is finite and try to compute its order by means of the Todd-Coxeter algorithm.

```
> Order(G);
165888
```

Hence the group is finite and soluble and G is isomorphic to Q. We may use the tools for finite soluble groups to investigate the structure of G. For example we can easily find the number of involutions in G.

```
> cls := ConjugacyClasses(Q);
> &+ [ cl[2] : cl in cls | cl[1] eq 2 ];
511
```

70.6 Subgroups

70.6.1 Specification of a Subgroup

`sub< G | L >`

Construct the subgroup H of the fp-group G generated by the words specified by the terms of the *generator list* $L = L_1, \ldots, L_r$.

A term L_i of the generator list may consist of any of the following objects:

(a) A word;

(b) A set or sequence of words;

(c) A sequence of integers representing a word;

(d) A set or sequence of sequences of integers representing words;

(e) A subgroup of an fp-group;

(f) A set or sequence of subgroups.

The collection of words and groups specified by the list must all belong to the group G and H will be constructed as a subgroup of G.

The generators of H consist of the words specified directly by terms L_i together with the stored generating words for any groups specified by terms of L_i. Repetitions of an element and occurrences of the identity element are removed (unless H is trivial).

If the `sub`-constructor is invoked with an empty list L, the trivial subgroup will be constructed.

`sub< G | f >`

Given a homomorphism f from G onto a transitive subgroup of $\mathrm{Sym}(n)$, construct the subgroup of G which affords this permutation representation.

`ncl< G | L >`

Construct the subgroup N of the fp-group G as the normal closure of the subgroup H generated by the words specified by the terms of the *generator list* L.

The possible forms of a term of the generator list are the same as for the `sub`-constructor.

This constructor may be applied even when H has infinite index in G, provided that its normal closure N has finite index. The subgroup N is obtained by computing the coset table of the trivial subgroup in the group defined by the relations of G together with relators corresponding to the words generating H. For a sample application of this function, see Example H70E17.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

`ncl< G | f >`

Given a homomorphism f from G onto a transitive subgroup of $\mathrm{Sym}(n)$, construct the subgroup of G that is the normal closure of the subgroup K of G which affords this permutation representation.

`CommutatorSubgroup(G)`

`DerivedSubgroup(G)`

`DerivedGroup(G)`

Given an fp-group G, try to construct the derived subgroup G' of G as finite index subgroup of G. The construction fails if no presentation for G is known or can be constructed, or if the index of G' in G is too large or infinite.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function SetGlobalTCParameters. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

Example H70E41

The group $(8, 7 \mid 2, 3)$ is defined by the presentation

$$< a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 >,$$

and has a subgroup of index 448 generated by the words a^2 and $a^{-1}b$:

```
> G<a, b> := Group<a, b| a^8, b^7, (a * b)^2, (a^-1 * b)^3>;
> G;
Finitely presented group G on 2 generators
Relations
     a^8 = Id(G)
       b^7 = Id(G)
       (a * b)^2 = Id(G)
> H<x, y> := sub< G | a^2, a^-1 * b >;
> H;
Finitely presented group H on 2 generators
Generators as words in group G
      x = a^2
      y = a^-1 * b
```

Example H70E42

Given the group G defined by the presentation

$$< a, b \mid a^8, b^7, (ab)^2, (a, b)^9 >,$$

there is a homomorphism into Sym(9) defined by

$$a \to (2,4)(3,5)(6,7)(8,9)$$
$$b \to (1,2,3)(4,6,7)(5,8,9)$$

We construct the subgroup H of G that is the preimage of the stabiliser of the point 1 in G.

```
> G<a, b> := Group< a, b | a^2, b^3, (a*b)^7, (a, b)^9>;
> T := PermutationGroup< 9 | (2, 4)(3, 5)(6, 7)(8, 9),
>    (1, 2, 3)(4, 6, 7)(5, 8, 9) >;
> f := hom< G -> T | a -> T.1, b ->T.2 >;
> H := sub< G | f >;
> H;
Finitely presented group H
Subgroup of group G defined by coset table
> Index(G, H);
9
```

Using the function GeneratingWords, we obtain a set of generators for H.

```
> print GeneratingWords(G, H);
      { a, b^-1 * a * b^3 * a * b, b * a * b * a * b * a * b^-1,
        b^3, b^-1 * a * b * a * b * a * b, b * a * b^3 * a * b^-1 }
```

70.6.2 Index of a Subgroup: The Todd-Coxeter Algorithm

This section describes the simplest use of coset enumeration techniques in MAGMA. MAGMA also provides interactive facilities for coset enumeration. For information on these more advanced uses, the user is referred to Chapter 71.

The Todd-Coxeter implementation installed in MAGMA is based on the stand alone coset enumeration programme ACE3 developed by George Havas and Colin Ramsay at the University of Queensland. The reader should consult [CDHW73] and [Hav91] for an explanation of the terminology and a general description of the algorithm. A manual for ACE3 as well as the sources of ACE3 can be found online [Ram].

Experienced users can control the Todd-Coxeter procedures invoked by the functions described in this section with a wide range of parameters. For a complete description of these parameters and their meanings we refer to the manual entry for the function CosetEnumerationProcess in Chapter 71. We just mention briefly the most important ones:

CosetLimit RNGINTELT *Default :* 0

If CosetLimit is set to n, where n is a positive integer, then the coset table may have at most n rows. In other words, a maximum of n cosets can be defined at any instant during the enumeration. It is ensured in this case, that enough memory is allocated to store the requested number of cosets, regardless of the value of the parameter Workspace.

If `CosetLimit` is set to 0 (default), the maximal number of active cosets is determined by the size of the coset table (cf. parameter `Workspace`) and the number of columns of the coset table (i.e. the number of group generators).

`Workspace` RngIntElt *Default :* 4000000

The number of words allocated for the coset table. Note that if `CosetLimit` is set, at least as much memory is allocated as is necessary to store the requested number of cosets.

`Strategy` MonStgElt *Default :*

Using this parameter one of several predefined strategies can be selected. (See the manual entry for the function `CosetEnumerationProcess` in Chapter 71 for a complete list.) The most important values of this parameter are:

- "Easy": This selects a combination of parameters, which in situations where an overflow is not expected is likely to produce a result more quickly and using less memory than the default strategy. This strategy will fail for more complicated enumerations.
- "Hard": This selects a combination of parameters, which is more likely to produce a finite result for difficult coset enumerations than the default strategy. This strategy will use more memory and enumerations usually will take longer.

`ToddCoxeter(G, H: parameters)`

Given a subgroup H of the fp-group G, this function attempts to build up a coset table of H in G using the Todd-Coxeter procedure.

The first return value is the index of H in G (or 0, if the enumeration fails to complete). The other values returned are the coset table map, the maximum number of simultaneously active cosets, and the total number of cosets defined during the enumeration.

Experienced users can control the Todd-Coxeter procedure invoked by this function with a wide range of parameters. This function accepts the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

`Index(G, H: parameters)`

`FactoredIndex(G, H: parameters)`

Given a subgroup H of the fp-group G, these functions attempt to determine the index of H in G by enumerating the cosets of H using the Todd-Coxeter procedure. The index is returned as a positive integer (`Index`) or as a sequence of factors (`FactoredIndex`), respectively. If the coset enumeration fails to complete with a closed coset table, `Index` returns a value of 0, whereas `FactoredIndex` reports an error. No conclusion can be drawn in this case.

Experienced users can control the Todd-Coxeter procedure invoked by these functions with a wide range of parameters. Both functions accept the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

Example H70E43

The classical test example for Todd-Coxeter programmes is the enumeration of the 448 cosets of the subgroup $H = < a^2, a^{-1}b >$ in the group $G = < a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 >$.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a^2,a^-1*b>;
> Index(G, H);
448
```

`Order(G: parameters)`

`FactoredOrder(G: parameters)`

Given an fp-group G, this function attempts to determine the order of G or to prove that G is infinite. If a finite order can be computed, the function `Order` returns the order as a positive integer, whereas the function `FactoredOrder` returns a sequence of prime power factors. The function `FactoredOrder` reports an error in all other cases, whereas the function `Order` returns the object `Infinity`, if G can be shown to be infinite and returns a value of 0 if neither a finite value for the group order nor a proof for the infinity of G can be obtained. No conclusions can be drawn from a return value 0 of `Order`.

In addition to the parameters controlling possibly invoked coset enumerations, there exist some other parameters controlling the strategy used by the functions `Order` and `FactoredOrder`. These parameters are described below.

Example H70E44

We use the function `Order` without any parameters to compute the order of the group

$$G = < a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 > .$$

```
> G<x, y> := Group<x,y | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> G;
Finitely presented group G on 2 generators
Relations
    x^8 = Id(G)
    y^7 = Id(G)
    (x * y)^2 = Id(G)
    (x^-1 * y)^3 = Id(G)
> Order(G);
10752
```

The strategy employed by the functions `Order` and `FactoredOrder` may involve trying to obtain information on certain subgroups of G. Whether or not an attempt is made to construct a presentation for a subgroup arising in the course of the computation by means of Reidemeister-Schreier rewriting, is controlled by three parameters:

`UseRewrite`	BoolElt	*Default :* `true`
`MinIndex`	RngIntElt	*Default :* 10
`MaxIndex`	RngIntElt	*Default :* 1000

If `UseRewrite` is set to `false`, attempts to construct presentations for subgroups are not made. Otherwise, `MinIndex` and `MaxIndex` specify for subgroups of which index range Reidemeister-Schreier rewriting is done.

The following strategy is used for trying to determine the order of G.

(1) Check whether G is free. If so, G is either trivial or infinite.

(2) Check whether the presentation for G is deficient (i.e. whether the number of relations is smaller than the number of generators). If it is, G is infinite.

(3) Check the subgroups of G with known order. If such a subgroup is known to be infinite or if we can compute its index in G, we're done.

(4) Try to compute the index of G in a supergroup of known order. (An infinite supergroup in which G has finite index proves G to be infinite.)

(5) Try to enumerate the cosets of the trivial subgroup in G.

(6) Check the subgroups of known or easily computable index in G. If we can compute the order of such a subgroup or prove that it is infinite, we're done.

(7) Try to enumerate the cosets of some subgroups occurring "naturally" in the presentation of G.

(8) Check the supergroups in which G has known or easily computable index. If we can compute the order of a supergroup or prove that it is infinite, we're done.

(9) Try to rewrite G w.r.t. some supergroup and to enumerate the cosets of the trivial subgroup using the resulting presentation.

Steps requiring coset enumeration in G or a supergroup of G are skipped, if no relations are known for this group. Steps involving Reidemeister-Schreier rewriting may be skipped according to the values of the parameters mentioned above.

Experienced users can control the behaviour of coset enumerations which may be invoked by the functions `Order` and `FactoredOrder` with a wide range of parameters. Both functions – in addition to the parameters mentioned above – accept the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

Example H70E45

The Harada-Norton simple group has the presentation

$$\begin{aligned} < x, a, b, c, d, e, f, g \mid & x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2, (x,a), (x,g), \\ & (bc)^3, (bd)^2, (be)^2, (bf)^2, (bg)^2, (cd)^3, (ce)^2, (cf)^2, (cg)^2, \\ & (de)^3, (df)^2, (dg)^2, (ef)^3, (eg)^2, (fg)^3, (b, xbx), (a, edcb), \\ & (a,f)dcbdcd, (ag)^5, (cdef, xbx), (b, xcdefx), (cdef, xcdefx) > \end{aligned}$$

The subgroup generated by x, b, c, d, e, f, g has index 1,140,000. We use the parameter `CosetLimit` to request a sufficiently large coset table. For the enumeration we choose the predefined strategy `Hard` with the modification of a complete C-style lookahead (`Lookahead := 2`).

```
> HN<x, a, b, c, d, e, f, g> :=
>     Group< x, a, b, c, d, e, f, g |
>               x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
>               (x, a), (x, g),
>               (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>               (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>               (d*e)^3, (d*f)^2, (d*g)^2,
>               (e*f)^3, (e*g)^2,
>               (f*g)^3,
>               (b, x*b*x),
>               (a, e*d*c*b), (a, f)*d*c*b*d*c*d, (a*g)^5,
>               (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>               (c*d*e*f, x*c*d*e*f*x)
>           >;
> H := sub<HN | x,b,c,d,e,f,g >;
> idx := Index(HN, H: Print := true, CosetLimit := 1200000,
>                     Strategy := "Hard", Lookahead := 2);
INDEX = 1140000
 (a=1140000 r=1471 h=1168483 n=1168483;
  l=2945 c=201.17;
  m=1142416 t=1470356)
> idx;
1140000
```

Example H70E46

We use a function representing a parametrised presentation to determine the order of a collection of groups obtained by systematically varying one relation. We select the predefined coset enumeration strategy `Easy` for the order computations.

```
> Grp := func< p, q, r, s |
>
>   Group<
>     x, y, z, h, k, a |
```

```
>     x^2, y^2, z^2, (x,y), (y,z), (x,z), h^3, k^3, (h,k),
>     (x,k), (y,k), (z,k), x^h*y, y^h*z, z^h*x, a^2, a*x*a*y,
>     a*y*a*x, (a,z), (a,k), x^p*y^q*z^r*k^s*(a*h)^2 >
>        >;
> [ < <i,j,k,l>, Order(Grp(i,j,k,l) : Strategy := "Easy") >
>   : i, j, k in [0..1], l in [0..2] ];
[ <<0, 0, 0, 0>, 144>, <<0, 0, 1, 0>, 18>, <<0, 1, 0, 0>, 72>,
  <<0, 1, 1, 0>, 36>, <<1, 0, 0, 0>, 18>, <<1, 0, 1, 0>, 144>,
  <<1, 1, 0, 0>, 36>, <<1, 1, 1, 0>, 72>, <<0, 0, 0, 1>, 144>,
  <<0, 0, 1, 1>, 18>, <<0, 1, 0, 1>, 72>, <<0, 1, 1, 1>, 36>,
  <<1, 0, 0, 1>, 18>, <<1, 0, 1, 1>, 144>, <<1, 1, 0, 1>, 36>,
  <<1, 1, 1, 1>, 72>, <<0, 0, 0, 2>, 144>, <<0, 0, 1, 2>, 18>,
  <<0, 1, 0, 2>, 72>, <<0, 1, 1, 2>, 36>, <<1, 0, 0, 2>, 18>,
  <<1, 0, 1, 2>, 144>, <<1,1, 0, 2>, 36>, <<1, 1, 1, 2>, 72> ]
```

70.6.3 Implicit Invocation of the Todd-Coxeter Algorithm

Several functions working with finitely presented groups at some point require a coset table of a subgroup and may invoke a coset enumeration indirectly, e.g. the function `meet` or the function `Normaliser`. The default behaviour for such implicitly called coset enumerations is the same as the one for coset enumerations invoked explicitly, e.g. using the function `ToddCoxeter`.

If such an implicitly called coset enumeration fails to produce a closed coset table, the calling function may terminate with a runtime error.

Experienced users can control the behaviour of indirectly invoked coset enumerations with a set of global parameters. These global parameters are valid for all implicitly called coset enumerations. For a detailed description of the available parameters and their meanings, we refer to Chapter 71. Note that coset enumerations which are *explicitly* invoked, e.g. by a call to the function `Index`, are not affected by this global set of parameters. Parameters for these functions have to be specified in the function call.

`SetGlobalTCParameters(: parameters)`

This function sets the parameter values used for indirect invocations of the Todd-Coxeter coset enumeration procedure. The parameters accepted and their default values are the same as for the function `CosetEnumerationProcess` described in Chapter 71.

`UnsetGlobalTCParameters()`

This function restores the default values for the parameters used for indirect invocations of the Todd-Coxeter coset enumeration procedure. For a description of the meanings of the parameters and their default values, see `CosetEnumerationProcess` in Chapter 71.

Example H70E47

We consider again the Harada-Norton simple group with the presentation

$$< x, a, b, c, d, e, f, g \mid x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2, (x,a), (x,g),$$
$$(bc)^3, (bd)^2, (be)^2, (bf)^2, (bg)^2, (cd)^3, (ce)^2, (cf)^2, (cg)^2,$$
$$(de)^3, (df)^2, (dg)^2, (ef)^3, (eg)^2, (fg)^3, (b, xbx), (a, edcb),$$
$$(a,f)dcbdcd, (ag)^5, (cdef, xbx), (b, xcdefx), (cdef, xcdefx) >$$

and the subgroup H generated by x, b, c, d, e, f, g.

```
> HN<x, a, b, c, d, e, f, g> :=
>     Group< x, a, b, c, d, e, f, g |
>              x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
>              (x, a), (x, g),
>              (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>              (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>              (d*e)^3, (d*f)^2, (d*g)^2,
>              (e*f)^3, (e*g)^2,
>              (f*g)^3,
>              (b, x*b*x),
>              (a, e*d*c*b), (a, f)*d*c*b*d*c*d, (a*g)^5,
>              (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>              (c*d*e*f, x*c*d*e*f*x) >;
> H := sub<HN | x,b,c,d,e,f,g >;
```

H has index 1,140,000 in HN. Using the default settings, the normaliser of H in HN cannot be computed.

```
>  N := Normaliser(HN, H);

>> N := Normaliser(HN, H);
                  ^
Runtime error in 'Normaliser': Coset table is not closed
```

We change the global parameters for implicitly called coset enumerations and try again.

```
> SetGlobalTCParameters( : Strategy := "Hard");
> N := Normaliser(HN, H);
```

With these parameters, the computation works. We see that H is self-normalising in HN.

```
> Index(HN, N);
1140000
> IsSelfNormalising(HN, H);
true
```

70.6.4 Constructing a Presentation for a Subgroup

70.6.4.1 Introduction

Let H be a subgroup of finite index in the finitely presented group G. It frequently happens that it is desirable to construct a set of defining relations for H from those of G. Such a presentation can be obtained either on a set of Schreier generators for H or on the given generators of H using the Reidemeister-Schreier rewriting technique [MKS76], if necessary together with extended coset enumeration [AR84, HKRR84].

We emphasise that if the user wishes only to determine the structure of the maximal abelian quotient of H, then the function `AbelianQuotientInvariants` should be used. In this case there is no need to first construct a presentation for H using the `Rewrite` function described below, since `AbelianQuotientInvariants` employs a special form of the Reidemeister-Schreier rewriting process which abelianises each relator as soon as it is constructed. Thus, compared to the function `Rewrite`, the function `AbelianQuotientInvariants` can be applied to subgroups of much larger index.

70.6.4.2 Rewriting

`Rewrite(G, H : parameters)`

Given a finitely presented group G and a subgroup H having finite index in G, return a group R isomorphic to H with a presentation on (some of) the Schreier generators of H in G. The group R will be created as a subgroup of G and defining relations of R on its generators will be available. Note that the generators of R will, in general, not correspond to the generators of H. The isomorphism from H onto R is returned as second return value.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

`Simplify`	BoolElt	*Default :* `true`

If this Boolean-valued parameter is given the value `true`, then the resulting presentation for H will be simplified (default). The function `Rewrite` returns a finitely presented group that is isomorphic to H. If simplification is requested (by setting `Simplify := true`) then the simplification procedures are invoked (see next section). These procedures perform a sequence of Tietze transformations which typically result in a considerable simplification of the presentation produced by the rewriting process. Alternatively, the user can set `Simplify := false` and then perform the simplification directly if desired. (See next section). If simplification is not requested as part of `Rewrite`, a small amount of simplification is performed on the presentation before it is returned.

`EliminationLimit`	RngIntElt	*Default :* 100
`ExpandLimit`	RngIntElt	*Default :* 150
`GeneratorsLimit`	RngIntElt	*Default :* 0

`LengthLimit`	RngIntElt	*Default :* ∞
`SaveLimit`	RngIntElt	*Default :* 10
`SearchSimultaneous`	RngIntElt	*Default :* 20
`Iterations`	RngIntElt	*Default :* 10000
`Print`	RngIntElt	*Default :* 0

These parameters control the simplification. See the description of `Simplify` for an explanation of these parameters.

`Rewrite(G, ~H : parameters)`

Given a finitely presented group G and a subgroup H having finite index in G, compute a defining set of relations for H on the existing generators, using extended coset enumeration and Reidemeister-Schreier rewriting, and change the presentation of H accordingly.

If the computation is successful, defining relations for H on its generators will be available at the end; any previously computed relations of H will be discarded. If the computation is unsuccessful, H is not changed. In any case, both the isomorphism type of H and its embedding into G as a subgroup is preserved by this function.

`Simplify`	BoolElt	*Default :* `true`

If this parameter is given the value `true` (default), then an attempt will be made to simplify the constructed set of relations by substring searches, that is, Tietze transformations not changing any generators. The generating set of H is not modified by this process.

Moreover, the extended coset enumeration can be controlled by a wide range of parameters. The function `Rewrite` – in addition to the parameter `Simplify` – accepts the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

Example H70E48

Starting with the group G defined by

$$< x, y \mid x^2, y^3, (xy)^{12}, (xy)^6(xy^{-1})^6 >,$$

we construct a subgroup K of index 3 generated by the words x, yxy^{-1} and $yxy^{-1}xy^{-1}xy$. We present the subgroup K, compute its abelian quotient structure and then show that the class 30 2-quotient of K has order 2^{62}.

```
> G<x, y> := Group< x, y | x^2, y^3, (x*y)^12, (x*y)^6*(x*y^-1)^6 >;
> G;
Finitely presented group G on 2 generators
Relations
       x^2 = Id(G)
       y^3 = Id(G)
       (x * y)^12 = Id(G)
       x * y * x * y * x * y * x * y * x * y * x * y * x * y * x * y^-1 * x *
```

```
          y^-1 * x * y^-1 * x * y^-1 * x * y^-1 * x * y^-1 = Id(G)
> K := sub< G | x, y*x*y^-1, y*x*y^-1*x*y^-1*x*y >;
> K;
Finitely presented group K on 3 generators
Generators as words in group G
       K.1 = x
       K.2 = y * x * y^-1
       K.3 = y * x * y^-1 * x * y^-1 * x * y
> Index(G, K);
3
> T := Rewrite(G, K);
> T;
Finitely presented group T on 3 generators
Generators as words in group G
    T.1 = x
    T.2 = y * x * y^-1
    T.3 = x^y
Relations
    T.1^2 = Id(T)
    T.2^2 = Id(T)
    T.3^2 = Id(T)
    (T.3 * T.2 * T.1 * T.3 * T.2)^2 = Id(T)
    (T.1 * T.3 * T.2 * T.1 * T.3)^2 = Id(T)
    (T.1 * T.2 * T.1 * T.3 * T.2)^2 = Id(T)
> AbelianQuotientInvariants(T);
[ 2, 2, 2 ]
> Q2 := pQuotient(T, 2, 30);
> FactoredOrder(Q2);
[ <2, 62> ]
```

Example H70E49

In this example we illustrate how the function `Rewrite` can be used to obtain a presentation of a finitely presented group on a different set of generators.

We start with a presentation of $L_2(7)$ on two generators x and y.

```
> F<x,y> := Group< x, y | x^3 = 1, y^3 = 1, (x*y)^4 = 1,
>                         (y*y^x)^2 = y^x*y >;
```

The group is also generated by the elements $a = (xy)^2$ and $b = y$.

```
> H<a,b> := sub<F | (x*y)^2, y >;
> Index(F,H);
1
```

At the moment, no defining relations of $H \cong F$ on the generators a and b are known.

```
> H;
Finitely presented group H on 2 generators
```

```
Index in group F is 1
Generators as words in group F
    a = (x * y)^2
    b = y
```

We apply the function Rewrite to H as a subgroup of F in order to compute defining relations on the generators a and b.

```
> Rewrite(F, ~H);
> H;
Finitely presented group H on 2 generators
Index in group F is 1
Generators as words in group F
    a = (x * y)^2
    b = y
Relations
    a^2 = Id(H)
    b^3 = Id(H)
    (a * b)^7 = Id(H)
    (a * b^-1 * a * b)^4 = Id(H)
    (b * a * b^-1 * a * b * a)^4 = Id(H)
```

The last relation turns out to be redundant; a and b are standard generators for $L_2(7)$.

```
> Order(DeleteRelation(H,5)) eq Order(H);
true
```

70.7 Subgroups of Finite Index

The functions in this section are concerned with the construction of subgroups of finite index. We first describe a method for computing all subgroups whose index does not exceed some (modest) integer bound. The next family of functions are concerned with constructing new subgroups of finite index from one or more known ones.

70.7.1 Low Index Subgroups

`LowIndexSubgroups(G, R : parameters)`

Given a finitely presented group G (possibly the free group), and an expression R defining a positive integer range (see below), determine the conjugacy classes of subgroups of G whose indices lie in the range specified by R. The subgroups are generated by systematically building all coset tables consistent with the defining relations for G and which satisfy the range condition R. The argument R is one of the following:

(a) An integer n representing the range $[1, n]$;

(b) A tuple $< a, b >$ representing the range $[a, b]$;

The generation of subgroups can be controlled by a set of parameters described below. The returned sequence contains the subgroups found and is sorted in order of increasing index in G.

Example H70E50

(Peter Lorimer) The two graphs known as Tutte's 8-cage and the Conder graph may be constructed as the Cayley graphs of two conjugacy classes of subgroups having index 10 in the finitely presented group

$$< q, r, s, h, a \mid h^3 = a^2 = p^2 = 1, p^h = p, p^a = q, q^h = r, r^a = s,$$
$$h^s = h^{-1}, r^h = pqr, sr = pqrs, pq = qp, pr = rp, ps = sp, qr = rq, qs = sq > .$$

We use the low index function to construct these subgroups.

```
> G<p, q, r, s, h, a> := Group<p, q, r, s, h, a |
>                         h^3 = a^2 = p^2 = 1, p^h = p, p^a = q,
>                         q^h = r, r^a = s, h^s = h^-1, r^h = p * q * r,
>                         s * r = p * q * r * s, p * q = q * p,
>                         p * r = r * p, p * s = s * p, q * r = r * q,
>                         q * s = s * q>;
> LowIndexSubgroups(G, <10, 10>);
[
       Finitely presented group on 6 generators
       Index in group G is 10 = 2 * 5
       Generators as words in group G
              $.1 = p
              $.2 = s
              $.3 = h
              $.4 = q^-2
              $.5 = a * h^-1 * a * r^-1
              $.6 = a * h * a * h^-1 * a * q^-1,

       Finitely presented group on 6 generators
       Index in group G is 10 = 2 * 5
       Generators as words in group G
              $.1 = p
              $.2 = s
              $.3 = h
              $.4 = q^-2
              $.5 = a * h^-1 * a * r^-1
              $.6 = a * h * a * h^-1 * a
]
```

Example H70E51

A fairly surprising application for the low index subgroup algorithm is the enumeration of the conjugacy classes of a finite fp-group. In this example, we consider the group

$$G \simeq \mathrm{PGL}_2(9) = \langle a, b | a^2, b^3, (ab)^8, [a,b]^5, [a,(ba)^3b^{-1}]^2 \rangle.$$

```
> G<a,b> := Group< a,b | a^2, b^3, (a*b)^8, (a,b)^5,
>                       (a,(b*a)^3*b^-1)^2  >;
> Order(G);
720
```

In an infinite fp-group, finding all classes of subgroups up to an index of 720 by applying the low index subgroup algorithm, would be extremely hard. In the case of the finite group G, however, we succeed.

```
> time sgG := LowIndexSubgroups(G, Order(G));
Time: 31.859
> #sgG;
26
```

We get a list of 26 representatives of the conjugacy classes of subgroups. For every representative, its index in G and a set of generating words are known. We just have a look at two of them.

```
> sgG[10];
Finitely presented group on 2 generators
Index in group G is 60 = 2^2 * 3 * 5
Generators as words in group G
    $.1 = b
    $.2 = a * b * a * b * a * b^-1 * a * b * a * b * a * b^-1 * a
       * b^-1 * a
> sgG[21];
Finitely presented group on 1 generator
Index in group G is 180 = 2^2 * 3^2 * 5
Generators as words in group G
    $.1 = (b * a)^2
```

The function LowIndexSubgroups constructs all conjugacy classes of subgroups of G satisfying the following two conditions:

(i) The index of each subgroup is in the range defined by R;

(ii) If the parameter Subgroup defines a subgroup H, then at least one subgroup in each conjugacy class contains the subgroup H.

The subgroups are returned as a sequence of subgroups G, unless otherwise specified by the parameter GeneratingSets (see below). The sequence is sorted by increasing index of the subgroups in G.

The subgroups are constructed using an algorithm due to Sims [Sim94, sect. 5.6]. This algorithm constructs the coset tables by using a backtrack algorithm. At a given position in the coset table, coset definitions are made systematically. Once a new definition has been made, the group relations are traced in an attempt to deduce further entries or to infer that this partial table will not extend to a table corresponding to a new class of subgroups. When either it cannot define a new entry, or when a complete table has been constructed, the algorithm backtracks to try the next possibility (this may introduce a new row, increasing the index). This algorithm may also be run as a process in such a way that the subgroups are returned one at a time, thereby allowing the user to analyze each subgroup as soon as it is found.

`ColumnMajor` BoolElt *Default :* `false`

If `ColumnMajor` is set `false` (*default*), then the location for a new definition in the coset table is determined by searching the table in row major order for undefined entries. If `ColumnMajor` is set `true`, then the position for a new definition is determined by searching the table in column major order. If the presentation for G contains explicit relators expressing the fact that certain of the generators have large order, then the presentation should be organized so that these generators appear first and the column major order should be selected for new coset definition. This strategy often leads to greatly improved performance.

`GeneratingSets` BoolElt *Default :* `false`

The conjugacy classes of subgroups are returned in the form of a sequence of sets of words, where the i-th set is a generating set for a representative subgroup from the i-th conjugacy class of subgroups satisfying the given conditions. This is a much more compact representation than returning the subgroups as a sequence of actual *subgroups* of G and should be used when a very large number of subgroups is expected, as there may be insufficient space to store each of them as a subgroup.

`Limit` RngIntElt *Default :* ∞

Terminate after finding n conjugacy classes of subgroups satisfying the designated conditions.

`Long` [RngIntElt] *Default :* []

This option enables the user to designate certain of the defining relators for G as *long relators*. The relators of G are numbered from 1 to r, in the order they appear in the `quo`- or `Group`-constructors. The value L of `Long` is a subset of the integer set $\{1, \ldots, r\}$. Magma interprets the relators whose numbers appear in L as long relators. A relator designated as long is not used during the construction of a coset table. Rather, it is applied once a complete table has been found. There is some evidence to suggest that better performance is achieved in those groups having one or more very long relators by deferring application of these relators until such time as a complete coset table has been obtained.

`Print` RngIntElt *Default :* 0

A description of each class of subgroups may be printed immediately after it is constructed. The value n assigned to the `Print` parameter specifies just what information is to be printed, according to the following rules:

$n = 0$: No printing (*default*).

$n = 1$: For each class, print a heading and a set of generators for the class representative.

$n = 2$: The information printed for $n = 1$, together with the permutation representation of G on the right cosets of the class representative.

$n = 3$: The information printed for $n = 2$, together with generators for the normalizer N of the class representative, and a system of right coset representatives for N in G.

`Subgroup` GrpFP *Default :* `sub< G | >`

By specifying a value H for `Subgroup`, only subgroups containing H will be constructed.

`TimeLimit` RngIntElt *Default :* 0 (no limit)

A time limit in seconds. A value of 0 (default) means no limit.

`LowIndexProcess(G, R : parameters)`

`ColumnMajor`	BoolElt	*Default :* `false`
`GeneratingSets`	BoolElt	*Default :* `false`
`Long`	[RngIntElt]	*Default :* []
`Print`	RngIntElt	*Default :* 0
`Subgroup`	GrpFP	*Default :* `sub< G | >`
`TimeLimit`	RngIntElt	*Default :* 0 (no limit)

Create a low index subgroups *process*. This process may be used to create the conjugacy classes of proper subgroups one at time, with control being handed back to the Magma language processor each time a new class of subgroups is found. This function returns a process which is used by the function `NextSubgroup` to actually produce the subgroups.

The arguments and parameters have the same interpretation as for the function `LowIndexSubgroups`, except that `Limit` is not available (since the same effect can be achieved by limiting the number of calls to `NextSubgroup`).

Setting a time limit for a process P limits the total amount of time spent in calls to `NextSubgroup`. If the time limit is exceeded in a call to `NextSubgroup`, this call is aborted and P becomes invalid. Further attempts to access P will cause a runtime error. The function `IsValid` can be used to check whether a process is valid in order to avoid runtime errors in loops or user written functions.

`NextSubgroup(~P)`

`NextSubgroup(~P, ~G)`

Given a low index subgroups process P, construct the next conjugacy class of proper subgroups. The process P must have been previously created using the function `LowIndexProcess` and must be valid. Calling `NextSubgroup` for an empty process has no effect.

`ExtractGroup(P)`

Extract a representative subgroup for the conjugacy class currently defined by the low index process P. The subgroup extracted will be the one found by the previous invocation of `NextSubgroup`, or the first subgroup if `NextSubgroup` has never been invoked on this process. Note that `ExtractGroup` will not search for a new subgroup. If P is empty or invalid, a runtime error will result.

`ExtractGenerators(P)`

Extract a generating set for the representative subgroup of the conjugacy class currently defined by the low index process P. The subgroup extracted will be the one found by the previous invocation of `NextSubgroup`, or the first subgroup if `NextSubgroup` has never been invoked on this process. Note that `ExtractGenerators` will not search for a new subgroup. If P is empty or invalid, a runtime error will result.

`IsEmpty(P)`

Return `true` if the low index process P has already found all conjugacy classes of subgroups. If `IsEmpty` is called immediately following the creation of the low index process, then it will return the value `false` if there are no subgroups satisfying the specified conditions or advance P to the first such subgroup otherwise.

`IsValid(P)`

Return `true` if the low index process P is valid, that is, no limit has been exceeded. If `IsValid` is called immediately following the creation of the low index process, then it will return the value `false` if no subgroups satisfying the specified conditions can be found within the specified time or advance P to the first such subgroup otherwise.

Example H70E52

We determine all conjugacy classes of subgroups having index at most 15 in the triangle group

$$< a, b \,|\, a^2, b^3, (ab)^7 > .$$

```
> G<a, b> := Group< a, b | a^2, b^3, (a*b)^7 >;
> L := LowIndexSubgroups(G, 15: Print := 1);
Subgroup class 1        Index 7 Length 7        Subgroup generators :-
{ a, b * a * b^-1, b^-1 * a * b * a * b^-1 * a * b }
```

```
Subgroup class 2        Index 7 Length 7        Subgroup generators :-
{ a, b^-1 * a * b^-1 * a * b * a * b, b * a * b^-1 }
Subgroup class 3        Index 15        Length 15       Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b * a * b^-1 * a * b, b * a * b^-1 }

Subgroup class 4        Index 15        Length 15       Subgroup generators :-
{ a, b * a * b^-1, b^-1 * a * b^-1 * a * b * a * b^-1 * a * b * a * b }

Subgroup class 5        Index 14        Length 14       Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b * a * b * a * b^-1 * a * b, b * a * b^-1 }

Subgroup class 6        Index 14        Length 7        Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b * a * b^-1 * a * b * a * b^-1 * a * b, b *
a * b * a * b^-1 }

Subgroup class 7        Index 14        Length 14       Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b^-1 * a * b * a * b * a * b^-1 * a * b, b *
a * b * a * b^-1 }

Subgroup class 8        Index 14        Length 7        Subgroup generators :-
{ a, b * a * b * a * b^-1 * a * b^-1 * a * b * a * b * a * b^-1 * a * b^-1, b^-1
 * a * b * a * b }

Subgroup class 9        Index 15        Length 15       Subgroup generators :-
{ a, b * a * b * a * b^-1 * a * b^-1, b^-1 * a * b^-1 * a * b * a * b }

Subgroup class 10       Index 9 Length 9        Subgroup generators :-
{ a, b * a * b * a * b * a * b^-1 }
Subgroup class 11       Index 14        Length 7        Subgroup generators :-
{ a, b * a * b * a * b * a * b^-1 * a * b, b * a * b^-1 * a * b * a * b^-1 }
Subgroup class 12       Index 14        Length 7        Subgroup generators :-
{ a, b * a * b * a * b^-1 * a * b * a * b, b * a * b^-1 * a * b * a * b^-1 }
Subgroup class 13       Index 14        Length 7        Subgroup generators :-
{ a, b^-1 * a * b * a * b^-1 * a * b, b * a * b * a * b * a * b^-1 * a * b^-1 }

Subgroup class 14       Index 14        Length 7        Subgroup generators :-
{ a, b * a * b * a * b^-1 * a * b * a * b, b^-1 * a * b * a * b^-1 * a * b }

Subgroup class 15       Index 14        Length 14       Subgroup generators :-
{ a, b^-1 * a * b * a * b * a * b^-1 * a * b, b * a * b * a * b * a * b^-1 * a *
 b^-1 }
Subgroup class 16       Index 8 Length 8        Subgroup generators :-
{ b, a * b * a * b * a * b^-1 * a }
```

Example H70E53

In this example we illustrate the use of the low index subgroup process by using it to determine whether the simple group PSL$(2, 8)$ is a homomorphic image of the triangle group

$$< x, y \mid x^2, y^3, (xy)^7 > .$$

```
> F<x, y> := FreeGroup(2);
> G<x, y> := quo< F | x^2, y^3, (x*y)^7 >;
> LP := LowIndexProcess(G, 30);
> i := 0;
> while i le 100 and not IsEmpty(LP) do
>    H := ExtractGroup(LP);
>    NextSubgroup(~LP);
>    P := CosetImage(G, H);
>    if Order(P) eq 504 and IsSimple(P) then
>       Ψprint "The group G has L(2, 8) as a homomorphic image.";
>        print "It is afforded by the subgroup:-", H;
>       Ψbreak;
>    end if;
>    i +:= 1;
> end while;
The group G has L(2, 8) as a homomorphic image.
It is afforded by the subgroup:-
Finitely presented group H on 4 generators
Index in group G is 28 = 2^2 * 7
Generators as words in group G
       H.1 = x
       H.2 = y * x * y^-1
       H.3 = y^-1 * x * y * x * y^-1 * x * y * x * y^-1 * x * y * x * y^-1 *
       x * y * x * y
       H.4 = y^-1 * x * y * x * y^-1 * x * y^-1 * x * y * x * y^-1 * x * y *
       x * y * x * y^-1 * x * y
```

Example H70E54

This example shows how the low index subgroup machinery may be used as part of a function trying to prove that a group is infinite:

```
> function MyIsInfinite(G)
>
>  // ...
>
>  // Low index subgroup approach: check whether an obviously
>  //    infinite subgroup can be found in reasonable time.
>  P := LowIndexProcess(G, 30 : TimeLimit := 5);
>  while IsValid(P) and not IsEmpty(P) do
>     H := ExtractGroup(P);
```

```
>     NextSubgroup(~P);
>     if 0 in AbelianQuotientInvariants(H) then
>        print "The group G has subgroup:-", H;
>        print "whose abelian quotient is infinite";
>        print "Hence G is infinite.";
>        return true;
>     end if;
>  end while;
>  print "Low index approach fails; trying other methods...";
>
>  // ...
>
> end function;
```

We try the code fragment on the group

$$< x, z \mid z^3xz^3x^{-1}, z^5x^2z^2x^2 > .$$

```
> G<x, z> := Group<x,z | z^3*x*z^3*x^-1, z^5*x^2*z^2*x^2 >;
> MyIsInfinite(G);
The group G has subgroup:-
Finitely presented group H on 4 generators
Index in group G is 4 = 2^2
Generators as words in group G
    H.1 = x
    H.2 = z * x * z
    H.3 = z^3
    H.4 = z * x^-1 * z * x * z^-1
whose abelian quotient has structure [ 2, 6, 0 ]
Hence G is infinite.
true
```

`LowIndexNormalSubgroups(G, n: parameters)`

The normal subgroups of finitely presented group G up to index n, $n \leq 100\,000$. The subgroups are returned as a sequence of records (ordered by subgroup index) where the ith record contains fields

`Group`: A presentation of the ith normal subgroup.

`Index`: The index of the ith normal subgroup in G.

`Supegroups`: The set of positions in the sequence of the groups which are supergroups of the ith group.

`PrintLevel` RingIntElt *Default :* 0

This parameter may be set to 0, 1 or 2. At 0, the function prints no diagnostic output. At level 1, it outputs details of each normal subgroup being tested for further normal subgroups. Level 2 gives details of each test being performed on each normal subgroup.

`Simplify` MonStgElt *Default : "No"*

The possible values are `"No"`, `"Yes"` and `"LengthLimit"`. This determines the parameter values passed to the `Rewrite(G,H)` function, when this function is used. The value `"No"` sets parameter `Simplify:=false`. The value `"Yes"` sets parameter `Simplify:=true`. The value `"LengthLimit"` sets parameter `LengthLimit:=Index(G,H)`.

70.7.2 Subgroup Constructions

Most operations described in this subsection require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function `SetGlobalTCParameters`. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

`H ^ u`
`Conjugate(H, u)`

Given an fp-group H and a word u in an fp-group K, such that H and K are subgroups of some common fp-group G and words in terms of the generators of G are known for the generators of both H and K, construct the subgroup of G obtained by conjugating H by u.

`H meet K`

Given subgroups H and K, both of finite index in some fp-group G, return the subgroup which is the intersection of H and K.

This function requires closed coset tables for both, H and K in G.

`Core(G, H)`

Given a subgroup H of finite index in the fp-group G, construct the core of H in G.

This function requires a closed coset table for H in G.

`GeneratingWords(G, H)`

Given a subgroup H of the fp-group G, this function returns a set of words in the generators of G, generating H as a subgroup of G (assuming such words are known or can be constructed). Note that the returned generating set does not necessarily correspond to the internal generators of H. In particular, generating words obtained using the function `GeneratingWords` cannot be used to coerce elements from H to G.

`MaximalOvergroup(G, H)`

Given a subgroup H of finite index in the fp-group G, construct a maximal overgroup of H in G. A *maximal overgroup* of H is a maximal subgroup of G that contains H. If H is already maximal, the group G is returned.

This function requires a closed coset table for H in G.

`MinimalOvergroup(G, H)`

Given a subgroup H of finite index in the fp-group G, construct a minimal overgroup of H in G. A *minimal overgroup* of a subgroup H is a subgroup K of G such that K contains H as a maximal subgroup. If H is already maximal in G, the group G is returned.

This function requires a closed coset table for H in G.

`H ^ G`

`NormalClosure(G, H)`

Given a subgroup H of finite index in the fp-group G, construct the normal closure of H in G.

This function requires a closed coset table for H in G.

`Normaliser(G, H)`

`Normalizer(G, H)`

Given a subgroup H of finite index in the fp-group G, construct the normaliser of H in G. For a sample application of this function, see Example H70E47.

This function requires a closed coset table for H in G.

`SchreierGenerators(G, H :` *parameters*`)`

`Simplify` BoolElt *Default :* `true`

Given a subgroup H of finite index in the fp-group G, return the Schreier generators for H as a set of words in G.

If the parameter `Simplify` is set to `true` (default), a heuristic method of eliminating redundant Schreier generators is applied. To switch this feature off, set `Simplify` to `false`.

This function requires a closed coset table for H in G.

`SchreierSystem(G, H)`

`Transversal(G, H)`

Given a subgroup H of finite index in the fp-group G, construct a (right) Schreier system of coset representatives for H in G. The function returns

(a) the Schreier system as a set of words in G;

(b) the corresponding Schreier coset function.

This function requires a closed coset table for H in G.

`Transversal(G, H, K)`

Given subgroups H and K, both of finite index in the fp-group G, return an indexed set of words which comprise a set of representatives for the double cosets HuK of H and K in G, as well as a map from G to the representatives. It should be noted that this function is evaluated by first constructing the right cosets of H in G and then computing the orbits of the cosets under the action of the generators of the subgroup K.

This function requires a closed coset table for H in G.

Example H70E55

We illustrate some of the subgroup constructions by using them to construct subgroups of small index in the two-dimensional space group p4g which has the presentation

$$< r, s \,|\, r^2, s^4, (r,s)^2 > .$$

```
> p4g<r, s> := Group< r, s | r^2 = s^4 = (r*s^-1*r*s)^2 = 1 >;
> p4g;
Finitely presented group p4g on 2 generators
Relations
    r^2 = Id(p4g)
    s^4 = Id(p4g)
    (r * s^-1 * r * s)^2 = Id(p4g)
```

We define two subgroups of $p4g$ and compute their indices in $p4g$.

```
> h := sub< p4g | (s^-1*r)^4, s*r >;
> k := sub< p4g | (s^-1*r)^2, (s*r)^2 >;
> Index(p4g, h);
8
> Index(p4g, k);
8
```

We construct the normal closure of h in $p4g$.

```
> n := NormalClosure(p4g, h);
> n;
Finitely presented group n on 6 generators
Index in group p4g is 2
Generators as words in group p4g
    n.1 = (s^-1 * r)^4
    n.2 = s * r
    n.3 = r * s
    n.4 = r^-1 * s * r^2
    n.5 = s^2 * r * s^-1
    n.6 = r * s
```

Next, we construct a subgroup of $p4g$ containing h as maximal subgroup...

```
> m := MinimalOvergroup(p4g, h);
```

```
> m;
Finitely presented group m on 3 generators
Index in group p4g is 4 = 2^2
Generators as words in group p4g
    m.1 = (s^-1 * r)^4
    m.2 = s * r
    m.3 = (r * s)^2
```

... and a maximal subgroup of $p4g$ containing k.

```
> n := MaximalOvergroup(p4g, k);
> n;
Finitely presented group n on 4 generators
Index in group p4g is 2
Generators as words in group p4g
    n.1 = (s^-1 * r)^2
    n.2 = (s * r)^2
    n.3 = r
    n.4 = s^2
```

Finally, we construct a transversal in $p4g$ for the normaliser of h in $p4g$...

```
> T := Transversal(p4g, Normaliser(p4g, h));
> T;
{@ Id(p4g), r, s^-1, r * s @}
```

... compute the intersection of h and the conjugate of h by r...

```
> l := h meet h^r;
> l;
Finitely presented group l
Index in group p4g is 32 = 2^5
Subgroup of group p4g defined by coset table
```

... and construct the core of h in $p4g$.

```
> c := Core(p4g, h);
> c;
Finitely presented group c
Index in group p4g is 32 = 2^5
Subgroup of group p4g defined by coset table
```

Note, that the two subgroups l and c constructed last are defined as finite index subgroups of $p4g$ by a coset table and that there are no generators known for them. Generators can be obtained e.g. by using the function GeneratingWords. We show this for the subgroup l.

```
> GeneratingWords(p4g, l);
{ (s * r)^4, (s^-1 * r)^4 }
```

Once computed, these generators are memorised. Compare the result of printing l to the output obtained above.

```
> l;
```

```
Finitely presented group l on 2 generators
Index in group p4g is 32 = 2^5
Generators as words in group p4g
    l.1 = (s * r)^4
    l.2 = (s^-1 * r)^4
```

Example H70E56

Consider the group G given by the presentation $\langle x, y \mid x^2, y^3, (xy)^7 \rangle$.

```
> G<x,y> := Group< x,y | x^2, y^3, (x*y)^7 >;
```

We construct the subgroups of index less than or equal to 7 using the low index algorithm.

```
> L := LowIndexSubgroups(G, 7);
> L;
[
    Finitely presented group on 2 generators
    Index in group G is 1
    Generators as words in group G
        $.1 = x
        $.2 = y,
     Finitely presented group on 3 generators
    Index in group G is 7
    Generators as words in group G
        $.1 = x
        $.2 = y * x * y^-1
        $.3 = y^-1 * x * y^-1 * x * y * x * y,
     Finitely presented group on 3 generators
    Index in group G is 7
    Generators as words in group G
        $.1 = x
        $.2 = y * x * y^-1
        $.3 = y^-1 * x * y * x * y^-1 * x * y
]
```

We define a subgroup as the core of one of the subgroups of index 7. The function `Core` returns a subgroup of G defined by a coset table.

```
> H := Core(G, L[2]);
> H;
Finitely presented group H
Index in group G is 168 = 2^3 * 3 * 7
Subgroup of group G defined by coset table
```

A set of generators for H can be obtained e.g. with the function `SchreierGenerators`.

```
> sgH := SchreierGenerators(G, H);
> #sgH;
```

```
6
```

By default, SchreierGenerators returns a reduced generating set. The unreduced set of Schreier generators can be obtained by setting the value of the parameter Simplify to false.

```
> sgHu := SchreierGenerators(G, H : Simplify := false);
> #sgHu;
85
```

70.7.3 Properties of Subgroups

The operations described in this subsection all require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function SetGlobalTCParameters. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

`u in H`

Given an fp-group H and a word u in an fp-group K, such that H and K are subgroups of some common fp-group G, H is of finite index in G, and words for the generators of K in terms of the generators of G are known, return `true`if u is an element of H and `false` otherwise.

This function requires a closed coset table for H in G.

`u notin H`

Given an fp-group H and a word u in an fp-group K, such that H and K are subgroups of some common fp-group G, H is of finite index in G, and words for the generators of K in terms of the generators of G are known, return `true`if u is not an element of H and `false` otherwise.

This function requires a closed coset table for H in G.

`H eq K`

Given subgroups H and K, both of finite index in the fp-group G, return `true`if H and K are equal and `false` otherwise.

This function may require closed coset tables for both, H and K in G.

`H ne K`

Given subgroups H and K, both of finite index in the fp-group G, return `true`if H and K are not equal and `false` otherwise.

This function may require closed coset tables for both, H and K in G.

`H subset K`

Given subgroups H and K, both of finite index in the fp-group G, return `true`if H is contained in K and `false` otherwise.

This function requires a closed coset table for K in G.

`H notsubset K`

Given subgroups H and K, both of finite index in the fp-group G, return `true`if H is not contained in K and `false` otherwise.

This function requires a closed coset table for K in G.

`IsConjugate(G, H, K)`

Given subgroups H and K, both of finite index in the fp-group G, return `true`if H and K are conjugate subgroups of G and `false` otherwise. If H and K are conjugate in G, a conjugating element is returned as second return value.

This function requires a closed coset table for both, H and K in G.

`IsNormal(G, H)`

Given a subgroup H of finite index in the fp-group G, return `true`if H is a normal subgroup of G and `false` otherwise.

This function requires a closed coset table for H in G.

`IsMaximal(G, H)`

Given a subgroup H of finite index in the fp-group G, return `true`if H is a maximal subgroup of G and `false` otherwise.

This function requires a closed coset table for H in G.

`IsSelfNormalizing(G, H)`

Given a subgroup H of finite index in the fp-group G, return `true`if H is a self-normalizing subgroup of G and `false` otherwise. For a sample application of this function, see Example H70E47.

This function requires a closed coset table for H in G.

Example H70E57

We illustrate some of the subgroup predicates by applying them to some subgroups of the two-dimensional space group $p4g =< r, s | r^2, s^4, (r,s)^2 >$ from Example H70E55.

```
> p4g<r, s> := Group< r, s | r^2 = s^4 = (r*s^-1*r*s)^2 = 1 >;
> h := sub< p4g | (s^-1*r)^4, s*r >;
> k := sub< p4g | (s^-1*r)^2, (s*r)^2 >;
```

h and k have the same index in $p4g\ldots$

```
> Index(p4g, h);
8
> Index(p4g, k);
```

```
8
```

. . . but they are not equal.

```
> h eq k;
false
```

We check for normality of h and k in $p4g$.

```
> IsNormal(p4g, h);
false
> IsNormal(p4g, k);
true
```

We construct the normal closure of h in $p4g$.

```
> n := NormalClosure(p4g, h);
```

We see that it is maximal. . .

```
> IsMaximal(p4g, n);
true
```

. . . and that it contains k.

```
> k subset n;
true
```

We define another subgroup of $p4g$.

```
> l := sub< p4g | (s*r)^4, s^-1*r >;
```

In fact, it is conjugate to h.

```
> IsConjugate(p4g, h, l);
true r^-1
```

I.e. $l = h^{r^{-1}}$. The intersection of h and $h^{r^{-1}}$ already yields the core of h in $p4g$.

```
> h meet l eq Core(p4g, h);
true
```

Example H70E58

The constructions of the previous section together with the Boolean function `IsMaximal` may be used to locate large maximal subgroups in a finite group. Consider the Hall-Janko group J_2, which may be defined by the presentation

$$\begin{aligned} < a, b, c \mid\ & a^3, b^3, c^3, abab^{-1}a^{-1}b^{-1}, (ca)^5, (cb)^5, \\ & (cb^{-1}cb)^2, a^{-1}baca^{-1}bac^{-1}a^{-1}b^{-1}ac^{-1}, \\ & aba^{-1}caba^{-1}c^{-1}ab^{-1}a^{-1}c^{-1} > . \end{aligned}$$

We examine subgroups generated by pairs of randomly chosen short words. Whenever we obtain a proper subgroup, if it is not already maximal we replace it by a maximal subgroup that contains it.

```
> J2<a, b, c> := Group<a, b, c | a^3, b^3, c^3, a*b*a*b^-1*a^-1*b^-1, (c*a)^5,
>                                  (c*b)^5, (c*b^-1*c*b)^2,
>                                  a^-1*b*a*c*a^-1*b*a*c^-1*a^-1*b^-1*a*c^-1,
>                                  a*b*a^-1*c*a*b*a^-1*c^-1*a*b^-1*a^-1*c^-1>;
>
> Seen := { 0, 1};
> Found := { };
> Sgs := [ ];
> for i := 1 to 30 do
>     u := Random(J2, 1, 1);
>     v := Random(J2, 3, 5);
>     H := sub< J2 | u, v >;
>     Indx := Index(J2, H);
>     if Indx notin Seen then
>        Include(~Seen, Indx);
>          if not IsMaximal(J2, H) then
>            H := MaximalOvergroup(J2, H);
>         end if;
>        if Indx notin Found then
>            Include(~Sgs, H);
>            Include(~Seen, Indx);
>            Include(~Found, Indx);
>        end if;
>    end if;
> end for;
> Sgs;
[
      Finitely presented group on 3 generators
      Index in group J2 is 315 = 3^2 * 5 * 7
      Generators as words in group J2
             $.1 = b^-1
             $.2 = a^-2 * c * a^-1
             $.3 = c,

      Finitely presented group on 3 generators
      Index in group J2 is 1008 = 2^4 * 3^2 * 7
      Generators as words in group J2
             $.1 = b^-1
             $.2 = c^-1 * a^-1 * c^-1 * b
             $.3 = a * c * b * c^-1 * a^-1 * c * b * c^-1,

      Finitely presented group on 3 generators
      Index in group J2 is 100 = 2^2 * 5^2
      Generators as words in group J2
```

```
            $.1 = c
            $.2 = (b * a^-1)^2
            $.3 = a * b^-1
]
```

Thus after taking 30 2-generator random subgroups, we have obtained three maximal subgroups, including the two largest maximal subgroups.

70.8 Coset Spaces and Tables

Let $G =< X|R >$ be a finitely presented group. Suppose that H is a subgroup of G having finite index m. Let $V = \{c_1(= H), c_2, \ldots, c_m\}$ denote the set of distinct right cosets of H in G. This set admits a natural G-action

$$f : V \times G \to V$$

where

$$f :< c_i, x >= c_k \quad \Longleftrightarrow \quad c_i * x = c_k,$$

for $c_i \in V$ and $x \in G$. The set V together with this action is a G-set called a *right coset space* for H in G. The action may also be represented using a coset table T.

If certain of the products $c_i * x$ are unknown, the corresponding images under f are undefined. In this case, T is not closed, and V is called an *incomplete coset space* for H in G.

70.8.1 Coset Tables

A coset table is represented in MAGMA as a mapping. Given a finitely-presented group G and a subgroup H, the corresponding (right) coset table is a mapping $f : \{1, \ldots, r\} \times G \to \{0, \ldots, r\}$, where r is the index of H in G. $f(i, x)$ is the coset to which coset i is mapped under the action of $x \in G$. The value 0 is only included in the codomain if the coset table is not closed, and it denotes that the coset is not known.

`CosetTable(G, H: parameters)`

The (right) coset table for G over subgroup H, constructed by means of the Todd-Coxeter procedure. If the coset table does not close then the codomain will include the value 0.

Experienced users can control the Todd-Coxeter procedure invoked by this function with a wide range of parameters. This function accepts the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

`CosetTableToRepresentation(G, T)`

Given a coset table T for a subgroup H of G, construct the permutation representation of G given by its action on the cosets of H, using the columns of T. The function returns:

(a) The homomorphism $f : G \to P$;

(b) The permutation group image P;

(c) The kernel K of the action (a subgroup of G).

`CosetTableToPermutationGroup(G, T)`

Given a coset table T for a subgroup H of G, construct the permutation group image of G given by its action on the cosets of H, using the columns of T. This is the second return value of `CosetTableToRepresentation(G, T)`.

Example H70E59

Consider the infinite dihedral group.

```
> G<a,b> := DihedralGroup(GrpFP, 0);
> G;
Finitely presented group G on 2 generators
Relations
    b^2 = Id(G)
    (a * b)^2 = Id(G)
```

We define a subgroup S of G and compute the coset table map for S in G.

```
> S := sub<G|a*b, a^10>;
> ct := CosetTable(G, S);
> ct;
Mapping from: Cartesian Product<{ 1 .. 10 }, GrpFP: G>
        to { 1 .. 10 }
     $1  $2 -$1
 1.   2   2   3
 2.   4   1   1
 3.   1   4   5
 4.   6   3   2
 5.   3   6   7
 6.   8   5   4
 7.   5   8   9
 8.  10   7   6
 9.   7  10  10
10.   9   9   8
```

When printing the coset table, the action of the generators and of the non-trivial inverses of generators on the enumerated transversal is shown in table form.

Using the coset table, we now construct the permutation representation of G on the cosets of S in G. We assign the representation (a homomorphism), the image (a permutation group of degree $[G : S] = 10$) and the kernel of the permutation representation (a subgroup of G).

```
> fP, P, K := CosetTableToRepresentation(G, ct);
> fP;
Homomorphism of GrpFP: G into GrpPerm: P, Degree 10,
Order 2^2 * 5 induced by
    a |--> (1, 2, 4, 6, 8, 10, 9, 7, 5, 3)
    b |--> (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)
```

Note that the images of a and b correspond to the first two columns of the printed coset table above.

```
> P;
Permutation group P acting on a set of cardinality 10
Order = 20 = 2^2 * 5
    (1, 2, 4, 6, 8, 10, 9, 7, 5, 3)
    (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)
> K;
Finitely presented group K
Index in group G is 20 = 2^2 * 5
Subgroup of group G defined by coset table
```

Now, we define a subgroup of infinite index in G and compute a coset table for it.

```
> H := sub<G|b>;
> ct := CosetTable(G, H);
```

Of course, the coset table cannot be complete; note that 0 is in its codomain, indicating unknown images of cosets.

```
> ct;
Mapping from: Cartesian Product<{ 1 .. 1333331 }, GrpFP: G>
        to { 0 .. 1333331 }
```

70.8.2 Coset Spaces: Construction

The indexed (right) coset space V of the subgroup H of the group G is a G-set consisting of the set of integers $\{1, ..., m\}$, where i represents the right coset c_i of H in G. The action of G on this G-set is that induced by the natural G-action

$$f : V \times G \to V$$

where

$$f :< c_i, x >= c_k \quad \Longleftrightarrow \quad c_i * x = c_k,$$

for $c_i \in V$ and $x \in G$. If certain of the products $c_i * x$ are unknown, the corresponding images under f are undefined, and V is called an *incomplete coset space* for H in G.

`CosetSpace(G, H: parameters)`

This function attempts to construct a coset space for the subgroup H in the group G by means of the Todd-Coxeter procedure. If the enumeration fails to complete, the function returns an incomplete coset space. The coset space is returned as an indexed right coset space. For a sample application of this function see Example H70E61.

Experienced users can control the Todd-Coxeter procedure invoked by this function with a wide range of parameters. This function accepts the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

`RightCosetSpace(G, H: parameters)`

`LeftCosetSpace(G, H: parameters)`

The explicit right coset space of the subgroup H of the group G is a G-set containing the set of right cosets of H in G. The elements of this G-set are the pairs $< H, x >$, where x runs through a transversal for H in G. Similarly, the explicit left coset space of H is a G-set containing the set of left cosets of H in G, represented as the pairs $< x, H >$. These functions use the Todd-Coxeter procedure to construct the explicit right (left) coset space of the subgroup H of the group G. For a sample application see Example H70E61.

Experienced users can control the Todd-Coxeter procedure invoked by these functions with a wide range of parameters. Both functions accept the same parameters as the function `CosetEnumerationProcess` described in Chapter 71.

70.8.3 Coset Spaces: Elementary Operations

`H * g`

Right coset of the subgroup H of the group G, where g is an element of G (as an element of the right coset of H).

`C * g`

Coset to which the right coset C of the group G is mapped by the (right) action of g, where g is an element of G.

`C * D`

and D.

Given two right cosets of the same normal subgroup H of the group G, return the right coset that is the product of C and D.

`g in C`

Return `true` if element g of group G lies in the coset C.

`g notin C`

Return `true` if element g of group G does not lie in the coset C.

`C1 eq C2`

Returns `true` if the coset $C1$ is equal to the coset $C2$.

`C1 ne C2`

Returns `true` if the coset $C1$ is not equal to the coset $C2$.

70.8.4 Accessing Information

`#V`

The cardinality of the coset space V.

`Action(V)`

The mapping $V \times G \to V$ giving the action of G on the coset space V. This mapping is a coset table.

`<i, w> @ T`
`T(i, w)`

The image of coset i as defined in the coset table T, under the action of word w.

`ExplicitCoset(V, i)`

The element of the explicit coset space that corresponds to indexed coset i.

`IndexedCoset(V, w)`

The element of the indexed coset space V to which the element w of G corresponds.

`IndexedCoset(V, C)`

The element of the indexed coset space V to which the explicit coset C of G corresponds.

`Group(V)`

The group G for which V is a coset space.

`Subgroup(V)`

The subgroup H of G such that V is a coset space for G over H.

`IsComplete(V)`

Returns `true` if the coset space V is complete.

`ExcludedConjugates(V)`

`ExcludedConjugates(T)`

Given a partial or complete coset space V for the group G over the subgroup H, or a coset table T corresponding to this coset space, this function returns the set of words $E = \{g_i^{-1} h_j g_i \mid g_i$ a generator of G, h_j a generator of H, and, modulo V, $g_i^{-1} h_j g_i$ does not lie in $H\}$. If E is empty, then H is a normal subgroup of G, while if E is non-empty, the addition of the elements of E to the generators of H will usually be a larger subgroup of the normal closure of H. This function may be used with incomplete coset spaces for G over H; it may then happen that some of the elements of E actually lie in H but there is insufficient information for this to be detected. This function is normally used in conjunction with the Todd-Coxeter algorithm when seeking some subgroup having index sufficiently small so that the Todd-Coxeter procedure completes. The conjugates are returned as a set of words.

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a finitely presented group G and a subgroup H of G, this function returns

(a) A set of elements T of G forming a right transversal for G over H; and

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and $g \in G$, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

These functions may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function SetGlobalTCParameters. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

Example H70E60

Consider the infinite dihedral group.

```
> G<a,b> := DihedralGroup(GrpFP, 0);
```

We define a subgroup H of index 10 in G...

```
> H := sub< G | a*b, a^10 >;
> Index(G, H);
10
```

... and construct a right transversal for H in G and the associate transversal map.

```
> RT, transmap := Transversal(G, H);
> RT;
{@ Id(G), a, a^-1, a^2, a^-2, a^3, a^-3, a^4, a^-4, a^5 @}
> transmap;
Mapping from: GrpFP: G to SetIndx: RT
```

From this a left transversal is easily obtained:

```
> LT := {@ x^-1 : x in RT @};
```

```
> LT;
{@ Id(G), a^-1, a, a^-2, a^2, a^-3, a^3, a^-4, a^4, a^-5 @}
```

We construct the coset table and check whether the enumeration of the cosets is compatible to the enumeration of the right transversal RT.

```
> ct := CosetTable(G, H);
> forall(culprit){ i : i in [1..Index(G, H)]
>                    | ct(1, RT[i]) eq i};
true
```

It is. Thus, we can very easily define a function $RT \times G \to RT$, describing the action of G on the set of right cosets of H in G.

```
> action := func< r, g | RT[ct(Index(RT, r), g)] >;
```

Note that the definition of the function `action` relies on the fact that the computed right transversal and its enumeration match the ones internally used for the coset table ct.

```
> action(Id(G), b);
a
```

I.e. $H * b = H * a$.

```
> action(a^-4, a*b);
a^4
```

I.e. $Ha^{-4} * (ab) = Ha^4$.

Example H70E61

Consider the group $G =< a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3 >$ and the subgroup H of G, generated by a^2 and $a^{-1}b$.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo< F | x^8, y^7, (x*y)^2, (x^-1*y)^3 >;
> H := sub< G | a^2,a^-1*b >;
```

We construct an indexed right coset space V and an explicit right coset space Vr for H in G.

```
> V := CosetSpace(G, H);
> Vr := RightCosetSpace(G, H);
```

The coset H always has index 1.

```
> trivial_coset := ExplicitCoset(Vr, 1);
> trivial_coset;
<GrpFP: H, Id(G)>
> IndexedCoset(V, trivial_coset);
1
```

We now pick a coset...

```
> coset := ExplicitCoset(Vr, 42);
> coset;
```

```
<GrpFP: H, a^-1 * b^-1 * a^3 * b^-1>
```

... multiply from the right with b...

```
> coset * b;
<GrpFP: H, a^-1 * b^-1 * a^3>
```

... and check where this gets us in the indexed coset space V.

```
> IndexedCoset(V, coset * b);
23
```

Example H70E62

We present a function which computes the derived subgroup G' for the finitely presented group G. It assumes that the Todd-Coxeter procedure can enumerate the coset space of G' in G.

```
> function DerSub(G)
>
> /* Initially define S to contain the commutator of each pair of distinct
>  generators of G */
>
>     S := { (x,y) : x, y in Generators(G) | (x,y) ne Id(G) };
>
> /* successively extend S until it is closed under conjugation by the
>   generators of G */
>
>         repeat
>             V := CosetSpace(G, sub<G | S>);
>             E := ExcludedConjugates(V);
>             S := S join E;
>         until # E eq 0;
>     return sub<G | S>;
> end function;
```

Example H70E63

Given a subgroup H of the finitely presented group G, for which the Todd-Coxeter procedure does not complete, add excluded conjugates one at a time to the generators of G until a subgroup K is reached such that either K is normal in G, or K has sufficiently small index for the Todd-Coxeter method to complete. The set *hgens* contains a set of generating words for H.

```
> function NormClosure(G, hgens)
> xgens := hgens;
> kgens := hgens;
> indx := 0;
> while # xgens ne 0 do
>       Include(~kgens, Representative(xgens));
>       V := CosetSpace(G, sub<G | kgens>);
>       if IsComplete(V) then break; end if;
```

```
>       xgens := ExcludedConjugates(V);
> end while;
> if IsComplete(V) then
>           print "The subgroup generated by", kgens, "has index", #V;
>           return kgens;
> else
>           print "The construction was unsuccessful";
>           return {};
> end if;
> end function;
```

70.8.5 Double Coset Spaces: Construction

`DoubleCoset(G, H, g, K )`

The double coset $H * g * K$ of the subgroups H and K of the group G, where g is an element of G.

`DoubleCosets(G, H, K)`

Set of double cosets $H * g * K$ of the group G.

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function SetGlobalTCParameters. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

Example H70E64

Consider again the infinite dihedral group G...

```
> G<a,b> := DihedralGroup(GrpFP, 0);
```

... and the subgroup H of index 10 in G.

```
> H := sub< G | a*b, a^10 >;
```

The set of H-H double cosets in G can be obtained with the statement

```
> DoubleCosets(G, H, H);
{ <GrpFP: H, Id(G), GrpFP: H>, <GrpFP: H, a^5, GrpFP: H>,
  <GrpFP: H, a^4, GrpFP: H>, <GrpFP: H, a^2, GrpFP: H>,
  <GrpFP: H, a, GrpFP: H>, <GrpFP: H, a^3, GrpFP: H> }
```

70.8.6 Coset Spaces: Selection of Cosets

`CosetsSatisfying(T, S: parameters)`

`CosetSatisfying(T, S: parameters)`

`CosetsSatisfying(V, S: parameters)`

`CosetSatisfying(V, S: parameters)`

Given a fp-group G, and a partial or complete coset space V or coset table T for G over the subgroup H generated by the set of words S, these functions return representatives for the cosets of V which satisfy the conditions defined in the parameters. In the description of the parameters below, the symbol l will denote a Boolean value, while the symbol n will denote a positive integer in the range $[1, \#V]$.

The functions are not identical. `CosetsSatisfying` returns a set of coset representatives for V as defined in the parameters. `CosetSatisfying` is the same as `CosetsSatisfying` with the `Limit` parameter equal to 1; thus it returns a set containing a single coset representative, or an empty set if no cosets satisfy the conditions.

`First` RngIntElt *Default :* 1

Start looking for coset representatives satisfying the designated conditions beginning with coset i of V.

`Last` RngIntElt *Default :* $\#V$

Stop looking for coset representatives after examining coset j of V.

`Limit` RngIntElt *Default :* ∞

Terminate the search for coset representatives as soon as l have been found which satisfy the designated conditions. This parameter is not available for `CosetSatisfying`, since `Limit` is set to 1 for this function.

`Normalizing` BoolElt *Default :* `false`

If `true`, select coset representatives x such that, modulo V, the word $x^{-1}h_1x$, ..., x_1h_sx is contained in H.

`Order` RngIntElt *Default :* 0

Select coset representatives x such that, modulo V, the word x^n is contained in H.

`Print` RngIntElt *Default :* 0

If $t > 0$, print the coset representatives found to satisfy the designated conditions.

Example H70E65

Consider the braid group G on 4 strings with Artin generators a, b and c and the subgroup H of G generated by a and b.

```
> G<a,b,c> := BraidGroup(GrpFP, 4);
```

```
> H := sub< G | a,b >;
```

We construct an – obviously incomplete – explicit right coset space for H in G.

```
> V := RightCosetSpace(G, H);
```

Using the function CosetSatisfying, we compute an element of G, not contained in H, which normalises H.

```
> cs := CosetSatisfying(V, Generators(Subgroup(V))
>                       : Normalizing := true, First := 2);
> cs;
{
    <GrpFP: H, c * b * a^2 * b * c>
}
> rep := c * b * a^2 * b * c;
```

The conjugates of a and b by this element had better be in H...

```
> rep^-1 * a * rep in H;
true
> rep^-1 * b * rep in H;
true
```

... OK.

70.8.7 Coset Spaces: Induced Homomorphism

`CosetAction(G, H)`

Given a subgroup H of the group G, this function constructs the permutation representation ϕ of G given by the action of G on the cosets of H. It returns:

(a) The homomorphism ϕ;

(b) The image group $\phi(G)$.

(c) (if possible) the kernel of ϕ.

The permutation representation is obtained by using the Todd-Coxeter procedure to construct the coset table for H in G. Note that G may be an infinite group: it is only necessary that the index of H in G be finite.

`CosetAction(V)`

Construct the permutation representation L of G given by the action of G on the coset space V. It returns the permutation representation ϕ (as a map) and its image.

`CosetImage(G, H)`

Construct the image of G given by its action on the (right) coset space of H in G.

`CosetImage(V)`

Construct the image of G as defined by its action on the coset space V.

`CosetKernel(G, H)`

The kernel of G in its action on the (right) coset space of H in G. (Only available when the index of H in G is very small).

This function may require the computation of a coset table. Experienced users can control the behaviour of a possibly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function SetGlobalTCParameters. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

`CosetKernel(V)`

The kernel of G in its action on the (right) coset space V. (Only available when the index of the subgroup H of G defining the coset space is very small).

Example H70E66

The first Conway group has a representation as the image of the group

$$\begin{aligned} G = < a,b,c,d,e,f,g,h \mid\ & a^2, b^2, c^2, d^2, e^2, f^2, g^2, h^2, \\ & (ab)^3, (ac)^2, (ad)^2, (ae)^4, (af)^2, (ag)^2, (ah)^3, \\ & (bc)^5, (bd)^2, (be)^2, (bf)^2, (bg)^4, (bh)^4, \\ & (cd)^3, (ce)^3, (cf)^4, (cg)^2, (ch)^2, \\ & (de)^2, (df)^3, (dg)^2, (dh)^2, \\ & (ef)^6, (eg)^2, (eh)^2, \\ & (fg)^4, (fh)^6, \\ & (gh)^2, \\ & a(cf)^2 = (adfh)^3 = b(ef)^3 = (baefg)^3 = 1, \\ & (cef)^7 = d(bh)^2 = d(aeh)^3 = e(bg)^2 = 1 > \end{aligned}$$

under the homomorphism defined by the action of G on the cosets of the subgroup

$$H = < a,b,c,d,e,f,g,(adefcefgh)^{39} > .$$

The permutation group can be constructed as follows:

```
> F<s, t, u, v, w, x, y, z> := FreeGroup(8);
> G<a, b, c, d, e, f, g, h> := quo<F | s^2, t^2, u^2, v^2, w^2, x^2, y^2, z^2,
> (s*t)^3, (s*u)^2, (s*v)^2, (s*w)^4, (s*x)^2, (s*y)^2, (s*z)^3,
> (t*u)^5, (t*v)^2, (t*w)^2, (t*x)^2, (t*y)^4, (t*z)^4,
> (u*v)^3, (u*w)^3, (u*x)^4, (u*y)^2, (u*z)^2,
> (v*w)^2, (v*x)^3, (v*y)^2, (v*z)^2,
```

```
> (w*x)^6, (w*y)^2, (w*z)^2,
> (x*y)^4, (x*z)^6,
> (y*z)^2,
> s*(u*x)^2 = (s*v*x*z)^3 = t*(w*x)^3 = (t*s*w*x*y)^3 = 1,
> (u*w*x)^7 = v*(t*z)^2 = v*(s*w*z)^3 = w*(t*y)^2 = 1>;
> H := sub< G | a, b, c, d, e, f, g, (a*d*e*f*c*e*f*g*h)^39 >;
> V := CosetSpace(G, H: FillFactor := 100000);
> Co1 := CosetImage(V);
> Degree(Co1);
    98280
```

Example H70E67

The group $G_2(3)$ is a homomorph of the fp-group G defined below. We construct a permutation representation $G1$ for $G_2(3)$ on 351 points, and then compute the subgroup generated by the images of the first four generators of G in $G1$. (The functions applied to permutation groups are described in Chapter 58.)

```
> F<a,b,c,d,y,x,w> := FreeGroup(7);
> z := y*c*a^2*b;
> u := x*d;
> t := w*c*a*d*b^2*c;
> G<a,b,c,d,y,x,w>, g :=
>            quo< F | a^4, b^4, c^2, d^2, (a,b), (a*c)^2, (b*c)^2,
>                       (c*d)^2, d*a*d*b^-1, y^3, (a^-1*b)^y*d*a^-1*b^-1,
>                       (c*d*a^-1*b)^y*b^-1*a*d*c, a*d*y*d*a^-1*y, x^3,
>                       a^x*b^-1, b^x*a*b, c^x*c, (x*d)^2, (u*z)^6, w^3,
>                       (w,y), (a*b)^w*b^-1*a*d*c, (c*d*a^-1*b)^w*d*c*b^2,
>                       (b^2*c*d)^w*a^-1*b^-1, (c*a^2*b*w)^2,
>                      (a^-1*b)^w*d*a^-1*b^-1, (t*u)^3 >;
> z1 := g(z);
> u1 := g(u);
> t1 := g(t);
> H := sub< G | z1*a^2*b^2, u1*c, t1*a^2*b^2 >;
> f, G1, K := CosetAction(G, H);
> Degree(G1);
351
> print Order(G1), FactoredOrder(G1);
4245696 [ <2, 6>, <3, 6>, <7, 1>, <13, 1> ]
> CompositionFactors(G1);
    G
    |  G(2, 3)
    1
> S := sub< G1 | f(a), f(b), f(c), f(d) >;
> BSGS(S);
> S;
Permutation group S of degree 351
```

```
Order = 64 = 2^6
```

Thus the images of a, b, c and d in G1 generate the Sylow 2-subgroup

---

70.9 Simplification

70.9.1 Reducing Generating Sets

Subgroups of finitely presented groups constructed in certain ways may be created with a generating set containing redundant generators. The most important case in which this situation may occur is a subgroup of the domain of a homomorphism f of groups, defined as the preimage under f of some given subgroup of the codomain of f. In this case, the generating set of the preimage will contain generators of the kernel of f and is likely to be not minimal.

Since reducing the generating set may be expensive and is not necessary in all situations, a reduction is not done automatically. Instead, MAGMA provides a function for reducing generating sets of finitely presented groups.

`ReduceGenerators(G)`

Given a finitely presented group G, attempt to construct a presentation H on fewer generators. H is returned as a subgroup of G (which of course is equal to G), so that element coerce is possible. The isomorphism from G to H is returned as second return value.

If a presentation for G is known, this function attempts to simplify this presentation (cf. section 70.9.2). Otherwise, it tries to rewrite G with respect to a suitable supergroup to obtain a presentation on fewer generators.

For a sample application of this function, see Example H70E73.

70.9.2 Tietze Transformations

Given a finitely presented group G, the user can attempt to simplify its presentation using Tietze transformations and substring searching. The choice of simplification strategy can either be left to MAGMA or selected by the user.

`Simplify(G: parameters)`

Given a finitely presented group G, attempt to simplify the presentation of G by repeatedly eliminating generators and subsequently shortening relators by substitution of substrings that correspond to the left or right hand side of a relation. The order in which transformations are applied is determined by a set of heuristics. The transformation process terminates when no more eliminations of generators and no more length reducing substring replacements are possible.

A new group K isomorphic to G is returned which is (hopefully) defined by a simpler presentation than G. K is returned as a subgroup of G. The isomorphism $f : G \to K$ is returned as second return value.

The simplification process can be controlled by a set of parameters described below.

Example H70E68

```
Consider the Fibonacci group $F(8)$.
> F<x1, x2, x3, x4, x5, x6, x7, x8> := FreeGroup(8);
> F8<x1, x2, x3, x4, x5, x6, x7, x8> :=
>     quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>              x5*x6=x7, x6*x7=x8, x7*x8=x1, x8*x1=x2>;
```

We use the function `Simplify` in order to obtain a presentation of $F(8)$ on two generators.

```
> H<[y]>, f := Simplify(F8);
> H;
Finitely presented group H on 2 generators
Generators as words in group F8
    y[1] = x3
    y[2] = x4
Relations
    y[2] * y[1]^-2 * y[2] * y[1]^-1 * y[2]^2 * y[1] * y[2]^2 *
       y[1]^-1 = Id(H)
    y[1] * y[2] * y[1] * y[2]^2 * y[1] * y[2] * y[1]^2 * y[2]^-1
       * y[1] = Id(H)
```

The isomorphism f can be used to express the old generators in terms of the new ones.

```
> f;
Mapping from: GrpFP: F8 to GrpFP: H
> f(x1);
y[1]^2 * y[2]^-1
```

The strategy employed by the function `Simplify` can be controlled using the following set of parameters.

`Preserve` [RngIntElt] *Default :* []

This parameter can be used to indicate that certain generators of G should not be eliminated (*default*: no restrictions). `Preserve` is assigned a sequence of integers in the range $[1, \ldots, n]$, where n is the number of generators of G, containing the numbers of those generators of G which should be preserved. See Example H70E70 for a sample application.

`Iterations` RngIntElt *Default :* 10000

This parameter sets the maximal number of iterations of the main elimination / simplification cycle which will be performed.

`EliminationLimit` RngIntElt *Default :* 100

This parameter sets the maximal number of generators which may be eliminated in each elimination phase.

`LengthLimit` RngIntElt *Default :* ∞

If `LengthLimit` is set to n, any eliminations which would make the total length of the relators grow beyond n will not be performed (*default*: no limit).

`ExpandLimit` RngIntElt *Default :* 150

If `ExpandLimit` is set to n, the total length of the relators is not permitted to grow by more than a factor of $n\%$ in any elimination phase (*default*: 150%). If this limit is reached, the elimination phase is aborted.

`GeneratorsLimit` RngIntElt *Default :* 0

Any eliminations which would reduce the number of generators below the value of `GeneratorsLimit` will not be performed (*default*: 0).

`SaveLimit` RngIntElt *Default :* 10

If `SaveLimit` is set to n, any simplification phase is repeated, if the reduction in the total length of the relators achieved during this phase exceeds $n\%$ (*default*: 10%).

`SearchSimultaneous` RngIntElt *Default :* 20

This parameter sets the number of relators processed simultaneously in a simplification phase.

`Print` RngIntElt *Default :* 0

This parameter controls the volume of printing. By default its value is that returned by `GetVerbose("Tietze")`, which is 0 unless it has been changed through use of `SetVerbose`.

`SimplifyLength(G: parameters)`

Given a finitely presented group G, attempt to eliminate generators and shorten relators by locating substrings that correspond to the left or right hand side of a relation. The order in which transformations are applied is determined by a set of heuristics. As opposed to the function `Simplify`, this function terminates the transformation process when the total length of the presentation starts to increase with the elimination of further generators.

A new group K isomorphic to G is returned which is (hopefully) defined by a simpler presentation than G. K is returned as a subgroup of G. The isomorphism $f : G \to K$ is returned as second return value. This function accepts the same set of parameters as the function `Simplify`.

`TietzeProcess(G: parameters)`

Create a Tietze process that takes the presentation for the fp-group G as its starting point. This process may now be manipulated by various procedures that will be described below.

`Preserve` [RngIntElt] *Default :* []

`Iterations` RngIntElt *Default :* 10000

`EliminationLimit`	RngIntElt	*Default :* 100
`LengthLimit`	RngIntElt	*Default :* ∞
`ExpandLimit`	RngIntElt	*Default :* 150
`GeneratorsLimit`	RngIntElt	*Default :* 0
`SaveLimit`	RngIntElt	*Default :* 10
`SearchSimultaneous`	RngIntElt	*Default :* 20
`Print`	RngIntElt	*Default :* 0

These parameters define the defaults used for the Tietze operations. Each of the various procedures described below allows some or all of these control parameters to be overridden.

For the meanings of the parameters, see the description under `Simplify` above.

`ShowOptions(~P : parameters)`

Display the defaults associated with the Tietze process P. The current status of all the control parameters may be viewed by using this function.

`SetOptions(~P : parameters)`

Change the defaults associated with the Tietze process P. All of the control parameters may be overridden permanently by using this function.

`Simplify(~P : parameters)`

`SimplifyPresentation(~P : parameters)`

Use the default strategy to simplify the presentation as much as possible. The transformation process is terminated when no more eliminations of generators and no more length reducing substring replacements are possible. All the control parameters may be overridden for this function.

`SimplifyLength(~P : parameters)`

Use the default strategy to simplify the presentation as much as possible. The transformation process is terminated when the total length of the presentation starts to increase with the elimination of further generators. All the control parameters may be overridden for this function.

`Eliminate(~P: parameters)`

`EliminateGenerators(~P: parameters)`

Eliminate generators in the presentation defined by the Tietze process P under the control of the parameters. First any relators of length one are used to eliminate trivial generators. Then, if there are any non-involutory relators of length two, the generator with the higher number is eliminated. Of the control parameters, only `EliminationLimit, ExpandLimit, GeneratorsLimit` and `LengthLimit` may be overridden by this function.

`Relator` RngIntElt *Default :* 0

If $n > 0$, try to eliminate a generator using the n-th relator. If no generator is specified by the parameter `Generator` below, then the generator which is eliminated will be the one occurring once in the relator that produced the smallest total relator length.

`Generator` RngIntElt *Default :* 0

If $n > 0$, try to eliminate the n-th generator. If no relation is specified by the parameter `Relator` above, then the shortest relator in which the n-th generator occurs exactly once (if any) is used.

`Search(~P: parameters)`

Simplifies the presentation by repeatedly searching for common substrings in pairs of relators where the length of the substring is greater than half the length of the shorter relator and making the corresponding transformations. Relators of length 1 or 2 are also used to generate simplifications. The control parameters `SaveLimit` and `SearchSimultaneous` can be overridden.

`SearchEqual(~P: parameters)`

Modifies the presentation by repeatedly searching for common substrings in pairs of relators where the length of the substring is exactly half the length of the shorter relator and making the corresponding transformations. The control parameter `SearchSimultaneous` can be overridden.

`Group(P)`

Extract the group G defined by the current presentation associated with the Tietze process P, together with the isomorphism between the original group and G. G is returned as a subgroup of the original group underlying P.

`NumberOfGenerators(P)`

`Ngens(P)`

The number of generators for the presentation currently stored by the Tietze process P.

`NumberOfRelations(P)`

`Nrels(P)`

The number of relations in the presentation currently stored by the Tietze process P.

`PresentationLength(P)`

The sum of the lengths of the relators in the presentation currently stored by the Tietze process P.

Example H70E69

The Fibonacci group $F(n)$ is generated by $\{x_1, \ldots, x_n\}$ with defining relations

$$x_i x_{i+1} = x_{i+2}, i \in \{1, \ldots, n\},$$

where the subscripts are taken modulo n. Consider the Fibonacci group $F(7)$, which is defined in terms of the presentation

$$< x_1, x_2, x_3, x_4, x_5, x_6, x_7 \mid x_1x_2 = x_3, x_2x_3 = x_4, x_3x_4 = x_5, \\ x_4x_5 = x_6, x_5x_6 = x_7, x_6x_7 = x_1, x_7x_1 = x_2 > .$$

The following code will produce a 2-generator, 2-relator presentation for $F(7)$:

```
> F<x1, x2, x3, x4, x5, x6, x7> := FreeGroup(7);
> F7<x1, x2, x3, x4, x5, x6, x7> :=
>    quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>             x5*x6=x7, x6*x7=x1, x7*x1=x2 >;
> P := TietzeProcess(F7);
> for i := 7 to 3 by -1 do
>    Eliminate(~P: Generator := i);
> end for;
> Search(~P);
> H<x, y>, f := Group(P);
> H;
Finitely presented group H on 2 generators
Generators as words in group F7
    x = x1
    y = x2
Relations
    x^-1 * y^-1 * x^-1 * y^-2 * x^-1 * y * x^-1 * y * x^-1 = Id(H)
    x * y^3 * x^-1 * y * x^-1 * y^2 * x * y * x * y^-1 = Id(H)
```

The resulting presentation is

$$< a, b \mid a^{-1}b^{-1}a^{-1}b^{-2}a^{-1}ba^{-1}ba^{-1}, ab^3a^{-1}ba^{-1}b^2abab^{-1} > .$$

We can use the isomorphism f returned by the function `Group` to express arbitrary words in the original generators of $F(7)$ in terms of the new generators x and y:

```
> f;
Mapping from: GrpFP: F7 to GrpFP: H
> f(x7);
x * y^2 * x * y^2 * x * y * x * y^2 * x * y
```

Alternatively, a similar effect may be obtained using the `Simplify` function:

```
> F<x1, x2, x3, x4, x5, x6, x7> := FreeGroup(7);
> F7<x1, x2, x3, x4, x5, x6, x7> :=
>    quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
```

```
>       x5*x6=x7, x6*x7=x1, x7*x1=x2>;
> H<x, y>, f := Simplify(F7: Iterations := 5);
> H;
Finitely presented group H on 2 generators
Generators as words in group F7
    x = x2
    y = x3
Relations
    x * y^-1 * x * y^2 * x * y * x^2 * y^-1 = Id(H)
    y * x * y^2 * x^-1 * y * x^-2 * y * x^-2 = Id(H)
```

Again, we can use the isomorphism f returned by the function `Simplify` to express arbitrary words in the original generators of $F(7)$ in terms of the new generators x and y:

```
> f;
Mapping from: GrpFP: F7 to GrpFP: H
> f(x7);
y * x * y * x * y^2 * x * y
```

Example H70E70

In a situation where some proper subset S of the original generating set of a finitely group G is sufficient to generate G, the function `Simplify` can also be used to rewrite words in the original generators in terms of the elements of S. Consider again one of the Fibonacci groups, say $F(8)$.

```
> F<x1, x2, x3, x4, x5, x6, x7, x8> := FreeGroup(8);
> F8<x1, x2, x3, x4, x5, x6, x7, x8> :=
>     quo< F | x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>        x5*x6=x7, x6*x7=x8, x7*x8=x1, x8*x1=x2>;
```

Obviously, $F(8)$ is generated by x_1 and x_2. We utilise the function `Simplify` to obtain a presentation H of $F(8)$ on x_1 and x_2, using the parameter `Preserve` to indicate that x_1 and x_2 – i.e. the first and the second generator – are to be retained in the new presentation. We also compute the isomorphism $f : F(8) \to H$.

```
> H<x, y>, f := Simplify(F8: Preserve := [1,2]);
```

Mapping elements of $F(8)$ to H using the map f basically means to rewrite these elements in terms of the generators $x = x_1$ and $y = x_2$. Since H is returned as a subgroup of $F(8)$, the resulting words can be coerced from H back into $F(8)$, yielding words explicitly in x_1 and x_2.

```
> F8 ! f(x5*x6);
x1 * x2^2 * x1 * x2 * x1^2 * x2^-1 * x1 * x2^-1
```

Example H70E71

The finiteness of the last of the Fibonacci groups, $F(9)$, was settled in 1988 by M.F. Newman using the following result:

Theorem. *Let G be a group with a finite presentation on b generators and r relations, and suppose p is an odd prime. Let d denote the rank of the elementary abelian group $G_1 = [G, G]G^p$ and let e denote the rank of $G_2 = [G_1, G]G^p$. If*

$$r - b < d^2/2 - d/2 - d - e$$

or

$$r - b \leq d^2/2 - d/2 - d - e + (e + d/2 - d^2/4)d/2,$$

then G has arbitrary large quotients of p-power order.

We present a proof that $F(9)$ is infinite using this result.

```
> Left := func< b, r | r - b >;
> Right := func< d, e | d^2 div 2 - d div 2 - d - e +
>                       (e + d div 2 - d^2 div 4)*(d div 2) >;
>
>
> F< x1,x2,x3,x4,x5,x6,x7,x8,x9 > :=
>  ΨGroup< x1, x2, x3, x4, x5, x6, x7, x8, x9 |
>              x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6, x5*x6=x7,
>              x6*x7=x8, x7*x8=x9, x8*x9=x1, x9*x1=x2 >;
>
> F;
Finitely presented group F on 9 generators
Relations
       x1 * x2 = x3
       x2 * x3 = x4
       x3 * x4 = x5
       x4 * x5 = x6
       x5 * x6 = x7
       x6 * x7 = x8
       x7 * x8 = x9
       x8 * x9 = x1
       x9 * x1 = x2
> AbelianQuotientInvariants(F);
[ 2, 38 ]
```

Thus the nilpotent quotient of F is divisible by 2 and 19. We examine the 2- and 19-quotients of F.

```
> Q1 := pQuotient(F, 2, 0: Print := 1);
Class limit set to 127.
Lower exponent-2 central series for F
Group: F to lower exponent-2 central class 1 has order 2^2
Group: F to lower exponent-2 central class 2 has order 2^3
Group completed. Lower exponent-2 central class = 2, Order = 2^3
> Q2 := pQuotient(F, 19, 0: Print := 1);
Class limit set to 127.
Lower exponent-19 central series for F
```

```
Group: F to lower exponent-19 central class 1 has order 19^1
Group completed. Lower exponent-19 central class = 1, Order = 19^1
```

Thus, the nilpotent residual of F has index 152. We try to locate this subgroup. We first take a 2-generator presentation for F.

```
> G := Simplify(F);
> G;
Finitely presented group G on 2 generators
Generators as words in group F
   G.1 = x4
   G.2 = x5
Relations
   G.2 * G.1 * G.2 * G.1 * G.2^2 * G.1 * G.2^2 * G.1^-1 * G.2 * G.1^-2 * G.2 *
      G.1^-2 = Id(G)
   G.1 * G.2^2 * G.1 * G.2 * G.1^2 * G.2^-1 * G.1^2 * G.2^-1 * G.1 * G.2^-1 *
      G.1^2 * G.2^-1 * G.1 * G.2^-1 = Id(G)
> H := ncl< G | (G.1, G.2) >;
> H;
Finitely presented group H
Index in group G is 76 = 2^2 * 19
Subgroup of group G defined by coset table
```

We haven't got the full nilpotent residual yet, so we try again.

```
> H := ncl< G | (G.1*G.1, G.2) >;
> H;
Finitely presented group H
Index in group G is 152 = 2^3 * 19
Subgroup of group G defined by coset table
```

Now, we have it.

```
> AbelianQuotientInvariants(H);
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
```

The nilpotent residual H has a 5-quotient. We construct a presentation for H and then calculate d and e for its 5-quotient.

```
> K := Rewrite(G, H: Simplify := false);
> KP := pQuotientProcess(K, 5, 1);
> d := FactoredOrder(ExtractGroup(KP))[1][2];
> NextClass(~KP);
> e := FactoredOrder(ExtractGroup(KP))[1][2] - d;
> "D = ", d, "e = ", e;
d =  18 e =  81
> "Right hand side = ", Right(d, e);
Right hand side =  135
> "Left hand side = ", Left(Ngens(K), #Relations(K));
```

```
Left hand side =  151
```

Since `Left` is greater than `Right`, this presentation for H doesn't work. Thus we start eliminating generators.

```
> K := Simplify(K: Iterations := 1);
> "Left hand side = ", Left(Ngens(K), #Relations(K));
Left hand side =  136
> K := Simplify(K: Iterations := 1);
> "Left hand side = ", Left(Ngens(K), #Relations(K));
Left hand side =  123
```

Got it! By Newman's theorem, H has an infinite 5-quotient and so F must be infinite.

Example H70E72

In this example, we consider a – quite unpleasant – presentation for some group G. In fact, it is a presentation for the group $\mathrm{PSL}_3(7)$:2, but we assume that we do not know this and want to compute the order of the finitely presented group G using coset enumeration.
We note in passing that the strategy outlined in this example is, together with other approaches, applied by the function `Order`.

```
> F<a,b,c> := FreeGroup(3);
> rels := [ a^4, b^2, (a,b)^3, (a*b)^8, ((a*b)^2*(a^-1*b)^3)^2,
>  c^2, (c*a*c*a^2)^2, (c*a)^3*(c*a^-1)^3,
>  c*a*b*c*a^-1*b*a^-1*c*a*b*c*a^2*a*b*a^-1,
>  c*a*b*c*b*a*c*a*c*a^-1*b*c*b*a^-1*c*a^-1,
>  c*a*b*a^-1*c*a*b*a^-1*c*a*b*a^-1*c*a*b*a^-1,
>  c*b*a^2*b*c*b*c*a^2*c*b*c*b*a^2*b,
>  c*a^2*c*b*a*c*b*a*c*b*a*c*a^-1*c*a*b*a^-1,
>  c*a^-1*b*a*c*a^-1*b*a*c*b*a*b*a^2*b*a^-1*b,
>  c*a*b*a^-1*b*c*b*a^-1*b*c*a^-1*b*a*b*a*c*b*c*b,
>  c*a*c*b*a*b*c*a*c*b*a*b*c*a*c*b*a*b,
>  c*b*a^-1*b*c*a^-1*c*a^-1*b*a*b*c*b*c*a^2*b*a*b*a^-1,
>  c*b*a^-1*b*a*b*c*b*a^-1*b*a*b*c*b*a^-1*b*a*b,
>  c*a^2*b*a^-1*b*c*b*c*b*a^-1*b*a*c*b*a^2*b*a^-1*b
> ];
> G<a,b,c> := quo< F | rels >;
```

As it happens, trying to determine the order of G by enumerating the cosets of the trivial subgroup is quite hard. – Even the predefined enumeration strategy `"Hard"` (cf. `ToddCoxeter` and `CosetEnumerationProcess`) does not give a finite result.

```
>  time ToddCoxeter(G, sub<G|> : Strategy := "Hard");
0
Time: 199.620
```

Of course we could try to increase the workspace for the coset enumeration, but we decide to be more clever. Trying random words in the generators of G, we easily find some subgroup S of G with pretty small index in G.

```
> S := sub< G | b, c*a*c*b*a*b >;
```

```
> time Index(G, S);
114
Time: 0.120
```

We now have to compute the order of S. In order to be able to do this using coset enumeration, we have to construct a presentation for S by rewriting S w.r.t. G.

```
> time R := Rewrite(G, S : Simplify := false);
Time: 0.030
```

However, the presentation obtained by Reidemeister-Schreier rewriting without any simplification is not suitable for coset enumeration: it contains too many generators and its total length is quite high.

```
> NumberOfGenerators(R);
133
> PresentationLength(R);
14384
```

An enumeration with the predefined enumeration strategy `"Hard"` does not produce a finite result. (Note that in consideration of the high total relator length, we select a coset table based enumeration style; cf. `CosetEnumerationProcess` in Chapter 71.)

```
> time ToddCoxeter(R, sub<R|> : Strategy := "Hard", Style := "C");
0
Time: 4.330
```

On the other hand, simplifying the presentation by reducing the number of generators as much as possible is not a good idea either, since the total relator length grows massively.

```
> time Rs := Simplify(R);
Time: 43.900
> NumberOfGenerators(Rs);
3
> PresentationLength(Rs);
797701
```

Again, an enumeration with the predefined enumeration strategy `"Hard"` does not produce a finite result.

```
>  time ToddCoxeter(Rs, sub<Rs|> : Strategy := "Hard", Style := "C");
0
Time: 22015.849
```

The best strategy is, to simplify the presentation obtained from the Reidemeister-Schreier procedure by eliminating generators until the total length of the relators starts to grow.

```
> time Rsl := SimplifyLength(R);
Time: 0.330
> NumberOfGenerators(Rsl);
48
> PresentationLength(Rsl);
```

```
7152
```

A coset enumeration for this presentation produces a finite result in a reasonable amount of time.

```
> time ToddCoxeter(Rsl, sub<Rsl|> : Strategy := "Hard", Style := "C");
32928
Time: 289.410
```

This finally proves that G is finite and has order $32928 * 114 = 3753792$.

70.10 Representation Theory

This section describes some functions for creating $R[G]$-modules for a finitely presented group G, which are unique for this category or have special properties when called for fp-groups. For a complete description of the functions available for creating and working with $R[G]$-modules we refer to chapter 89.

Note that the function GModuleAction can be used to extract the matrix representation associated to an $R[G]$-module.

All operations described in this subsection may require a closed coset table for at least one subgroup of an fp-group. If a closed coset table is needed and has not been computed, a coset enumeration will be invoked. If the coset enumeration does not produce a closed coset table, a runtime error is reported.

Experienced users can control the behaviour of such indirectly invoked coset enumeration with a set of global parameters. These global parameters can be changed using the function SetGlobalTCParameters. For a detailed description of the available parameters and their meanings, we refer to Chapter 71.

GModulePrimes(G, A)

Let G be a finitely presented group and A a normal subgroup of G of finite index. Given any prime p, the maximal p-elementary abelian quotient of A can be viewed as a $\mathbf{F}_p[G]$-module M_p. This function determines all primes p such that M_p is not trivial (i.e. zero-dimensional) and the dimensions of the corresponding modules M_p. The return value is a multiset S. If $0 \notin S$, the maximal abelian quotient of A is finite and the multiplicity of p is the dimension of M_p. If S contains 0 with multiplicity m, the maximal abelian quotient of A contains m copies of $\mathbf{Z}$. In this case, M_p is non-trivial for every prime p. The rank of M_p in this case is the sum of m and the multiplicity of p in S.

`GModulePrimes(G, A, B)`

Let G be a finitely presented group, A a normal subgroup of finite index in G and B a normal subgroup of G contained in A. Given any prime p, the maximal p-elementary abelian quotient of A/B can be viewed as a $\mathbf{F}_p[G]$-module M_p. This function determines all primes p such that M_p is not trivial (i.e. zero-dimensional) and the dimensions of the corresponding modules M_p. The return value is a multiset S. If $0 \notin S$, the maximal abelian quotient of A/B is finite and the multiplicity of p is the dimension of M_p. If S contains 0 with multiplicity m, the maximal abelian quotient of A/B contains m copies of $\mathbf{Z}$. In this case, M_p is non-trivial for every prime p. The rank of M_p in this case is the sum of m and the multiplicity of p in S.

`GModule(G, A, p)`

Given a finitely presented group G, a normal subgroup A of finite index in G and a prime p, create the $\mathbf{F}_p[G]$-module M corresponding to the conjugation action of G on the maximal p-elementary abelian quotient of A. The function also returns the epimorphism $\pi : A \to M$.

Note that normality of A in G is not checked. The results for invalid input data are undefined.

`GModule(G, A, B, p)`

`GModule(G, A, B)`

Given a finitely presented group G, a normal subgroup A of G of finite index, a normal subgroup B of G contained in A and a prime p, create the $\mathbf{F}_p[G]$-module M corresponding to the conjugation action of G on the maximal p-elementary abelian quotient of A/B.

p can be omitted, if the maximal elementary abelian quotient of A/B is a p-group for some prime p. Note, however, that the computation is much faster, if a prime is specified.

The function also returns the epimorphism $\pi : A \to M$.

Note that normality of A and B in G is not checked. The results for invalid input data are undefined.

`Pullback(f, N)`

Given a map $f : A \to M$ from a normal subgroup A of an fp-group G onto a $\mathbf{F}_p[G]$-module M and a submodule N of M, try to compute the preimage of N under f using a fast pullback method. If successful, the preimage is returned as subgroup of A.

If the pullback works, it is in general faster than a direct computation of the preimage using the preimage operator and it produces a more concise generating set for the preimage; see the following example. In cases where the pullback fails, a runtime error is reported and a preimage construction should be used instead.

Example H70E73

Consider the group G defined by the presentation

$$\langle a, b, c, d, e \mid a^4, b^{42}, c^6, e^3, b^a = b^{-1}, [a, c], [a, d], [a, e], \\ c^b = ce, d^b = d^{-1}, e^b = e^2, d^c = de, e^c = e^2, [d, e] \rangle.$$

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< F | a^4, b^42, c^6, e^3,
>                         b^a=b^-1, (a,c), (a,d), (a,e),
>                         c^b=c*e, d^b=d^-1, e^b=e^2,
>                         d^c=d*e, e^c=e^2,
>                         (d,e) >;
```

The finite index subgroup H of G generated by c, d, e is normal in G.

```
> H := sub< G | c,d,e >;
> Index(G, H);
168
> IsNormal(G, H);
true
```

We check, for which characteristics the action of G on H yields non-trivial modules.

```
> GModulePrimes(G, H);
{* 0, 2, 3 *}
```

We construct the $\mathbf{F}_3[G]$-module M given by the action of G on the maximal 3-elementary abelian quotient of H and the natural epimorphism π from H onto the additive group of M.

```
> M, pi := GModule(G, H, 3);
> M;
GModule M of dimension 2 over GF(3)
```

Using the function Submodules, we obtain the submodules of M. Their preimages under π are precisely the normal subgroups of G which are contained in H and contain $\ker(\pi)$.

```
> submod := Submodules(M);
> time nsgs := [ m @@ pi : m in submod ];
Time: 11.640
> [ Index(G, s) : s in nsgs ];
[ 1512, 504, 504, 168 ]
```

The generating sets for the normal subgroups obtained in this way, contain in general many redundant generators. (E.g. each will contain a generating set for $\ker(\pi)$.)

```
> [ NumberOfGenerators(s) : s in nsgs ];
[ 19, 20, 20, 21 ]
```

Optimised generating sets can be obtained using the function ReduceGenerators.

```
> nsgs_red := [ ReduceGenerators(s) : s in nsgs ];
> [ NumberOfGenerators(s) : s in nsgs_red ];
```

```
[ 2, 2, 3, 2 ]
```

Alternatively, and in fact this is the recommended way, we can use the function Pullback to compute the preimages of the submodules under π. Note that the generating sets for the preimages computed this way contain fewer redundant generators.

```
> time nsgs := [ Pullback(pi, m) : m in submod ];
Time: 8.560
> [ Index(G, s) : s in nsgs ];
[ 1512, 504, 504, 168 ]
> [ NumberOfGenerators(s) : s in nsgs ];
[ 4, 4, 3, 2 ]
```

Example H70E74

Consider the group defined by the presentation

$$< a, b, c, d, e \mid a^5, b^5, c^6, d^5, e^3, b^a = bd, \\ (a,c), (a,d), (a,e), (b,c), (b,d), (b,e), (c,d), (c,e), (d,e) > .$$

```
> G<a,b,c,d,e> := Group< a,b,c,d,e |
>                            a^5, b^5, c^6, d^5, e^3, b^a = b*d,
>                            (a,c), (a,d), (a,e), (b,c), (b,d), (b,e),
>                            (c,d), (c,e), (d,e) >;
```

Obviously the subgroup of G generated by b, c, d, e is normal in G.

```
> H := sub< G | b,c,d,e >;
> IsNormal(G, H);
true
```

We use the function GModulePrimes to determine the set of primes p for which the action of G on the maximal p-elementary abelian quotient of H induces a nontrivial $\mathbf{F}_p[G]$-module.

```
> P := GModulePrimes(G, H);
> 0 in P;
false
```

0 is not contained in P, i.e. the maximal free abelian quotient of H is trivial. Hence, there are only finitely many primes, satisfying the condition above.

We loop over the distinct elements of P and for each element p we construct the induced $\mathbf{F}_p[G]$-module, print its dimension and check whether it is decomposable. Note that the dimension of the module for p must be equal to the multiplicity of p in P.

```
> for p in MultisetToSet(P) do
>     M := GModule(G, H, p);
>     dim := Dimension(M);
>     decomp := IsDecomposable(M);
>
>     assert dim eq Multiplicity(P, p);
```

```
>
>    print "prime", p, ": module of dimension", dim;
>    if decomp then
>       print "  has a nontrivial decomposition";
>     else
>       print "  is indecomposable";
>    end if;
> end for;
prime 2 : module of dimension 1
  is indecomposable
prime 3 : module of dimension 2
  has a nontrivial decomposition
prime 5 : module of dimension 2
  is indecomposable
```

70.11 Small Group Identification

This section describes some special issues arising with the identification of a finitely presented group using the database of small groups described in Section 66.2.

`IdentifyGroup(G)`

Locate the pair of integers $\langle o, n \rangle$ so that `SmallGroup(o, n)` is isomorphic to G. If the construction of a permutation representation for G fails or if there is no group isomorphic to G in the database, then an error will result.

When trying to look up a finitely presented group G in the database of small groups, MAGMA tries to construct a permutation representation of G by enumerating the cosets of the trivial subgroup in G. Assuming that a group isomorphic to G is contained in the database, the resulting coset table will be fairly small. Hence for performance reasons, a coset limit of $100 \cdot o$ is imposed, where o is the maximal order of groups in the database, unless the order of G is *known* to be less or equal to o. If, on the other hand, $|G| \leq o$ is known, the global set of parameters for implicitly invoked coset enumerations applies. This set of parameters can be changed using the function SetGlobalTCParameters.

If the coset enumeration for G fails with the coset limit $100 \cdot o$, this can be seen as a reasonable indication that G is probably too large to be contained in the database of small groups.

To deal with cases where the coset enumeration fails although G is known or suspected to be small enough, it is recommended to attempt to compute the order of G using the function Order before the actual group identification. If G that way can be shown to be small enough to be contained in the database, the function SetGlobalTCParameters can be used to control the behaviour of coset enumerations in a subsequent call to `IdentifyGroup`. The following example illustrates this.

Example H70E75

Consider the group defined by the presentation

$$\langle\, a, b \,|\, (ba^{-1})^3,\ (ba^{-1}b)^2,\ a^{12}b^2a^7b^2ab^2 \,\rangle.$$

```
> G := Group<a,b | (b*a^-1)^3, (b*a^-1*b)^2,
>                  a^12*b^2*a^7*b^2*a*b^2 >;
```

We suspect (for some reason) that G is a small group and want to identify its isomorphism type in the database of small groups.

```
>  IdentifyGroup(G);
IdentifyGroup(
    G: GrpFP: G
)
In file "/home/magma/package/Group/Grp/smallgps2.m", line 220,
column 25:
>>     res := IdentifyGroup(db, G);
                           ^
Runtime error in 'IdentifyGroup': Coset enumeration failed; group
may be too large (see handbook entry for details)
```

The group couldn't be identified, because it was not possible to obtain a permutation representation with the default value of the coset limit.
Since we still think the group should be small, we try to prove this using the function Order.

```
> Order(G : Print := true);
INDEX = 6 (a=6 r=66709 h=999999 n=999999; l=1247 c=1.56; m=969169
t=999998)
6
```

We were correct; the group is in fact very small. However, we can see from the output of the `Order` command that this wasn't that easy to find out: almost one million cosets were used in the coset enumeration.
Since the order of G is now known to the system, a subsequent call to `IdentifyGroup` will now use the global parameters for implicitly invoked coset enumerations. To be on the safe side, we tell the system to be prepared to work a little harder using the function `SetGlobalTCParameters` and try again to identify G.

```
> SetGlobalTCParameters( : Strategy := "Hard");
>
> IdentifyGroup(G);
<6, 1>
```

Now G can be identified.

70.11.1 Concrete Representations of Small Groups

When an finitely-presented group is known to small, it may be useful to write it concretely as a permutation or PC-group. There are two utilities provided for this purpose.

`PermutationGroup(G)`

Construct a faithful permutation representation of G, and the isomorphism from G to the representation. The algorithm first computes the order of G, then takes the regular representation of G, and reduces the degree. Thus the command is restricted to small groups.

`PCGroup(G)`

Construct a faithful PC-group representation of G, and the isomorphism from G to the representation. The algorithm first computes the order of G, then computes the soluble quotient of G with the order found. Thus the command is restricted to small soluble groups.

70.12 Bibliography

[**AR84**] D. G. Arrell and E. F. Robertson. A modified Todd-Coxeter algorithm. In *Computational group theory (Durham, 1982)*, pages 27–32. Academic Press, London, 1984.

[**CDHW73**] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463–490, 1973.

[**CHN11**] M. Conder, G. Havas, and M. Newman. On one-relator quotients of the modular group. In *Proc. Groups St Andrews 2009 in Bath*, number 387 in London Mathematical Society Lecture Note Series, pages 183–197. Cambridge University Press, 2011.

[**COS08**] A. Cavicchioli, E. O'Brien, and F. Spaggiari. On some questions about a family of cyclically presented groups. *J. Algebra*, 320(11):4063–4072, 2008.

[**Fab09**] Anna Fabianska. *Algorithmic analysis of presentations of groups and modules*. Dissertation, RWTH Aachen University, 2009.

[**Hav91**] G. Havas. Coset enumeration strategies. In *ISSAC'91*, pages 191–199. ACM Press, 1991.

[**HH10**] G. Havas and D.F. Holt. On Coxeter's families of group presentations. *J. Algebra*, 324(5):1076–1082, 2010.

[**HKRR84**] George Havas, P. E. Kenne, J. S. Richardson, and E. F. Robertson. A Tietze transformation program. In *Computational group theory (Durham, 1982)*, pages 69–73. Academic Press, London, 1984.

[**MKS76**] Wilhelm Magnus, Abraham Karrass, and Donald Solitar. *Combinatorial group theory*. Dover Publications Inc., New York, revised edition, 1976. Presentations of groups in terms of generators and relations.

[**Nic96**] Werner Nickel. Computing nilpotent quotients of finitely presented groups. In *Geometric and computational perspectives on infinite groups (Minneapolis, MN and New Brunswick, NJ, 1994)*, pages 175–191. Amer. Math. Soc., Providence, RI, 1996.

[**NO96**] M. F. Newman and E. A. O'Brien. Application of computers to questions like those of Burnside. II. *Internat. J. Algebra Comput.*, 6(5):593–605, 1996.

[**PF09**] W. Plesken and A. Fabianska. An L2-quotient algorithm for finitely presented groups. *J. Algebra*, 322(3):914–935, 2009.

[**Ram**] Colin Ramsay. ACE. URL:http://www.csee.uq.edu.au/˜cram/.

[**Sim94**] Charles C. Sims. *Computation with finitely presented groups.* Cambridge University Press, Cambridge, 1994.

71 FINITELY PRESENTED GROUPS: ADVANCED

Chapter 71
FINITELY PRESENTED GROUPS: ADVANCED

71.1 Introduction

This section presents some more advanced techniques available for computing with finitely-presented groups (fp-groups for short) within MAGMA. The features considered here are regarded as more advanced, either because they are technically or theoretically more complex and they are therefore expected to be used mainly by specialists, or because their efficient (and in a few cases even their merely correct) use requires some more detailed knowledge on the user's part.

Trying to summarise the expected main purpose of the functions described in this section, one could think of two main situations: On the one hand, user written functions, which may benefit from low-level tools for manipulating presentations or words, or which make use of interruptible process versions of some standard MAGMA functions for fp-groups. On the other hand, the solution of very hard problems, which require careful fine-tuning of the strategy employed or for which some iterative approach, using feedback of information obtained during the computation, is necessary.

The following topics are discussed in detail. First, some rather low-level operations on presentations and elements of fp-groups (words) are described. Then, the features for interactive coset enumeration in MAGMA are presented. This section also contains the complete description of all the parameters available for controlling the execution of the Todd-Coxeter procedure, which also applies to the appropriate standard functions documented in Chapter 70. After that, we describe the process version of the p-quotient algorithm. Note that some care has to be taken when interpreting results obtained with this interactive p-quotient computation; incorrect use of the existing functions may result in incomplete or wrong answers. The chapter ends with a treatise of the soluble quotient algorithm available in MAGMA. This final section contains a brief review of the theory underlying the soluble quotient algorithm, a description of the parameters available for functions computing soluble quotients, and the documentation of the interactive soluble quotient facilities.

71.2 Low Level Operations on Presentations and Words

In this section, we describe some rather low level operations on presentations and on elements of fp-groups. The main purpose of the functions described here, is to provide some efficient machinery for manipulating presentations and elements of fp-groups for user written functions.

71.2.1 Modifying Presentations

The functions described in this section construct a new fp-group from an existing one by adding or deleting a generator or by adding, deleting or changing a relation. The new group is created without any relationship to the existing group.

`AddGenerator(G)`

Given an fp-group G with presentation $\langle X \mid R \rangle$, create a new fp-group with presentation $\langle X \cup \{z\} \mid R \rangle$, where z is a symbol not in X.

`AddGenerator(G, w)`

Given an fp-group G with presentation $\langle X \mid R \rangle$, and given also a word w in the generators X, create a new fp-group having the presentation $\langle X \cup \{z\} \mid R \cup \{z = w\} \rangle$, where z is a symbol not in X.

`AddRelation(G, r)`

Given an fp-group G, and a relation r on the generators of G, create a new fp-group whose presentation consists of the relations of G together with the relation r.

`AddRelation(G, g)`

Given an fp-group G, and an element g of G, create a new fp-group whose presentation consists of the relations of G together with the relation $g = \mathrm{Id}(G)$.

`AddRelation(G, r, i)`

Given an fp-group G, and a relation r on the generators of G, create a new fp-group which has as its presentation the relations of G together with the relation r inserted after the i-th existing relation of G.

`AddRelation(G, g, i)`

Given an fp-group G, and an element g of G, create a new fp-group which has as its presentation the relations of G together with the relation $g = \mathrm{Id}(G)$ inserted after the i-th existing relation of G.

`DeleteGenerator(G, x)`

Given an fp-group G with presentation $\langle X \mid R \rangle$, and given also an element z in X, create a new fp-group with presentation $\langle X \setminus \{z\} \mid R' \rangle$, where the relations R' are obtained from R by deleting all relations containing an occurrence of z.

`DeleteRelation(G, r)`

Given an fp-group G, which includes the relation r amongst its relations, create a new fp-group which has as its presentation the relations of G with relation r omitted.

`DeleteRelation(G, g)`

Given an fp-group G, which includes the relation $g = \mathrm{Id}(G)$ amongst its relations, create a new fp-group which has as its presentation the relations of G with this relation omitted.

`DeleteRelation(G, i)`

Given an fp-group G, create a new fp-group which has as its presentation the relations for G with the i-th relation deleted.

`ReplaceRelation(G, s, r)`

`ReplaceRelation(G, h, r)`

`ReplaceRelation(G, s, g)`

`ReplaceRelation(G, h, g)`

Given an fp-group G, which includes the relation s or $h = \mathrm{Id}(G)$ amongst its relations, create a new fp-group which has as its presentation the relations for G with the relation s replaced by the relation r or $g = \mathrm{Id}(G)$.

`ReplaceRelation(G, i, r)`

Given an fp-group G and a relation r in the generators of G, create a new fp-group which has as its presentation the relations for G with relation number i replaced by the relation r.

`ReplaceRelation(G, i, g)`

Given an fp-group G and an element g of G, create a new fp-group which has as its presentation the relations for G with relation number i replaced by the relation $g = \mathrm{Id}(G)$.

Example H71E1

We use the function `ReplaceRelation` to vary a particular relation in a presentation. The order of the resulting group together with the index of a particular subgroup is determined.

```
> G<x,y,z,h,k,a> := Group< x, y, z, h, k, a |
>    x^2, y^2, z^2, (x,y), (y,z), (x,z), h^3, k^3, (h,k),
>    (x,k), (y,k), (z,k), x^h*y, y^h*z, z^h*x, a^2, a*x*a*y,
>    a*y*a*x, (a,z), (a,k), (a*h)^2 >;
> for i := 0 to 1 do
>     for j := 0 to 1 do
>         for k := 0 to 1 do
>              for l := 0 to 2 do
>                 rel := G.1^i*G.2^j*G.3^k*G.5^l*(G.6*G.4)^2 = Id(G);
>                 K := ReplaceRelation(G, 21, rel);
>                 print Order(K), Index(K, sub< K | K.6, K.4>);
>             end for;
>         end for;
```

```
>     end for;
> end for;
<0, 0, 0, 0> 144 24
<0, 0, 0, 1> 144 8
<0, 0, 0, 2> 144 8
<0, 0, 1, 0> 18 3
<0, 0, 1, 1> 18 1
<0, 0, 1, 2> 18 1
<0, 1, 0, 0> 72 3
<0, 1, 0, 1> 72 1
<0, 1, 0, 2> 72 1
<0, 1, 1, 0> 36 6
<0, 1, 1, 1> 36 2
<0, 1, 1, 2> 36 2
<1, 0, 0, 0> 18 3
<1, 0, 0, 1> 18 1
<1, 0, 0, 2> 18 1
<1, 0, 1, 0> 144 6
<1, 0, 1, 1> 144 2
<1, 0, 1, 2> 144 2
<1, 1, 0, 0> 36 6
<1, 1, 0, 1> 36 2
<1, 1, 0, 2> 36 2
<1, 1, 1, 0> 72 12
<1, 1, 1, 1> 72 4
<1, 1, 1, 2> 72 4
```

71.2.2 Low Level Operations on Words

The functions described in this section perform low level string operations like substitution, elimination or substring matching on elements of fp-groups.

`Eliminate(u, x, v)`

Given words u and v, and a generator x, all belonging to a group G, return the word obtained from u by replacing each occurrence of x by v and each occurrence of x^{-1} by v^{-1}.

`Eliminate(U, x, v)`

Given a set of words U, a word v, and a generator x, all belonging to a group G, return the set of words obtained by taking each element u of U in turn, and replacing each occurrence of x in u by v and each occurrence of x^{-1} by v^{-1}.

`Match(u, v, f)`

Suppose u and v are words belonging to the same group G, and that f is an integer such that $1 \leq f \leq \#u$. The function seeks the least integer l such that:

(a) $l \geq f$; and

(b) v appears as a subword of u, starting at the l-th letter of u.

If such an integer l is found `Match` returns the value true and l. If no such l is found, `Match` returns the value false.

`RotateWord(u, n)`

The word obtained by cyclically permuting the word u by n places. If n is positive, the rotation is from left to right, while if n is negative the rotation is from right to left. In the case where n is zero, the function returns u.

`Substitute(u, f, n, v)`

Given words u and v belonging to a group G, and non-negative integers f and n, this function replaces the substring of u of length n, starting at position f, by the word v. Thus, if $u = x_{i_1}^{e_1} \cdots x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}} \cdots x_{i_m}^{e_m}$ then the substring $x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}}$ is replaced by v. If the function is invoked with $v = \mathrm{Id(G)}$, then the substring $x_{i_f}^{e_f} \cdots x_{i_{f+n-1}}^{e_{f+n-1}}$ of u is deleted.

`Subword(u, f, n)`

The subword of the word u comprising the n consecutive letters commencing at the f-th letter of u.

Example H71E2

We demonstrate some of these operations in the context of the free group on generators x, y, and z.

```
> F<x, y, z> := FreeGroup(3);
> u := (x, y*z);
> w := u^(x^2*y);
> #w;
12
> w;
y^-1 * x^-3 * z^-1 * y^-1 * x * y * z * x^2 * y
```

We replace each occurrence of the generator x in w by the word $y * z^-1$.

```
> Eliminate(w, x, y*z^-1);
y^-1 * z * y^-1 * z * y^-1 * z * y^-1 * z^-2 * y * z * y * z^-1 * y * z^-1 * y
```

We count the number of occurrences of each generator in w.

```
> [ ExponentSum(w, F.i) : i in [1..Ngens(F)] ];
[ 0, 0, 0 ]
> GeneratorNumber(w);
```

```
-2
```

We locate the start of the word u in the word w.

```
> b, p := Match(w, u, 1);
> b, p;
true 4
```

We now replace the subword u in w by the word $y * x$.

```
> t := Substitute(w, p, #u, y*x);
> t;
y^-1 * x^-2 * y * x^3 * y
```

We create the set of all distinct cyclic permutations of the word u.

```
> rots := { RotateWord(u, i) : i in [1 ..#u] };
> rots;
{ y^-1 * x * y * z * x^-1 * z^-1,  x * y * z * x^-1 * z^-1 * y^-1,
x^-1 * z^-1 * y^-1 * x * y * z,  z * x^-1 * z^-1 * y^-1 * x * y,
z^-1 * y^-1 * x * y * z * x^-1,  y * z * x^-1 * z^-1 * y^-1 * x }
```

71.3 Interactive Coset Enumeration

71.3.1 Introduction

This section presents the interactive coset enumeration facility of MAGMA. This concept makes it possible to restart an enumeration after changing enumeration parameters or adding relators or subgroup generators, while making use of information obtained up to that point as much as possible. It is thus particularly suitable for very hard enumerations, requiring a careful and interactive choice of enumeration parameters or for series of similar enumerations.

The Todd-Coxeter implementation installed in MAGMA is based on the stand alone coset enumeration programme `ACE3` developed by George Havas and Colin Ramsay at the University of Queensland. The reader should consult [CDHW73] and [Hav91] for an explanation of the terminology and a general description of the algorithm. A manual for `ACE3` as well as the sources of `ACE3` can be found online [Ram].

In MAGMA an interactive coset enumeration is realised as an object of the category `GrpFPCosetEnumProc` which can be created and modified, allows starting and restarting of coset enumerations and provides access to internal data like the coset table and various status information.

71.3.2 Constructing and Modifying a Coset Enumeration Process

`CosetEnumerationProcess(G, H: parameters)`

This function creates a coset enumeration process for enumerating the cosets of the subgroup H of the finitely presented group G. Note that no actual coset enumeration is started for the created coset enumeration process. This can be done with the function `StartEnumeration`.

The user can control in detail the way a subsequent enumeration will be performed with the help of the following parameters. For a more thorough explanation of the parameters and of how they affect the coset enumeration, we refer to the `ACE3` manual.

`Compact` RngIntElt *Default :* 10

This parameter controls the compaction of the coset table during an enumeration. A compaction will be done if a new coset definition is required, there is no space for a new coset available in the coset table and the percentage of dead cosets exceeds the value of the parameter `Compact`.

`CosetLimit` RngIntElt *Default :* 0

If `CosetLimit` is set to n, where n is a positive integer, then the coset table may have at most n rows. In other words, a maximum of n cosets can be defined at any instant during the enumeration. It is ensured in this case, that enough memory is allocated to store the requested number of cosets, regardless of the value of the parameter `Workspace`.

If `CosetLimit` is set to 0 (default), the maximal number of active cosets is determined by the size of the coset table (cf. parameter `Workspace`) and the number of columns of the coset table (i.e. the number of group generators).

`Workspace` RngIntElt *Default :* 4000000

The number of words allocated for the coset table. Note that if `CosetLimit` is set, at least as much memory is allocated as is necessary to store the requested number of cosets.

`FillFactor` RngIntElt *Default :* 0

In certain situations it is necessary to ensure that a certain proportion, the *fill fraction* (see Havas (1991)), of the coset table is always kept filled, even if *preferred definitions* are made. The parameter `FillFactor` allows the user to specify the fill fraction which is the reciprocal of `FillFactor`. The default value of 0 selects a fill factor of $\lfloor (5 * (c + 2))/4 \rfloor$, where c is the number of columns in the coset table.

`CTFactor` RngIntElt *Default :* 1000

`RTFactor` RngIntElt *Default :* $2000/l$

`Style` MonStgElt *Default :* "*R_CR*"

These parameters control the enumeration style and the balance between coset table style definitions (C-style, Felsch style) and relator table style definitions (R-style, HLT style).

The parameters `CTFactor` and `RTFactor` set the number of C-style coset definitions and R-style coset applications, respectively, which are performed in every step of the appropriate type. The default value for `RTFactor` is $2000/l$, where l is the total length of the relators.

The possible arguments to the parameter `Style` are `"R"`, `"C"`, `"R_CR"`, `"Rc"`, `"Cr"`, `"Rt"` and `"CR"`.

In style R all definitions are made via relator scans, i.e. this is HLT mode.

In style C all definitions are made by filling the next empty position in the coset table and testing the new coset in all essentially different positions in the relators, i.e. this is Felsch mode.

In style R_CR we run in style R until an overflow occurs, perform a lookahead on the entire table and then switch to style CR (see below).

The styles Rc and Cr are like the styles R and C, except that a single style C or style R pass is done after the initial R or style C pass, respectively.

In style Rt definitions are made as in style R, but the new cosets are tested in all essentially different positions in the relators as in style C.

In style CR alternate passes of style C and style R are performed, with all definitions tested in all essentially different positions in the relators as in style C.

`Lookahead` RngIntElt *Default :* 0

In HLT style enumerations, possible implications of new definitions or deductions made while tracing relators may not be detected until much later. One possible solution to this problem is to perform lookaheads occasionally, i.e. to process the coset table, looking for deductions or coincidences. A lookahead can be done using the entire table (*complete*) or only that part of the table above the current coset (*partial*) and it can be done in R-style (scanning cosets from the beginning of relators) or in C-style (testing all definitions in all essentially different relator positions). The following lookahead modes are supported.

For `Lookahead := 0` (default) no lookahead is done.
For `Lookahead := 1` a partial R-style lookahead is done.
For `Lookahead := 2` a complete C-style lookahead is done.
For `Lookahead := 3` a complete R-style lookahead is done.
For `Lookahead := 4` a partial C-style lookahead is done.

`Mendelsohn` BoolElt *Default :* `false`

If `Mendelsohn` is set, coset applications are done at all cyclic permutations of the relators. This is usually very expensive but may nevertheless be helpful in certain cases.

`RelationsInSubgroup` RngIntElt *Default :* -1

This parameter controls, whether relators are treated as additional subgroup generators, i.e. whether they are applied to coset 1 at the start of an enumeration. An argument of -1 (default) includes all relators, an argument of 0 turns this feature off and a positive argument includes the appropriate number of relators, in order.

`RowFilling` BoolElt *Default :* `true`

In R-style it is usual to scan each row of the table after its coset has been applied to all relators, and to make definitions to fill any holes encountered. Failure to do so can cause even simple enumerations to overflow. To switch row filling off, set the parameter `RowFilling` to false.

`PrefDefMode` RngIntElt *Default :* 3

If the argument is 0, then Felsch style definitions are made using the next empty position in the coset table. Otherwise, gaps of length one found during relator scans are preferentially filled. If the argument is 1, they are filled immediately, and if it is 2, the consequent deduction is also made immediately. If the argument is 3, then the gaps are noted in the preferred definition queue and the next coset definition will be made to fill the oldest gap of length one.

`PrefDefSize` RngIntElt *Default :* 8

This parameter controls the size of the preferred definition queue, which is implemented as a ring buffer, dropping earliest entries. Setting `PrefDefSize` to n allocates a buffer of size 2^n.

`DeductionMode` RngIntElt *Default :* 4

A completed table is only valid if every table entry has been tested in all essentially different relator positions. Untested deductions are stored on a stack. This parameter allows the user to specify how deductions should be handled. The possible actions are:

For `DeductionMode := 0` : discard deductions if there is no stack space left.

For `DeductionMode := 1` : as 0, but redundant cosets are purged off the top of the stack whenever a coincidence is found.

For `DeductionMode := 2` : as 0, but all redundant cosets are purged from the stack whenever a coincidence is found.

For `DeductionMode := 3` : discard the entire stack if it overflows.

For `DeductionMode := 4` : if the stack overflows, then double the stack size and purge all redundant cosets from the stack.

If deductions are discarded for any reason during an enumeration, then a final relator application pass will be done at the end of the enumeration automatically to check the result.

`DeductionSize` RngIntElt *Default :* 1000

Sets the (initial) size of the deduction stack in words, with one deduction taking two words. A value of 0 selects the default size of 1000 words.

`PathCompression` BoolElt *Default :* `false`

Switching this option on reduces the amount of data movement during coincidence processing at the expense of tracing and compressing coincidence paths, which involves many coset table accesses. The value of this parameter has no effect on the result but may influence the running time.

`TimeLimit` RngIntElt *Default :* -1

This parameter puts a time limit in seconds on the length of an enumeration. A value of -1 (default) means no limit. A value of 0 performs exactly one pass through the main enumeration loop.

`LoopLimit` RngIntElt *Default :* 0

The core enumerator is organised as a state machine, with each step performing an action (e.g. lookahead, compaction) or a block of actions (e.g. `CTFactor` coset definitions or `RTFactor` coset applications). Using this parameter, a limit on the total number of steps can be imposed. A value of 0 (default) turns this feature off.

`LowerBound` RngIntElt *Default :* 1

This may be set to any known lower bound for the index. Should it happen that the coset table has no holes, and the number of active cosets is equal to the given bound, the enumeration will terminate. When the given bound is equal to the index, this will save tracing relators at many cosets when there is no possibility of finding coincidences.

`Print` BoolElt *Default :* `false`

If this parameter is set to true, the enumerator prints a single message at the end of an enumeration and possibly some progress messages during an enumeration (cf. parameter `Messages`).

`Messages` RngIntElt *Default :* 0

If the argument n of this parameter is non-zero (and `Print` is set to true), then a progress message is printed after every $|n|$ "actions" (i.e. coset definitions, deductions, coincidences, etc.) A negative value of n turns hole monitoring on.

`Strategy` MonStgElt *Default :*

Using this parameter one of several predefined strategies can be selected. The effect is equivalent to selecting an appropriate combination of arguments for some of the parameters explained above. Any predefined strategy can be modified by explicitly overriding parameter values.

The following predefined strategies exist: "Default", "Easy", "Hard", "Felsch", "HLT", "CT", "RT", "Sims1", "Sims3", "Sims5", "Sims7", "Sims9".

All predefined strategies set `PrefDefSize := 8`, `DeductionSize := 1000`, `PathCompression := false` and `LoopLimit := 0`. For the remaining parameter settings see Table 1 on page 2215, Table 2 on page 2215 and Table 3 on page 2216.

`AddRelator(~P, w)`

Add a word w in the generators of the group G underlying the coset enumeration process P to the defining relations of G. This means that a coset enumeration process P for the cosets of H in G is transformed into a coset enumeration process for the cosets of $\pi(H)$ in $\pi(G)$, where $\pi : G \to \pi(G)$ is an epimorphism with kernel $\langle w \rangle^G$. (Here $\langle w \rangle^G$ denotes the normal closure in G of the subgroup of G generated by w.)

	Default	Easy	Hard	Felsch	HLT
`Compact`	10	100	10	10	10
`Workspace` (in 10^6)	4	1	10	4	4
`FillFactor`	0	1	0	0	1
`CTFactor`	1000	0	1000	1000	0
`RTFactor`	$2000/l$	1000	1	0	1000
`Style`	R_CR	R	CR	C	R
`Lookahead`	0	0	0	0	1
`Mendelsohn`	`false`	`false`	`false`	`false`	`false`
`RelationsInSubgroup`	-1	0	-1	-1	0
`RowFilling`	`true`	`true`	`true`	`false`	`true`
`PrefDefMode`	3	0	3	3	0
`DeductionMode`	4	0	4	4	0

Table 1: Strategies

	CT	RT	Sims1	Sims3	Sims5
`Compact`	100	100	10	10	10
`Workspace` (in 10^6)	4	4	4	4	4
`FillFactor`	1	1	1	1	1
`CTFactor`	1000	0	0	0	0
`RTFactor`	0	1000	1000	1000	1000
`Style`	C	R	R	Rt	R
`Lookahead`	0	0	0	0	0
`Mendelsohn`	`false`	`false`	`false`	`false`	`true`
`RelationsInSubgroup`	0	0	0	0	0
`RowFilling`	`false`	`false`	`true`	`true`	`true`
`PrefDefMode`	0	0	0	0	0
`DeductionMode`	4	0	0	4	0

Table 2: Strategies (continued)

`AddSubgroupGenerator(~P, w)`

Add an element w of the group G underlying the coset enumeration process P to the generators of the subgroup. This means that a coset enumeration process P for the cosets of H in G is transformed into a coset enumeration process for the cosets of $\langle H, w\rangle$ in G, where $\langle H, w\rangle$ denotes the subgroup of G generated by H and w.

	Sims7	Sims9
Compact	10	10
Workspace (in 10^6)	4	4
FillFactor	1	1
CTFactor	0	1000
RTFactor	1000	0
Style	Rt	C
Lookahead	0	0
Mendelsohn	true	false
RelationsInSubgroup	0	0
RowFilling	true	false
PrefDefMode	0	0
DeductionMode	4	4

Table 3: Strategies (continued)

`SetProcessParameters(~P: parameters)`

Change enumeration parameters of the coset enumeration process P. The set of parameters accepted by this function is the same as for the function CosetEnumerationProcess; see there for a description.

All parameters which are not explicitly changed or modified by selecting one of the predefined strategies retain their old values.

It should be noted that it is not possible to decrease the workspace allocated by a coset enumeration process, once an enumeration has been started. However the workspace can be extended without invalidating any information contained in the process.

71.3.3 Starting and Restarting an Enumeration

There are several ways of starting and restarting an enumeration for a coset enumeration process, which retain information from previous enumerations to a varying extent.

`StartEnumeration(~P: parameters)`

Start a new enumeration for P. All information in P is discarded. This function can be called at any time for an existing coset enumeration process. The enumeration parameters for P can be modified by passing parameters to this function. (This is equivalent to calling the function SetProcessParameters before calling StartEnumeration.) The set of parameters accepted by this function is the same as for the function CosetEnumerationProcess; see there for a description.

`RedoEnumeration(~P: parameters)`

Restart an enumeration for P. All information in P is retained and the enumeration is restarted at coset number 1. This function can be called for any coset enumeration process, which contains a valid coset table. (If P does not contain a valid coset table, a call to `RedoEnumeration` causes a runtime error. Use the function `CanRedoEnumeration` to check whether a call to `RedoEnumeration` is legal for a certain coset enumeration process.) Note that the coset table of P need not be complete to use this function.

This function is intended for the case where additional relators and/or subgroup generators have been introduced. The coset table contained in P is still valid. However, the additional data may allow the enumeration to compete, or cause a collapse to a smaller index.

The enumeration parameters for P can be modified by passing parameters to this function. (This is equivalent to calling the function `SetProcessParameters` before calling `RedoEnumeration`.) The set of parameters accepted by this function is the same as for the function `CosetEnumerationProcess`; see there for a description.

`CanRedoEnumeration(P)`

Returns `true`, if a call to `RedoEnumeration` is legal for the coset enumeration process P.

`ContinueEnumeration(~P: parameters)`

Continue an enumeration for P, which has been interrupted because some limit was exceeded. All information in P is retained and the enumeration is restarted at the coset where the previous enumeration was stopped. This function can only be called for a coset enumeration process, if the previous enumeration produced a valid coset table and the subgroup underlying P has not been changed since then. (Otherwise, a call to `ContinueEnumeration` causes a runtime error. Use the function `CanContinueEnumeration` to check whether a call to `ContinueEnumeration` is legal for a certain coset enumeration process.) Note that the coset table of P need not be complete to use this function.

This function is intended for the case where an enumeration stopped without producing a finite index. This function allows to continue the enumeration with modified enumeration parameters with the minimal possible overhead.

The enumeration parameters for P can be modified by passing parameters to this function. (This is equivalent to calling the function `SetProcessParameters` before calling `ContinueEnumeration`.) The set of parameters accepted by this function is the same as for the function `CosetEnumerationProcess`; see there for a description.

`CanContinueEnumeration(P)`

Returns `true`, if a call to `ContinueEnumeration` is legal for the coset enumeration process P.

`ResumeEnumeration(~P: parameters)`

Resume or start an enumeration for P in the "cheapest" way permitted by the state of the coset enumeration process P. The call

```
ResumeEnumeration(~P : parameters);
```

is equivalent to

```
if CanContinueEnumeration(P) then
   ContinueEnumeration(~P : parameters);
elif CanRedoEnumeration(P) then
   RedoEnumeration(~P : parameters);
else
   StartEnumeration(~P : parameters);
end if;
```

The enumeration parameters for P can be modified by passing parameters to this function. (This is equivalent to calling the function `SetProcessParameters` before calling `ResumeEnumeration`.) The set of parameters accepted by this function is the same as for the function `CosetEnumerationProcess`; see there for a description.

71.3.4 Accessing Information

`CosetsSatisfying(P : parameters)`

`CosetSatisfying(P : parameters)`

Given a coset enumeration process P with underlying group G and subgroup H, which contains a valid coset table, these functions return a set of words representing cosets which satisfy the conditions defined in the parameters.

A call to `CosetSatisfying` is equivalent to calling `CosetsSatisfying` with `Limit` set to 1 and `First` set to 2 (unless a higher value for `First` is specified explicitly), i.e. the function returns when the first coset, other than H, satisfying the specified conditions has been found.

`First` RngIntElt *Default :* 1

This parameter determines at which coset the search for coset representatives satisfying the designated conditions is started. The coset H always is coset number 1, i.e. setting `First` to 2 restricts the search to non-trivial cosets.

For the function `CosetSatisfying`, the minimal possible value for this parameter is 2; setting it to 1 does not cause an error but will be ignored.

`Last` RngIntElt *Default :* 0

This parameter determines at which coset the search for coset representatives satisfying the designated conditions is stopped. If it is set to 0 (default), all active cosets (starting from `First`) are searched.

`Limit` RngIntElt *Default :* 0

If this parameter is set to $l > 0$, the search for coset representatives is aborted as soon as l cosets satisfying the designated condition have been found. If it is set to 0 (default for `CosetsSatisfying`), no limit is in force.

This parameter is not available for the function `CosetSatisfying`.

`Normalizing` BoolElt *Default :* `false`

If true, select coset representatives x such that $x^{-1}h_ix$ is known to be contained in H from the information in the coset table of P for every generator h_i of H. (I.e. x is recognised as an element of the normaliser of H in G.)

`Order` RngIntElt *Default :* 0

Select coset representatives x such that x^n is known to be contained in H from the information in the coset table of P.

`Print` RngIntElt *Default :* 0

If the value of this parameter is positive, print the coset representatives found to satisfy the designated conditions.

These functions can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to any of these functions causes a runtime error. You can use the function `HasValidCosetTable` to check whether a call to these functions is legal for a certain coset enumeration process.

`CosetTable(P)`

The current coset table of P as a map $f : \{1, \ldots, r\} \times G \to \{0, \ldots, r\}$, where G and H are the finitely presented group and the subgroup underlying P, respectively, and r is number of active cosets. $f(i, x)$ is the coset to which coset i is mapped under the action of $x \in G$. The value 0 is only included in the codomain if the coset table is not complete, and it denotes that the image of i under x is not known.

This function can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to `CosetTable` causes a runtime error. Use the function `HasValidCosetTable` to check whether a call to `CosetTable` is legal for a certain coset enumeration process. Note that the coset table of P need not be complete to use this function.

`HasValidCosetTable(P)`

Returns `true`, if P contains a valid (but not necessarily closed) coset table, i.e. if a call to `CosetTable` is legal for the coset enumeration process P.

`HasClosedCosetTable(P)`

`HasCompleteCosetTable(P)`

Returns `true`, if P contains a closed, valid coset table.

Note that if `HasClosedCosetTable` returns `true` for a coset enumeration process P, then in particular a call to `CosetTable` is legal for P.

`ExcludedConjugate(P)`

`ExcludedConjugates(P)`

Given a coset enumeration process P with underlying group G and subgroup H, which contains a valid coset table, these functions return a set E containing either at most one word (`ExcludedConjugate`) or all words (`ExcludedConjugates`) of the form $g_i^{-1}h_jg_i$, where g_i is a generator of G and h_j is a generator of H, such that $g_i^{-1}h_jg_i$ is not known to lie in H from the information contained in the coset table of P.

If E is empty, then H is a normal subgroup of G. Otherwise the addition of elements of E to the generators of H may yield a larger subgroup of the normal closure of H in G. In the case of a non-complete coset table it may happen, however, that excluded conjugates are found which actually lie in H.

These functions can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to `ExcludedConjugate` or `ExcludedConjugates` causes a runtime error. You can use the function `HasValidCosetTable` to check whether a call to these functions is legal for a certain coset enumeration process.

`ExistsCosetSatisfying(P :` *parameters*`)`

Given a coset enumeration process P with underlying group G and subgroup H, which contains a valid coset table, return whether or not there exists a coset other than H, which satisfies the conditions defined in the parameters. If such a coset exists, a representing word is returned as second return value. This function accepts the same set of parameters as the function `CosetSatisfying`; see there for a description.

This function can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to `ExistsCosetSatisfying` causes a runtime error. You can use the function `HasValidCosetTable` to check whether a call to `ExistsCosetSatisfying` is legal for a certain coset enumeration process.

`ExistsExcludedConjugate(P)`

Given a coset enumeration process P with underlying group G and subgroup H, which contains a valid coset table, return whether or not there exists a word of the form $g_i^{-1}h_jg_i$, where g_i is a generator of G and h_j is a generator of H, such that $g_i^{-1}h_jg_i$ is not known to lie in H from the information contained in the coset table of P. If the answer is positive, such a word is returned as second return value.

This function can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to `ExistsExcludedConjugate` causes a runtime error. You can use the function `HasValidCosetTable` to check whether a call to `ExistsExcludedConjugate` is legal for a certain coset enumeration process.

Note that the coset table of P need not be complete to call this function. A negative result of `ExistsExcludedConjugate` always implies that H is a normal subgroup of G, even if the coset table of P is not complete. In the case of a non-complete coset table it may happen, however, that excluded conjugates are found which actually lie in H.

`ExistsNormalisingCoset(P)`

`ExistsNormalizingCoset(P)`

Returns `true`, if an element of $G \setminus H$ which normalises H can be found from the coset table contained in P. (Here, G and H are the finitely presented group and the subgroup underlying P, respectively.) If the answer is positive, such an element is returned as second return value.

This function can be called for any coset enumeration process, which contains a valid coset table. If P does not contain a valid coset table, a call to `ExistsNormalisingCoset` causes a runtime error. You can use the function `HasValidCosetTable` to check whether a call to `ExistsNormalisingCoset` is legal for a certain coset enumeration process.

Note that the coset table of P need not be complete to call this function. However, no conclusion can be drawn from a negative result in the case of a non-complete coset table.

`Group(P)`

Returns the group underlying P as a finitely presented group.

`Index(P)`

Returns the index of H in G. (Here, G and H denote the finitely presented group and the subgroup underlying P, respectively.)

This function can only be called, if the last enumeration done for P has completed successfully with a finite index. Otherwise, a call to `Index` will cause a runtime error. Use the function `HasValidIndex` to check whether a call to `Index` is legal for a certain coset enumeration process.

`HasValidIndex(P)`

Returns `true`, if the last enumeration done for P has completed successfully with a finite index, i.e., if a call to `Index` is legal for the coset enumeration process P.

`MaximalNumberOfCosets(P)`

Returns the maximal number of cosets which were simultaneously active during the last enumeration done for P, or 1 if no enumeration has been done for P.

This function may be useful for assessing the performance of a certain set of enumeration parameters.

`Subgroup(P)`

Returns the subgroup H underlying the coset enumeration process P. H is returned as a subgroup of G, where G is the finitely presented group underlying P.

`TotalNumberOfCosets(P)`

Returns the total number of cosets defined during the last enumeration done for P, or 1 if no enumeration has been done for P.

This function may be useful for assessing the performance of a certain set of enumeration parameters.

Example H71E3

In the Harada-Norton sporadic simple group

$$\begin{aligned} < x, a, b, c, d, e, f, g \mid\ & x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2, \\ & (x, a), (x, g), \\ & (bc)^3, (bd)^2, (be)^2, (bf)^2, (bg)^2, \\ & (cd)^3, (ce)^2, (cf)^2, (cg)^2, \\ & (de)^3, (df)^2, (dg)^2, \\ & (ef)^3, (eg)^2, \\ & (fg)^3, \\ & (b, xbx), \\ & (a, edcb), (a, f)dcbdcd, (ag)^5, \\ & (cdef, xbx), (b, xcdefx), (cdef, xcdefx) > \end{aligned}$$

we want to construct the coset table for the subgroup generated by x, b, c, d, e, f, g interactively. First, we create a coset enumeration process. Since the index of the chosen subgroup is 1 140 000, we request a coset limit of 1 200 000. Then we start the enumeration.

```
> HN<x, a, b, c, d, e, f, g> :=
>     Group< x, a, b, c, d, e, f, g |
>               x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
>               (x, a), (x, g),
>               (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>               (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>               (d*e)^3, (d*f)^2, (d*g)^2,
>               (e*f)^3, (e*g)^2,
>               (f*g)^3,
>               (b, x*b*x), (a, e*d*c*b), (a, f)*d*c*b*d*c*d,
>               (a*g)^5, (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>               (c*d*e*f, x*c*d*e*f*x)
>           >;
> H := sub<HN | x,b,c,d,e,f,g >;
> P := CosetEnumerationProcess(HN, H : CosetLimit := 1200000, Print := true);
```

```
> StartEnumeration(~P);
Overflow
 (a=1110331 r=58605 h=101193 n=1200001;
  l=5444 c=105.36;
  m=1110331 t=2001537)
```

The enumeration could not be completed successfully. Though P does contain a valid coset table, but this coset table fails to be closed.

```
> HasValidCosetTable(P);
true
> HasClosedCosetTable(P);
false
```

We extract the coset table map, check its domain and its codomain, and determine the position of the first "hole" in the coset table.

```
> ct := CosetTable(P);
> Domain(ct) : Minimal;
Cartesian Product<{ 1 .. 1110331 }, GrpFP: HN>
> Codomain(ct);
{ 0 .. 1110331 }
> row := 1;
> while forall(col){ gen : gen in {x, a, b, c, d, e, f, g}
>                        | ct(<row, gen>) ne 0 } do
>    row +:= 1;
> end while;
> row;
41881
> col;
x
```

We change the enumeration parameters for the process P, selecting the predefined strategy `Hard`. Since this predefined strategy sets a workspace of 10 000 000, we must expressly override this, if we want the old value to be retained. Because the workspace size never is decreased once an enumeration has been started, we can retain the old value simply by setting `Workspace` to 0; this overrides the setting done by selecting the predefined strategy, but actually doesn't change the process' workspace. We then continue the enumeration with the new set of parameters, after checking that this is legal.

```
> SetProcessParameters(~P : Strategy := "Hard",
>                           Workspace := 0);
> CanContinueEnumeration(P);
true
> ContinueEnumeration(~P);
INDEX = 1140000
 (a=1140000 r=58512 h=1144534 n=1144534;
  l=70 c=6.50;
```

```
  m=1140000 t=2035739)
```

We have obtained a closed coset table. Note that the codomain of the new coset table map does not contain 0.

```
> HasClosedCosetTable(P);
true
> ct := CosetTable(P);
> Domain(ct) : Minimal;
Cartesian Product<{ 1 .. 1140000 }, GrpFP: HN>
> Codomain(ct);
{ 1 .. 1140000 }
```

Example H71E4

First, we create a coset enumeration process for the trivial subgroup of the finite group $G := \langle a, b \mid a^8, b^7, (a*b)^2, (a^{-1}*b)^3 \rangle$ and start the enumeration.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | >;
> P := CosetEnumerationProcess(G, H : Print := true);
> StartEnumeration(~P);
INDEX = 10752
 (a=10752 r=57263 h=1 n=57263;
  l=292 c=0.11;
  m=47825t=57262)
```

We want to enumerate the cosets of the two non-trivial subgroups of G generated by $a^{-1}*b$ and a^2, respectively. To do this, we create two copies of the coset enumeration process `P` and use the function `AddSubgroupGenerator` for each of them. Since the copies inherit all information contained in `P`, we then can call the function `RedoEnumeration` to enumerate the cosets of the two non-trivial subgroups, making use of the existing coset table.

```
> P1 := P;
> AddSubgroupGenerator(~P1, a^-1*b);
> Subgroup(P1);
Finitely presented group on 2 generators
Generators as words in group G
    $.1 = Id(G)
    $.2 = a^-1 * b
> CanRedoEnumeration(P1);
true
> RedoEnumeration(~P1);
INDEX = 3584
 (a=3584 r=57263 h=1 n=57263;
  l=49 c=0.02;
  m=47825 t=57262)
>
> P2 := P;
```

```
> AddSubgroupGenerator(~P2, a^2);
> Subgroup(P2);
Finitely presented group on 2 generators
Generators as words in group G
    $.1 = Id(G)
    $.2 = a^2
> CanRedoEnumeration(P2);
true
> RedoEnumeration(~P2);
INDEX = 2688
 (a=2688 r=57263 h=1 n=57263;
  l=37 c=0.02;
  m=47825 t=57262)
```

Finally, we are interested in the quotient of G by the normal closure of the subgroup generated by a^4 and want to enumerate the cosets of the image of the subgroup generated by a^2 in this quotient. Since this is the subgroup used in `P2`, we create a copy of `P2` and add the relation a^4. Again, we are able to make use of information obtained earlier, by continuing the inherited enumeration.

```
> P3 := P2;
> AddRelator(~P3, a^4);
> CanContinueEnumeration(P3);
true
> ContinueEnumeration(~P3);
INDEX = 84
 (a=84 r=57263 h=1 n=57263;
 l=2 c=0.00;
 m=47825 t=57262)
```

We extract the quotient and its subgroup from the process `P3`, using the appropriate access functions.

```
> G3<a3,b3> := Group(P3);
> G3;
Finitely presented group G3 on 2 generators
Relations
    a3^8 = Id(G3)
    b3^7 = Id(G3)
    (a3 * b3)^2 = Id(G3)
    (a3^-1 * b3)^3 = Id(G3)
    a3^4 = Id(G3)
> H3<u3, v3> := Subgroup(P3);
> H3;
Finitely presented group H3 on 2 generators
Generators as words in group G3
    u3 = Id(G3)
    v3 = a3^2
```

Example H71E5

Consider the subgroup H of the (infinite) group $G := \langle a, b \mid b^7, (a*b)^2, (a^{-1}*b)^3 \rangle$ generated by a. We create a coset enumeration process and start an enumeration with the default parameters.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a>;
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P : Print := true);
Overflow
 (a=957026 r=415230 h=415230 n=999999;
  l=3553 c=2.38;
  m=960050 t=999998)
```

The enumeration produces a valid (albeit not complete) coset table.

```
> HasValidCosetTable(P);
true
> HasCompleteCosetTable(P);
false
```

Even the partial coset table is sufficient to find an element in the normaliser of H in G.

```
> found, elt := ExistsNormalisingCoset(P);
> found;
true
> elt;
b^-4 * a * b^-2
```

Example H71E6

Consider again the subgroup H of the (infinite) group $G := \langle a, b \mid b^7, (a*b)^2, (a^{-1}*b)^3 \rangle$ generated by a. We create a coset enumeration process and start an enumeration with the default parameters.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a>;
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P : Print := true);
Overflow
 (a=957026 r=415230 h=415230 n=999999;
  l=3553 c=2.38;
  m=960050 t=999998)
```

We check, whether the coset table exhibits excluded conjugates.

```
> ExistsExcludedConjugate(P);
```

```
true a^b
```

It does. This means in particular, that H is not normal in G. We create a copy `P1` of the coset enumeration process `P`, extend the subgroup of `P1` by the excluded conjugates found in the previous step and restart the enumeration for `P1`.

```
> P1 := P;
> for c in ExcludedConjugates(P) do
>   AddSubgroupGenerator(~P1, c);
> end for;
> RedoEnumeration(~P1);
INDEX = 1 (a=1 r=2 h=2 n=2; l=2 c=0.94; m=960050 t=999998)
```

The new subgroup is equal to G. In particular, the normal closure of H in G is the whole of G. We return to the coset enumeration process `P` and check whether we can find a non-trivial element $x \in \mathbf{N}_G(H)$ such that $x^2 \in H$.

```
> ExistsCosetSatisfying(P : Order := 2, Normalizing := true);
true b^-4 * a * b^-2
```

We can. In fact, $b^{-4} \cdot a \cdot b^{-2}$ is the only non-trivial coset which is known to satisfy this condition...

```
> CosetsSatisfying(P : Order := 2, Normalizing := true);
{ Id(G), b^-4 * a * b^-2 }
```

...and we can't find in a similar way a non-trivial element $x \in \mathbf{N}_G(H)$ such that $x^3 \in H$.

```
> ExistsCosetSatisfying(P : Order := 3, Normalizing := true);
false
```

Note the difference in the output of `CosetSatisfying` and `CosetsSatisfying`: The former takes into account only cosets other than H, whereas the latter (unless we set the parameter `First`) includes the coset H, which obviously satisfies the specified conditions.

```
> CosetSatisfying(P : Order := 3, Normalizing := true);
{}
> CosetsSatisfying(P : Order := 3, Normalizing := true);
{ Id(G) }
```

71.3.5 Induced Permutation Representations

Given a finite index subgroup H of a group G, the action of G on the set of right cosets of H in G by right multiplication defines a permutation representation $\rho : G \to S$ of G onto a suitable subgroup S of the symmetric group on $[G : H]$ letters. The kernel of ρ is the core of H in G, the maximal normal subgroup of G contained in H.

`CosetAction(P)`

Given a coset enumeration process P with underlying group G and subgroup H for which a valid finite index has been obtained, this function returns

(a) The permutation representation ρ of G, induced by the action of G on the set of right cosets of H in G.

(b) The image group $\rho(G)$.

(c) (if possible) the kernel of ρ.

This function can only be called, if the last enumeration done for P has completed successfully with a finite index. Otherwise, a call to `CosetAction` will cause a run-time error. Use the function `HasValidIndex` to check whether a call to `CosetAction` is legal for a certain coset enumeration process.

`CosetImage(P)`

Given a coset enumeration process P with underlying group G and subgroup H for which a valid finite index has been obtained, this function returns the image of the permutation representation ρ of G induced by the action of G on the set of right cosets of H in G as a permutation group on $[G : H]$ digits.

This function can only be called, if the last enumeration done for P has completed successfully with a finite index. Otherwise, a call to `CosetImage` will cause a runtime error. Use the function `HasValidIndex` to check whether a call to `CosetImage` is legal for a certain coset enumeration process.

`CosetKernel(P)`

Given a coset enumeration process P with underlying group G and subgroup H for which a valid finite index has been obtained, this function returns the kernel of the permutation representation ρ of G induced by the action of G on the set of right cosets of H in G. This function is only available if the index of H in G is sufficiently small.

This function can only be called, if the last enumeration done for P has completed successfully with a finite index. Otherwise, a call to `CosetKernel` will cause a run-time error. Use the function `HasValidIndex` to check whether a call to `CosetKernel` is legal for a certain coset enumeration process.

71.3.6 Coset Spaces and Transversals

The (right) indexed coset space V of the subgroup H of the group G is a G-set consisting of the set of integers $\{1, ..., m\}$, where i represents some right coset c_i of H in G. The action of G on this G-set is that induced by the natural G-action

$$f : V \times G \to V$$

where

$$f : \langle c_i, x \rangle = c_k \quad \Longleftrightarrow \quad c_i * x = c_k,$$

for $c_i \in V$ and $x \in G$. If certain of the products $c_i * x$ are unknown, the corresponding images under f are undefined, and V is called an *incomplete coset space* for H in G.

`CosetSpace(P)`

The coset space defined by the current state of the coset enumeration process P.

This function can be called for any coset enumeration process, which contains a valid coset table. (If P does not contain a valid coset table, a call to `CosetSpace` causes a runtime error. Use the function `HasValidCosetTable` to check whether a call to `CosetSpace` is legal for a certain coset enumeration process.) Note that the coset table of P need not be complete to use this function.

`RightCosetSpace(P)`

`LeftCosetSpace(P)`

The explicit right coset space of a subgroup H of some group G is a G-set containing the set of right cosets of H in G. The elements of this G-set are the pairs $\langle H, x\rangle$, where x runs through a transversal for H in G. Similarly, the explicit left coset space of H is a G-set containing the set of left cosets of H in G, represented as the pairs $\langle x, H\rangle$.

These functions return the explicit right (left) coset space defined by the current state of the coset enumeration process P.

This function can be called for any coset enumeration process, which contains a valid coset table. (If P does not contain a valid coset table, a call to any of these functions causes a runtime error. Use the function `HasValidCosetTable` to check whether a call to these functions is legal for a certain coset enumeration process.) Note that the coset table of P need not be complete to use these functions.

`Transversal(P)`

`RightTransversal(P)`

Given a coset enumeration process P with underlying group G and subgroup H for which a valid finite index has been obtained, these functions return

(a) An indexed set T of elements of G forming a right transversal for H in G; and

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = \{@\ t_1, \ldots, t_r\ @\}$ and $g \in G$, then ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

These functions can only be called, if the last enumeration done for P has completed successfully with a finite index. Otherwise, a call to any of these functions will cause a runtime error. Use the function `HasValidIndex` to check whether a call to these functions is legal for a certain coset enumeration process.

Example H71E7

We construct a coset enumeration process for the subgroup $H = \langle a^2, a^{-1}b\rangle$ in the group $G = \langle a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3\rangle$ and start an enumeration.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | x^8, y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a^2,a^-1*b>;
```

```
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P);
```

After checking that a finite index has been obtained, we extract a transversal and the corresponding transversal map from P.

```
> HasValidIndex(P);
true
> T, f := Transversal(P);
> #T;
448
> f;
Mapping from: GrpFP: G to SetIndx: T
```

Finally, we construct the permutation representation of G on the cosets of H in G, its image and its kernel.

```
> r, S, K := CosetAction(P);
> r : Minimal;
Homomorphism of GrpFP: G into GrpPerm: S, Degree 448, Order 2^9 *
3 * 7
> S;
Permutation group S acting on a set of cardinality 448
Order = 10752 = 2^9 * 3 * 7
> K;
Finitely presented group K
Index in group G is 10752 = 2^9 * 3 * 7
Subgroup of group G defined by coset table
```

The kernel turns out to be trivial, i.e. the permutation representation is faithful.

```
> Order(K);
1
```

Example H71E8

Consider the subgroup H of the (infinite) group $G := \langle a, b \mid b^7, (a*b)^2, (a^{-1}*b)^3 \rangle$ generated by a. We create a coset enumeration process and start an enumeration with the default parameters.

```
> F<x, y> := FreeGroup(2);
> G<a, b> := quo<F | y^7, (x*y)^2, (x^-1*y)^3>;
> H := sub<G | a>;
> P := CosetEnumerationProcess(G, H);
> StartEnumeration(~P : Print := true);
Overflow
 (a=957026 r=415230 h=415230 n=999999;
  l=3553 c=2.38;
  m=960050 t=999998)
```

The enumeration produces a valid (albeit not complete) coset table.

```
> HasValidCosetTable(P);
```

```
true
> HasCompleteCosetTable(P);
false
```

We extract the incomplete coset space from the process.

```
> V := CosetSpace(P);
> #V;
957026
> IsComplete(V);
false
```

71.4 p-Quotients (Process Version)

Let F be a finitely presented group, p a prime and c a positive integer. A p-quotient algorithm constructs a consistent power-conjugate presentation for the largest p-quotient of F having lower exponent-p class at most c. For details of this algorithm, see [NO96].

Assume that the p-quotient has order p^n, Frattini rank d, and that its generators are $a_1, \ldots, a_n$. Then the power-conjugate presentation constructed has the following additional structure. The set $\{a_1, \ldots, a_d\}$ is a generating set for G. For each a_k in $\{a_{d+1}, \ldots, a_n\}$, there is at least one relation whose right hand side is a_k. One of these relations is taken as the *definition* of a_k. (The list of definitions is also returned by `pQuotient`.) The power-conjugate generators also have a *weight* associated with them: a generator is assigned a weight corresponding to the stage at which it is added and this weight is extended to all normal words in a natural way.

The p-quotient process and its associated commands allows the user to construct a power-conjugate presentation (pcp) for a p-group.

71.4.1 The p-Quotient Process

`pQuotientProcess(F, p, c`: *parameters*`)`

Given an fp-group F, a prime p and a positive integer c, create a p-quotient process for the group F with the indicated arguments. As part of the initialisation of the process, a pcp for the largest p-quotient of F having class at most c will be constructed. If c is given as 0, then the limit 127 is placed on the class. This function supports the same parameters as `pQuotient` and returns a process P.

`NextClass(~P : parameters)`

`NextClass(~P, k : parameters)`

`Exponent`	RngIntElt	*Default :*
`Metabelian`	BoolElt	*Default :*
`Print`	RngIntElt	*Default :*
`MaxOccurrence`	[RngIntElt]	*Default :* []

Assumes that a pcp has already been constructed for the class c quotient of F. It seeks to construct a pcp for the class $c+1$ p-quotient of F. If `k` is supplied, continue to construct until the pcp for the largest quotient of class k is constructed.

The parameters `Exponent`, `Print`, and `Metabelian` are used as before. If `MaxOccurrence := Q`, then the sequence Q has length equal to the rank of the class 1 quotient of F; its entries are integers which specify the maximum number of occurrences of the class 1 generators in the definitions of pcp generators of F. An entry of 0 for a particular generator indicates that no limit is placed on the number of occurrences of this generator.

Care should be exercised when supplying values for parameters. Once set, they retain their values until explicitly reassigned.

71.4.2 Using p-Quotient Interactively

We assume that we have constructed a pcp for the largest class c p-quotient of F and now seek to construct a pcp for the largest class $c+1$ p-quotient.

The following options allow the user to construct a pcp for the next class of the group interactively. The steps are laid out in one of a number of natural sequences in which they may be executed. Some of them may be interleaved; however, the user should pay particular attention to the assumptions mentioned below. The procedures that drive the process do not verify that the assumptions are satisfied.

`StartNewClass(~P: parameters)`

If P is a process for a class c p-quotient, commence construction of class $c+1$.

`Tails(~P: parameters)`

`Tails(~P, k: parameters)`

`Metabelian`	BoolElt	*Default :* `false`

Add tails to the current pcp; default is to add all tails for this class. If `k` is supplied, then tails for weight k only are added; in this case, it is assumed that the tails for each of weight $c+1, c, \ldots, k+1$ have already been added. The valid range of k is $2, \ldots, c+1$. The one valid parameter is `Metabelian`; if `true`, then only the tails for the metabelian p-quotient are inserted.

`Consistency(~P: parameters)`

`Consistency(~P, k: parameters)`

`Metabelian` BoolElt *Default :* `false`

Apply the consistency algorithm to the pcp to compute any redundancies among the tails already added. Default is to apply it to all tails; in this case, it is assumed that all tails have been added. If `k` is supplied, it is assumed that tails for weight k have been added; in this case, the tails added for weight k only are checked. The range of k is $3, \ldots, c+1$. The one valid parameter is `Metabelian`; if `true`, we assume that the tails inserted were those for a metabelian p-quotient and hence invoke the (less expensive) metabelian consistency algorithm.

`CollectRelations(~P)`

Collect the defining relations (if any) in the current pcp. If the tails operation is not complete, then the relations may be evaluated incorrectly.

`ExponentLaw(~P : parameters)`

`ExponentLaw(~P, Start, Fin: parameters)`

Enforce the supplied exponent law on the current pcp. If `Start` and `Fin` are supplied, then enforce the law for those weights between `Start` and `Fin`; otherwise, enforce the law for all weights. It is assumed that the tails operation is complete. If the display parameter `DisplayLevel` (which may be set using `SetDisplayLevel`) has value 2, those words whose powers are collected and give redundancies among the pcp generators are printed out. If `DisplayLevel` has value 3, all words whose powers are collected are printed out. The following additional parameters are available:

`Exponent` RngIntElt *Default :* 0

If `Exponent := m`, enforce the exponent law, $x^m = 1$, on the group.

`Print` RngIntElt *Default :* 1

As for `pQuotient`.

`Trial` BoolElt *Default :* `false`

Generate the list of words used to enforce the exponent law and print out statistics but do not power words or echelonise the results.

`ShortList` BoolElt *Default :* `false`

Generate the list of enforcement words whose entries have the form w or $1 * w$ where w is an element of the Frattini subgroup of F.

`DisplayList` BoolElt *Default :* `false`

Display the list of all enforcement words generated – not just those which are collected.

`IdentifyFilters` BoolElt *Default :* `false`

Identify filters used to eliminate words from list.

`InitialSegment` [<GrpFPElt, RngIntElt>] *Default :* []

If `InitialSegment := w`, generate only those enforcement words which have w as an initial segment, where w is supplied as a sequence of generator-exponent pairs.

`Report` RngIntElt *Default :* 0

If `Report := n`, report after computing the powers of each collection of n enforcement words.

`EliminateRedundancy(~P)`

Eliminate all redundant generators from the pcp defined by process P. This operation may be performed at any time.

We now list the remaining functions which can be applied to a pQuotient process.

`Display(P)`

`Display(P, DisplayLevel)`

Display the pcp for the p-quotient G of the fp-group F. The argument `DisplayLevel` may be 1, 2, or 3, and is used to control the amount of information given:

`1` : Display order and class of G;

`2` : Display non-trivial relations for G;

`3` : Display the structure of pcp generators of G, non-trivial relations of G, and the map from the defining generators of F to the pcp generators of G.

The presentation displayed by this function is in power-commutator form. If `DisplayLevel` is not supplied, the information displayed is determined by its existing (or default) value.

`RevertClass(~P)`

Given a pcp for the class $c+1$ p-quotient of F, this procedure reverts to the pcp for the class c p-quotient of F. Note that this command can be applied only **once** during construction of a single class.

`pCoveringGroup(~P)`

`pCoveringGroup(`G`)`

Given a process or a pcp for a p-group, this procedure computes a pcp for the p-covering group of this group. In the process case, it is equivalent to `Tails(~P); Consistency(~P); EliminateRedundancy(~P)`.

`GeneratorStructure(P)`

`GeneratorStructure(P, Start, Fin)`

Display the structure of the generators in the pcp. If `Start` and `Fin` are given, then print out the structure of those pcp generators numbered from `Start` to `Fin`.

`Jacobi(~P, c, b, a, ~r)`

`Jacobi(~P, c, b, a)`

Calculate the Jacobi c, b, a and echelonise the resulting relation against the current pcp. If a redundant generator results from the echelonisation, the optional variable r is the number of that generator; otherwise r has value 0.

`Collect(P, Q)`

The sequence Q, consisting of generator-exponent pairs, defines a word w in the pcp generators of the group defined by the process P. Collect this word and return the resulting normal word as an exponent vector.

`EcheloniseWord(~P, ~r)`

`EcheloniseWord(~P)`

Echelonise the word most recently collected using `Collect` against the relations of the pcp. If a redundant generator results from the echelonisation, the optional variable r is the number of that generator; otherwise r has value 0. This function must be called immediately after `Collect`.

`SetDisplayLevel(~P, Level)`

This procedure alters the display level for the process to the supplied value, `Level`.

`ExtractGroup(P)`

Extract the group G defined by the pcp associated with the process P, as a member of the category `GrpPC` of finite soluble groups. The function also returns the natural homomorphism π from the original group F to G, a sequence S describing the definitions of the pc-generators of G and a flag indicating whether G is the maximal p-quotient of F.

The k-th element of S is a sequence of two integers, describing the definition of the k-th pc-generator $G.k$ of G as follows.

- If $S[k] = [0, r]$, then $G.k$ is defined via the image of $F.r$ under π.
- If $S[k] = [r, 0]$, then $G.k$ is defined via the power relation for $G.r$.
- If $S[k] = [r, s]$, then $G.k$ is defined via the conjugate relation involving $G.r^{G.s}$.

`Order(P)`

The order of the group defined by the pcp associated with the process P.

`FactoredOrder(P)`

The factored order of the group defined by the pcp associated with the process P.

`NumberOfPCGenerators(P)`

The number of pc-generators of the group defined by the pcp associated with the process P.

pClass(P)

The lower exponent-p class of the group defined by the pcp associated with the process P.

NuclearRank(G)

NuclearRank(P)

Return the rank of the p-multiplicator of the p-group G, where G may be supplied or defined by the process P.

pMultiplicatorRank(G)

pMultiplicatorRank(P)

Return the rank of the p-multiplicator of the p-group G, where G may be supplied or defined by the process P.

Example H71E9

Starting with an exponent 9 group having two generators of order 3, we set up the largest class 4 quotient and then interactively compute two more classes.

```
> G<a, b> := Group<a, b | a^3, b^3>;
> q := pQuotientProcess(G, 3, 4: Exponent := 9, Print :=1);

Lower exponent-3 central series for G

Group: G to lower exponent-3 central class 1 has order 3^2

Group: G to lower exponent-3 central class 2 has order 3^3

Group: G to lower exponent-3 central class 3 has order 3^5

Group: G to lower exponent-3 central class 4 has order 3^7
> Display(q, 2);

Group: G to lower exponent-3 central class 4 has order 3^7

Non-trivial powers:
 .3^3 = .6^2

Non-trivial commutators:
[ .2, .1 ] = .3
[ .3, .1 ] = .4
[ .3, .2 ] = .5
[ .4, .1 ] = .6
[ .4, .2 ] = .7
[ .5, .1 ] = .7
```

```
[ .5, .2 ] = .6
```

We construct the class 5 quotient using the function NextClass.

```
> NextClass(~q);
```

```
Group: G to lower exponent-3 central class 5 has order 3^9
```

We now construct the class 6 quotient step by step. For this, we set the output level to 1.

```
> SetDisplayLevel(~q, 1);
```

Now we start the next class.

```
> StartNewClass(~q);
```

The first step is to add the tails.

```
> Tails(~q);
```

After that, we apply the consistency algorithm,...

```
> Consistency(~q);
```

... collect the defining relations,...

```
> CollectRelations(~q);
```

... and enforce the exponent law.

```
> ExponentLaw(~q);
```

Finally, we eliminate redundant generators.

```
> EliminateRedundancy(~q);
```

This results in the following presentation for class 6 quotient.

```
> Display(q, 2);
```

```
Group: G to lower exponent-3 central class 6 has order 3^11

Non-trivial powers:
 .3^3 = .6^2 .8^2 .10 .11

Non-trivial commutators:
[ .2, .1 ] = .3
[ .3, .1 ] = .4
[ .3, .2 ] = .5
[ .4, .1 ] = .6
[ .4, .2 ] = .7
[ .4, .3 ] = .8 .10^2 .11^2
[ .5, .1 ] = .7 .8 .9^2 .10^2 .11^2
[ .5, .2 ] = .6 .8 .9^2 .10^2
[ .5, .3 ] = .9^2 .11
[ .5, .4 ] = .10 .11
```

```
[ .6, .2 ] = .10^2
[ .7, .1 ] = .8
[ .7, .2 ] = .9
[ .7, .3 ] = .10^2 .11
[ .8, .2 ] = .10
[ .9, .1 ] = .11
```

Example H71E10

Starting with the free product of two cyclic groups of order 5, we bound the number of occurrences of pcp generators of the class 1 quotient in definitions of new pcp generators.
We start with setting up the class 1 quotient of the group.

```
> G := Group<a, b | a^5, b^5>;
> q := pQuotientProcess(G, 5, 1);
> Display(q, 1);
Group: G to lower exponent-5 central class 1 has order 5^2
```

Now we start the next class, setting bounds on the number of occurrences of the pcp generators of the class 1 quotient in the definitions of new pcp generators.

```
> NextClass(~q, 6: MaxOccurrence := [3, 2]);
Group: G to lower exponent-5 central class 2 has order 5^3
Group: G to lower exponent-5 central class 3 has order 5^5
Group: G to lower exponent-5 central class 4 has order 5^7
Group: G to lower exponent-5 central class 5 has order 5^9
> Display(q, 2);
Group: G to lower exponent-5 central class 5 has order 5^9
Non-trivial powers:
Non-trivial commutators:
[ .2, .1 ] = .3
[ .3, .1 ] = .4
[ .3, .2 ] = .5
[ .4, .1 ] = .6
[ .4, .2 ] = .7
[ .4, .3 ] = .8^4 .9
[ .5, .1 ] = .7 .8^4 .9
[ .6, .2 ] = .8
[ .7, .1 ] = .9
```

Example H71E11

We construct the class 6 quotient q of $R(2,5)$ and then partially construct the class 7 quotient interactively to find out how many normal words having initial segment $q.1^2$ need to be considered when imposing the exponent law.

```
> F := FreeGroup(2);
> q := pQuotientProcess(F, 5, 6: Exponent := 5);
Lower exponent-5 central series for F
```

```
Group: F to lower exponent-5 central class 1 has order 5^2
Group: F to lower exponent-5 central class 2 has order 5^3
Group: F to lower exponent-5 central class 3 has order 5^5
Group: F to lower exponent-5 central class 4 has order 5^8
Group: F to lower exponent-5 central class 5 has order 5^10
Group: F to lower exponent-5 central class 6 has order 5^14
> StartNewClass(~q);
> Tails(~q);
> Consistency(~q);
> SetDisplayLevel(~q, 3);
> ExponentLaw(~q, 1, 6: InitialSegment := [<1, 2>], Trial := true);
0 Relations of class 1 will be collected
0 Relations of class 2 will be collected
Will collect power 5 of the following word: 1^2 2^1
1 Relation of class 3 will be collected
Will collect power 5 of the following word: 1^2 2^2
1 Relation of class 4 will be collected
Will collect power 5 of the following word: 1^2 2^3
Will collect power 5 of the following word: 1^2 5^1
2 Relations of class 5 will be collected
Will collect power 5 of the following word: 1^2 2^4
Will collect power 5 of the following word: 1^2 2^1 4^1
2 Relations of class 6 will be collected
```

Example H71E12

We demonstrate several of the remaining procedures in the course of interactively extending the class 6 quotient of the group

$$< a, b, c, d \mid (bc^{-1}d)^7, (cd^{-1})^7, (b, a) = c^{-1}, (c, a) = 1, (c, b) = d^{-1} >$$

to class 7.

```
> G := Group<a, b, c, d | (b * c^-1 * d)^7, (c * d^-1)^7, (b,a) = c^-1,
>                         (c,a) = 1, (c,b) = d^-1>;
> q := pQuotientProcess(G, 7, 6);
Lower exponent-7 central series for G
Group: G to lower exponent-7 central class 1 has order 7^2
Group: G to lower exponent-7 central class 2 has order 7^4
Group: G to lower exponent-7 central class 3 has order 7^6
Group: G to lower exponent-7 central class 4 has order 7^8
Group: G to lower exponent-7 central class 5 has order 7^11
Group: G to lower exponent-7 central class 6 has order 7^14
> StartNewClass(~q);
> Tails(~q);
> GeneratorStructure(q, 15, 34);
Class 7
15 is defined on [12, 1] = 2 1 2 2 1 2 1
```

```
16 is defined on [12, 2] = 2 1 2 2 1 2 2
17 is defined on [13, 1] = 2 1 2 2 2 2 1
18 is defined on [13, 2] = 2 1 2 2 2 2 2
19 is defined on [14, 1] = 1 1 1 1 1 1 1
20 is defined on [14, 2] = 1 1 1 1 1 1 2
21 is defined on 14^7 = 1 1 1 1 1 1 1
22 is defined on [9, 1] = 2 1 2 2 1 1
23 is defined on [10, 1] = 2 1 2 2 2 1
24 is defined on [11, 1] = 1 1 1 1 1 1
25 is defined on [11, 2] = 1 1 1 1 1 2
26 is defined on [8, 1] = 1 1 1 1 1
27 is defined on [8, 2] = 1 1 1 1 2
28 is defined on [5, 1] = 2 1 2 1
29 is defined on [6, 1] = 1 1 1 1
30 is defined on [6, 2] = 1 1 1 2
31 is defined on [3, 1] = 2 1 1
32 is defined on [4, 1] = 1 1 1
33 is defined on [4, 2] = 1 1 2
34 is defined on 2^7 = 2 2
> Jacobi(~q, 6, 6, 1);
Generator 26 is trivial
Jacobi was 6 6 1
> Jacobi(~q, 3, 2, 1);
Generator 28 is redundant
Jacobi was 3 2 1
> v := Collect(q,  [<29, 2>, <26, -3>]);
> v;
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 4, 0, 0, 2, 0, 0, 0, 0, 0 ]
> EcheloniseWord(~q, ~redgen);
Generator 29 is trivial
> Display(q, 1);
Group: G to lower exponent-7 central class 7 has order 7^34
```

Now we enforce the relations...

```
> CollectRelations(~q);
```

...and apply the consistency algorithm.

```
> Consistency(~q);
> Display(q, 1);
Group: G to lower exponent-7 central class 7 has order 7^36
> EliminateRedundancy(~q);
> Display(q, 1);
Group: G to lower exponent-7 central class 7 has order 7^19
```

71.5 Soluble Quotients

71.5.1 Introduction

This section presents a short overview towards the theory of the soluble quotient algorithm, the functions designed for computing soluble quotients and the functions designed for dealing with soluble quotient processes.

71.5.2 Construction

For a finite group G, the property of being soluble means that the derived series $G = G^{(0)} > G^{(1)} > ... > G^{(n)}$ terminates with $G^{(n)} = \langle 1 \rangle$. Each section $G^{(i)}/G^{(i+1)}$ is a finite abelian group, hence it can be identified with a $\mathbf{Z}G/G^{(i)}$-module M, where the action of $G/G^{(i)}$ on M is given by the conjugation action on $G^{(i)}/G^{(i+1)}$.

From module theory we see that there exists a series $M = M^{(0)} > M^{(1)} > ... > M^{(r_i)}$, where each section is an irreducible $GF(p)$ $G/G(i)$-module for some prime p. Using these series we obtain a refinement $G = G_{(0)} > G_{(1)} > ... > G_{(n)} = \langle 1 \rangle$ of the commutator series with the properties:

- The series is normal,
- Each section $G_{(i)}/G_{(i+1)}$ is elementary abelian of prime power order, and is irreducible as a $G/G_{(i)}$-module.
- If $G = H_{(0)} > H_{(1)} > ... > H_{(t)} = \langle 1 \rangle$ is another series with these properties, then $n = t$ and there exists a permutation $\pi \in S_n$ such that $G_{(i)}/G_{(i+1)}$ is isomorphic to $H_{(i\pi)}/H_{(i\pi+1)}$ (as $GF(p)$ modules).

Note: A PC-presentation defined by a further refinement of this series leads to a "conditioned" presentation. The converse is not always true, because the irreducibility of the sections is not required for a conditioned presentation.

The soluble quotient algorithm uses these series to construct soluble groups. Starting with the trivial group $G/G_{(0)}$, it successively chooses irreducible $G/G_{(i)}$ modules M_i and extensions

$$\zeta_i \in H^2(G/G_{(i)}, M_i)$$

which give rise to exact sequences

$$1 \to M_i \to G/G_{(i+1)} = G/G_{(i)}.M_i \to G/G_{(i)} \to 1.$$

To describe the algorithmic approach, we consider the following situation. Let G be a finite soluble group, M a normal elementary abelian subgroup of G such that M is a $H = G/M$ irreducible module. (The action of H on M is identified with the conjugation action of G/M on the subgroup M.) Then the relations of the group G have the shape

$$g_i^{p_i} = w_i m_i \text{ resp. } g_j^{g_i} = w_{ij} m_{ij}, m_i, m_{ij} \in M$$

where w_i, w_{ij} are the canonical representatives of H in G. Then $\bar{g_i}^{p_i} = \bar{w_i}$ resp. $\bar{g_j}^{\bar{g_i}} = \bar{w_{ij}}$ is a PC-presentation of H and the set of images $\left((g_i^{p_i-1}, g_i) \to m_i, (g_j, g_i) \mapsto m_{ij}\right)$ determines a unique element of the cocycle space $C^2(H, M)$.

According to the p-group situation, we call the system $t = (m_i, m_{ij})$, the tail defining $G = H.M_t$ as an extension of M by H.

Not every choice of a system $t = (m_i, m_{ij})$ defines an extension. To make sure that t corresponds to an element s of $C^2(G, M)$ it must satisfy a certain equation system, the so-called consistency equations (see Vaughan-Lee). These are linear homogeneous equations in M, so the solution space can be determined.

For the construction of soluble quotients we also have to find the epimorphism $\varepsilon : F \twoheadrightarrow G$. The existence of the epimorphism is obviously a restriction of the possible images G, and it can be checked simultaneously with constructing G. Let G be an extension $G = H.M$, where M is a H-module and H is a known soluble quotient $\delta : F \twoheadrightarrow H$. Then δ is uniquely determined by the images $\delta(f_i) = h_i, 1 \leq i \leq r$, where $\{f_1, \ldots, f_r\}$ is a generating set of F. We want to find a lift ε of δ, i.e. $\varepsilon(f) = \delta(f) \bmod M$ for all $f \in F$. This means that $\varepsilon(f_i) = h_i x_i$ for all $i \leq r$, where $x_i \in M$ are to be determined. Since ε will be a homomorphism, we require $\varepsilon(r_j(f_1, \ldots, f_r)) = 1_G$ for the defining relations r_j of F. This leads to a linear equation system for the variables $(x_1, \ldots, x_r) \in M^r$. We solve this equation system together with the consistency equations, hence we find a subspace S of $M^r \times H^2(H, M)$ of those extensions for which the epimorphism δ has a lift. Let us call an element of S an extended tail.

Let K be the minimal splitting field of M, i.e. the character field of the unique character of H which corresponds to M. Then M is obviously a KH-module and S is also a K-space. The space S has a K-subspace S_S of split extensions, i.e. the projection of S_S into $H^2(H, M)$ is only the trivial element. For any element $t \in S/S_S$ the corresponding map $\varepsilon_t : F \to H.M_t$ is necessarily surjective, hence defines a soluble quotient. Let s_1, s_2 be two elements of S/S_S and let $\varepsilon_i : F \twoheadrightarrow H.M_{s_i} =: G_i$ denote the corresponding soluble quotients. If s_1 and s_2 are K-linear dependent, the groups G_1 and G_2 will be isomorphic. Unfortunately, the converse is not true, even K-linear independent elements may lead to isomorphic groups. Nevertheless, if s_1 and s_2 are independent, the kernels of the epimorphisms will be different, hence one can iterate the lifting to

$$\varepsilon_{1,2} : F \twoheadrightarrow (G_1).M_{s_2} = (H.M_{s_1}).M_{s_2} = H.(M \oplus M)_{s_1 \cdot s_2}.$$

(The last equality can be read as a definition of $s_1 \cdot s_2$ in the righthand side term.)

The dimension $a = dim_K(S/S_S)$ is therefore characterised as the maximal multiplicity

$$a = max\left\{z \in \mathbf{Z}^+ \mid \exists \varepsilon : F \twoheadrightarrow H, M^z\right\}.$$

S_S itself also has a K-subspace S_C, for which the map

$$\varepsilon_s : F \to H.M_s, s \in S_C$$

is not surjective. The K-dimension $b = dim_K(S_S/S_C)$ is again characterised as the maximal multiplicity

$$b = \max\left\{z \in \mathbf{Z}^+ \mid \exists \varepsilon : F \twoheadrightarrow H, M^z\right\}.$$

Moreover, after taking a maximal extension $\varepsilon : F \twoheadrightarrow H.M^b$, M never has to be considered for split extensions again.

71.5.3 Calculating the Relevant Primes

A crucial step in finding finite soluble quotients $\varepsilon : F \twoheadrightarrow G$ is the calculation of the relevant primes, i.e. the prime divisors of $|G|$. This is the most time consuming part of the algorithm, so it is crucial to apply it as efficiently as possible, and any other information about possible prime divisors is very helpful. We do not explain this calculation in detail. However, to use the functions and options provided by MAGMA in a correct manner, we outline the idea of the calculation.

Let $\varepsilon : F \twoheadrightarrow G$ be a finite soluble quotient and let N denote the kernel of ε. If a module M allows an extension $\bar{\varepsilon} : F \twoheadrightarrow G.M$, then M must be a constituent of N/N', and the relevant primes are the prime divisors of N/N'. Again N/N' can be viewed as an F/N-module, hence an H-module. Therefore it is a direct sum $N/N' \cong M_{p_1} \oplus M_{p_2} \oplus \ldots \oplus \mathbf{Z}^d$, where the M_{p_i} are finite modules of order a power of the prime p_i. Let p not divide $|H|$. Then by Zassenhaus the extension $H.M_p$ must split. We consider an irreducible module M in the head of M_p, i.e. there is an H-epimorphism $M_p \twoheadrightarrow M$. Then there is a valid soluble quotient $\varepsilon : F \twoheadrightarrow H.M$ and the extension $H.M$ splits.

Now there exists an irreducible $\mathbf{Z}H$- module L such that M is a $GF(p)$-modular constituent of L/pL. Moreover, $\Delta : H \to GL(L)$ is the corresponding representation of H and any Δ-module will have this property.

Now using the theory of space groups, one can construct homomorphisms $\varepsilon_1 : F \twoheadrightarrow H.L$ and $\varepsilon_2 : H.L \twoheadrightarrow H.M$ such that the composition is surjective. Hence p can be detected in Δ. Let P_Δ denote the set of primes obtained from Δ. To find these primes, we have to know a set D of representatives of the irreducible rational representations of F/N, hence of H. The set of prime divisors of K/K' is a subset of

$$P = \bigcup_{\Delta \in D} P_\Delta \cup \{p \mid p \text{ prime}, p \mid |G|\}.$$

We want to point out the following:

- Usually the set of primes dividing $|K/K'|$ is a proper subset of P, because the primes dividing $|G|$ may or may not divide $|K/K'|$.

  Conversely, if p does not divide $|G|$, then there certainly exists a module M in characteristic p such that there is a soluble quotient $F \twoheadrightarrow G.M$, and the extension splits.
- The algorithm can also recognise infinite abelian sections. This means that already $\varepsilon_1 : F \twoheadrightarrow H.L$ is surjective. All primes are relevant and no maximal finite soluble quotient exists.
- Let $\phi : F \twoheadrightarrow H$ be a lift of $\varepsilon : F \twoheadrightarrow G$, i.e. G is a quotient of H: $\phi : H \twoheadrightarrow G$. If the relevant primes of G are known, these are also relevant primes of H, and only those representations Δ of H must be considered for which $\ker \phi \not\subset \ker \Delta$ is valid.

71.5.4 The Functions

MAGMA provides two different ways to calculate finite soluble quotients: a main function for the whole calculation, and a process which gives control over each individual step.

Let F be a finitely presented group.

`SolubleQuotient(F, n : parameters)`
`SolvableQuotient(F, n : parameters)`
`SolubleQuotient(F : parameters)`
`SolvableQuotient(F : parameters)`
`SolubleQuotient(F, P : parameters)`
`SolvableQuotient(F, P : parameters)`

Find a soluble quotient $\varepsilon : F \twoheadrightarrow G$ with a specified order. n must be a nonnegative integer. P must be a set of primes.

The three forms reflect possible information about the order of an expected soluble quotient. In the first form the order of G is given by n, if n is greater than zero. If n equals zero, nothing about the order is known and the relevant primes will be calculated completely.

The second form, with no n argument, is equivalent to the first with $n = 0$. This is a standard argument, and usually it is the most efficient way to calculate soluble quotients.

Note that, if $n > 0$ is not the order of the maximal finite soluble quotient, it may happen that no group of order n can be found, since an epimorphic image of size n may not be exhibited by the chosen series.

In the third form a set P of relevant primes is given. The algorithm calculates the biggest quotient such that the order has prime divisors only in P. P may have a zero as element, this is just for consistency reasons. It is equivalent to the first form with n equal zero.

The returned values are the group G and the epimorphism $\varepsilon : F \twoheadrightarrow G$. The third returned value is a sequence describing the series and modules by which G has been constructed.

The fourth value is a string which explain the reason for termination. The following list gives the termination conditions. The algorithm terminates normally if:

1. A quotient of the given order has been constructed.
2. A maximal quotient (with respect to the conditions given on the order) has been constructed.

The algorithm will be aborted and returns a warning if:

3. A bound on the length of a series or subseries has been hit.
4. A limit on the size of the quotient or a section has been hit.
5. The algorithm detects a free abelian section.

With the following options one can define abort conditions corresponding to the third and fourth item. The idea of all these conditions is to control the occurrence of infinite soluble quotients.

`SeriesLength` RngIntElt *Default :* 0

Limits the length of the chief series to r. For sag-series it is the nilpotent length of the series, for derived series it is the derived length. The default value of zero means no limit.

If the algorithm hits the limit, it gives a warning message and returns the last soluble quotient.

`SubseriesLength` RngIntElt *Default :* 0

Limits the length of a series in a section. If the sag-series is used, it is the length of the lower descending series. For the derived series, the limit is applied to the exponent of an element of a section with maximal prime power order. For example, if the limit is set to 3, the limit would be hit by a section of type C_8, but not by C_2^3 and not even by C_4^2.

`#QuotientSize` RngIntElt *Default :* 0

If the value n is bigger than zero, the algorithm returns if a quotient of order bigger or equal to n has been found. A warning message will be printed.

`#SectionSize` RngIntElt *Default :* 0

If the value s is bigger than zero, the algorithm returns if the order of a section is bigger than or equal to s. A warning message will be printed.

Note: Since an additive bound on a group order is not as meaningful as a multiplicative bound, the latter options are only useful as break conditions when the quotient gets too big for further calculations. The return quotient which hits such a bound is somewhat randomly chosen, since only a change in the order of checking modules may lead to other quotients.

With the following options the strategy of the algorithm and some subalgorithms can be chosen.

`#MSQ_Series` MonStgElt *Default :* "*sag*"

Determines the series which is used for the construction of soluble groups.

The default value is `"sag"`, since it is usually the most efficient choice. Of course, there is a value `"derived"`, exhibiting the derived series.

Another choice is `"lowercentral"`, choosing the lower central series. This restricts the algorithm to finite nilpotent quotients. The choice `"pcentral"` only exhibits p-groups as quotients.

The nilpotent resp. p-quotient algorithms are usually more efficient, so these options may only be useful to obtain additional information needed for a SQ-process.

`MSQ_PrimeSearchModus` RngIntElt *Default :* 3

Defines at what status of the algorithm the relevant prime search is called.

The possible choices reflect the different intentions of constructing a soluble quotient; for the general situation, (i.e. finding a finite soluble quotient without any information about relevant primes and check its maximality) this option makes only little difference in runtime behaviour.

0: No calculation of relevant primes. This is the default value, if the second argument does not request a prime calculation, e.g. if the order of the quotient or its relevant primes are known.

1: The relevant primes will be calculated after the soluble quotient algorithm (with the given input) terminates normally. Possibly new relevant primes are returned in a message. If, for example, the second argument is a set S (so no limit on the exponent) and no new relevant primes have been found, the maximality of the soluble quotient is proved.

2: As 1, but continues the algorithm when finding new relevant primes.

3: Perform the relevant prime calculation after a "main" step in the series, i.e. after completing a nilpotent section in a sag-series resp. a commutator section for the derived series. This is the default value, if prime calculation is required by the second argument. It is a good choice if one wants to construct large finite quotients quickly.

Note: This option can cause problems in case of a sag-series when infinite soluble quotients exist. For finite quotients, it seems to be the best choice.

4: Perform the relevant prime calculation after calculating an elementary abelian layer. This option is preferable, if the sag-series is used and an infinite soluble quotient is possible.

5: Perform the relevant prime calculation after each successful lift of a quotient. This option is preferable when infinite sections shall be detected as soon as possible (with respect to the chosen series).

`MSQ_ModulCalcModus` RngIntElt *Default :* 0

In the construction of soluble quotients using a sag-series one can restrict the number of modules by using tensor products and skew symmetric products. This can improve the performance in the case of big soluble quotients, for small quotients the overhead may invalidate the improvement. For other series this option has no meaning. The possible values are:

0: Do not apply this technique (default).

1: Fast version, just apply those parts which can be calculated quickly.

2: Full version, this is only recommended for "big" soluble quotients, i.e. quotients with long descending series in nilpotent sections.

`MSQ_CollectorModus` RngIntElt *Default :* 2

Defines the setup modus for the symbolic collector, i.e. the ratio of precalculation to dynamic setup:

0: Full precalculation, preferable for small soluble groups.

1: Partial precalculation (test version).

2: Dynamic setup (default).

The function also provides a general print option determining the amount of timings status information during the function call. Additionally there are some

verbose flags which determine the amount of information given about various sub-algorithms. If both a general print value and a verbose flag are given, the verbose flag has higher preference.

`Print` RngIntElt *Default :* 0

Determines what timing information and status messages are given during the calculation (0 = no printing, 5 = maximal information).

Verbose `MSQ_Messages` *Maximum :* 2

If set to 1, the sizes of new soluble quotients are printed.

Verbose `MSQ_PrimeSearch` *Maximum :* 15

Bitflag for print levels during the calculation of relevant primes:

1: Timings and statistics about the calculation of rational representations.

2: Timings for transforming rational into integral representations.

4: Timing for finding the relevant primes.

8: Printing of new relevant primes.

Verbose `MSQ_RepsCheck` *Maximum :* 3

1: Timing for checking extensions of modules.

2: Statistics about the modules to be checked.

Verbose `MSQ_RepsCalc` *Maximum :* 3

1: Timing information about the module calculation.

2: Statistics about the module calculation.

Verbose `MSQ_Collector` *Maximum :* 1

If set to 1, the timing for the setup of the symbolic collector is printed.

Verbose `MSQ_TraceFunc` *Maximum :* 2

Give messages about the main function calls (1) resp. most function calls (2) in the Magma language during the algorithm.

71.6 Bibliography

[**CDHW73**] John J. Cannon, Lucien A. Dimino, George Havas, and Jane M. Watson. Implementation and analysis of the Todd-Coxeter algorithm. *Math. Comp.*, 27:463–490, 1973.

[**Hav91**] G. Havas. Coset enumeration strategies. In *ISSAC'91*, pages 191–199. ACM Press, 1991.

[**NO96**] M. F. Newman and E. A. O'Brien. Application of computers to questions like those of Burnside. II. *Internat. J. Algebra Comput.*, 6(5):593–605, 1996.

[**Ram**] Colin Ramsay. ACE. URL:http://www.csee.uq.edu.au/~cram/.

72 POLYCYCLIC GROUPS

Chapter 72
POLYCYCLIC GROUPS

72.1 Introduction

In this chapter, we consider a class of finitely presented groups for which the word problem is solvable, the category of – possibly infinite – polycyclic groups. The corresponding MAGMA category is called `GrpGPC`. To distinguish this class from finite solvable groups described by a power-conjugate presentation (MAGMA category `GrpPC`, cf. Chapter 63), we use the term *general polycyclic group*.

An introduction to the theory of polycyclic groups and a collection of some basic algorithms can be found in [Sim94, ch. 9]. Unless otherwise mentioned, implementations of MAGMA functions are mostly based on ideas described in this reference.

72.2 Polycyclic Groups and Polycyclic Presentations

72.2.1 Introduction

A polycyclic group is a group G with a subnormal series $G = G_1 \triangleright G_2 \triangleright \ldots \triangleright G_{n+1} = 1$ in which each of the quotients G_i/G_{i+1} is cyclic. Every polycyclic group G has a presentation of the form

$$\begin{aligned} < a_1, \ldots, a_n \mid\ & a_i^{m_i} = w_{i,i} \quad (i \in I), \\ & a_j^{a_i} = w_{i,j} \quad (1 \leq i < j \leq n), \\ & a_j^{a_i^{-1}} = w_{-i,j} \quad (1 \leq i < j \leq n, i \notin I) > \end{aligned}$$

where

(i) $I \subseteq \{1, \ldots, n\}$,

(ii) $m_i > 1$ for $i \in I$, and

(iii) the words $w_{i,j}$ are of the form $w_{i,j} = a_{|i|+1}^{l(i,j,|i|+1)} \ldots a_n^{l(i,j,n)}$, with $0 \leq l(i,j,k) < m_k$ if $k \in I$.

Such a presentation is called a *polycyclic presentation* for G. For $1 \leq i \leq n$, let G_i be the subgroup of G generated by $a_i, \ldots, a_n$ and define G_{n+1} to be the trivial group. The presentation is called *consistent*, if $|G_i/G_{i+1}| = m_i$ whenever $i \in I$ and G_i/G_{i+1} is infinite whenever $i \notin I$. The generators $a_1, \ldots a_n$ are referred to as *polycyclic generators* (*pc-generators*) for G and the values m_i $(i \in I)$ are called the corresponding *pc-exponents*.

In MAGMA, the user can define a polycyclic group by providing a consistent polycyclic presentation or by using one of the existing category transfer functions.

Given a consistent polycyclic presentation for G, every element a of G can be written uniquely in the form $a = a_1^{e_1} \ldots a_n^{e_n}$, where the e_i are integers satisfying $0 \leq e_i < m_i$

if $i \in I$. This form is called *normal form*. There exists an algorithm (the *collection algorithm*), which given an arbitrary word in the generators $a_1, \ldots, a_n$, will determine the corresponding normal word. MAGMA uses a collection algorithm written by Volker Gebhardt. The cost of collection for this algorithm grows logarithmically in a bound on the absolute values of the exponents e_i occurring during the collection [Geb02].

Infinite polycyclic groups are a comparatively new topic in computational group theory and the number of available algorithms is much smaller than in the case of finite polycyclic groups. For this reason, the data type `GrpPC` (cf. Chapter 63) should be preferred for finite polycyclic groups.

72.2.2 Specification of Elements

Elements of polycyclic groups are words. A *word* is defined inductively as follows:

(i) A generator is a word;

(ii) The expression (u) is a word, where u is a word;

(iii) The product $u * v$ of the words u and v is a word;

(iv) The conjugate u^v of the word u by the word v is a word (u^v expands into the word $v^{-1} * u * v$);

(v) The power of a word u^n, where u is a word and n is an integer, is a word;

(vi) The commutator (u, v) of the words u and v is a word ((u, v) expands into the word $u^{-1} * v^{-1} * u * v$).

`G ! Q`

Given the polycyclic group G and a sequence Q of length n, containing integers $e_1 \ldots e_n$, where $0 \leq e_i < m_i$ if $i \in I$, construct the element x of G given by

$$x = a_1^{e_1} \ldots a_n^{e_n}.$$

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity element of the polycyclic group G.

72.2.3 Access Functions for Elements

Throughout this subsection, G will be a polycyclic group with pc-generators $a_1, \ldots, a_n$.

`ElementToSequence(x)`

`Eltseq(x)`

Given an element x belonging to the polycyclic group G, where $x = a_1^{e_1} \ldots a_n^{e_n}$ in normal form, return the sequence Q of n integers defined by $Q[i] = e_i$, for $i = 1, \ldots, n$.

`LeadingTerm(x)`

Given an element x of a polycyclic group G with n pc-generators, where x is of the form $a_1^{e_1} \dots a_n^{e_n}$, return $a_i^{e_i}$ for the smallest i such that $e_i > 0$. If x is the identity of G, then the identity is returned.

`LeadingGenerator(x)`

Given an element x of a polycyclic group G with n pc-generators, where x is of the form $a_1^{e_1} \dots a_n^{e_n}$, return a_i for the smallest i such that $e_i > 0$. If x is the identity of G, then the identity is returned.

`LeadingExponent(x)`

Given an element x of a polycyclic group G with n pc-generators, where x is of the form $a_1^{e_1} \dots a_n^{e_n}$, return e_i for the smallest i such that $e_i > 0$. If x is the identity of G, then 0 is returned.

`Depth(x)`

Given an element x of a polycyclic group G with n pc-generators, where x is of the form $a_1^{e_1} \dots a_n^{e_n}$, return the smallest i such that $e_i > 0$. If x is the identity of G, then $n + 1$ is returned.

`Depth` returns the maximal value of i, such that $x \in G_i$.

72.2.4 Arithmetic Operations on Elements

Throughout this subsection, G will be a polycyclic group with pc-generators $a_1, \dots, a_n$.

`g * h`

Product of the element g and the element h, where g and h belong to some common subgroup G of a polycyclic group U. If g and h are given as elements belonging to the same proper subgroup G of U, then the result will be returned as an element of G; if g and h are given as elements belonging to distinct subgroups H and K of U, then the product is returned as an element of G, where G is the smallest subgroup of U known to contain both elements.

`g *:= h`

Replace g with the product of element g and element h.

`g ^ n`

The n-th power of the element g, where n is a positive or negative integer.

`g ^:= n`

Replace g with the n-th power of the element g.

`g / h`

Quotient of the element g by the element h, i.e. the element $g * h^{-1}$. Here g and h must belong to some common subgroup G of a polycyclic group U. The rules for determining the parent group of g/h are the same as for $g * h$.

`g /:= h`

Replace g with $g * h^{-1}$.

`g ^ h`

Conjugate of the element g by the element h, i.e. the element $h^{-1} * g * h$. Here g and h must belong to some common subgroup G of a polycyclic group U. The rules for determining the parent group of g^h are the same as for $g * h$.

`g ^:= h`

Replace g with the conjugate of the element g by the element h.

`(g1, ..., gn)`

Given the n words $g_1, \ldots, g_n$ belonging to some common subgroup G of a polycyclic group U, compute the *left-normed* commutator of $g_1, \ldots, g_n$. This is defined inductively as follows: $(g_1, g_2) = g_1^{-1} * g_2^{-1} * g_1 * g_2$ and $(g_1, \ldots, g_n) = ((g_1, \ldots, g_{n-1}), g_n)$.

If $g_1, \ldots, g_n$ are given as elements belonging to the same proper subgroup G of U, then the result will be returned as an element of G; if $g_1, \ldots, g_n$ are given as elements belonging to distinct subgroups of U, then the product is returned as an element of G, where G is the smallest subgroup of U known to contain all elements.

72.2.5 Operators for Elements

`Order(x)`

The order of the element x.

`Parent(x)`

The parent group G of the element x.

72.2.6 Comparison Operators for Elements

`g eq h`

Given elements g and h belonging to a common polycyclic group, return `true` if g and h are the same element, `false` otherwise.

`g ne h`

Given elements g and h belonging to a common polycyclic group, return `true` if g and h are distinct elements, `false` otherwise.

`IsIdentity(g)`

`IsId(g)`

Returns `true` if g is the identity element, `false` otherwise.

72.2.7 Specification of a Polycyclic Presentation

`quo< GrpGPC : F | R : parameters >`

`Check` BoolElt *Default :* `true`

Given a free group F of rank n with generating set X, and a collection R of polycyclic relations on X, construct the polycyclic group G defined by the polycyclic presentation $< X|R >$.

The construct R denotes a list of polycyclic relations. Thus, an element of R must be one of:

(a) A *power relation* $a_i^{m_i} = w_{i,i}$, $i \in I \subseteq \{1, \ldots, n\}$, where $m_i > 1$ is an integer, and $w_{i,i}$ is $Id(F)$ or a word in the generators $a_{i+1}, \ldots, a_n$ for $i < n$, and $w_{i,i} = Id(F)$ for $i = n$.

(b) A *conjugate relation* $a_j^{a_i^e} = w_{e\cdot i,j}$, $1 \leq i < j \leq n$, where $w_{e\cdot i,j}$ is a word in the generators $a_{i+1}, \ldots, a_n$, $e = 1$ if $i \in I$ and $e \in \{-1, 1\}$ if $i \notin I$.

(c) A *power* $a_i^{m_i}$, $i \in I \subseteq \{1, \ldots, n\}$, where $m_i > 1$ is an integer, which is treated as the *power relation* $a_i^{m_i} = Id(F)$.

(d) A set of (a) – (c).

(e) A sequence of (a) – (c).

Note the following points:

(i) If there is no power relation for some generator a_i, $i = 1, \ldots, n$ (i.e. $i \notin I$), conjugate relations $a_j^{a_i^{-1}} = w_{-i,j}$ must be present for $i < j \leq n$ (except for pairs of commuting generators). If there is a power relation for a_i, on the other hand, conjugate relations involving a_i^{-1} are not permitted.

(ii) Conjugate relations for pairs of commuting generators may be omitted. If no conjugate relations are given for a certain pair of generators, the corresponding generators are assumed to commute.

(iii) The words $w_{i,j}$ must be in normal form.

(iv) The presentation must be consistent. In particular, right hand sides of conjugate relations must not be the empty word.

This constructor returns a polycyclic group because the category `GrpGPC` is stated. If no category were stated, it would return an fp-group.

The parameter `Check` may be used to indicate whether or not the presentation should be checked for consistency. Disabling this check can be useful for avoiding runtime errors if the constructor is called in user written loops or functions. The boolean valued function `IsConsistent` is provided to check presentations obtained

with disabled consistency check. It should be noted that the results of working with an inconsistent presentation are undefined, hence it is strongly advised to enable the consistency check in the constructor or to use the function IsConsistent.

The natural homomorphism from $F \to G$ is returned as second return value.

PolycyclicGroup< x$_1$, ..., x$_n$ | R : *parameters* >

Check	BoolElt	*Default :* true
Class	MonStgElt	*Default :*

Construct the polycyclic group G defined by the consistent polycyclic presentation $< x_1, \ldots, x_n | R >$.

The construct $x_1, \ldots, x_n$ defines names for the generators of G that are local to the constructor, i.e. they are used when writing down the relations to the right of the bar. However, no assignment of names to these generators is made. If the user wants to refer to the generators by these (or other) names, then the *generators assignment* construct must be used on the left hand side of an assignment statement.

The construct R denotes a list of polycyclic relations. The syntax and semantics for the relations clause is identical to that appearing in the quo-constructor above.

A map f from the free group on $x_1, \ldots, x_n$ to G is returned as second return value.

The parameter Check may be used as described for the quo-constructor.

If R is both, a valid power-conjugate presentation for a finite soluble group (cf. Chapter 63) and a consistent polycyclic presentation, this constructor by default returns a group in the category GrpPC. To force creation of a group in the category GrpGPC, the parameter Class must be set to "GrpGPC" in these situations.

Example H72E1

(1) Consider the infinite polycyclic group defined by the presentation

$$< a, b, c \mid b^a = b * c, (a, c), (b, c) > .$$

Starting from a free group and giving the relations in the form of a sequence, this presentation would be specified as follows:

```
> F<a,b,c> := FreeGroup(3);
> rels := [ b^a = b*c, b^(a^-1) = b*c^-1 ];
> G<a,b,c> := quo< GrpGPC : F | rels >;
> G;
GrpGPC : G of infinite order on 3 PC-generators
PC-Relations:
    b^a = b * c,
    b^(a^-1) = b * c^-1
```

Note, that the relation $b^{a^{-1}} = b * c^{-1}$ has to be included, although it can be derived from the relations $b^a = b * c$ and (a, b).

(2) The infinite dihedral group is obtained as epimorphic image of the free group of rank two as follows:

```
> F<a,b> := FreeGroup(2);
> D<u,v>, pi := quo<GrpGPC: F | a^2, b^a = b^-1>;
> D;
GrpGPC : D of infinite order on 2 PC-generators
PC-Relations:
    u^2 = Id(D),
    v^u = v^-1
> pi;
Mapping from: GrpFP: F to GrpGPC: D
```

(3) We create an element e of the group D defined above from a sequence of coefficients and extract both its leading generator and its leading exponent.

```
> e := D ! [1,42];
> e;
u * v^42
> gen := LeadingGenerator(e);
> gen;
u
> Parent(gen);
GrpGPC : D of infinite order on 2 PC-generators
PC-Relations:
    u^2 = Id(D),
    v^u = v^-1
> exp := LeadingExponent(e);
> exp;
1
```

We obtain an element of depth 2 from e, by replacing e with its quotient by the appropriate power of the leading generator.

```
> e /:= gen^exp;
> Depth(e);
2
> e;
v^-42
> ElementToSequence(e);
[ 0, -42 ]
```

Example H72E2

Using the constructor PolycyclicGroup with different values of the parameter Class, we construct the dihedral group of order 10 first as a finite soluble group given by a power-conjugate presentation (GrpPC) and next as a general polycyclic group (GrpGPC). Note that the presentation $\langle a, b \mid a^2, b^5, b^a = b^4 \rangle$ is both a valid power-conjugate presentation and a consistent polycyclic

presentation, so we have to set the parameter `Class` to `"GrpGPC"` if we want to construct a group in the category `GrpGPC`.

```
> G1<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 >;
> G1;
GrpPC : G1 of order 10 = 2 * 5
PC-Relations:
    a^2 = Id(G1),
    b^5 = Id(G1),
    b^a = b^4
> G2<a,b> := PolycyclicGroup< a,b | a^2, b^5, b^a=b^4 : Class := "GrpGPC">;
> G2;
GrpGPC : G2 of order 10 = 2 * 5 on 2 PC-generators
PC-Relations:
    a^2 = Id(G2),
    b^5 = Id(G2),
    b^a = b^4
```

We construct the infinite dihedral group as a group in the category `GrpGPC` from a consistent polycyclic presentation. We do not have to use the parameter `Class` in this case.

```
> G3<a,b> := PolycyclicGroup< a,b | a^2, b^a=b^-1>;
> G3;
GrpGPC : G3 of infinite order on 2 PC-generators
PC-Relations:
    a^2 = Id(G3),
    b^a = b^-1
```

The presentation $\langle a, b \,|\, a^2, b^4, b^a = b^3 \rangle$ is not a valid power-conjugate presentation for the dihedral group of order 8, since the exponent of b is not prime. However, it is a consistent polycyclic presentation. Consequently, the constructor `PolycyclicGroup` without specifying a value for the parameter `Class` returns a group in the category `GrpGPC`.

```
> G4<a,b> := PolycyclicGroup< a,b | a^2, b^4, b^a=b^3 >;
> G4;
GrpGPC : G4 of order 2^3 on 2 PC-generators
PC-Relations:
    a^2 = Id(G3),
    b^4 = Id(G3),
    b^a = b^3
```

72.2.8 Properties of a Polycyclic Presentation

`IsConsistent(G)`

Returns `true` if the stored presentation for G is consistent, `false` otherwise.

`IsIdenticalPresentation(G, H)`

Returns `true` if the polycyclic presentations for G and H are identical, `false` otherwise.

`PresentationIsSmall(G)`

Returns `true` if only small integers occur in the presentation of G, `false` otherwise. The category transfer functions `FPGroup` and `PCGroup` currently support only groups with a small presentation.

72.3 Subgroups, Quotient Groups, Homomorphisms and Extensions

72.3.1 Construction of Subgroups

`sub< G | L >`

Construct the subgroup H of the polycyclic group G generated by the elements specified by the terms of the *generator list* L.
A term $L[i]$ of the generator list may consist of any of the following objects:

(a) An element liftable to G;

(b) A sequence of integers representing an element of G;

(c) A subgroup of G;

(d) A set or sequence of (a), (b), or (c). The collection of words and groups specified by the list must all belong to the group G and H will be constructed as a subgroup of G.

The generators of H consist of the words specified directly by terms of $L[i]$ together with the stored generating words for any groups specified by terms of $L[i]$. Repetitions of an element and occurrences of the identity element are removed.

The inclusion map from H to G is returned as a second value.

`ncl< G | L >`

Construct the subgroup N of the polycyclic group G as the normal closure of the subgroup generated by the elements specified by the terms of the *generator list* L.

The possible forms of a term $L[i]$ of the generator list are the same as for the `sub`-constructor.

The inclusion map from N to G is returned as a second value.

72.3.2 Coercions Between Groups and Subgroups

`G ! g`

Given an element g belonging to the subgroup H of the group G, rewrite g as an element of G.

`H ! g`

Given an element g belonging to the group G, and given a subgroup H of G containing g, rewrite g as an element of H.

`K ! g`

Given an element g belonging to the group H, and a group K, such that H and K are subgroups of G, and both H and K contain g, rewrite g as an element of K.

`InclusionMap(G, H)`

The map from the subgroup H of G to G.

Example H72E3

Consider again the infinite polycyclic group G defined by the polycyclic presentation

$$< a, b, c \mid b^a = b * c, (a, c), (b, c) > .$$

```
> F<a,b,c> := FreeGroup(3);
> rels := [ b^a = b*c, b^(a^-1) = b*c^-1 ];
> G<a,b,c> := quo< GrpGPC : F | rels >;
> G;
GrpGPC : G of infinite order on 3 PC-generators
PC-Relations:
    b^a = b * c,
    b^(a^-1) = b * c^-1
```

Using the function PCGenerators which is described later, the groups $G_1, \ldots, G_4$ and the corresponding inclusion maps can be defined as follows:

```
> G_ := []; incl_ := [ PowerStructure(Map) | ];
> for i := 1 to #PCGenerators(G)+1 do
>   G_[i], incl_[i] := sub< G | [ g : g in PCGenerators(G) |
>                                       Depth(g) ge i ] >;
> end for;
> for i := 1 to #G_ do
>   printf "G_%o = <%o>", i, {@ G!x : x in
>                   PCGenerators(G_[i]) @}; print "";
> end for;
G_1 = <{@ a, b, c @}>
G_2 = <{@ b, c @}>
G_3 = <{@ c @}>
```

```
G_4 = <{@ @}>
```

Note that we must set the universe of the sequence `incl_` to `PowerStructure(Map)` manually, since we want to store maps which do not have a common domain. If we failed to do this, the universe would be chosen automatically to be the set of all maps from G_1 to G when the first map is inserted into the sequence. Inserting the second map, which does not have the domain G_1, would then cause a runtime error.

72.3.3 Construction of Quotient Groups

`quo< G | L >`

Construct the quotient Q of the polycyclic group G by the normal subgroup N, where N is the smallest normal subgroup of G containing the elements specified by the terms of the *generator list* L.

The possible forms of a term $L[i]$ of the generator list are the same as for the `sub`-constructor.

The quotient group Q and the corresponding natural homomorphism $f : G \to Q$ are returned.

`G / N`

Given a normal subgroup N of the polycyclic group G, construct the quotient of G by N.

72.3.4 Homomorphisms

For a general description of homomorphisms, we refer to Chapter 16. This section describes some special aspects of homomorphisms the domain of which is a polycyclic group.

72.3.4.1 General remarks

The kernel of a homomorphism with a domain of type `GrpGPC` can be computed using the function `Kernel`, if the codomain is of one of the types `GrpGPC`, `GrpPC` (cf. Chapter 63), `GrpAb` (cf. Chapter 69), `GrpPerm` (cf. Chapter 58), `ModAlg` or `ModGrp` (cf. Chapter 89) or if the codomain is of the type `GrpMat` (cf. Chapter 59) and the image is finite.

In particular, preimages of substructures can be computed in these situations. The kernel of a map will be computed automatically, if the preimage of a substructure of the codomain is to be computed.

The kernel (and hence preimages of substructures) may also be computable, if the codomain is of the type `GrpFP` (cf. Chapter 70) and the domain is nilpotent.

72.3.4.2 Construction of Homomorphisms

`hom< P -> G | S : parameters >`

Returns the homomorphism from the polycyclic group P to the group G defined by the assignment S. S can be the one of the following:

(i) A list, sequence or indexed set containing the images of the n polycyclic generators $P.1, \ldots, P.n$ of P. Here, the i-th element of S is interpreted as the image of $P.i$, i.e. the order of the elements in S is important.

(ii) A list, sequence, enumerated set or indexed set, containing r tuples $< x_i, y_i >$ or arrow pairs $x_i -> y_i$, where $x_i \in P$, $y_i \in G$ $(i = 1, \ldots, r)$ and the set $\{x_1, \ldots, x_r\}$ is a generating set for P. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important. Note that the preimages x_i need not be the polycyclic generators of P.

If the data type of the codomain supports element arithmetic and element comparison, by default the constructed homomorphism is checked by verifying that the would-be images of the polycyclic generators satisfy the defining relations of P and that this assignment is consistent with the assignments made by the user. In this case, it is assured that the returned map is a well-defined homomorphism with the desired images. The most important situation in which it is not possible to perform checking is the case in which the domain is a finitely presented group (`FPGroup`; cf. Chapter 70) which is not free. Checking may be disabled using the parameter `Check`.

`Check` BoolElt *Default :* `true`

If `Check` is set to `false`, checking of the homomorphism is disabled.

72.3.5 Construction of Extensions

`DirectProduct(G, H)`

The direct product K of the polycyclic groups G and H. The second value returned is a sequence containing the inclusion maps $I_G : G \to K$ and $I_H : H \to K$. The third value returned is a sequence containing the projection maps $P_G : K \to G$ and $P_H : K \to H$.

72.3.6 Construction of Standard Groups

A number of functions are provided which construct polycyclic presentations for various standard groups.

`AbelianGroup(GrpGPC, Q)`

Construct the abelian group defined by the sequence $Q = [n_1, \ldots, n_r]$ as a polycyclic group. The entries n_i may be either zero, indicating an infinite cyclic factor, or integers greater than 1. The function returns the polycyclic group which is the direct product of the cyclic groups $Z_1 \times \cdots \times Z_r$, where Z_i is a cyclic group of infinite order if $n_i = 0$, or a cyclic group of order n_i if $n_i > 1$.

`CyclicGroup(GrpGPC, n)`

For $n > 0$, the cyclic group of order n is returned, while for $n = 0$, the infinite cyclic group is returned as a polycyclic group.

`DihedralGroup(GrpGPC, n)`

For $n \geq 3$, the dihedral group of order $2n$ is returned, while for $n = 0$, the infinite dihedral group is returned as a polycyclic group.

`ElementaryAbelianGroup(GrpGPC, p, n)`

Given a prime p and a positive integer n, construct the elementary abelian group of order p^n as a polycyclic group.

`ExtraSpecialGroup(GrpGPC, p, n : parameters)`

Given a prime p and a small positive integer n, construct an extra-special group G of order p^{2n+1} as a polycyclic group. The isomorphism type of G can be selected using the parameter `Type`.

`Type` MonStgElt *Default :* "+"

Possible values for this parameter are "+" (default) and "−".

If `Type` is set to "+", the function returns for $p = 2$ the central product of n copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p.

If `Type` is set to "−", the function returns for $p = 2$ the central product of a quaternion group of order 8 and $n - 1$ copies of the dihedral group of order 8, and for $p > 2$ it returns the unique extra-special group of order p^{2n+1} and exponent p^2.

`FreeAbelianGroup(GrpGPC, n)`

Given a positive integer n, construct the free abelian group of rank n as a polycyclic group.

`FreeNilpotentGroup(r, e)`

The free nilpotent group of rank r and class e is returned as a polycyclic group.

Example H72E4

Consider the dihedral group D_4.

```
> G<a,b> := DihedralGroup(GrpGPC, 4);
```

We obtain a well-defined homomorphism f of G onto a group $A = \mathbf{Z}_2 \times \mathbf{Z}_2$, by assigning two generators of A as the images of a and ab, respectively. Note that in the definition of f we can use the generating set $\{a, ab\}$ of G, instead of the defining polycyclic generating sequence of G.

```
> A<u,v> := AbelianGroup([2,2]);
```

```
> f := hom< G->A | a->u, a*b->v >;
```

We compute the kernel K of f and express the generators of K as elements of G, using the function PCGenerators described later.

```
> K := Kernel(f);
> PCGenerators(K, G);
{@ b^2 @}
```

Example H72E5

A polycyclic representation for the group $D_3 \times D_\infty$ may be obtained as follows:

```
> G1<a,b> := DihedralGroup(GrpGPC, 3);
> G2<u,v> := DihedralGroup(GrpGPC, 0);
> D, incl, proj := DirectProduct(G1, G2);
> D;
GrpGPC : D of infinite order on 4 PC-generators
PC-Relations:
    D.1^2 = Id(D),
    D.2^3 = Id(D),
    D.3^2 = Id(D),
    D.2^D.1 = D.2^2,
    D.4^D.3 = D.-4
```

Using the inclusion maps returned by DirectProduct, we define a subgroup and compute the quotient by its normal closure:

```
> S := sub<D| incl[1](a)*incl[2](u), incl[1](b)*incl[2](v)>;
> S;
GrpGPC : S of infinite order on 3 PC-generators
PC-Relations:
    S.1^2 = Id(S),
    S.2^3 = S.3,
    S.2^S.1 = S.2^2 * S.-3,
    S.3^S.1 = S.-3
> Q, pi := quo<D|S>;
> Q;
GrpGPC : Q of order 2 on 1 PC-generators
PC-Relations:
    Q.1^2 = Id(Q)
> Q.1 @@ pi;
D.3
```

72.4 Conversion between Categories

This section describes category transfers from polycyclic groups to other MAGMA categories of groups. For category transfers to permutation groups (cf. Chapter 58) the functions CosetAction and CosetImage can be used. For category transfers to matrix groups (cf. Chapter 59) we refer to section 72.12.

AbelianGroup(G)

A GrpAb (cf. Chapter 69) representation A of the abelian polycyclic group G and the isomorphism from G to A.

FPGroup(G)

A GrpFP (cf. Chapter 70) representation F of G and the isomorphism from F to G. This category transfer is currently only possible provided that all exponents occurring in the presentation of G are small integers. This can be checked by means of the function PresentationIsSmall.

PCGroup(G)

A GrpPC (cf. Chapter 63) representation F of the finite polycyclic group G and the isomorphism from G to F.

This category transfer is currently only possible provided that all exponents occurring in the presentation of G are small integers. This can be checked by means of the function PresentationIsSmall.

GPCGroup(G)

A GrpGPC representation P of the solvable group G and the isomorphism from G to P. The group G can be of one of the following types: GrpPerm (cf. Chapter 58), GrpMat (cf. Chapter 59), GrpAb (cf. Chapter 69), GrpPC (cf. Chapter 63). Currently G must be finite, if it is of type GrpMat.

Example H72E6

We define a finite, solvable matrix group and convert it to a (general) polycyclic group G.

```
> a := GL(2,3) ! [1,1,0,1];
> b := GL(2,3) ! [0,1,1,0];
> M := sub<Parent(a)|a,b>;
> IsSolvable(M);
true
> IsFinite(M);
true
> G, f := GPCGroup(M);
```

We now compute the direct product D of G and the infinite dihedral group H, define a subgroup S of D, its normal closure N and construct the quotient Q of D by N.

```
> H<u,v> := DihedralGroup(GrpGPC, 0);
> D, incl, proj := DirectProduct(G, H);
```

```
> S := sub<D | incl[1](f(a*b)), incl[2]((u,v)^2)>;
> N := ncl<D|S>;
> Q := D/N;
> Q;
GrpGPC : Q of order 2^3 on 2 PC-generators
PC-Relations:
    Q.1^2 = Id(Q),
    Q.2^4 = Id(Q),
    Q.2^Q.1 = Q.2^3
```

Since Q is finite, it can be transformed into a group of type `GrpPC` using the function `PCGroup`. This should (in non-trivial examples) be done, if further computations with it are intended.

```
> Q_ := PCGroup(Q);
> Q_;
GrpPC : Q_ of order 8 = 2^3
PC-Relations:
    Q_.2^2 = Q_.3,
    Q_.2^Q_.1 = Q_.2 * Q_.3
```

72.5 Access Functions for Groups

The functions described here provide access to basic information stored for a polycyclic group G.

`G . i`

The i-th polycyclic generator for G if $i > 0$, the inverse of the $|i|$-th polycyclic generator for G if $i < 0$ and the identity element of G if $i = 0$.

`Generators(G)`
`PCGenerators(G)`

An indexed set containing the polycyclic generators of G.

`Generators(H, G)`
`PCGenerators(H, G)`

An indexed set containing the polycyclic generators of H as elements of G.

`NumberOfGenerators(G)`
`Ngens(G)`
`NumberOfPCGenerators(G)`
`NPCgens(G)`

The number of polycyclic generators for the polycyclic group G.

`PCExponents(G)`

The orders of the cyclic factors in the polycyclic series defined by the polycyclic presentation of G. The orders are returned in a sequence Q. $|G_i/G_{i+1}| = m_i = Q[i]$ if $Q[i] > 0$ and G_i/G_{i+1} is infinite (i.e. $i \notin I$) if $Q[i] = 0$.

`HirschNumber(G)`

The Hirsch number of G, i.e. the number of infinite cyclic factors in the polycyclic series defined by the polycyclic presentation of G.

The Hirsch number of G is equal to $n - |I|$, i.e. to the number of polycyclic generators of G for which there is no power relation. A polycyclic group G is finite if and only if its Hirsch number is 0.

72.6 Set-Theoretic Operations in a Group

72.6.1 Functions Relating to Group Order

`FactoredIndex(G, H)`

Given a group G and a subgroup H of G of finite index, return the factored index of H in G

`FactoredOrder(G)`

The factored order of the finite group G.

`Index(G, H)`

The index of the subgroup H in the group G, returned as an ordinary integer.

`Order(G)`
`#G`

The order of the group G, returned as an ordinary integer.

72.6.2 Membership and Equality

`g in G`

Given an element g and a group G, return `true` if g is an element of G, `false` otherwise.

`g notin G`

Given an element g and a group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G`

Given an group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G`

Given a group G and a set S of elements belonging to a group H, where G and H have some covering group, return `true` if S is not a subset of G, `false` otherwise.

`H subset G`

Given groups G and H, having some covering group, return `true` if H is a subgroup of G, `false` otherwise.

`H notsubset G`

Given groups G and H, having some covering group, return `true` if H is not a subgroup of G, `false` otherwise.

`G eq H`

Given groups G and H, having some covering group, return `true` if G and H are the same group, `false` otherwise.

`G ne H`

Given groups G and H, having some covering group, return `true` if G and H are distinct groups, `false` otherwise.

72.6.3 Set Operations

`Representative(G)`

`Rep(G)`

A representative element of G.

`RandomProcess(G)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 100

Create a process to generate pseudo-randomly chosen elements from the group G. The process uses an 'expansion' procedure to construct a set of elements corresponding to fairly long words in the generators of G [CLGM+95]. At all times, N elements forming a generating set for G are stored. Here, N is the maximum of $n + 1$ and the specified value for `Slots`, where n is the number of generators of G. Initially, these are just the generators of G and products of pairs of generators of G. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses an element previously stored by the process as the new random element. The process then replaces this stored element with the product of this element and another one of the stored elements (on the left or the right). Setting `Scramble:= m` causes m such operations to be performed before the process is returned.

Care should be taken when trying to apply this function to infinite polycyclic groups. Firstly, the computations may take a considerable amount of time and secondly, the quality of the pseudo-random element generator may be extremely poor, depending on the required properties of the sequence of pseudo-random elements.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the group G, construct a pseudo-random element of G by forming a random product over the expanded generating set currently stored by the process. The remarks concerning random elements of infinite polycyclic groups given in the description of `RandomProcess` apply here.

`Random(G)`

`Random(G, max)`

An element, pseudo-randomly chosen, from the group G. An exponent vector in normal form is chosen at random. Exponents of polycyclic generators for which there is no power relation, are chosen to have absolute value less or equal to `max`. A default value for `max` is 10.

It should be kept in mind that the distribution of the elements returned by `Random` is uniform only in the case that G is finite.

72.7 Coset Spaces

`CosetTable(G, H)`

The (right) coset table for G over the subgroup H of finite index, relative to the polycyclic generators. This is defined to be a map

$$\{1, \ldots, |G : H|\} \times G \to \{1, \ldots, |G : H|\}$$

describing the action of G on the enumerated set of right cosets of H in G by right multiplication.

The underlying set of right coset representatives is identical to the right transversal returned by `Transversal` and `RightTransversal` and the same enumeration of the elements is used.

`Transversal(G, H)`

`RightTransversal(G, H)`

Given a group G and a subgroup H of G, this function returns:

(a) An indexed set of elements T of G forming a right transversal for G over H. The right transversal and its enumeration are identical to those internally used by the function `CosetTable`.

(b) The corresponding transversal mapping $\phi : G \to T$. If $T = [t_1, \ldots, t_r]$ and g in G, ϕ is defined by $\phi(g) = t_i$, where $g \in H * t_i$.

Example H72E7

We compute a right transversal of a subgroup H of the infinite dihedral group G.

```
> G<a,b> := DihedralGroup(GrpGPC, 0);
> H := sub<G|a*b, b^10>;
> Index(G, H);
10
> RT, transmap := Transversal(G, H);
> RT;
{@ Id(G), b^-1, b^-2, b^-3, b^-4, b^-5, b^-6, b^-7, b^-8, b^-9 @}
> transmap;
Mapping from: GrpGPC: G to SetIndx: RT
```

From this a left transversal is easily obtained:

```
> LT := {@ x^-1 : x in RT @};
> LT;
{@ Id(G), b, b^2, b^3, b^4, b^5, b^6, b^7, b^8, b^9 @}
```

We construct the coset table and define a function $RT \times G \to RT$, describing the action of G on the set of right cosets of H in G.

```
> ct := CosetTable(G, H);
> action := func< r, g | RT[ct(Index(RT, r), g)] >;
> action(Id(G), b);
b^-9
```

I.e. $H * b = Hb^{-9}$.

```
> action(b^-4, a*b);
b^-6
```

I.e. $Hb^{-4} * (ab) = Hb^{-6}$.

Note that the definition of the function `action` relies on the fact that the computed right transversal and its enumeration are identical to those internally used by the function `CosetTable`.

`CosetAction(G, H)`

Given a subgroup H of the group G of finite index, construct the permutation representation of G, induced by the action of G on the set of (right) cosets of H in G. The function returns:

(a) The permutation representation $f : G \to L \leq \mathrm{Sym}(|G : H|)$, induced by the action of G on the set of (right) cosets of H in G;

(b) The epimorphic image L of G under the representation f;

(c) The kernel K of the representation f.

`CosetImage(G, H)`

Given a subgroup H of the group G of finite index, construct the permutation group, induced by the action of G on the set of (right) cosets of H in G. The returned group is the epimorphic image L of G under the permutation representation $f : G \to L \leq \mathrm{Sym}(|G:H|)$, induced by the action of G on the set of (right) cosets of H in G.

`CosetKernel(G, H)`

Given a subgroup H of the group G of finite index, construct the kernel of the permutation representation $f : G \to L \leq \mathrm{Sym}(|G:H|)$, induced by the action of G on the set of (right) cosets of H in G.

Example H72E8

We use the function `CosetAction` to construct a (non-faithful) permutation representation of the group G defined by the polycyclic presentation

$$< a, b, c \mid b^a = b * c, (a, c), (b, c) > .$$

```
> F<a,b,c> := FreeGroup(3);
> rels := [ b^a=b*c, b^(a^-1)=b*c^-1 ];
> G<a,b,c> := quo<GrpGPC: F | rels>;
>
> S := sub<G|(a*b)^3, c^7, b^21>;
> Index(G, S);
441
> pi, P, K := CosetAction(G, S);
> P;
Permutation group P acting on a set of cardinality 441
> K;
GrpGPC : K of infinite order on 3 PC-generators
PC-Relations:
    K.2^K.1 = K.2 * K.3^63,
    K.2^(K.-1) = K.2 * K.-3^63
> Index(G, K);
3087
```

We express the generators of the kernel K in terms of the generators of G:

```
> {@ G!x : x in PCGenerators(K) @};
{@ a^21, b^21, c^7 @}
```

$pi(S)$ is a point stabiliser in the transitive permutation group P of degree 441 and hence should have index 441 in P:

```
> pi(S);
Permutation group acting on a set of cardinality 441
> Index(P, pi(S));
441
```

72.8 The Subgroup Structure

72.8.1 General Subgroup Constructions

The operators and functions which construct a subgroup of a polycyclic group always return the subgroup as a polycyclic group.

`H ^ g`

`Conjugate(H, g)`

Construct the conjugate $g^{-1} * H * g$ of the group H under the action of the element g. The group H and the element g must belong to a common group.

`H ^ G`

`ncl< G | H >`

`NormalClosure(G, H)`

Given a subgroup H of the group G, construct the normal closure of H in G.

`CommutatorSubgroup(G, H, K)`

`CommutatorSubgroup(H, K)`

Construct the commutator subgroup of groups H and K, where H and K are subgroups of a common group G.

72.8.2 Subgroup Constructions Requiring a Nilpotent Covering Group

The operators and functions described in this section require the existence of a nilpotent covering group. They are based on algorithms published in [Lo98]. Again, the constructed subgroup is returned as a polycyclic group.

`H meet K`

Given two groups H and K, contained in some common group G which is nilpotent, construct the intersection of H and K.

`H meet:= K`

Given two groups H and K, contained in some common group G which is nilpotent, replace H with the intersection of H and K.

`Centraliser(G, g)`

`Centralizer(G, g)`

The subgroup of G centralising g. Both g and G must be contained in some common nilpotent group.

`Centraliser(G, H)`
`Centralizer(G, H)`

The subgroup of G centralising H. Both H and G must be subgroups of some common nilpotent group.

`Core(G, H)`

The maximal normal subgroup of the nilpotent group G that is contained in the subgroup H of G.

`Normaliser(G, H)`
`Normalizer(G, H)`

The subgroup of G normalising H. Both H and G must be subgroups of some common nilpotent group.

72.9 General Group Properties

`IsAbelian(G)`

Returns `true` if the group G is abelian, `false` otherwise.

`IsCyclic(G)`

Returns `true` if the group G is cyclic, `false` otherwise.

`IsElementaryAbelian(G)`

Returns `true` if the group G is elementary abelian, `false` otherwise. The following definition is used:

A group G is called elementary abelian if it is an abelian p-group of exponent p for some prime p.

`IsFinite(G)`

Returns `true` if the group G is finite, `false` otherwise.

`IsNilpotent(G)`

Returns `true` if the group G is nilpotent, `false` otherwise. This function uses an algorithm described in [Lo98].

`IsPerfect(G)`

Returns `true` if the group G is perfect, `false` otherwise. A polycyclic group G is perfect, if and only if it is trivial.

`IsSimple(G)`

Returns `true` if the group G is simple, `false` otherwise. A polycyclic group is simple, if and only if it is cyclic of prime order.

IsSoluble(G)

IsSolvable(G)

Returns `true` if the group G is solvable, `false` otherwise. Every polycyclic group is solvable.

72.9.1 General Properties of Subgroups

IsCentral(G, H)

Returns `true` if the subgroup H of the group G lies in the centre of G, `false` otherwise.

IsNormal(G, H)

Returns `true` if the subgroup H of the group G is a normal subgroup of G, `false` otherwise.

72.9.2 Properties of Subgroups Requiring a Nilpotent Covering Group

The functions described in this section require the existence of a nilpotent covering group. The are based on algorithms published in [Lo98].

IsConjugate(G, H, K)

Given groups G, H and K with a nilpotent common covering group, return the value `true` if there exists $c \in G$ such that $H^c = K$. If so, the function returns such a conjugating element as second value.

IsSelfNormalising(G, H)

IsSelfNormalizing(G, H)

Returns `true` if the subgroup H of the nilpotent group G is self-normalising in G, `false` otherwise.

Example H72E9

We define a group G on 5 generators $a, \ldots, e$ of infinite order by fixing the commutators of the generators:

$$(b, a) = e^2, \quad (d, c) = e^3$$

All other pairs of generators commute.

```
> F<a,b,c,d,e> := FreeGroup(5);
> rels := [ b^a = b*e^2, b^(a^-1) = b*e^-2, d^c = d*e^3,
>           d^(c^-1) = d*e^-3 ];
> G<a,b,c,d,e> := quo< GrpGPC: F | rels >;
> IsNilpotent(G);
true
```

Since G is nilpotent, we can compute intersections of subgroups of G.

We define the subgroups generated by $a, \ldots, e$ and their nontrivial commutator groups as subgroups of G.

```
> H1 := sub<G|a>;
> H2 := sub<G|b>;
> H3 := sub<G|c>;
> H4 := sub<G|d>;
> H5 := sub<G|e>;
>
> C12 := CommutatorSubgroup(H1, H2);
> {@ G!x : x in PCGenerators(C12) @};
{@ e^2 @}
> C12 subset H5;
true
>
> C34 := CommutatorSubgroup(H3, H4);
> {@ G!x : x in PCGenerators(C34) @};
{@ e^3 @}
> C34 subset H5;
true
```

Finally, we compute the intersection C of $C12$ and $C13$.

```
> C := C12 meet C34;
> {@ G!x : x in PCGenerators(C) @};
{@ e^6 @}
```

This intersection C is cyclic and central in G.

```
> IsCyclic(C);
true
> IsCentral(G, C);
true
```

Example H72E10

Consider the nilpotent group $G := D_{16} \wr 2$ generated by the 5 generators a, b, c, d, t with the relations

$$a^2 = 1, \quad b^{16} = 1, \quad b^a = b^{15}$$

$$c^2 = 1, \quad d^{16} = 1, \quad d^c = d^{15}$$

$$t^2 = 1, \quad a^t = c, \quad b^t = d, \quad c^t = a, \quad d^t = b$$

(All other pairs of generators commute.)

```
> F<t, a,b, c,d> := FreeGroup(5);
> G<t, a,b, c,d> := quo<GrpGPC: F | a^2, b^16, b^a=b^15,
>                                   c^2, d^16, d^c=d^15,
>                      t^2, a^t=c, b^t=d, c^t=a, d^t=b>;
> IsNilpotent(G);
```

```
true
```

Since G is nilpotent, we can compute normalisers and centralisers in G.
We define the (dihedral) subgroup $D3$ of G generated by ac and bd and compute its normaliser in G and its centraliser in the (dihedral) subgroup $D2$ of G generated by c and d.

```
> D2 := sub<G|c,d>;
>
> D3<u,v> := sub<G|a*c, b*d>;
> D3;
GrpGPC : D3 of order 2^5 on 2 PC-generators
PC-Relations:
    u^2 = Id(D3),
    v^16 = Id(D3),
    v^u = v^15
>
> N3 := Normaliser(G, D3);
> PCGenerators(N3, G);
{@ t, a * c, b * d, d^8 @}
>
> C3 := Centraliser(D2, D3);
> PCGenerators(C3, G);
{@ d^8 @}
```

Finally we compute the centraliser of the element t in G.

```
> Ct := Centraliser(G, t);
> PCGenerators(Ct, G);
{@ t, a * c, b * d @}
```

72.10 Normal Structure and Characteristic Subgroups

72.10.1 Characteristic Subgroups and Subgroup Series

`Centre(G)`
`Center(G)`

The centre of the group G. For nilpotent groups the centre is computed using the centraliser algorithm [Lo98]. Otherwise, it is computed as the simultaneous fixed point space of the action of the generators of G on the centre of the Fitting subgroup of G [Eic01].

`DerivedLength(G)`

The derived length of the group G.

`DerivedSeries(G)`

The derived series of the group G. The series is returned as a sequence of subgroups.

`DerivedSubgroup(G)`

`DerivedGroup(G)`

The derived subgroup of the group G.

`EFASeries(G)`

Returns a normal series of G, the factors of which are either elementary abelian p-groups or free abelian groups.

`FittingLength(G)`

The Fitting length of the group G, i.e. the smallest integer k such that $F_k = G$ where the groups F_i are defined recursively by $F_0 := 1$ and $F_i/F_{i-1} := \mathrm{Fit}(G/F_{i-1})$ $(i > 0)$. Note that such a k exists for every polycyclic group G.

`FittingSeries(G)`

The Fitting series of the group G, where the groups F_i are defined recursively by $F_0 := 1$ and $F_i/F_{i-1} := \mathrm{Fit}(G/F_{i-1})$ $(i > 0)$. The series is returned as the sequence $[F_0, \ldots, F_k]$ of subgroups of G. Note that every polycyclic group G has a finite Fitting series ending in G.

`FittingSubgroup(G)`

`FittingGroup(G)`

The Fitting subgroup of the group G, i.e. the maximal nilpotent normal subgroup of G. This function uses an algorithm described in [Eic01].

`HasComputableLCS(G)`

This function returns the value `true` if the lower central series of G is computable, otherwise it returns the value `false`. This is useful to avoid runtime errors, when `LowerCentralSeries` is called in user written loops or functions.

`LowerCentralSeries(G)`

The lower central series for the group G. The series is returned as a sequence of subgroups. Since infinite polycyclic groups need not satisfy the descending chain condition for subgroups, computation of the lower central series may fail. To determine if the series can be computed and thereby avoid runtime errors, the function `HasComputableLCS` may be used. This function uses an algorithm described in [Lo98].

`NilpotencyClass(G)`

The nilpotency class of the group G. If G is not nilpotent, then -1 is returned.

`NilpotentPresentation(G)`

A polycyclic presentation is called *nilpotent*, if for all $i = 1, \ldots, n$, G_{i+1} is normal in G and G_i/G_{i+1} is central in G/G_{i+1}. Every nilpotent polycyclic group has a nilpotent polycyclic presentation. A suitable polycyclic series can be obtained by refining the lower central series.

The function `NilpotentPresentation` computes a group N isomorphic to G, given by a nilpotent polycyclic presentation and the isomorphism from G to N.

`SemisimpleEFASeries(G)`

Returns a normal series of G, the factors of which are either elementary abelian p-groups which are semisimple as $\mathbf{F}_p[G]$-modules or free abelian groups which are semisimple as $\mathbf{Q}[G]$-modules.

The normal series returned by the function `SemisimpleEFASeries` is a refinement of the normal series returned by the function `EFASeries`.

`UpperCentralSeries(G)`

The upper central series $[Z_0, \ldots, Z_k]$ of the group G, where the groups Z_k are defined recursively by $Z_0 := 1$ and $Z_i/Z_{i-1} := \mathrm{Z}(G/Z_{i-1})$ $(i > 0)$. The series is returned as a sequence of subgroups of G. Note that since polycyclic groups satisfy the ascending chain condition for subgroups, every polycyclic group G has a finite upper central series.

Example H72E11

The dihedral group of order 32 is nilpotent; we compute its lower central series.

```
> D16<a,b> := DihedralGroup(GrpGPC, 16);
> IsNilpotent(D16);
true
> NilpotencyClass(D16);
4
> L := LowerCentralSeries(D16);
```

The generators of the subgroups in the lower central series expressed as elements of $D16$ are:

```
> for i := 1 to 1+NilpotencyClass(D16) do
>   print i, ":", {@ D16!x : x in PCGenerators(L[i]) @};
> end for;
1 : {@ a, b @}
2 : {@ b^2 @}
3 : {@ b^4 @}
4 : {@ b^8 @}
5 : {@ @}
```

We compute a nilpotent presentation and express the new generators in terms of a and b:

```
> N<p,q,r,s,t>, f := NilpotentPresentation(D16);
> N;
```

```
GrpGPC : N of order 2^5 on 5 PC-generators
PC-Relations:
    p^2 = Id(N),
    q^2 = r,
    r^2 = s,
    s^2 = t,
    t^2 = Id(N),
    q^p = q * r * s * t,
    r^p = r * s * t,
    s^p = s * t
> {@ x@@f : x in PCGenerators(N) @};
{@ a, b, b^2, b^4, b^8 @}
```

The infinite dihedral group has an infinite, strictly descending, lower central series which cannot be computed:

```
> D := DihedralGroup(GrpGPC, 0);
> HasComputableLCS(D);
false
```

It is easy to see, that $b^{2^{i-1}}$ would be a generator of L_i.

The symmetric group on 3 letters is not nilpotent, but has a lower central series which becomes stationary and which can be computed:

```
> F2<a,b> := FreeGroup(2);
> rels := [ a^2 = F2!1, b^3 = F2!1, b^a = b^2 ];
> G<a,b> := quo<GrpGPC : F2 | rels>;
> G;
GrpGPC : G of order 6 = 2 * 3 on 2 PC-generators
PC-Relations:
    a^2 = Id(G),
    b^3 = Id(G),
    b^a = b^2
> IsNilpotent(G);
false
> HasComputableLCS(G);
true
> L := LowerCentralSeries(G);
> for i := 1 to #L do
>   print i, ":", {@ G!x : x in PCGenerators(L[i]) @};
> end for;
1 : {@ a, b @}
2 : {@ b @}
```

72.10.2 The Abelian Quotient Structure of a Group

`AbelianQuotient(G)`

The maximal abelian quotient G/G' of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism is returned as second return value.

`AbelianQuotientInvariants(G)`

`AQInvariants(G)`

Returns a sequence containing the invariants of the maximal abelian quotient G/G' of the group G. Each infinite cyclic factor of G/G' is represented by zero.

`ElementaryAbelianQuotient(G, p)`

The maximal p-elementary abelian quotient of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism is returned as second return value.

`FreeAbelianQuotient(G)`

The maximal free abelian quotient of the group G as `GrpAb` (cf. Chapter 69). The natural epimorphism is returned as second return value.

72.11 Conjugacy

The functions described in this section require the existence of a nilpotent covering group. They are based on algorithms published in [Lo98].

`IsConjugate(G, g, h)`

Given elements g and h and a group G, which are contained in some nilpotent common group, return the value `true` if there exists $c \in G$ such that $g^c = h$. If so, the function returns such a conjugating element as second value.

`IsConjugate(G, H, K)`

Given groups G, H and K with a nilpotent common covering group, return the value `true` if there exists $c \in G$ such that $H^c = K$. If so, the function returns such a conjugating element as second value.

Example H72E12

We again consider the nilpotent group $G := D_{16} \wr 2$.

```
> F<t, a,b, c,d> := FreeGroup(5);
> G<t, a,b, c,d> := quo<GrpGPC: F | a^2, b^16, b^a=b^15,
>                                   c^2, d^16, d^c=d^15,
>                      t^2, a^t=c, b^t=d, c^t=a, d^t=b>;
> IsNilpotent(G);
true
```

Since G is nilpotent, a test for conjugacy in G is available.

We define the following subgroups of G: $D1$ generated by a and b, $D2$ generated by c and d and $D3$ generated by ac and bd.

```
> D1 := sub<G|a,b>;
> D2 := sub<G|c,d>;
> D3<u,v> := sub<G|a*c, b*d>;
>
```

$D1$ and $D2$ are, of course, conjugate in G; t is a conjugating element.

```
> IsConjugate(G, D1, D2);
true t
```

The elements b and d^{-1} are conjugate in G; we compute a conjugating element.

```
> IsConjugate(G, b, d^-1);
true t * a * c
```

However, neither the subgroups $D1$ and $D2$ nor the elements b and d^{-1}, are conjugate in the subgroup $D3$.

```
> IsConjugate(D3, D1, D2);
false
> IsConjugate(D3, b, d^-1);
false
```

72.12 Representation Theory

This section describes some functions for creating $R[G]$-modules for a polycyclic group G, which are unique for this category or have special properties when called for polycyclic groups. For a complete description of the functions available for creating and working with $R[G]$-modules we refer to Chapter 89.

Note that the function GModuleAction can be used to extract the matrix representation associated to an $R[G]$-module.

`EFAModuleMaps(G)`

Every polycyclic group G has a normal series $G = N_1 \triangleright N_2 \triangleright \ldots \triangleright N_{r+1} = 1$, such that every quotient $M_i := N_i/N_{i+1}$ is either free abelian or p-elementary abelian for some prime p. The action of G by conjugation induces a $\mathbf{Z}[G]$-module structure on M_i if M_i is free abelian and an $\mathbf{F}_p[G]$-module structure if M_i is p-elementary abelian.

This function returns a sequence $[f_1, \ldots, f_r]$, where $f_i : N_i \to M_i$ is the natural epimorphism onto the additive group of an $R_i[G]$-module M_i ($R_i \in \{\mathbf{F}_p, \mathbf{Z}\}$), constructed as above.

The functions EFAModuleMaps and EFAModules use the normal series returned by the function EFASeries.

Note that the kernels of the epimorphisms f_i can be computed and hence it is possible to form preimages of submodules of M_i, which are normal subgroups of G contained in N_i and containing N_{i+1}.

`EFAModules(G)`

Every polycyclic group G has a normal series $G = N_1 \triangleright N_2 \triangleright \ldots \triangleright N_{r+1} = 1$, such that every quotient $M_i := N_i/N_{i+1}$ is either free abelian or p-elementary abelian for some prime p. The action of G by conjugation induces a $\mathbf{Z}[G]$-module structure on M_i if M_i is free abelian and an $\mathbf{F}_p[G]$-module structure if M_i is p-elementary abelian.

This function returns a sequence $[M_1, \ldots, M_r]$ of $R_i[G]$-modules (where $R_i \in \{\mathbf{F}_p, \mathbf{Z}\}$), constructed as above.

The functions `EFAModuleMaps` and `EFAModules` use the normal series returned by the function `EFASeries`.

`GModule(G, A, p)`

`GModule(G, A)`

Let A be a normal subgroup of the polycyclic group G. If $p = 0$, the function returns the $\mathbf{Z}[G]$-module corresponding to the conjugation action of G on the maximal free abelian quotient of A. If p is a prime, it returns the $\mathbf{F}_p[G]$-module corresponding to the conjugation action of G on the maximal p-elementary abelian quotient of A. The epimorphism $\pi : A \to M$ onto the additive group of the constructed module M is returned as second return value. Note that the kernel of π can be computed and hence it is possible to form preimages of submodules of M, which are normal subgroups of G contained in A.

If the maximal abelian quotient A/A' of A is either free abelian or p-elementary abelian for some prime p, p can be omitted in the function call.

Note that it is the user's responsibility to ensure that A is in fact normal in G.

`GModule(G, A, B, p)`

`GModule(G, A, B)`

Let A and $B < A$ be normal subgroups of the polycyclic group G. If $p = 0$, the function returns the $\mathbf{Z}[G]$-module corresponding to the conjugation action of G on the maximal free abelian quotient of A/B. If p is a prime, it returns the $\mathbf{F}_p[G]$-module corresponding to the conjugation action of G on the maximal p-elementary abelian quotient of A/B. The epimorphism $\pi : A \to M$ onto the additive group of the constructed module M is returned as second return value. Note that the kernel of π can be computed and hence it is possible to form preimages of submodules of M, which are normal subgroups of G contained in A and containing B.

If the maximal abelian quotient of A/B is either free abelian or p-elementary abelian for some prime p, p can be omitted in the function call.

Note that it is the user's responsibility to ensure that A and B are in fact normal in G.

`GModulePrimes(G, A)`

Let G be a polycyclic group and A a normal subgroup of G. Given any prime p, the maximal p-elementary abelian quotient of A can be viewed as an $\mathbf{F}_p[G]$-module M_p. The maximal free abelian quotient of A can be viewed as a $\mathbf{Z}[G]$-module M_0. This function determines those primes p for which the module M_p is non-trivial (i.e. not zero-dimensional) and the dimensions of the corresponding modules M_p. The return value is a multiset S. If $0 \notin S$, the maximal abelian quotient of A is finite and the multiplicity of p is the dimension of M_p. If S contains 0 with multiplicity m, the maximal abelian quotient of A contains m copies of $\mathbf{Z}$. In this case, the rank of M_0 is m and M_p is non-trivial for every prime p and its rank is the sum of m and the multiplicity of p in S.

`GModulePrimes(G, A, B)`

Let G be a polycyclic group, A a normal subgroup of G and B a normal subgroup of G contained in A. Given any prime p, the maximal p-elementary abelian quotient of A/B can be viewed as an $\mathbf{F}_p[G]$-module M_p. The maximal free abelian quotient of A/B can be viewed as a $\mathbf{Z}[G]$-module M_0. This function determines those primes p for which the module M_p is non-trivial (i.e. not zero-dimensional) and the dimensions of the corresponding modules M_p. The return value is a multiset S. If $0 \notin S$, the maximal abelian quotient of A/B is finite and the multiplicity of p is the dimension of M_p. If S contains 0 with multiplicity m, the maximal abelian quotient of A/B contains m copies of $\mathbf{Z}$. In this case, the rank of M_0 is m and M_p is non-trivial for every prime p and its rank is the sum of m and the multiplicity of p in S.

`SemisimpleEFAModuleMaps(G)`

Every polycyclic group G has a normal series $G = N_1 \triangleright N_2 \triangleright \ldots \triangleright N_{r+1} = 1$, such that every quotient $M_i := N_i/N_{i+1}$ is either free abelian and semisimple as a $\mathbf{Q}[G]$-module or p-elementary abelian and semisimple as an $\mathbf{F}_p[G]$-module for some prime p.

This function returns a sequence $[f_1, \ldots, f_r]$, where $f_i : N_i \to M_i$ is the natural epimorphism onto the additive group of an $R_i[G]$-module M_i ($R_i \in \{\mathbf{F}_p, \mathbf{Z}\}$), constructed as above.

The functions `SemisimpleEFAModules` and `SemisimpleEFAModuleMaps` use the normal series returned by the function `SemisimpleEFASeries`. Moreover, this normal series is a refinement of the normal series returned by the function `EFASeries`.

Note that the kernels of the epimorphisms f_i can be computed and hence it is possible to form preimages of submodules of M_i, which are normal subgroups of G contained in N_i and containing N_{i+1}.

`SemisimpleEFAModules(G)`

Every polycyclic group G has a normal series $G = N_1 \triangleright N_2 \triangleright \ldots \triangleright N_{r+1} = 1$, such that every quotient $M_i := N_i/N_{i+1}$ is either free abelian and semisimple as a $\mathbf{Q}[G]$-module or p-elementary abelian and semisimple as an $\mathbf{F}_p[G]$-module for some prime p.

This function returns a sequence $[M_1, \ldots, M_r]$ of $R_i[G]$-modules (where $R_i \in \{\mathbf{F}_p, \mathbf{Z}\}$), constructed as above.

The functions SemisimpleEFAModules and SemisimpleEFAModuleMaps use the normal series returned by the function SemisimpleEFASeries. Moreover, this normal series is a refinement of the normal series returned by the function EFASeries.

Example H72E13

Consider the group G defined by the polycyclic presentation

$$\langle a, b, c, d, e \,|\, c^6, e^3, d^c = de, e^c = e^2, b^a = b^{-1}, b^{a^{-1}} = b^{-1}, \\ c^b = ce, c^{b^{-1}} = ce, d^b = d^{-1}, d^{b^{-1}} = d^{-1}, e^b = e^2, e^{b^{-1}} = e^2 \rangle.$$

(Trivial commutator relations have been omitted.)

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F | c^6 = F!1, e^3 = F!1,
>                                   d^c=d*e, e^c=e^2,
>                                   b^a=b^-1,
>                                   b^(a^-1)=b^-1,
>                                   c^b=c*e,
>                                   c^(b^-1)=c*e,
>                                   d^b=d^-1,
>                                   d^(b^-1)=d^-1,
>                                   e^b=e^2,
>                                   e^(b^-1)=e^2 >;
```

The subgroup H of G generated by c, d, e is normal in G.

```
> H := sub< G | c,d,e >;
> IsNormal(G, H);
true
```

We determine the primes such that the action of G on H yields non-trivial modules.

```
> GModulePrimes(G, H);
{* 0, 2, 3 *}
```

We construct the $\mathbf{F}_3[G]$-module M given by the action of G on the maximal 3-elementary abelian quotient of H and the natural epimorphism π from H onto the additive group of M.

```
> M, pi := GModule(G, H, 3);
> M;
GModule M of dimension 2 over GF(3)
```

Using the function Submodules, we obtain the submodules of M. Their preimages under π are precisely the normal subgroups of G which are contained in H and contain $\ker(\pi)$.

```
> submod := Submodules(M);
> nsgs := [ m @@ pi : m in submod ];
> [ PCGenerators(s, G) : s in nsgs ];
```

```
[
    {@ c^3, d^3, e @},
    {@ c, d^3, e @},
    {@ c^3, d, e @},
    {@ c, d, e @}
]
```

Example H72E14

Consider the group defined by the polycyclic presentation

$$< a, b, c, d, e \mid a^5, b^5, c^6, d^5, e^3, b^a = bd > .$$

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F |
>                      a^5, b^5, c^6, d^5, e^3, b^a = b*d >;
```

Obviously the subgroup of G generated by b, c, d, e is normal in G.

```
> H := sub< G | b,c,d,e >;
> IsNormal(G, H);
true
```

We use the function GModulePrimes to determine the set of primes p for which the action of G on the maximal p-elementary abelian quotient of H induces a nontrivial $\mathbf{F}_p[G]$-module.

```
> P := GModulePrimes(G, H);
> 0 in P;
false
```

0 is not contained in P, i.e. the maximal free abelian quotient of H is trivial. Hence, there are only finitely many primes, satisfying the condition above.

We loop over the distinct elements of P and for each element p we construct the induced $\mathbf{F}_p[G]$-module, print its dimension and check whether it is decomposable. Note that the dimension of the module for p must be equal to the multiplicity of p in P.

```
> for p in MultisetToSet(P) do
>    M := GModule(G, H, p);
>    dim := Dimension(M);
>    decomp := IsDecomposable(M);
>
>    assert dim eq Multiplicity(P, p);
>
>    print "prime", p, ": module of dimension", dim;
>    if decomp then
>       print "  has a nontrivial decomposition";
>     else
>       print "  is indecomposable";
>    end if;
```

```
> end for;
prime 2 : module of dimension 1
  is indecomposable
prime 3 : module of dimension 2
  has a nontrivial decomposition
prime 5 : module of dimension 2
  is indecomposable
```

Example H72E15

The Fitting subgroup of a polycyclic group G can be characterised as the intersection of the centralisers in G of the semisimple G-modules defined by the action of G on the factors of a semisimple EFA-series of G. The centraliser in G of a G-module is just the kernel of the action map.
We illustrate this with the group G defined in the example above.

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F | c^6 = F!1, e^3 = F!1,
>                                   b^a = b * d,
>                                   b^(a^-1) = b * d^-1 >;
```

We first construct the G-modules defined by the action of G on the factors of a semisimple EFA-series of G.

```
> modules := SemisimpleEFAModules(G);
> modules;
[
    GModule of dimension 2 over Integer Ring,
    GModule of dimension 1 over Integer Ring,
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(3)
]
```

Now, we compute the intersection of the kernels of the module action maps, which can be obtained using the function GModuleAction.

```
> S := G;
> for m in modules do
>    S meet:= Kernel(GModuleAction(m));
> end for;
```

Finally, we compare the result with the Fitting subgroup of G, returned by the Magma function FittingSubgroup.

```
> S eq FittingSubgroup(G);
true
```

Example H72E16

The functions EFAModuleMaps and SemisimpleEFAModuleMaps are useful whenever it is desired to refine an EFA-series or a semisimple EFA-series by computing the subgroups corresponding to submodules of the G-modules given by the factors of the series. Consider again the group defined above.

```
> F<a,b,c,d,e> := FreeGroup(5);
> G<a,b,c,d,e> := quo< GrpGPC : F | c^6 = F!1, e^3 = F!1,
>                                   b^a = b * d,
>                                   b^(a^-1) = b * d^-1 >;
```

We extract the map f from G (the first group in any EFA-series of G) onto the module given by the first factor of an EFA-series of G.

```
> f := EFAModuleMaps(G)[1];
> f;
Mapping from: GrpGPC: G to GModule of dimension 2 over Integer
Ring
```

The module itself can be accessed as the codomain of f.

```
> M := Codomain(f);
> M;
GModule M of dimension 2 over Integer Ring
```

Spinning up random elements, we try to construct a submodule S of M.

```
> repeat
>    S := sub<M|[Random(-1, 1): i in [1 .. Dimension(M)]]>;
>    until Dimension(S) gt 0 and S ne M;
> S;
GModule S of dimension 1 over Integer Ring
```

The preimage N of S under f is a normal subgroup of G, which lies between the first and the second subgroup of the original EFA-series for G.

```
> N := S @@ f;
> PCGenerators(N, G);
{@ a^2 * b^4, c, d, e @}
```

72.13 Power Groups

`Parent(G)`

The PowerStructure of category GrpGPC.

`PowerGroup(G)`

The set of all subgroups of G. This is very useful when constructing sets of polycyclic groups. If the user will be building a set of subgroups of a polycyclic group G, then it is best to specify the set's universe to be PowerGroup(G). If the set's universe is not specified it will be the parent structure of G as returned by Parent(G).

72.14 Bibliography

[**CLGM+95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

[**Eic01**] Bettina Eick. On the Fitting subgroup of a polycyclic-by-finite group and its applications. *J. Algebra*, 242(1):176–187, 2001.

[**Geb02**] Volker Gebhardt. Efficient collection in infinite polycyclic groups. *J. Symbolic Comput.*, 34(3):213–228, 2002.

[**Lo98**] Eddie H. Lo. Finding intersections and normalizers in finitely generated nilpotent groups. *J. Symbolic Comput.*, 25(1):45–59, 1998.

[**Sim94**] Charles C. Sims. *Computation with finitely presented groups.* Cambridge University Press, Cambridge, 1994.

73 BRAID GROUPS

Chapter 73
BRAID GROUPS

73.1 Introduction

This chapter deals with another, quite specialised, class of finitely presented groups for which the word problem is solvable, the category of braid groups. The corresponding MAGMA category is called `GrpBrd`.

The notion of braid groups was introduced by Artin [Art47], who considered a sequence B_n $(n = 1, 2, \ldots)$ of groups, where B_n is presented on $n-1$ generators $\sigma_1, \ldots, \sigma_{n-1}$ with the defining relations

$$\begin{aligned} \sigma_i\sigma_j &= \sigma_j\sigma_i && (1 \leq i < j < n, \quad j - i > 1), \\ \sigma_i\sigma_{i+1}\sigma_i &= \sigma_{i+1}\sigma_i\sigma_{i+1} && (1 \leq i < n-1) \quad . \end{aligned}$$

B_n is called the *braid group on n strings*. In the sequel, we refer to the above presentation as *Artin presentation* and to $\sigma_1, \ldots, \sigma_{n-1}$ as *Artin generators* of B_n.

Birman, Ko and Lee introduced an alternative way of presenting braid groups [BKL98]. Here, B_n is presented on $n(n-1)/2$ generators $a_{r,t}$ $(n \geq r > t \geq 1)$ with the defining relations

$$\begin{aligned} &a_{t,s}a_{r,q} = a_{r,q}a_{t,s} \quad \text{for } n \geq t > s \geq 1,\ n \geq r > q \geq 1,\ (t-r)(t-q)(s-r)(s-q) > 0 \\ &a_{t,s}a_{s,r} = a_{t,r}a_{t,s} = a_{s,r}a_{t,r} \quad \text{for } n \geq t > s > r > 0 \quad . \end{aligned}$$

We refer to this presentation as *BKL presentation* and to $a_{r,t}$ $(n \geq r > t \geq 1)$ as *BKL generators* of B_n.

A possible choice for the BKL generators in terms of Artin generators is $a_{r,t} = (\sigma_{r-1} \cdots \sigma_{t+1})\sigma_t(\sigma_{t+1}^{-1} \cdots \sigma_{r-1}^{-1})$. This identification is used in MAGMA.

Recently, braid groups came under consideration as possible sources for public key cryptosystems [AAG99, KLC+00]. The features of braid groups which make them interesting for public key cryptography are the following.

- The basic group operations in braid groups can be implemented efficiently on a computer.
- The word problem in braid groups is solvable, that is, there is a normal form for elements of a braid group and elements can be compared. Moreover, there are algorithms which are able to compute the normal form of an element efficiently.
- There are several problems in braid groups which are believed to be mathematically hard and whose use for cryptographic purposes has been suggested. The most important examples are variations of the conjugacy problem.

However, both recent attacks on particular cryptosystems [GKT+02, HS03, Hug02, LL02, LP03] and advances in the analysis of the conjugacy problem [GM02, Geb03] in general shed some doubts on the security of braid group cryptosystems. At the time of this writing it is an open question whether braid group cryptosystems can be made secure by an appropriate choice of parameters and keys or whether they have to be considered as insecure. More research into these issues is necessary.

The MAGMA category `GrpBrd` was introduced mainly with these applications in mind. Focus was put on providing fast operations with elements and on giving the user as much control over the details of computations as possible.

73.1.1 Lattice Structure and Simple Elements

In this section we briefly recall the basic terminology used for describing elements of braid groups. More detailed descriptions can be found in [ECH+92] for the Artin presentation and in [BKL98] for the BKL presentation.

We remark that both Artin presentation and BKL presentation are special cases of so-called *Garside groups* [Deh02].

In the sequel, let M be either the Artin presentation or the BKL presentation of the braid group B on n strings, let X denote the generators of M and let R denote the relations of M. As the relations in R do not contain inverses of generators, we can interpret M as monoid presentation. We denote the finitely presented monoid defined by M by B^+. The natural homomorphism from B^+ to B can be shown to be injective. We identify its image with B^+ and call it the set of *positive* elements of B. Finally, we denote the identity of B by 1.

We can now define two partial orderings on B. For elements $u, v \in B$ we say $u \preceq v$, if there exists a positive element a such that $ua = v$, and we say $v \succeq u$, if there exists a positive element a such that $v = au$. Note that these partial orderings are different; $u \preceq v$ is not equivalent to $v \succeq u$.

B can be shown to be a lattice with respect to both partial orderings, that is, for elements $u, v \in B$ there are elements $d_l, m_l, d_r, m_r \in B$ such that

$$\begin{aligned}
d_l \preceq u,\ d_l \preceq v \quad &\text{and} \quad d \preceq u,\ d \preceq v \ \text{ implies } \ d \preceq d_l \ \text{ for all } \ d \in B \\
u \preceq m_l,\ v \preceq m_l \quad &\text{and} \quad u \preceq m,\ v \preceq m \ \text{ implies } \ m_l \preceq m \ \text{ for all } \ m \in B \\
u \succeq d_r,\ v \succeq d_r \quad &\text{and} \quad u \succeq d,\ v \succeq d \ \text{ implies } \ d_r \succeq d \ \text{ for all } \ d \in B \\
m_r \succeq u,\ m_r \succeq v \quad &\text{and} \quad m \succeq u,\ m \succeq v \ \text{ implies } \ m \succeq m_r \ \text{ for all } \ m \in B \quad .
\end{aligned}$$

We call d_l, m_l, d_r and m_r the *left-gcd*, the *left-lcm*, the *right-gcd* and the *right-lcm*, respectively, of u and v.

It can be shown that the left-lcm of the elements of X and the right-lcm of the elements of X are equal; we call this element the *fundamental element* of the presentation M and denote it by D. The fundamental element is crucial for the study of braid groups. One of its most important properties is that a certain power D^N of D generates the centre of B. ($N = 2$ for the Artin presentation and $N = n$ for the BKL presentation.) Moreover, $u \preceq D^k$ is equivalent to $D^k \succeq u$ and $D^k \preceq u$ is equivalent to $u \succeq D^k$ for all $k \in \mathbf{Z}, u \in B$.

In MAGMA, the partial ordering $\preceq$ is provided as operator `le` and the partial ordering $\succeq$ is provided as operator `ge`; see Section 73.4.4. For a description of the functions computing lcm and gcd of elements, see Section 73.4.5.

The positive elements c of B satisfying $c \preceq D$ are called *simple elements*; we denote the set of simple elements by C. simple elements can be uniquely described by permutations on n points. In MAGMA, a simple element c inducing a permutation π on the strings on which B is defined, is represented by the permutation π^{-1}.

If M is the Artin presentation, every permutation on n points corresponds to a simple element, that is, $|C| = n!$.

If M is the BKL presentation, $|C| = (2n)!/(n!(n+1)!)$ and only permutations on n points which are products of parallel, descending cycles correspond to simple elements. Here, a cycle $(i_1, \ldots, i_r)$ is called *descending* if $i_1 > \ldots > i_r$ and two descending cycles $(i_1, \ldots, i_r)$ and $(j_1, \ldots, j_s)$ are called *parallel* if $(i_k - j_l)(i_k - j_{l'})(i_{k'} - j_l)(i_{k'} - j_{l'}) > 0$ for all $1 \leq k, k' \leq r$ and $1 \leq l, l' \leq s$. The descending cycle $(i_1, \ldots, i_r)$ corresponds to the element $a_{i_1,i_2} a_{i_2,i_3} \cdots a_{i_{r-1},i_r}$ of B. It is obvious from the defining relations that the simple elements defined by two parallel descending cycles commute.

Every element u of B can be written in the form $u = D^l c_1 \cdots c_k$, where l is a suitable integer and $c_1, \ldots c_k$ are simple elements. We call representations of this form *simple element representations* or *canonical factor products* (CFP).

73.1.2 Representing Elements of a Braid Group

This section describes the ways in which elements of a braid group can be represented internally by MAGMA. From the user's point of view, this mainly affects input and printing of elements. This section is intended to be a concise overview; for a detailed description of functions and for examples we refer to Section 73.2, Section 73.3 and Section 73.4.1.

Since an element of a braid group B can be represented either as word in the generators or as product of simple elements (see Section 73.4.5) with respect to either the Artin presentation or the BKL presentation of B, there are four different ways of representing elements of B, which can be used for entering or printing elements and for computing with elements.

73.1.2.1 Automatic Conversions

MAGMA can work with all the above representations and conversions are done automatically when necessary, for example, when multiplying an element defined as word in the Artin generators with an element given as product of simple elements for the BKL presentation. Hence, the user normally does not have do give too much thought about how elements are represented. It should be noted, however, that automatic conversions can affect performance and that in time critical situations, the best results in general are obtained if automatic conversions are avoided.

73.1.2.2 Default Presentations

When creating a braid group B using the command `BraidGroup`, the user can specify whether the Artin presentation or the BKL presentation should be used as *default presentation* for B. Unless specified otherwise by the user, this presentation is used in all

subsequent operations with B or with elements of B. In particular, group operations and printing of elements are performed with respect to this presentation. It is possible to change the default presentation using the command SetPresentation. Certain commands accept a parameter Presentation, which can be used to perform that command with respect to a presentation other than the default presentation.

73.1.2.3 Representation Used for Group Operations

By default, group operations with elements of a braid group B are performed using representations of the elements as products of simple elements for the default presentation of B. Experienced users can change this behaviour using the command SetForceCFP. If this flag is set to false, arguments of a group operation are not automatically converted into CFP representation if both arguments are represented as words in the generators of the default presentation of B, but the operation is performed, if possible, using the word representations instead.

73.1.2.4 Printing of Elements

The default printing format for an element u of a braid group B is that both a representation of u as word in the generators of the default presentation of B and a representation of u as product of simple elements for the default presentation of B are printed.

Depending on the application, the user may wish to change the print format so that only one of the above representations of u is printed. This can be achieved using the command SetElementPrintFormat.

73.1.3 Normal Form for Elements of a Braid Group

This section briefly describes the normal form for elements of braid groups. For details we refer to [ECH+92] and [BKL98]. The MAGMA commands for computing normal forms are described in Section 73.4.2.

Let B be the braid group on n strings and fix a presentation M for B, either the Artin presentation or the BKL presentation. A product of simple elements $D^l c_1 \cdots c_k$ is said to be in *left normal form* with respect to M, if $c_1 \neq D$, $c_k \neq 1$ and $(c_i^{-1} D) \wedge_l c_{i+1} = 1$ for $i = 1, \ldots, k-1$, where $(c_i^{-1} D) \wedge_l c_{i+1}$ denotes the left-gcd of $c_i^{-1} D$ and c_{i+1} with respect to the presentation M.

Similarly, we define $c_1 \cdots c_k D^l$ to be in *right normal form* with respect to M, if $c_k \neq D$, $c_1 \neq 1$ and $c_i \wedge_r (D c_{i+1}^{-1}) = 1$ for $i = 1, \ldots, k-1$, where $\wedge_r$ denotes right-gcd with respect to the presentation M.

It can be shown that the numbers of simple elements and the powers of D in the left and right normal forms of an element are equal, that is, if $x \in B$ has left normal form $D^l c_1 \cdots c_k$ and right normal form $\bar{c}_1 \cdots \bar{c}_{k'} D^{l'}$ then $k' = k$ and $l' = l$. In this situation we call l the *infimum* of x, denoted by $\inf(x)$, k the *canonical length* of x, denoted by $\mathrm{len}(x)$, and $l + k$ the *supremum* of x, denoted by $\sup(x)$. l is the maximal integer d satisfying $D^d \preceq x$ and $l + k$ is the minimal integer d satisfying $x \preceq D^d$.

To bring a product $D^l c_1 \cdots c_k$ of simple elements into left normal form, we proceed by induction, assuming that $D^l c_1 \cdots c_{k-1}$ is in left normal form. For $i = k-1, \ldots, 1$ we now

compute $d = (c_i^{-1}D) \wedge_l c_{i+1}$ and, if $d \neq 1$, replace c_i by $c_i d$ and c_{i+1} by $d^{-1}c_{i+1}$. Finally, we delete trailing trivial simple elements and absorb simple elements equal to D into the leading power of D. The result can be shown to be in left normal form [ECH+92, BKL98].

Both the theoretical complexity of this algorithm and its performance in practice are determined by the gcd computations.

For the Artin presentation, the cost of computing the left-gcd of two simple elements is $O(n \log n)$ [ECH+92], whence the complexity of bringing a product of simple elements as above into left normal form is $O(k^2 n \log n)$.

For the Artin presentation, the cost of computing the left-gcd of two simple elements is $O(n)$ [BKL98], whence the complexity of bringing a product of simple elements as above into left normal form is $O(k^2 n)$.

Computing right normal forms is analogous.

73.1.4 Mixed Canonical Form and Lattice Operations

This section outlines the algorithms used for lattice operations in a braid group. Let u and v be elements of a braid group B and let M be either the Artin presentation or the BKL presentation of B. The MAGMA commands for computing mixed canonical forms are described in Section 73.4.2 and the commands providing lattice operations are described in Section 73.4.5.

Evaluating partial orderings for u and v with respect to M is straightforward. $u \preceq v$ if and only if $u^{-1}v$ is a positive element with respect to M. The latter can be decided by computing the left normal form $D^l c_1 \cdots c_k$ of $u^{-1}v$ with respect to M: $u^{-1}v$ is positive if and only if $l \geq 0$. Evaluating the partial ordering $\succeq$ is analogous.

We call the tuple $< a, b >$ the *left-mixed canonical form* of an element $x \in B$, if $a = a_1 \cdots a_k$ and $b = b_1 \cdots b_s$ are positive elements in left normal form ($a_1 = D$, $b_1 = D$ is permitted), $x = a^{-1}b$ and the left-gcd of a_1 and b_1 is trivial.

Similarly, we call the tuple $< a, b >$ the *right-mixed canonical form* of x, if $a = a_1 \cdots a_k$ and $b = b_1 \cdots b_s$ are positive elements in right normal form ($a_k = D$, $b_s = D$ is permitted), $x = ab^{-1}$ and the right-gcd of a_k and b_s is trivial.

It is not difficult to show that the left-gcd of u and v is given by ua^{-1}, where $< a, b >$ is the left-mixed canonical form of $u^{-1}v$, and that the right-gcd of u and v is given by $a^{-1}u$, where $< a, b >$ is the right-mixed canonical form of uv^{-1} [ECH+92].

Similarly, the left-lcm of u and v is given by ua, where $< a, b >$ is the right-mixed canonical form of $u^{-1}v$ and the right-lcm of u and v is given by au, where $< a, b >$ is the left-mixed canonical form of uv^{-1}

Computing the left-mixed canonical form of an element x can, after writing $x = a^{-1}b$ with two positive elements a and b, easily be reduced to computing repeatedly the left-normal forms of a and b and cancelling the left-gcd of the leading simple elements. Computing the right-mixed canonical form is analogous.

73.1.5 Conjugacy Testing and Conjugacy Search

Conjugacy testing, that is, deciding whether two given braids are conjugate, and conjugacy search, that is, computing a conjugating element for a pair of conjugate braids, are of particular importance to public key cryptosystems based on braid groups. Known algorithms for both conjugacy testing and conjugacy search require the (at least partial) computation of an invariant of the conjugacy classes of the elements in question, either the *super summit set* [Gar69, ERM94] or the *ultra summit set* [Geb03].

This section recalls the definition of these invariants and sketches the algorithms used for computing them, for conjugacy testing and for conjugacy search. The relevant MAGMA commands are described in Section 73.4.6.

For this section let B be a braid group and let M be either the Artin presentation or the BKL presentation of B.

73.1.5.1 Definition of the Class Invariants

We define two operations, the *cycling* operation $\mathbf{c}$ and the *decycling* operation $\mathbf{d}$, each mapping an arbitrary element $x \in B$ to a conjugate of x as follows. Let $x \in B$ be a braid with left normal form $x = D^l c_1 \cdots c_k$ as defined in Section 73.1.3. If $k = 0$, we define $\mathbf{c}(x) = x$ and $\mathbf{d}(x) = x$. Otherwise, we define $\mathbf{c}(x) = D^l c_2 \cdots c_k (c_1^{D^{-l}})$ and $\mathbf{d}(x) = D^l (c_k^{D^l}) c_1 \cdots c_{k-1}$.

We now fix an element $x \in B$ and consider the set C_x of all conjugates of x. Proofs for the following facts can be found in [ECH+92] or [BKL98].

- The set $\{\inf(y) : y \in C_x\}$ is bounded above; we denote its maximum by ss-inf(x).
- The set $\{\sup(y) : y \in C_x\}$ is bounded below; we denote its minimum by ss-sup(x).
- The maximum of inf on C_x and the minimum of sup on C_x can be achieved simultaneously.

We define three sets of conjugates of x as follows.

- The set $P_x = \{y \in C_x : y \in B^+\}$, containing the positive conjugates of x.
- The set $S_x = \{y \in C_x : \inf(y) = \text{ss-inf}(x), \sup(y) = \text{ss-sup}(x)\}$, called the *super summit set* of x.
- The set $U_x = \{y \in S_x : \exists\, i > 0 : \mathbf{c}^i(y) = y\}$, called the *ultra summit set* of x.

Clearly, the sets P_x, S_x and U_x only depend on the conjugacy class of x. Moreover, the set P_x is empty if ss-inf$(x) < 0$ and it contains S_x if ss-inf$(x) \geq 0$.

Proofs of the following properties can be found in [ECH+92] and [BKL98] for the sets P_x and S_x and in [Geb03] for the set U_x.

- The sets P_x, S_x and U_x are finite.
- The sets S_x and U_x are non-empty.
- Representatives of P_x, S_x and U_x, respectively, can be obtained from x by a finite number of cycling and decycling operations.

73.1.5.2 Computing the Class Invariants

The main tools for computing the class invariants introduced in Section 73.1.5.1 are the following "convexity" results established in [ERM94] and [FGM03] for the sets P_x and S_x and in [Geb03] for the set U_x. Let $I_x \in \{P_x, S_x, U_x\}$.

- For $y, z \in I_x$, there exists a finite sequence $y = y_0, \ldots, y_r = z$ such that for $i = 1, \ldots, r$, $y_i \in I_x$ and $y_i = y_{i-1}^{c_i}$ for a simple element c_i.
- For $y \in I_x$ and a simple element c, there exists a unique $\preceq$-minimal element $\iota_y(c)$ such that $c \preceq \iota_y(c)$ and $y^{\iota_y(c)} \in I_x$. Moreover, $\iota_y(c)$ is simple.

By the above results, any non-empty subset $I \subseteq I_x$ with the property that $y^{\iota_y(s)} \in I$ for all $y \in I$ and all generators s of the presentation M is equal to I_x. In particular, I_x can be computed, starting from a single representative, as closure under conjugation with minimal simple elements.

Algorithms for computing the minimal simple elements $\iota_y(c)$ are given in [FGM03] for the case $I_x \in \{P_x, S_x\}$ and in [Geb03] for the case $I_x = U_x$.

The MAGMA commands for computing the class invariants P_x, S_x and U_x as well as corresponding minimal simple elements $\iota_y(c)$ are described in Section 73.4.6.

73.1.5.3 Conjugacy Testing and Conjugacy Search

Testing conjugacy of two braids $x, y \in B$ can be performed using either super summit sets or ultra summit sets. It is obvious from the results cited in Section 73.1.5.1 that the following are equivalent.

- x and y are conjugate in B.
- $S_x = S_y$.
- $U_x = U_y$.
- $S_x \cap S_y \neq \emptyset$.
- $U_x \cap U_y \neq \emptyset$.

If x and y are conjugate, a conjugating element can be obtained by establishing an element $z \in S_x \cap S_y$ or $z \in U_x \cap U_y$ both as conjugate of x and of y and keeping track of the conjugating elements in each step.

The size of super summit sets grows rapidly with increasing values of braid index n and canonical length. In general, computing super summit sets is difficult or infeasible for braids on more than 5-10 strings, except for very short canonical lengths. Ultra summit sets, on the other hand tend to be much smaller and can frequently be computed for braids on up to 100 strings and canonical length up to 1000, provided sufficient memory is available [Geb03]. Conjugacy search may be successful even in situations where the entire class invariant is too large to be computed.

In MAGMA, conjugacy testing and conjugacy search based on ultra summit sets is provided by the function `IsConjugate`.

73.2 Constructing and Accessing Braid Groups

This section describes the facilities for creating a braid group and for accessing and changing its basic properties.

`BraidGroup(n: parameters)`

`BraidGroup(GrpBrd, n: parameters)`

Given a small integer $n > 0$, return the braid group on n strings, that is, $n-1$ Artin generators.

`Presentation` MonStgElt *Default :* "*Artin*"

The presentation can be selected using the parameter `Presentation`. Possible values for this parameter are `"Artin"` (default) and `"BKL"`.

`GetPresentation(B)`

Returns a string s indicating the presentation currently used for B. s is either `"Artin"` or `"BKL"`.

`SetPresentation(~B, s)`

Set the presentation used for B to the presentation indicated by s. Possible values for s are `"Artin"` and `"BKL"`.

`GetForceCFP(B)`

Returns whether arithmetic operations with elements of B are always done using representations of the elements as products of simple elements.

`SetForceCFP(~B, b)`

By default, arithmetic operations with elements of B are always done using representations of the elements as products of simple elements. If necessary, such representations are computed.

Experienced users can turn this feature off for a braid group B using the procedure `SetForceCFP` with b set to `false`.

`GetElementPrintFormat(B)`

Returns a string s indicating the format currently used for printing elements of B. s is one of `"Word"`, `"CFP"` or `"Both"`.

`SetElementPrintFormat(~B, s)`

When printing an element of a braid group B, by default both the representation as word in the generators and the representation as product of simple elements in the presentation selected for B are printed.

Experienced users can use the procedure `SetElementPrintFormat` for changing the print format. Possible values for s are `"Word"`, `"CFP"` and `"Both"`.

`NumberOfStrings(B)`

Given a braid group B, return the number of strings on which B is defined.

`NumberOfGenerators(B)`

`Ngens(B)`

Given a braid group B, return the number of Artin generators of B. Note that the number of Artin generators is returned regardless of the presentation selected for B.

73.3 Creating Elements of a Braid Group

This section describes the facilities for creating elements of a braid group.

`Representative(B)`

`Rep(B)`

Given a braid group B, return a representative of B.

`Identity(B)`

`Id(B)`

`B ! 1`

Given a braid group B, return the identity element of B.

`FundamentalElement(B: parameters)`

`Presentation` MonStgElt *Default :*

Return the fundamental element for the presentation of B indicated by the parameter `Presentation`. Possible values for this parameter are `"Artin"` and `"BKL"`. If the parameter `Presentation` is not used, the fundamental element for the presentation currently selected for B is returned.

`Generators(B: parameters)`

`Presentation` MonStgElt *Default :*

Return a sequence containing the generators for the presentation of B indicated by the parameter `Presentation`. Possible values for this parameter are `"Artin"` and `"BKL"`. If the parameter `Presentation` is not used, a sequence containing the generators for the presentation currently selected for B is returned.

`B . i`

Given a braid group B on n strings and an integer i, where $0 < |i| < n$, return the $|i|$-th Artin generator σ_i, if $i > 0$, or its inverse $\sigma_{|i|}^{-1}$, if $i < 0$.

`B . T`

Given a braid group B on n strings and an tuple $T =< r, t >$, where $1 \leq |t| < |r| \leq n$, return the BKL generator $a_{|r|,|t|}$, if $r, t > 0$, or its inverse $a^{-1}_{|r|,|t|}$ otherwise.

`B ! [ i1, ..., ik ]`

Given a braid group B on n strings and a sequence $[i_1, \cdots, i_k]$ of integers satisfying $0 < |i_j| < n$ $(j = 1, \ldots, k)$, return the element of B given by the product

$$\sigma_{|i_1|}^{\mathrm{sgn}(i_1)} \cdots \sigma_{|i_k|}^{\mathrm{sgn}(i_k)}.$$

`B ! [ T1, ..., Tk ]`

Given a braid group B on n strings and a sequence $[T_1, \cdots, T_k]$ of tuples satisfying $T_j =< r_j, t_j >$, $1 \leq |t_j| < |r_j| \leq n$ $(j = 1, \ldots, k)$, return the element of B given by the product

$$a_{|r_1|,|t_1|}^{e_1} \cdots a_{|r_k|,|t_k|}^{e_k}$$

where $e_j = 1$ if $r_j, t_j > 0$ and $e_j = -1$ otherwise $(j = 1, \ldots, k)$.

`B ! p`

Given a braid group B on n strings and a permutation p on n points, return the simple element defined by p in the presentation currently selected for B as new element of B.

Note that the result in general depends on the presentation selected for B. Note further that in the BKL presentation, only permutations which are products of parallel descending cycles correspond to simple elements; attempting to coerce an invalid permutation will result in a runtime error. The function `IsProductOfParallelDescendingCycles` can be used to test whether a given permutation corresponds to a BKL simple element.

`B ! [ p1, ...,pk ]`

Given a braid group B on n strings and a sequence $[p_1, \ldots, p_k]$ of permutations on n points, return the product $c_1 \cdots c_k$ as new element of B, where c_j is the simple element defined by p_j in the presentation currently selected for B $(j = 1, \ldots, k)$.

Note that the result in general depends on the presentation selected for B. Note further that in the BKL presentation, only permutations which are products of parallel descending cycles correspond to simple elements; attempting to coerce a sequence containing an invalid permutation will result in a runtime error. The function `IsProductOfParallelDescendingCycles` can be used to test whether a given permutation corresponds to a BKL simple element.

`B ! T`

Given a braid group B on n strings and an tuple $T =< s, l, S, r >$, where s is either the string `"Artin"` or the string `"BKL"`, l and r are integers and S is a sequence $[p_1, \ldots, p_k]$ of permutations on n points, return the product $D^l c_1 \cdots c_k D^r$ as new element of B, where D is the fundamental element and c_j is the simple element defined by p_j $(j = 1, \ldots, k)$ in the presentation indicated by s.

Note that in the BKL presentation, only permutations which are products of parallel descending cycles correspond to simple elements; if S contains an invalid permutation, a runtime error will result. Whether the elements of a given sequence S correspond to BKL simple elements can be tested using the function `IsProductOfParallelDescendingCycles`.

`IsProductOfParallelDescendingCycles(p)`

Given a permutation p on n points, return whether p is a product of parallel descending cycles, that is, whether p defines a simple element in the BKL monoid on n strings.

`Random(B, r, s, m, n:` *parameters*`)`

`RandomCFP(B, r, s, m, n:` *parameters*`)`

`Random(B:` *parameters*`)`

`RandomCFP(B:` *parameters*`)`

`Presentation` MONSTGELT *Default :*

Given a braid group B and integers r, s, m, n, satisfying $r \leq s$ and $0 \leq m \leq n$, a pseudo-random element of B is constructed as follows. Let D be the fundamental element and C the set of simple elements for the presentation indicated by the parameter `Presentation`. First, integers $e \in [r, s]$ and $l \in [m, n]$ are chosen using uniform distributions on these sets. Then, for $i = 1, \ldots, l$, $c_i \in C$ is chosen using a uniform distribution on C and the element $D^e c_1 \cdots c_l$ is returned.

If no value is given for the parameter `Presentation`, the presentation selected for B is used.

The versions with a single argument are short for `Random(B, 0, 0, 0, 42)`.

`Random(B, m, n:` *parameters*`)`

`RandomWord(B, m, n:` *parameters*`)`

`RandomWord(B:` *parameters*`)`

`Presentation` MONSTGELT *Default :*

Given a braid group B and two integers $0 \leq m \leq n$, `Random(B, m, n)` returns a pseudo-random element of B constructed as follows. First, a length $l \in [m, n]$ is chosen using a uniform distribution. Then, for $i = 1, \ldots, l$, $g_i \in X \cup X^{-1} \setminus \{g_{i-1}^{-1}\}$ is chosen using a uniform distribution on this set. Here, X is the set of generators of the presentation indicated by the parameter `Presentation` and X^{-1} is the set of generator inverses.

If no value is given for the parameter `Presentation`, the presentation selected for B is used.

The signature `RandomWord(B)` is short for `RandomWord(B, 0, 42)`.

Example H73E1

We construct the braid group B on 6 strings and the symmetric group S on 6 points.

```
> S := Sym(6);
> B := BraidGroup(6);
> B;
GrpBrd : B on 6 strings
```

By default, B is created using the Artin presentation.

```
> GetPresentation(B);
Artin
```

We now define the fundamental element with respect to the BKL presentation of B and print this element with respect to the presentation currently used for B, that is, with respect to the Artin presentation. Note that both a word in the Artin generators and a representation of the element in terms of Artin simple elements are printed.

```
> D_BKL := FundamentalElement(B : Presentation := "BKL");
> D_BKL;
B.5 * B.4 * B.3 * B.2 * B.1
<Artin, 0, [
    (1, 6, 5, 4, 3, 2)
], 0>
> GetElementPrintFormat(B);
Both
```

We print the BKL generator $a_{3,1}$.

```
> B.<3,1>;
B.2 * B.1 * B.2^-1
<Artin, 0, [
    (1, 3, 2),
    (1, 6)(2, 5, 3, 4)
], -1>
```

Next we change the format for printing elements of B using the function `SetElementPrintFormat` so that only a representation in terms of simple elements is printed.

```
> SetElementPrintFormat(~B, "CFP");
```

We now define and print several elements of B, illustrating the use of some of the functions described in the previous section.

First we create a pseudo-random element of B as product of 3 random simple elements for the Artin presentation.

```
> u := Random(B, 0, 0, 3, 3);
> u;
```

```
<Artin, 0, [
    (1, 6)(3, 5, 4),
    (1, 3)(2, 6)(4, 5),
    (2, 3)
], 0>
```

Next we define an element of B by a product of simple elements for the BKL presentation using the coercion operator '!'. Note that printing of this element is still done with respect to the Artin presentation.

```
> v := B ! <"BKL", 0, [ S | (1,6)(3,5,4), (1,3)(4,5)], 0>;
> v;
<Artin, 0, [
    (1, 6, 5, 4, 3, 2),
    (1, 2, 6)(3, 5),
    (2, 4, 5, 6),
    (1, 6)(2, 4, 3, 5)
], -2>
```

Finally, we look at the simple elements defined by the permutations $p = (1, 3)(4, 2)$ and $q = (1, 4, 3)$ on 6 points.

```
> p := S ! (1,3)(4,2);
> q := S ! (1,4,3);
```

Creating the simple elements for the Artin presentation defined by p and q is straightforward using the coercion operator '!'.

```
> p_Artin := B!p;
> p_Artin;
<Artin, 0, [
    (1, 3)(2, 4)
], 0>
```

```
> q_Artin := B!q;
> q_Artin;
<Artin, 0, [
    (1, 4, 3)
], 0>
```

We now change the presentation used for B to the BKL presentation. Note that this also changes the presentation with respect to which elements are printed.

```
> SetPresentation(~B, "BKL");
> GetPresentation(B);
BKL
```

The attempt to define a simple element for the BKL presentation using the permutation p fails.

```
>  p_BKL := B!p;

>> p_BKL := B!p;
             ^
```

```
Runtime error in '!': Illegal coercion
LHS: GrpBrd
RHS: GrpPermElt
```

We should have been more careful: using the function IsProductOfParallelDescendingCycles we see that p is not a product of parallel descending cycles and hence does not define a simple element for the BKL presentation.

```
> IsProductOfParallelDescendingCycles(p);
false
```

q, on the other hand, does define a simple element for the BKL presentation and we can coerce q to an element of B using the operator '!'.

```
> IsProductOfParallelDescendingCycles(q);
true
> q_BKL := B!q;
> q_BKL;
<BKL, 0, [
    (1, 4, 3)
], 0>
```

Note however, that the simple element for the BKL presentation defined by q and the simple element for the Artin presentation defined by q are different elements of B! (The comparison operator eq is described in Section 73.4.4.)

```
> q_BKL eq q_Artin;
false
```

The representations of the Artin simple elements defined by p and q in terms of BKL simple elements have no obvious connection to p and q, respectively.

```
> p_Artin;
<BKL, 0, [
    (1, 3, 2),
    (2, 4, 3)
], 0>

> q_Artin;
<BKL, 0, [
    (1, 4, 3, 2),
    (2, 3)
], 0>
```

73.4 Working with Elements of a Braid Group

73.4.1 Accessing Information

This sections describes how the internal representations of an element and a number of basic invariants can be accessed.

`Parent(u)`

Given an element u of a braid group B, return the parent group of u, that is B.

`#u`

Given an element u of a braid group B, return the length of the representing word in the generators corresponding to the presentation selected for B. Note that this is not an invariant of u.

`CanonicalFactorRepresentation(u: parameters)`

`CFP(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return a tuple $T =< s, l, S, r >$ describing the representation in terms of simple elements for the presentation indicated by the value of the parameter `Presentation`. If no value for `Presentation` is given, the presentation selected for B is used.

The interpretation of the components of T is as follows: s is a string, either equal to `"Artin"` or equal to `"BKL"` indicating the presentation, l and r are integers and S is a sequence $[p_1, \ldots, p_k]$ of permutations on n points, such that

$$D^l c_1 \cdots c_k D^r$$

is the representation of u in terms of simple elements, where D is the fundamental element and c_j is the simple element defined by p_j $(j = 1, \ldots, k)$ in the presentation indicated by s.

`WordToSequence(u: parameters)`

`ElementToSequence(u: parameters)`

`Eltseq(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return a sequence describing the representing word in the generators corresponding to the presentation indicated by the value of the parameter `Presentation`. If no value for `Presentation` is given, the presentation selected for B is used.

For a representing word

$$\sigma_{i_1}^{e_1} \cdots \sigma_{i_k}^{e_k}$$

in the Artin generators with $0 < i_j < n$ and $e_j \in \{-1, 1\}$ for $j = 1, \ldots, k$, the sequence of integers

$$[e_1 i_1, \ldots, e_k i_k]$$

is returned.

For a representing word

$$a_{r_1,t_1}^{e_1} \ldots a_{r_k,t_k}^{e_k}$$

in the BKL generators with $1 \leq t_j < r_j \leq n$ and $e_j \in \{-1, 1\}$ for $j = 1, \ldots, k$, the sequence of tuples

$$[< e_1 r_1, e_1 t_1 >, \ldots, < e_k r_k, e_k t_k >]$$

is returned.

`InducedPermutation(u)`

Given an element u of a braid group B on n strings, return the permutation on n points induced by u acting on the strings on which B is defined.

For the following description of the functions `CanonicalLength`, `Infimum` and `Supremum`, let D be the fundamental element and let $D^l c_1 \cdots c_k$ be the left normal form of the element u of the braid group B in the presentation indicated by the value of the parameter `Presentation`. If no value for `Presentation` is given, the presentation selected for B is used.

`CanonicalLength(u:` *parameters*`)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return the canonical length k of u for the appropriate presentation of B. The argument is converted into left normal form.

`Infimum(u:` *parameters*`)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return the infimum l of u for the appropriate presentation of B. The argument is converted into left normal form.

`Supremum(u:` *parameters*`)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return the supremum $l + k$ of u for the appropriate presentation of B. The argument is converted into left normal form.

For the following description of the functions `SuperSummitCanonicalLength`, `SuperSummitInfimum` and `SuperSummitSupremum`, let D be the fundamental element and let $D^l c_1 \cdots c_k$ be the normal form of a representative of the super summit set the element u of the braid group B with respect to the presentation indicated by the value of the parameter `Presentation`. If no value for `Presentation` is given, the presentation selected for B is used.

`SuperSummitCanonicalLength(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return the canonical length k of a representative of the super summit set of u with respect to the appropriate presentation of B, that is, the minimal canonical length among all conjugates of u. The argument is converted into left normal form.

`SuperSummitInfimum(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return the infimum l of a representative of the super summit set of u with respect to the appropriate presentation of B, that is, the maximal infimum among all conjugates of u. The argument is converted into left normal form.

`SuperSummitSupremum(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return the supremum $l+k$ of a representative of the super summit set of u with respect to the appropriate presentation of B, that is, the minimal supremum among all conjugates of u. The argument is converted into left normal form.

Example H73E2

We define the braid group B on 6 strings using the BKL presentation and create an element u as a random word of length between 5 and 10 in the BKL generators.

```
> B := BraidGroup(6 : Presentation := "BKL");
> u := RandomWord(B, 5, 10);
```

The parent group of u can be accessed using the function `Parent`.

```
> Parent(u);
GrpBrd : B on 6 strings
```

As word in the BKL generators, u has length 5. We define a sequence describing the representation of u as word in the BKL generators.

```
> #u;
5
> seq_BKL := WordToSequence(u);
> seq_BKL;
[ <5, 1>, <5, 2>, <5, 4>, <4, 1>, <2, 1> ]
```

When we ask for a representation as word in the Artin generators, such a representation is created automatically.

```
> seq_Artin := WordToSequence(u : Presentation := "Artin");
> seq_Artin;
```

```
[ 4, 3, 2, 1, 2, 1, -2, -3, 1 ]
```

We now define the permutation p induced by u on the strings on which B is defined.

```
> p := InducedPermutation(u);
> p;
(4, 5)
```

The representation of u in terms of simple elements for the Artin presentation can be obtained using the function CanonicalFactorRepresentation.

```
> CanonicalFactorRepresentation(u : Presentation := "Artin");
<Artin, 0, [
    (1, 5, 4, 3),
    (1, 2),
    (1, 6)(2, 5, 3),
    (5, 6)
], -1>
```

We now compute the canonical lengths of u with respect to the Artin presentation and with respect to the BKL presentation. Note that these lengths are different.

```
> CanonicalLength(u : Presentation := "Artin");
4
> CanonicalLength(u : Presentation := "BKL");
3
```

Finally, we compute for both presentations the canonical lengths of a super summit representative of u.

```
> SuperSummitCanonicalLength(u : Presentation := "Artin");
2
> SuperSummitCanonicalLength(u : Presentation := "BKL");
3
```

Obviously, u does not belong to its super summit set with respect to the Artin presentation. (We cannot tell for the BKL presentation from the information we have computed.)

73.4.2 Computing Normal Forms of Elements

This section describes functions and procedures for computing various normal forms of elements of a braid group B. All normal forms are defined in terms of representations of elements as products of simple elements and depend on the presentation of B which is used. The functions documented in this section all accept a parameter `Presentation` which can be used to specify the presentation of B with respect to which the computation should be performed. Possible values for this parameter are the strings `"Artin"` and `"BKL"` If no value is given for `Presentation`, the presentation selected for B is used.

`LeftNormalForm(u: parameters)`

`NormalForm(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return a new element of B defined by the left normal form of u with respect to the indicated presentation.

`LeftNormalForm(~u: parameters)`

`NormalForm(~u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, bring u into left normal form with respect to the indicated presentation.

`RightNormalForm(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return a new element of B defined by the right normal form of u with respect to the indicated presentation.

`RightNormalForm(~u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, bring u into right normal form with respect to the indicated presentation.

`LeftMixedCanonicalForm(u: parameters)`

`MixedCanonicalForm(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return two tuples T_1 and T_2 defining products $v_1 \cdots v_k$ and $w_1 \cdots w_l$, respectively, of simple elements for the indicated presentation, such that $v_1 \cdots v_k$ and $w_1 \cdots w_l$ are in left normal form, the left-gcd of v_1 and w_1 is trivial and

$$u = (v_1 \cdots v_k)^{-1}(w_1 \cdots w_l).$$

See the entry for `CanonicalFactorRepresentation` for a description of the tuple format. Note that the tuples can be coerced into elements of B using the coercion operator '`!`'.

`RightMixedCanonicalForm(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return two tuples T_1 and T_2 defining products $v_1 \cdots v_k$ and $w_1 \cdots w_l$, respectively, of simple elements for the indicated presentation, such that $v_1 \cdots v_k$ and $w_1 \cdots w_l$ are in right normal form, the right-gcd of v_1 and w_1 is trivial and

$$u = (v_1 \cdots v_k)(w_1 \cdots w_l)^{-1}.$$

See the entry for `CanonicalFactorRepresentation` for a description of the tuple format. Note that the tuples can be coerced into elements of B using the coercion operator '`!`'.

Example H73E3

We define the braid group B on 6 strings using the Artin presentation, set the print format for elements to `"CFP"` and define an element u of B.

```
> B := BraidGroup(6);
> SetElementPrintFormat(~B, "CFP");
>
> u := B ! <"Artin",
>          0,
>          [ Sym(6) | (1,6)(3,5,4), (1,3)(2,6)(4,5), (2,3)],
>          0>;
> u;
<Artin, 0, [
    (1, 6)(3, 5, 4),
    (1, 3)(2, 6)(4, 5),
    (2, 3)
], 0>
```

We compute and print the left normal form of u with respect to the Artin presentation of B.

```
> u_Artin := LeftNormalForm(u);
> u_Artin;
<Artin, 1, [
    (1, 2, 6, 5, 4, 3),
    (2, 3)
], 0>
```

We now compute the left normal form of u with respect to the BKL presentation of B. Since elements are printed with respect to the presentation selected for the parent group, that is, in the Artin presentation in our example, we use the function `CanonicalFactorRepresentation` to print the representation in terms of simple elements for the BKL presentation.

```
> u_BKL := LeftNormalForm(u : Presentation := "BKL");
> CFP(u_BKL : Presentation := "BKL");
<BKL, 2, [
    (2, 6, 5, 4, 3),
```

```
    (2, 6, 5, 4),
    (2, 6, 5),
    (2, 6),
    (2, 3)
], 0>
```

We define another element v of B in left normal form.

```
> v := LeftNormalForm(B.5*B.2^-2*B.4*B.3^-1*B.5^-1*B.3^-1*B.5);
> v;
<Artin, -3, [
    (1, 6)(2, 4, 3, 5),
    (1, 2, 6)(3, 5),
    (1, 6, 2, 5)(3, 4),
    (3, 5)(4, 6)
], 0>
```

As can easily be read off the representation of v in terms of simple elements which is in left normal form, v has infimum -3, canonical length 4 and supremum 1 with respect to the Artin presentation.

```
> Infimum(v);
-3
> CanonicalLength(v);
4
> Supremum(v);
1
```

Note that infimum, canonical length and supremum of an element can also be obtained from its right normal form.

```
> RightNormalForm(v);
<Artin, 0, [
    (4, 6, 5),
    (1, 6)(2, 5, 3, 4),
    (1, 6)(2, 4, 5),
    (1, 6)(2, 5)
], -3>
```

73.4.3 Arithmetic Operators and Functions for Elements

This section describes the basic arithmetic operations for elements of a braid group. Strictly speaking, all functions should be considered as functions on *representatives* of elements, that is, words in the generators or products of simple elements.

Unless stated otherwise, arithmetic operations with elements of a braid group B are performed using representations with respect to the presentation selected for B. This presentation can be changed using the function SetPresentation.

By default, arithmetic operations with elements of B are performed using representations in terms of simple elements; such representations are created if necessary. Experienced users can change this behaviour, if desired, using the function SetForceCFP.

The complexity of all basic arithmetic operations is linear in the length of the representations of the input elements. No normalisations are performed automatically, as doing so would restrict the user's control of operations with elements. It is, however, recommended to use the function NormalForm or its procedural version in time critical situations to limit the length of representations of elements; see Example H73E4.

`u * v`

Given elements u and v belonging to the same braid group B, return the product uv as a new element of B.

`u *:= v`

Given elements u and v belonging to the same braid group B, replace u with the product uv.

`u / v`

Given elements u and v belonging to the same braid group B, return the product uv^{-1} as a new element of B.

`u /:= v`

Given elements u and v belonging to the same braid group B, replace u with the product uv^{-1}.

`u ^ n`

Given an element u of a braid group B and an integer n, return the power u^n as a new element of B.

`u ^:= n`

Given an element u of a braid group B and an integer n, replace u with the power u^n.

`u ^ v`

Given elements u and v belonging to the same braid group B, return the conjugate $u^v = v^{-1}uv$ as a new element of B.

`u ^:= v`

Given elements u and v belonging to the same braid group B, replace u with the conjugate $u^v = v^{-1}uv$.

`Inverse(u)`

Given an element u of a braid group B, return its inverse u^{-1} as a new element of B.

`Inverse(~u)`

Given an element u of a braid group B, replace u with its inverse u^{-1}.

`LeftConjugate(u, v)`

Given elements u and v belonging to the same braid group B, return the "left conjugate" vuv^{-1} as a new element of B.

`LeftConjugate(~u, v)`

Given elements u and v belonging to the same braid group B, replace u with the "left conjugate" vuv^{-1}.

`LeftDiv(u, v)`

Given elements u and v belonging to the same braid group B, return the product $u^{-1}v$ as a new element of B.

`LeftDiv(u, ~v)`

Given elements u and v belonging to the same braid group B, replace v with the product $u^{-1}v$.

The following functions `Cycle` and `Decycle` accept a parameter `Presentation` which can be set either to `"Artin"` or to `"BKL"`. The results of the cycling and decycling operations are defined in terms of the left normal form $D^l c_1 \cdots c_k$ of the argument $u \in B$ in terms of simple elements for a presentation of B and the results in general depend on the presentation used. The results of these functions are returned in left normal form.

If no value for the parameter `Presentation` is given, the presentation selected for the parent group of the argument will be used.

`Cycle(u: parameters)`

`Presentation` MONSTGELT *Default :*

Given an element $u \in B$ with left normal form $D^l c_1 \cdots c_k$, return the result of a cycling operation on u, that is,

$$u^{(c_1^{D^{-l}})}$$

as new element of B in left normal form.

`Cycle(~u: parameters)`

`Presentation` MONSTGELT *Default :*

Given an element $u \in B$ with left normal form $D^l c_1 \cdots c_k$, replace u by the result of a cycling operation on u, that is, by

$$u^{(c_1^{D^{-l}})}$$

in left normal form.

`Decycle(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element $u \in B$ with left normal form $D^l c_1 \cdots c_k$, return the result of a decycling operation on u, that is,

$$u^{(c_k^{-1})}$$

as new element of B in left normal form.

`Decycle(~u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element $u \in B$ with left normal form $D^l c_1 \cdots c_k$, replace u by the result of a decycling operation on u, that is, by

$$u^{(c_k^{-1})}$$

in left normal form.

Example H73E4

We illustrate the importance of limiting the length of representations of elements using the function `NormalForm` when performing a sequence of arithmetic operations on an element.
Consider the following computation in the braid group B on 6 strings. Starting with an element w, we repeatedly replace w by the product ww^{σ_1} where σ_1 is the first Artin generator of B.
A naive way of implementing this computation would be as follows.

```
> B := BraidGroup(6);
> u := B.5*B.2^-2*B.4*B.3^-1;
> v := B.1;
> N := 14;
>
> T := Cputime();
> w := u;
> for i := 1 to N do
>    w := w * w^v;
> end for;
```

This, however, yields an extremely complicated representation for the result; the representation in terms of simple elements has the length 114686.

```
> #CFP(w)[3];
114686
```

Performing subsequent computations with the result, for example computing its normal form, is very expensive.

```
> NormalForm(~w);
> print "total time used: ", Cputime()-T;
```

```
total time used:  149.229
```

One might be tempted to solve this problem by working only with elements in normal form, that is, by bringing every result of an arithmetic operation into normal form after computing it.
Using this approach, computing the result of the above iteration is indeed much faster.

```
> T := Cputime();
> w := u;
> for i := 1 to N do
>    t := w^v;
>    NormalForm(~t);
>    w := w * t;
>    NormalForm(~w);
> end for;
> print "total time used: ", Cputime()-T;
total time used:  0.53
```

However, this strategy is not optimal either. For the above example, the optimal performance is obtained if the result is normalised every third pass through the iteration.

```
> T := Cputime();
> w := u;
> for i := 1 to N do
>    w := w * w^v;
>    if i mod 3 eq 0 then
>       NormalForm(~w);
>    end if;
> end for;
> NormalForm(~w);
> print "total time used: ", Cputime()-T;
total time used:  0.171
```

Unfortunately, the frequency of normalisation giving best results depends heavily on the situation, that is, both on the arithmetic operations and on the characteristics of the arguments.
As a rule of thumb, the effects of normalising results too frequently are less of a problem than normalising results not often enough or not at all.

73.4.4 Boolean Predicates for Elements

This section describes the tests for membership, equality and partial orderings which are available for elements of a braid group B.

Unless stated otherwise, all computations are performed in the presentation selected for B or in the presentation specified by the value of the parameter `Presentation`, either `"Artin"` or `"BKL"` if a value for this parameter is given.

`u in B`

Given an element u of a braid group and a braid group B, return `true` if $u \in B$ and `false` otherwise.

`u notin B`

Given an element u of a braid group and a braid group B, return `false` if $u \in B$ and `true` otherwise.

`IsEmptyWord(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group, return `true` if u is the represented by the empty word in the specified presentation and `false` otherwise.

`AreIdentical(u, v: parameters)`

`Presentation` MonStgElt *Default :*

Given elements u and v belonging to the same braid group B, return `true` if u and v are represented by identical words in the specified presentation and `false` otherwise.

`IsSimple(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group, return `true` if u is a simple element with respect to the specified presentation and `false` otherwise. The argument is converted into normal form.

`IsSuperSummitRepresentative(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group, return `true` if u is an element of its super summit set with respect to the specified presentation and `false` otherwise. The argument is converted into normal form.

`IsUltraSummitRepresentative(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group, return `true` if u is an element of its ultra summit set with respect to the specified presentation and `false` otherwise. The argument is converted into normal form.

`IsIdentity(u: parameters)`

`IsId(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return `true` if u is the identity element of B and `false` otherwise. The argument is converted into normal form.

`u eq v`

Given elements u and v belonging to the same braid group B, return `true` if $u = v$ and `false` otherwise. Both arguments are converted into normal form.

`u ne v`

Given elements u and v belonging to the same braid group B, return `false` if $u = v$ and `true` otherwise. Both arguments are converted into normal form.

`u le v`

`IsLE(u, v: parameters)`

`IsLe(u, v: parameters)`

`Presentation` MonStgElt *Default :*

Given elements u and v belonging to the same braid group B, return `true` if $u \preceq v$, that is, if $u^{-1}v$ is a positive element, with respect to the specified presentation and `false` otherwise.

Note that the parameter `Presentation` is not available for the operator version of this predicate.

`u ge v`

`IsGE(u, v: parameters)`

`IsGe(u, v: parameters)`

`Presentation` MonStgElt *Default :*

Given elements u and v belonging to the same braid group B, return `true` if $u \succeq v$, that is, if uv^{-1} is a positive element, with respect to the specified presentation and `false` otherwise.

Note that the parameter `Presentation` is not available for the operator version of this predicate.

`IsConjugate(u, v: parameters)`

`Presentation` MonStgElt *Default :*

Given elements u and v belonging to the same braid group B, return `true` and an element $c \in B$ satisfying $u^c = v$ if u and v are conjugate and return `false` otherwise.

The function first computes representatives u_s and v_s of the ultra summit sets of u and v, respectively, with respect to the specified presentation. If this does not prove that the elements are not conjugate, the function tries to compute elements of the ultra summit set of u until either the element v_s is found, proving that u and v are conjugate, or the ultra summit set of u is seen not to contain v_s, proving that u and v are not conjugate. See Example H73E8 for a more detailed description.

Note that testing elements for conjugacy is a hard problem and may require significant amounts of memory and CPU time.

Example H73E5

We define the braid group B on 6 strings using the Artin presentation and set the print format for elements to "CFP".

```
> B:= BraidGroup(6);
> SetElementPrintFormat(~B, "CFP");
```

(1) We create pseudo-random elements of B until we find an element u which is contained in its super summit set with respect to the Artin presentation of B.

```
> repeat
>    u := Random(B, 5, 10);
> until IsSuperSummitRepresentative(u);
> NormalForm(u);
<Artin, -2, [
    (1, 5)(2, 3, 6),
    (1, 5, 6, 3, 2, 4),
    (2, 6)(3, 4, 5)
], 0>
```

u is not contained in its super summit set with respect to the BKL presentation of B, showing that the super summit set of an element in general depends on the presentation with respect to which it is defined.

```
> IsSuperSummitRepresentative(u : Presentation := "Artin");
true
> IsSuperSummitRepresentative(u : Presentation := "BKL");
false
```

(2) This example shows that the Artin presentation and the BKL presentation give rise to distinct partial orderings on B.

$\sigma_1^{-1}\sigma_2\sigma_1$ has negative infimum with respect to the Artin presentation and hence is not a positive element with respect to this presentation.

```
> Infimum(B.1^-1*B.2*B.1 : Presentation := "Artin");
-1
```

Consequently, $\sigma_1 \not\preceq \sigma_2\sigma_1$ in the partial ordering defined with respect to the Artin presentation.

```
> B.1 le B.2*B.1;
false
```

We can also use the function version to check this.

```
> IsLE(B.1, B.2*B.1 : Presentation := "Artin");
false
```

However, $\sigma_1^{-1}\sigma_2\sigma_1$ is equal to the BKL generator $a_{3,1}$ and hence is, in particular, a positive element with respect to the BKL presentation. in the BKL generators.

```
> B.1^-1*B.2*B.1 eq B.<3,1>;
```

```
true
```

Hence, $\sigma_1 \preceq \sigma_2\sigma_1$ in the partial ordering defined with respect to the BKL presentation.

```
> IsLE(B.1, B.2*B.1 : Presentation := "BKL");
true
```

(3) We change the print format for elements of B so that only words in the Artin generators are printed.

```
> SetElementPrintFormat(~B, "Word");
```

Inducing permutations with different cycle structure, σ_1 and $\sigma_1\sigma_2$ cannot be conjugate in B.

```
> InducedPermutation(B.1);
(1, 2)
> InducedPermutation(B.2*B.1);
(1, 2, 3)
> IsConjugate(B.1, B.2*B.1);
false
```

σ_1 and σ_2, however, are conjugate in B. We compute a conjugating element c.

```
> res, c := IsConjugate(B.1, B.2);
> res;
true
> NormalForm(c);
B.2 * B.1
```

c, as desired, conjugates σ_1: s_1: to σ_2: s_2:.

```
> B.1^c eq B.2;
true
```

73.4.5 Lattice Operations

This section describes the functions available for computing lattice operations, least common multiple and greatest common divisor, for elements of a braid group B. The results of all lattice operations depend on the presentation used for B and on the partial ordering considered.

The functions documented in this section all accept a parameter `Presentation` which can be used to specify the presentation of B with respect to which the computation should be performed. Possible values for this parameter are the strings `"Artin"` and `"BKL"`. If no value is given for `Presentation`, the presentation selected for B is used.

`LeftGCD(u, v: parameters)`
`LeftGcd(u, v: parameters)`
`LeftGreatestCommonDivisor(u, v: parameters)`
`GCD(u, v: parameters)`
`Gcd(u, v: parameters)`
`GreatestCommonDivisor(u, v: parameters)`

`Presentation` MonStgElt *Default :*

Given elements u and v belonging to the same braid group B, return the left-gcd of u and v, that is, the with respect to $\preceq$ maximal element d of B satisfying $d \preceq u$ and $d \preceq v$. Here, $\preceq$ is the partial ordering on B defined as follows: $a \preceq b$ iff $a^{-1}b$ is representable as a positive word in the specified presentation of B.

`RightGCD(u, v: parameters)`
`RightGcd(u, v: parameters)`
`RightGreatestCommonDivisor(u, v: parameters)`

`Presentation` MonStgElt *Default :*

Given elements u and v belonging to the same braid group B, return the right-gcd of u and v, that is, the with respect to $\succeq$ maximal element d of B satisfying $u \succeq d$ and $v \succeq d$. Here, $\succeq$ is the partial ordering on B defined as follows: $a \succeq b$ iff ab^{-1} is representable as a positive word in the specified presentation of B.

`LeftGCD(S: parameters)`
`LeftGcd(S: parameters)`
`LeftGreatestCommonDivisor(S: parameters)`
`GCD(S: parameters)`
`Gcd(S: parameters)`
`GreatestCommonDivisor(S: parameters)`

`Presentation` MonStgElt *Default :*

Given a set or a sequence S containing elements of a braid group B, return the left-gcd of the elements of S, that is, the with respect to $\preceq$ maximal element d of B satisfying $d \preceq s$ for all $s \in S$, where $\preceq$ is defined as above.

`RightGCD(S: parameters)`
`RightGcd(S: parameters)`
`RightGreatestCommonDivisor(S: parameters)`

`Presentation` MonStgElt *Default :*

Given a set or a sequence S containing elements of a braid group B, return the right-gcd of the elements of S, that is, the with respect to $\succeq$ maximal element d of B satisfying $s \succeq d$ for all $s \in S$, where $\succeq$ is defined as above.

`LeftLCM(u, v: parameters)`
`LeftLcm(u, v: parameters)`
`LeftLeastCommonMultiple(u, v: parameters)`
`LCM(u, v: parameters)`
`Lcm(u, v: parameters)`
`LeastCommonMultiple(u, v: parameters)`

`Presentation` MONSTGELT *Default :*

Given elements u and v belonging to the same braid group B, return the left-lcm of u and v, that is, the with respect to $\preceq$ minimal element d of B satisfying $u \preceq d$ and $v \preceq d$, where $\preceq$ is defined as above.

`RightLCM(u, v: parameters)`
`RightLcm(u, v: parameters)`
`RightLeastCommonMultiple(u, v: parameters)`

`Presentation` MONSTGELT *Default :*

Given elements u and v belonging to the same braid group B, return the right-lcm of u and v, that is, the with respect to $\succeq$ minimal element d of B satisfying $d \succeq u$ and $d \succeq v$, where $\succeq$ is defined as above.

`LeftLCM(S: parameters)`
`LeftLcm(S: parameters)`
`LeftLeastCommonMultiple(S: parameters)`
`LCM(S: parameters)`
`Lcm(S: parameters)`
`LeastCommonMultiple(S: parameters)`

`Presentation` MONSTGELT *Default :*

Given a set or a sequence S containing elements of a braid group B, return the left-gcd of the elements of S, that is, the with respect to $\preceq$ minimal element d of B satisfying $s \preceq d$ for all $s \in S$, where $\preceq$ is defined as above.

`RightLCM(S: parameters)`
`RightLcm(S: parameters)`
`RightLeastCommonMultiple(S: parameters)`

`Presentation` MonStgElt *Default :*

Given a set or a sequence S containing elements of a braid group B, return the right-lcm of the elements of S, that is, the with respect to $\succeq$ minimal element d of B satisfying $d \succeq s$ for all $s \in S$, where $\succeq$ is defined as above.

Example H73E6

We define the braid group B on 6 strings.

```
> B := BraidGroup(6);
> SetElementPrintFormat(~B, "CFP");
```

(1) For both Artin and BKL presentation, the fundamental element is the (left or right) least common multiple of the generators. We check this for the Artin presentation...

```
> D_Artin := LeftLCM({B.i : i in [1..NumberOfGenerators(B)]});
> D_Artin eq FundamentalElement(B);
true
> D_Artin eq RightLCM({B.i : i in [1..NumberOfGenerators(B)]});
true
```

...and for the BKL presentation.

```
> idx := { <r,t> : r,t in {1..NumberOfStrings(B)} | r gt t };
> D_BKL := LeftLCM({B.T : T in idx} : Presentation := "BKL");
> D_BKL eq FundamentalElement(B : Presentation := "BKL");
true
> D_BKL eq RightLCM({B.T : T in idx} : Presentation := "BKL");
true
```

In general, left and right least common multiple of elements are different.

```
> LeftLCM(B.1,B.1*B.2) eq RightLCM(B.1,B.1*B.2);
false
```

(2) For both Artin and BKL presentation, the following hold. Let D denote the fundamental element.

- The simple elements are those positive elements s satisfying $s \preceq D$ (or $D \succeq s$).
- A product $u_1 \cdots u_r$ of simple elements is in left normal form, if and only if the left greatest common divisor of $u_i^{-1}D$ and u_{i+1} is trivial for all $i = 1, \ldots, r-1$.

We illustrate this for the Artin presentation.

```
> D := FundamentalElement(B);
> forall{ s : s in Sym(6) | B!1 le B!s and B!s le D };
true
> forall{ s : s in Sym(6) | D ge B!s and B!s ge B!1 };
```

```
true
```

We create an element u as product of random simple elements.

```
> u := Random(B, 0, 0, 3, 5);
> u;
<Artin, 0, [
    (1, 5, 2)(3, 6),
    (1, 6, 5, 3),
    (1, 6, 5, 3, 2)
], 0>
```

We define a sequence of elements of B, containing the simple elements of the above representation of u using the function `CanonicalFactorRepresentation` and the coercion operator `!`.

```
> cfu := [ B!x : x in CFP(u)[3] ];
```

This representation is not in left normal form, as the above condition is violated for $i = 1$.

```
> IsId(LeftGCD(cfu[1]^-1*D, cfu[2]));
false
```

We now bring u into left normal form and extract again the sequence of simple elements.

```
> n := NormalForm(u);
> n;
<Artin, 0, [
    (1, 5, 3, 6, 2),
    (1, 6, 3, 2, 5)
], 0>
> cfn := [ B!x : x in CFP(n)[3] ];
```

This time, the above condition is satisfied.

```
> IsId(LeftGCD(cfn[1]^-1*D, cfn[2]));
true
```

73.4.6 Invariants of Conjugacy Classes

This section describes the functions for computing the set of positive conjugates, the super summit set and the ultra summit set for an element of a braid group B as defined in Section 73.1.5, as well as related MAGMA functions.

All the class invariants in general depend on the presentation of B used for their definition. Many functions documented in this section accept a parameter `Presentation` which can be used to specify the presentation of B with respect to which the computation should be performed. Possible values for this parameter are the strings `"Artin"` and `"BKL"`. If no value is given for `Presentation`, the presentation selected for B is used.

For any given element $u \in B$, all the invariants defined in Section 73.1.5 are finite and can be computed in principle. In practice, however, computations may fail because the sets can get very large with increasing canonical length of u or with increasin braid index of B. This is in particular the case for sets of positive conjuagtes and for super summit sets.

`PositiveConjugates(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return an indexed set containing the conjugates of u which can be represented as positive words in the specified presentation of B.

`SuperSummitRepresentative(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return an element u_s of the super summit set of u with respect to the specified presentation of B and an element c of B satisfying $u^c = u_s$.

Note that u_s is a positive conjugate of u, if u_s has non-negative infimum and that u does not have any positive conjugates if the infimum of u_s is negative.

`SuperSummitSet(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return the super summit set of u with respect to the specified presentation as indexed set of elements of B.

`UltraSummitRepresentative(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return an element u_s of the ultra summit set of u with respect to the specified presentation of B and an element c of B satisfying $u^c = u_s$.

Note that u_s is an element of the super summit set of u, that u_s is a positive conjugate of u, if u_s has non-negative infimum and that u does not have any positive conjugates if the infimum of u_s is negative.

`UltraSummitSet(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return the ultra summit set of u with respect to the specified presentation as indexed set of elements of B.

Example H73E7

(1) In the braid group B on 4 strings we compute the sets of positive conjugates and the super summit sets of $u = \sigma_1\sigma_2\sigma_1$ with respect to both Artin presentation and BKL presentation.

```
> B := BraidGroup(4);
> u := B.1*B.2*B.1;
> p_Artin := PositiveConjugates(u : Presentation := "Artin");
> p_BKL := PositiveConjugates(u : Presentation := "BKL");
> s_Artin := SuperSummitSet(u : Presentation := "Artin");
```

```
> s_BKL := SuperSummitSet(u : Presentation := "BKL");
```

Since the Artin generators form a subset of the BKL generators, every element which is positive with respect to the Artin presentation is also positive with respect to the BKL presentation. In particular, p_Artin is a subset of p_BKL.

```
> p_Artin subset p_BKL;
true
```

The converse inclusion does not hold.

```
> #p_Artin;
10
> #p_BKL;
36
```

For both presentations the super summit set is a subset of the set of positive conjugates, as u is positive. The converse inclusions do not hold.

```
> s_Artin subset p_Artin;
true
> s_BKL subset p_BKL;
true
> #s_Artin;
2
> #s_BKL;
12
```

(2) As we have seen in Section 73.1.5, we can decide whether two braids are conjugate by checking whether their super summit sets are equal.

We illustrate this approach with two elements of B, using the Artin presentation of B.

```
> u := B.2 * B.1 * B.2^2 * B.1 * B.2;
> v := B.2^2 * B.1 * B.3 * B.1 * B.3;
```

Suppose we want to prove that u and v are not conjugate in B. We could start by checking the cycle structure of the induced permutations on the strings on which B acts.

```
> CycleStructure(InducedPermutation(u));
[ <1, 4> ]
> CycleStructure(InducedPermutation(v));
[ <1, 4> ]
```

This does not help. Next we can check the infima and the suprema of super summit representatives.

```
> SuperSummitInfimum(u) eq SuperSummitInfimum(v);
true
> SuperSummitSupremum(u) eq SuperSummitSupremum(v);
true
```

Again, we cannot conclude anything. We decide to compare the super summit sets of u and v.

```
> SuperSummitSet(u) eq SuperSummitSet(v);
```

```
false
```

Success! The super summit sets of u and v are different, proving that u and v are not conjugate. For a more efficient version of conjugacy testing see Example H73E8.

(3) Finally, we illustrate the significant difference in the sizes of super summit sets and ultra summit sets for slightly larger values of braid index and canonical length.

```
> B := BraidGroup(8);
```

We create a pseudo-random element of B as product of 5 simple elements independently chosen at random.

```
> x := B.4 * B.3 * B.2 * B.1 * B.5 * B.4 * B.5 *
> B.6 * B.7 * B.6 * B.5;
> x := x^2;
> Sx := SuperSummitSet(x);
> #Sx;
10972
> Ux := UltraSummitSet(x);
> #Ux;
36
```

The ultra summit set is much smaller than the super summit set. We try again.

```
> x := B.4 * B.3 * B.2 * B.1 * B.5 * B.4 * B.5;
> x := x^3;
> Sx := SuperSummitSet(x);
> #Sx;
882
> Ux := UltraSummitSet(x);
> #Ux;
18
```

The difference in sizes is still large. The behaviour exhibited by these examples is quite typical. In particular, the sizes of super summit sets for braids on a given number of strings and with a given canonical length show much larger fluctuations than the sizes of ultra summit sets. For a more detailed analysis we refer to [Geb03].

73.4.6.1 Computing Class Invariants Interactively

This section describes the functions relevant for interactive computation of the set of positive conjugates, the super summit set and the ultra summit set for a given element of a braid group B as defined in Section 73.1.5.

Process versions of the algorithms used by the functions PositiveConjugates, SuperSummitSet and UltraSummitSet are available for computing these invariants one element at a time.

`PositiveConjugatesProcess(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return a process for constructing the conjugates of u which can be represented as positive words in the specified presentation of B.

The returned process contains the first positive conjugate of u if positive conjugates exist and is *empty* otherwise.

`SuperSummitProcess(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return a process for constructing the super summit elements of u with respect to the specified presentation of B.

The returned process contains the first super summit element of u.

`UltraSummitProcess(u: parameters)`

`Presentation` MonStgElt *Default :*

Given an element u of a braid group B, return a process for constructing the ultra summit elements of u with respect to the specified presentation of B.

The returned process contains the first ultra summit element of u.

`BaseElement(P)`

Return the element used for the construction of the process P.

`#P`

Return the number of elements that have been found by the process P.

`Representative(P)`

`Rep(P)`

Given a non-empty process P, return the element most recently found by P.

If P is empty, a runtime error will occur. The function `IsEmpty` can be used for checking whether a process is empty, in order to avoid runtime errors in loops and user written functions.

`IsEmpty(P)`

Return `true` if P is empty and `false` otherwise.

This function can be used to check whether `Representative` can be called for a process P.

`Elements(P)`

Return an indexed set containing the elements found so far by the process P.

`u in P`

Given an element u of a braid group and a process P for computing positive conjugates or super summit elements of the element b, return `true` and an element c satisfying $b^c = u$ if u is one of the elements that have been constructed by P and `false` otherwise.

`u notin P`

Given an element u of a braid group and a process P, return `false` if u is one of the elements that have been constructed by P and `true` otherwise.

`NextElement(~P)`

Given a process P, continue searching for elements until the next element is found or the search completes without finding a new element.

If a new element if found, it can subsequently be accessed using the function `Representative`. If the search completes without finding a new element, P is marked as *empty*. Calling `NextElement` on an empty process has no effect.

`Complete(~P)`

Given a process P, complete the search for elements. After executing this procedure, P is *empty* and the set of all elements found by P can be accessed using the function `Elements`. Calling `Complete` on an empty process has no effect.

Example H73E8

We sketch how the functions described in the preceeding section could be used for testing whether two elements are conjugate and for computing a conjugating element if they are.

The approach outlined here is basically the algorithm used by the function `IsConjugate`.

```
> function MyIsConjugate(u, v)
>
>  // check obvious invariants
>  infu := SuperSummitInfimum(u);
>  infv := SuperSummitInfimum(v);
>  supu := SuperSummitSupremum(u);
>  supv := SuperSummitSupremum(v);
>  if infu ne infv or supu ne supv then
>     return false, _;
>  end if;
>
>  // compute an ultra summit element for v
>  sv, cv := UltraSummitRepresentative(v);
>
>  // set up a process for computing the ultra summit set of u
>  P := UltraSummitProcess(u);
>
>  // compute ultra summit elements of u until sv is found
>  //    or sv is seen not to be in the ultra summit set of u
```

```
>  while sv notin P and not IsEmpty(P) do
>     NextElement(~P);
>  end while;
>
>  print #P, "elements computed";
>  isconj, c := sv in P;
>  if isconj then
>     // return true and an element conjugating u to v
>     return true, c*cv^-1;
>  else
>     return false, _;
>  end if;
>
> end function;
```

We test our function using two pairs of elements of the braid group B on 4 strings.

```
> B := BraidGroup(4);
```

As we have seen in Example H73E7, the following elements u and v are not conjugate.

```
> u := B.2 * B.1 * B.2^2 * B.1 * B.2;
> v := B.2^2 * B.1 * B.3 * B.1 * B.3;
```

To prove this, our function has to compute the whole ultra summit set of u.

```
> MyIsConjugate(u,v);
2 elements computed
false
> #UltraSummitSet(u);
2
```

We try our function on another pair of elements.

```
> r := B.3*B.2*B.3*B.2^2*B.1*B.3*B.1*B.2;
> s := B.3^-1*B.2^-1*B.3*B.2*B.3*B.2^2*B.1*B.3*B.1*B.2^2*B.3;
> isconj, c := MyIsConjugate(r,s);
3 elements computed
> isconj;
true
> r^c eq s;
true
```

The ultra summit representative of s was the 3rd ultra summit element of r found. Note that the function did not have to compute the whole ultra summit set of r to find the answer.

```
> #UltraSummitSet(r);
6
```

In this small example, we could also have used super summit sets for conjugacy testing, as the super summit set of r is not much larger than its ultra summit set.

```
>  #SuperSummitSet(r);
```

```
22
```

A more challenging application of the function `MyIsConjugate` from above will be presented in Example H73E10.

73.4.6.2 Computing Minimal Simple Elements

This section describes the functions for computing minimal simple elements as introduced in Section 73.1.5.2 and functions for computing the *transport* and the *pullback* as defined in [Geb03].

All functions documented in this section accept two parameters, `Presentation` and `CheckArguments`. The parameter `Presentation` can be used to specify the presentation of a braid group B with respect to which the computation should be performed. Possible values for this parameter are the strings `"Artin"` and `"BKL"`. If no value is given for `Presentation`, the presentation selected for B is used. The parameter `CheckArguments` can be used to turn off argument checking for performance reasons. It should be noted that the results are undefined if functions are called with invalid arguments and argument checking is disabled.

`MinimalElementConjugatingToPositive(x, s: parameters)`

`Presentation`	MonStgElt	*Default :*
`CheckArguments`	BoolElt	*Default :* `true`

Given a positive element x of a braid group B and a simple element s, return the minimal simple element $r_x(s)$ satisfying $s \preceq r_x(s)$ and $x^{r_x(s)} \in B^+$.

`MinimalElementConjugatingToSuperSummit(x, s: parameters)`

`Presentation`	MonStgElt	*Default :*
`CheckArguments`	BoolElt	*Default :* `true`

Given an element x of a braid group B which is contained in its super summit set S_x and a simple element s, return the minimal simple element $\rho_x(s)$ satisfying $s \preceq \rho_x(s)$ and $x^{\rho_x(s)} \in S_x$.

`MinimalElementConjugatingToUltraSummit(x, s: parameters)`

`Presentation`	MonStgElt	*Default :*
`CheckArguments`	BoolElt	*Default :* `true`

Given an element x of a braid group B which is contained in its ultra summit set U_x and a simple element s, return the minimal simple element $c_x(s)$ satisfying $s \preceq c_x(s)$ and $x^{c_x(s)} \in U_x$.

`Transport(x, s: parameters)`

`Presentation`	MonStgElt	*Default :*
`CheckArguments`	BoolElt	*Default :* `true`

Given an element x of a braid group B and a simple element s such that both x and x^s are super summit elements, return the *transport* of s along $x \to \mathbf{c}(x)$, that is, the element $\phi_x(s) = (D \wedge_l xD^{-\inf(x)})^{-1} \cdot s \cdot (D \wedge_l x^s D^{-\inf(x)})$, where D is the fundamental element of B. The transport is a simple element satisfying $\mathbf{c}(x^s) = \mathbf{c}(x)^{\phi_x(s)}$ [Geb03].

`Pullback(x, s: parameters)`

`Presentation`	MonStgElt	*Default :*
`CheckArguments`	BoolElt	*Default :* `true`

Given an element x of a braid group B which is contained in its super summit set S_x and a simple element s, return the *pullback* of s along $x \to \mathbf{c}(x)$, that is, the unique $\preceq$-minimal element $\pi_x(s)$ satisfying $x^{\pi_x(s)} \in S_x$ and $s \preceq \phi_x(\pi_x(s))$ [Geb03].

Example H73E9

The following function uses the technique sketched in Section 73.1.5 for computing the ultra summit set of a given braid. This is basically the algorithm used by the Magma function `UltraSummitSet`.

```
> function MyUltraSummitSet(x)
>
> // create a subset of the ultra summit set of x
> U := {@ UltraSummitRepresentative(x) @};
> gens := Generators(Parent(x));
> pos := 1;
>
> // close U under conjugation with minimal simple elements
> while pos le #U do
>    y := U[pos];
>    // add missing conjugates of y
>    for z in { y^MinimalElementConjugatingToUltraSummit(y, s)
>                   : s in gens } do
>       if z notin U then
>          Include(~U, z);
>       end if;
>    end for;
>    pos +:= 1;
> end while;
>
> return U;
>
> end function;
```

73.5 Homomorphisms

For a general description of homomorphisms, we refer to Chapter 16. This section describes some special aspects of homomorphisms whose domain is in the category GrpBrd.

73.5.1 General Remarks

An important special case of homomorphisms with domain in the category GrpBrd is the following. A homomorphism $f : B \to G$, where B and B' are braid groups on n and m strings, respectively, and f is an embedding of B in G induced by

$$\sigma_i \to \sigma'_{k+\epsilon i} \quad (i = 1, \ldots, n-1)$$

where the σ_i are the Artin generators of B, the σ'_j are the Artin generators of B', $|\epsilon| = 1$ and k is a suitable constant.

The MAGMA implementation uses special optimisation techniques, if a homomorphism with domain in the category GrpBrd has the additional properties listed above. Compared to the general case, this results in faster evaluation of such homomorphisms, in particular in the case $\epsilon = 1$.

Computing preimages under homomorphisms with domain in the category GrpBrd currently is only supported for the special case described above. Moreover, computing the preimage of an element u under a map f may fail, even if u is contained in the image of f.

73.5.2 Constructing Homomorphisms

`hom< B -> G | S : parameters >`

`Check` BOOLELT *Default :* `true`

Returns the homomorphism from the braid group B to the group G defined by the assignment S. S can be the one of the following:

(i) A list, sequence or indexed set containing the images of all k Artin generators $B.1, \ldots, B.k$ of B. Here, the i-th element of S is interpreted as the image of $B.i$, that is, the order of the elements in S is important.

(ii) A list, sequence, enumerated set or indexed set, containing k tuples $< x_i, y_i >$ or arrow pairs $x_i \text{ -> } y_i$, where x_i is a generator of B and $y_i \in G$ $(i = 1, \ldots, k)$ and the set $\{x_1, \ldots, x_k\}$ is the full set of Artin generators of B. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

Note, that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of B.

If the category of the codomain supports element arithmetic and element comparison, by default the constructed homomorphism is checked by verifying that the would-be images of the Artin generators satisfy the braid relations of B. In this case, it is assured that the returned map is a well-defined homomorphism. The most important situation in which it is not possible to perform checking is the case in which the domain is a finitely presented group (`FPGroup`; cf. Chapter 70) which is not free. Checking may be disabled by setting the parameter `Check` to `false`.

73.5.3 Accessing Homomorphisms

`e @ f`
`f(e)`

Given a homomorphism whose domain is a braid group B and an element e of B, return the image of e under f as element of the codomain of f.

`B @ f`
`f(B)`

Given a homomorphism whose domain is a braid group B, return the image of B under f as a subgroup of the codomain of f.

This function is not supported for all codomain categories.

`u @@ f`

Given a homomorphism whose domain is a braid group B and an element u of the image of f, return the preimage of u under f as an element of B.

This function currently is only supported if f is an embedding of one braid group into another as described in Section 73.5.1. Note, moreover, that computing the preimage of u may fail, even if u is contained in the image of f.

`Domain(f)`

The domain of the homomorphism f.

`Codomain(f)`

The codomain of the homomorphism f.

`Image(f)`

The image or range of the homomorphism f as a subgroup of the codomain of f.

This function is not supported for all codomain categories.

Example H73E10

(1) The symmetric group on n letters is an epimorphic image of the braid group on n strings, where for $0 < i < n$ the image of the Artin generator σ_i is given by the transposition $(i, i+1)$. We construct this homomorphism for the case $n = 10$.

```
> Bn := BraidGroup(10);
> Sn := Sym(10);
> f := hom< Bn->Sn | [ Sn!(i,i+1) : i in [1..Ngens(Bn)] ] >;
```

Of course, the image of `f` is the full symmetric group.

```
> Image(f) eq Sn;
true
```

Now we compute the image of a pseudo-random element of `Bn` under `f`.

```
> f(Random(Bn));
```

```
(1, 5, 8)(2, 4, 9, 7, 6, 3)
```

(2) (Key exchange as proposed in [KLC+00])

Consider a collection of $l+r$ strings $t_1, \ldots, t_{l+r}$ and the braid group B acting on $t_1, \ldots, t_{l+r}$ with Artin generators $\sigma_1, \ldots, \sigma_{l+r-1}$. The subgroups of B fixing the strings $t_{l+1}, \ldots, t_{l+r}$ and $t_1, \ldots, t_l$ may be identified with braid groups L on l strings and R on r strings, respectively, with the Artin generators of L and R corresponding to $\sigma_1, \ldots, \sigma_{l-1}$ and $\sigma_{l+1}, \ldots, \sigma_{l+r-1}$, respectively.

We set up these groups for $l = 6$ and $r = 7$ using the BKL presentations.

```
> l := 6;
> r := 7;
> B := BraidGroup(l+r : Presentation := "BKL");
> L := BraidGroup(l   : Presentation := "BKL");
> R := BraidGroup(r   : Presentation := "BKL");
```

We now construct the embeddings $f : L \to B$ and $g : R \to B$.

```
> f := hom< L-> B | [ L.i -> B.i : i in [1..Ngens(L)] ] >;
> g := hom< R-> B | [ R.i -> B.(l+i) : i in [1..Ngens(R)] ] >;
```

To complete the preparatory steps, we choose a random element of B which is not too short.

```
> x := Random(B, 15, 25);
```

The data constructed so far is assumed to be publicly available. Each time two users A and B require a shared key, the following steps are performed.

(a) A chooses a random secret element $a \in L$ and sends the normal form of $y_1 := x^a$ to B.
(b) B chooses a random secret element $b \in R$ and sends the normal form of $y_2 := x^b$ to A.
(c) A receives y_2 and computes the normal form of y_2^a.
(d) B receives y_1 and computes the normal form of y_1^b.

Note the following.

• Transmitting y_1 and y_2 in normal form disguises their structure as products $a^{-1}xa$ and $b^{-1}xb$, provided the words used are long enough and prevents simply reading off the conjugating elements a and b.
• Since the subgroups L and R of B commute, we have $ab = ba$, which implies $y_2^a = x^{ba} = x^{ab} = y_1^b$. Thus, the normal forms computed by A and B in steps (c) and (d), respectively, must be identical and can be used to extract a secret shared key.

We illustrate this, using the groups set up above. For step (a):

```
> a := Random(L, 15, 25);
> y1 := NormalForm(x^f(a));
```

Now for step (b):

```
> b := Random(R, 15, 25);
> y2 := NormalForm(x^g(b));
```

We now verify that A and B arrive at the same information in steps (c) and (d).

```
> K_A := NormalForm(y2^f(a));
```

```
> K_B := NormalForm(y1^g(b));
> AreIdentical(K_A, K_B);
true
```

We see that the information computed by A and B in steps (c) and (d) is indeed identical and hence can be used (in suitable form) as a common secret. Note, however, that the number of strings and the lengths of the elements used in the example above are much smaller than the values suggested for real cryptographic purposes.

(3) (Attack on key exchange)

We now show an attack on the key exchange outlined above using conjugacy search based on ultra summit sets.

An eavesdropper can try to compute an element c conjugating x to y_1. While this is not guaranteed to reproduce the braid a, the chances for a successful key recovery are quite good.

We decide to change to the Artin presentation of B and use the function `MyIsConjugate` from Example H73E8 for computing a conjugating element c as above.

```
> SetPresentation(~B, "Artin");
> time _, c := MyIsConjugate(x, y1);
42 elements computed
Time: 0.020
```

Finding a conjugating element is no problem at all. Using the conjugating element, we can try to recover the shared secret. In this example we are lucky.

```
> NormalForm(y2^c) eq K_A;
true
```

In the conjugacy search above, a conjugating element was found after computing 42 ultra summit elements. The ultra summit set itself is larger, but can still be computed very easily.

```
> time Ux := UltraSummitSet(x);
Time: 3.150
> #Ux;
1584
```

The super summit set is, even in this small example, too large to be computed; conjugacy search based on super summit sets would quite likely fail.

```
>  time Sx := SuperSummitSet(x);
Current total memory usage: 4055.1MB
System error: Out of memory.
```

Finally, we show that the attack using conjugacy search based on ultra summit sets is also applicable to larger examples. We try to recover a key, which is generated using elements with canonical lengths between 500 and 1000 in a braid group on 100 strings.

```
> l := 50;
> r := 50;
> B := BraidGroup(l+r);
> L := BraidGroup(l);
```

```
> R := BraidGroup(r);
>
> f := hom< L-> B | [ L.i -> B.i : i in [1..Ngens(L)] ] >;
> g := hom< R-> B | [ R.i -> B.(l+i) : i in [1..Ngens(R)] ] >;
>
> x := Random(B, 0, 1, 500, 1000);
>
> a := Random(L, 0, 1, 500, 1000);
> y1 := NormalForm(x^f(a));
>
> b := Random(R, 0, 1, 500, 1000);
> y2 := NormalForm(x^g(b));
>
> K_A := NormalForm(y2^f(a));
> K_B := NormalForm(y1^g(b));
> AreIdentical(K_A, K_B);
true
```

We again try to recover the key by computing an element conjugating x to y_1. This time, we use the built-in MAGMA function for efficiency reasons.

```
> time _, c := IsConjugate(x, y1);
Time: 18.350
> K_A eq NormalForm(y2^c);
false
```

Bad luck. – We managed to compute a conjugating element, but this failed to recover the key. We try with an element conjugating x to y_2.

```
> time _, c := IsConjugate(x, y2);
Time: 3.800
> K_B eq NormalForm(y1^c);
true
```

Success! – Good that we didn't use this key to encrypt our credit card number!

73.5.4 Representations of Braid Groups

This section describes the functions available for creating a number of well known representations of braid groups.

`SymmetricRepresentation(B)`

Given a braid group B on n strings, return the natural epimorphism from B onto the symmetric group on n points, induced by the action of B on the strings by which B is defined.

`BurauRepresentation(B)`

Given a braid group B on n strings, return the Burau representation of B as homomorphism from B to the matrix algebra of degree n over the rational function field over the integers.

`BurauRepresentation(B, p)`

Given a braid group B on n strings and a prime p, return the p-modular Burau representation of B as homomorphism from B to the matrix algebra of degree n over the rational function field over the field with p elements.

Example H73E11

We construct the Burau representation of the braid group on 4 strings.

```
> B := BraidGroup(4);
> f := BurauRepresentation(B);
```

Its codomain is a matrix algebra of degree 4 over the rational function field over the integers.

```
> A := Codomain(f);
> A;
GL(4, FunctionField(IntegerRing()))
> F := BaseRing(A);
> F;
Univariate rational function field over Integer Ring
Variables: $.1
```

To obtain nicer printing, we assign the name t to the generator of the function field F.

```
> AssignNames(~F, ["t"]);
```

Now we can check easily whether we remembered the definition of the Burau representation correctly.

```
> f(B.1);
[-t + 1      t      0      0]
[     1      0      0      0]
[     0      0      1      0]
[     0      0      0      1]

> f(B.2);
[     1      0      0      0]
[     0 -t + 1      t      0]
[     0      1      0      0]
[     0      0      0      1]

> f(B.3);
[     1      0      0      0]
[     0      1      0      0]
[     0      0 -t + 1      t]
[     0      0      1      0]
```

73.6 Bibliography

[**AAG99**] Iris Anshel, Michael Anshel, and Dorian Goldfeld. An algebraic method for public-key cryptography. *Math. Res. Lett.*, 6(3-4):287–291, 1999.

[**Art47**] E. Artin. Theory of braids. *Ann. of Math. (2)*, 48:101–126, 1947.

[**BKL98**] Joan Birman, Ki Hyoung Ko, and Sang Jin Lee. A new approach to the word and conjugacy problems in the braid groups. *Adv. Math.*, 139(2):322–353, 1998.

[**Deh02**] Patrick Dehornoy. Groupes de Garside. *Ann. Sci. École Norm. Sup. (4)*, 35(2):267–306, 2002.

[**ECH+92**] David B. A. Epstein, James W. Cannon, Derek F. Holt, Silvio V. F. Levy, Michael S. Paterson, and William P. Thurston. *Word processing in groups*. Jones and Bartlett Publishers, Boston, MA, 1992.

[**ERM94**] Elsayed A. El-Rifai and H. R. Morton. Algorithms for positive braids. *Quart. J. Math. Oxford Ser. (2)*, 45(180):479–497, 1994.

[**FGM03**] Nuno Franco and Juan González-Meneses. Conjugacy problem for braid groups and Garside groups. *J. Algebra*, 266(1):112–132, 2003.

[**Gar69**] F. A. Garside. The braid group and other groups. *Quart. J. Math. Oxford Ser. (2)*, 20:235–254, 1969.

[**Geb03**] Volker Gebhardt. A new approach to the conjugacy problem in Garside groups. Preprint; available at URL:http://www.arxiv.org/math.GT/0306199, 2003.

[**GKT+02**] D. Garber, S. Kaplan, M. Teicher, B. Tsaban, and U. Vishne. Length-based conjugacy search in the braid group. Preprint; available at URL:http://www.arxiv.org/math.GR/0209267, 2002.

[**GM02**] Juan González-Meneses. Improving an algorithm to solve Multiple Simultaneous Conjugacy Problems in braid groups. Preprint; available at URL:http://www.arxiv.org/math.GT/0212150, 2002.

[**HS03**] D. Hofheinz and R. Steinwandt. A practical attack on some braid group based cryptographic primitives. In *Public Key Cryptography, 6th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2003*, volume 2567 of *LNCS*, pages 187–198. Springer, 2003.

[**Hug02**] J. Hughes. A linear algebraic attack on the AAFG1 braid group cryptosystem. In *The 7th Australasian Conference on Information Security and Privacy, ACISP 2002*, volume 2384 of *LNCS*, pages 176–189. Springer, 2002.

[**KLC+00**] Ki Hyoung Ko, Sang Jin Lee, Jung Hee Cheon, Jae Woo Han, Ju-sung Kang, and Choonsik Park. New public-key cryptosystem using braid groups. In *Advances in cryptology—CRYPTO 2000*, pages 166–183. Springer, Berlin, 2000.

[**LL02**] Sang Jin Lee and Eonkyung Lee. Potential Weakness of the Commutator Key Agreement Protocol Based on Braid Groups. In *Advances in Cryptology—EuroCrypt 2002*, volume 2332 of *LNCS*, pages 14–28. Springer, 2002.

[**LP03**] E. Lee and J. H. Park. Cryptanalysis of the public key encryption based on braid groups. In *Advances in Cryptology—EuroCrypt 2003*, volume 2656 of *LNCS*, pages 477–490. Springer, 2003.

74 GROUPS DEFINED BY REWRITE SYSTEMS

Chapter 74
GROUPS DEFINED BY REWRITE SYSTEMS

74.1 Introduction

The class of finitely presented groups defined by finite rewrite systems provide a Magma level interface to Derek Holt's KBMAG programs, and specifically to the Knuth–Bendix completion procedure for groups defined by a finite (monoid) presentation. Much of the material in this chapter is taken from the KBMAG documentation [Hol97]. Familiarity with the Knuth–Bendix completion procedure is assumed. Some familiarity with KBMAG would be beneficial.

74.1.1 Terminology

A rewrite group G is a finitely presented group in which equality between elements of G, called *words* or *strings*, may be decidable via a sequence of rewriting equations, called *reduction relations, rules, or equations*. In the interests of efficiency the reduction rules are codified into a finite state automaton called a *reduction machine*. The words in a rewrite group G are ordered, as are the reduction relations of G. Several possible orderings of words are supported, namely short-lex, recursive, weighted short-lex and wreath-product orderings. A rewrite group can be *confluent* or *non-confluent*. If a rewrite group G is confluent its reduction relations, or more specifically its reduction machine, can be used to reduce words in G to their irreducible normal forms under the given ordering, and so the word problem for G can be efficiently solved.

74.1.2 The Category of Rewrite Groups

The family of all rewrite groups forms a category. The objects are the rewrite groups and the morphisms are group homomorphisms. The MAGMA designation for this category of groups is `GrpRWS`. Elements of a rewrite group are designated as `GrpRWSElt`.

74.1.3 The Construction of a Rewrite Group

A rewrite group G is constructed in a three-step process:

(i) We construct a free group FG.

(ii) We construct a quotient F of FG.

(iii) We create a monoid presentation of F and then run a Knuth–Bendix completion procedure on this presentation to create a rewrite group G.

The Knuth–Bendix procedure may or may not succeed. If it fails the user may need to perform the above steps several times, manually adjusting the parameters to the Knuth–Bendix procedure. If it succeeds then the rewrite system constructed will be confluent.

74.2 Constructing Confluent Presentations

74.2.1 The Knuth-Bendix Procedure

`RWSGroup(F: parameters)`

Suppose F is a finitely presented group. Internally, the first step is to construct a presentation for a monoid M. By default, the generators of M are taken to be $g_1, g_1^{-1}, \ldots, g_n, g_n^{-1}$, where $g_1, \ldots, g_n$ are the generators of F. The relations for M are taken to be the relations of F together with the trivial relations $g_1 g_1^{-1} = g_1^{-1} g_1 = 1$. The Knuth–Bendix completion procedure for monoids is now applied to M. Regardless of whether or not the completion procedure succeeds, the result will be a rewrite monoid, M, containing a reduction machine and a sequence of reduction relations. If the procedure succeeds M will be marked as confluent, and the word problem for M is therefore decidable. If, as is very likely, the procedure fails then M will be marked as non-confluent. In this case M will contain both the reduction relations and the reduction machine computed up to the point of failure. Reductions made using these relations will be correct in F, but words that are equal in F are not guaranteed to reduce to the same word.

The Knuth–Bendix procedure requires ordering to be defined on both the generators and the words. The default generator ordering is that induced by the ordering of the generators of F while the default ordering on strings is the `ShortLex` order. We give a simple example and then discuss the parameters that allow the user to specify these two orderings.

As the Knuth–Bendix procedure will more often than not run forever, some conditions must be specified under which it will stop. These take the form of limits that are placed on certain variables, such as the number of reduction relations. If any of these limits are exceeded during a run of the completion procedure it will fail, returning a non-confluent rewrite monoid. The optimal values for these limits vary from example to example. The various parameters that allow the user to specify the limits for these variables will be described in a subsequent section.

Example H74E1

We construct the Von Dyck $(2, 3, 5)$ group. Since a string ordering is not specified the default `ShortLex` ordering is used. Similarly, since a generator ordering is not specified, the default generator ordering, in this case $[a, a^{-1}, b, b^{-1}]$, is used.

```
> FG<a,b> := FreeGroup(2);
> F := quo< FG | a*a=1, b*b=b^-1, a*b^-1*a*b^-1*a=b*a*b*a*b>;
> G := RWSGroup(F);
> G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
Ordering = ShortLex.
The reduction machine has 39 states.
The rewrite relations are:
```

```
a^2 = Id(F)
b * b^-1 = Id(F)
b^-1 * b = Id(F)
b^2 = b^-1
b * a * b * a * b = a * b^-1 * a * b^-1 * a
b^-2 = b
b^-1 * a * b^-1 * a * b^-1 = a * b * a * b * a
a^-1 = a
a * b^-1 * a * b^-1 * a * b = b * a * b * a * b^-1
b * a * b^-1 * a * b^-1 * a = b^-1 * a * b * a * b
a * b * a * b * a * b^-1 = b^-1 * a * b^-1 * a * b
b^-1 * a * b * a * b * a = b * a * b^-1 * a * b^-1
b * a * b * a * b^-1 * a * b * a = a * b^-1 * a * b * a * b^-1 * a * b^-1
b^-1 * a * b^-1 * a * b * a * b^-1 * a = a * b * a * b^-1 * a * b * a * b
b^-1 * a * b * a * b^-1 * a * b * a * b^-1 = b * a * b^-1 * a * b * a * b^-1
    * a * b
b * a * b^-1 * a * b * a * b^-1 * a * b^-1 = (b^-1 * a * b * a)^2
b^-1 * a * b * a * b^-1 * a * b * a * b = (b * a * b^-1 * a)^2
b * a * b^-1 * a * b * a * b^-1 * a * b * a = a * b * a * b^-1 * a * b * a *
    b^-1 * a * b
```

74.2.2 Defining Orderings

`RWSGroup(F:` *parameters*`)`

Attempt to construct a confluent presentation for the finitely presented group F using the Knuth-Bendix completion algorithm. In this section we describe how the user can specify the generator order and the ordering on strings.

`GeneratorOrder`	SeqEnum	*Default :*

Give an ordering for the generators. This ordering affects the ordering of words in the alphabet. If not specified the ordering defaults to the order induced by F's generators, that is $[g_1, \ldots, g_n]$ where $g_1, \ldots, g_n$ are the generators of F.

`Ordering`	MonStgElt	*Default :* "*ShortLex*"
`Levels`	SeqEnum	*Default :*
`Weights`	SeqEnum	*Default :*

`Ordering := "ShortLex"`: Use the short-lex ordering on strings. Shorter words come before longer, and for words of equal length lexicographical ordering is used, using the given ordering of the generators.
`Ordering := "Recursive" | "RtRecursive"`: Use a recursive ordering on strings. There are various ways to define this. Perhaps the quickest is as follows. Let u and v be strings in the generators. If one of u and v, say v, is empty, then $u \geq v$. Otherwise, let $u = u'a$ and $v = v'b$, where a and b are generators. Then $u > v$ if and only if one of the following holds:

(i) $a = b$ and $u' > v'$;

(ii) $a > b$ and $u > v'$;

(iii) $b > a$ and $u' > v$.

The `RtRecursive` ordering is similar to the `Recursive` ordering, but with $u = au'$ and $v = bv'$. Occasionally one or the other runs significantly quicker, but usually they perform similarly.

`Ordering := "WtLex"`: Use a weighted-lex ordering. `Weights` should be a sequence of non-negative integers, with the i-th element of `Weights` giving the weight of the i-the generator. The length of `Weights` must equal the number of generators. The length of words in the generators is then computed by adding up the weights of the generators in the words. Otherwise, ordering is as for short-lex.

`Ordering := "Wreath"`: Use a wreath-product ordering. `Levels` should be a sequence of non-negative integers, with the i-th element of `Levels` giving the level of the i-the generator. The length of `Levels` must equal the number of generators. In this ordering, two strings involving generators of the same level are ordered using short-lex, but all strings in generators of a higher level are larger than those involving generators of a lower level. That is not a complete definition; one can be found in [Sim94, pp. 46–50]. Note that the recursive ordering is the special case in which the level of generator number i is i.

Example H74E2

A confluent presentation is constructed for an infinite non-hopfian group using the `Recursive` ordering.

```
> F<a, b> := Group< a, b | b^-1*a^2*b=a^3>;
> G := RWSGroup(F : Ordering :="Recursive");
> G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
Ordering = Recursive.
The reduction machine has 7 states.
The rewrite relations are:
    a * a^-1 = Id(FG)
    a^-1 * a = Id(FG)
    b * b^-1 = Id(FG)
    b^-1 * b = Id(FG)
    a^3 * b^-1 = b^-1 * a^2
    a^2 * b = b * a^3
    a^-1 * b^-1 = a^2 * b^-1 * a^-2
    a^-1 * b = a * b * a^-3
```

Example H74E3

A confluent presentation of a free nilpotent group of rank 2 and class 2 is constructed by the following code. Note that the lower weight generators (in the sense of nilpotency class) need to come first in the ordering of generators.

```
> FG<a,b,c> := FreeGroup(3);
> F := quo< FG | b*a=a*b*c, c*a=a*c, c*b=b*c>;
> G := RWSGroup(F : Ordering :="Recursive",
>                 GeneratorOrder := [c,c^-1,b,b^-1,a,a^-1]);
> G;
A confluent rewrite group.
Generator Ordering = [ c, c^-1, b, b^-1, a, a^-1 ]
Ordering = Recursive.
The reduction machine has 7 states.
The rewrite relations are:
    c * c^-1 = Id(FG)
    c^-1 * c = Id(FG)
    b * b^-1 = Id(FG)
    b^-1 * b = Id(FG)
    a * a^-1 = Id(FG)
    a^-1 * a = Id(FG)
    b * a = a * b * c
    c * a = a * c
    c * b = b * c
    c^-1 * a = a * c^-1
    c * a^-1 = a^-1 * c
    c^-1 * b = b * c^-1
    c * b^-1 = b^-1 * c
    b^-1 * a = a * b^-1 * c^-1
    b^-1 * a^-1 = a^-1 * b^-1 * c
    c^-1 * a^-1 = a^-1 * c^-1
    c^-1 * b^-1 = b^-1 * c^-1
    b * a^-1 = a^-1 * b * c^-1
```

74.2.3 Setting Limits

In this section we introduce the various parameters used to control the execution of the Knuth-Bendix procedure.

`RWSMonoid(F: parameters)`

Attempt to construct a confluent presentation for the finitely presented group F using the Knuth-Bendix completion algorithm. We present details of the various parameters used to control the execution of the Knuth-Bendix.

`MaxRelations` RngIntElt *Default :* 32767

Limit the maximum number of reduction equations to `MaxRelations`.

`TidyInt` RngIntElt *Default :* 100

After finding `TidyInt` new reduction equations, the completion procedure interrupts the main process of looking for overlaps, to tidy up the existing set of equations. This will eliminate any redundant equations performing some reductions on their left and right hand sides to make the set as compact as possible. (The point is that equations discovered later often make older equations redundant or too long.)

`RabinKarp` Tup *Default :*

Use the Rabin-Karp algorithm for word-reduction on words having length at least l, provided that there are at least n equations, where `RabinKarp := <l, n>`. This uses less space than the default reduction automaton, but it is distinctly slower, so it should only be used when seriously short of memory. Indeed this option is only really useful for examples in which collapse occurs - i.e. at some intermediate stage of the calculation there is a very large set of equations, which later reduces to a much smaller confluent set. Collapse is not uncommon when analysing pathological presentations of finite groups, and this is one situation where the performance of the Knuth–Bendix algorithm can be superior to that of Todd-Coxeter coset enumeration. The best setting for `RabinKarp` varies from example to example - generally speaking, the smaller l is, the slower things will be, so set it as high as possible subject to not running out of memory. The number of equations n should be set to a value greater than the expected final number of equations.

`MaxStates` RngIntElt *Default :*

Limit the maximum number of states of the finite state automaton used for word reduction to `MaxStates`. By default there is no limit, and the space allocated is increased dynamically as required. The space needed for the reduction automaton can also be restricted by using the `RabinKarp` parameter. This limit is not usually needed.

`MaxReduceLen` RngIntElt *Default :* 32767

Limit the maximum allowed length that a word can reach during reduction to `MaxReduceLen`. It is only likely to be exceeded when using the recursive ordering on words. This limit is usually not needed.

`ConfNum` RngIntElt *Default :* 500

If `ConfNum` overlaps are processed and no new equations are discovered, then the overlap searching process is interrupted, and a fast check for confluence performed on the existing set of equations. Doing this too often wastes time, but doing it at the

right moment can also save a lot of time. If `ConfNum` = 0, then the fast confluence check is performed only when the search for overlaps is complete.

Warning: Changing the default setting on any of the following parameters may either cause the procedure to terminate without having found a confluent presentation or may change the underlying group.

`MaxStoredLen` TUP *Default :*

Only equations in which the left and right hand sides have lengths at most l and r, respectively, where `MaxStoredLen := <l, r>` are kept. Of course this may cause the overlap search to complete on a set of equations that is not confluent. In some examples, particularly those involving collapse (i.e. a large intermediate set of equations, which later simplifies to a small set), it can result in a confluent set being found much more quickly. It is most often useful when using a recursive ordering on words. Another danger with this option is that sometimes discarding equations can result in information being lost, and the monoid defined by the equations changes.

`MaxOverlapLen` RNGINTELT *Default :*

Only overlaps of total length at most `MaxOverlapLen` are processed. Of course this may cause the overlap search to complete on a set of equations that is not confluent.

`Sort` BOOLELT *Default :* `false`

`MaxOpLen` RNGINTELT *Default :* 0

If `Sort` is set to `true` then the equations will be sorted in order of increasing length of their left hand sides, rather than the default, which is to leave them in the order in which they were found. `MaxOpLen` should be a non-negative integer. If `MaxOpLen` is positive, then only equations with left hand sides having length at most `MaxOpLen` are output. If `MaxOpLen` is zero, then all equations are sorted by length. Of course, if `MaxOpLen` is positive, there is a danger that the monoid defined by the output equations may be different from the original.

```
SetVerbose("KBMAG", v)
```

Set the verbose printing level for the Knuth-Bendix completion algorithm. Setting this level allows a user to control how much extra information on the progress of the algorithm is printed. Currently the legal values for v are 0 to 3 inclusive. Setting v to 0 corresponds to the '-silent' option of KBMAG in which no extra output is printed. Setting v to 2 corresponds to the '-v' (verbose) option of KBMAG in which a small amount of extra output is printed. Setting v to 3 corresponds to the '-vv' (very verbose) option of KBMAG in which a huge amount of diagnostic information is printed.

74.2.4 Accessing Group Information

The functions in this group provide access to basic information stored for a rewrite group G.

`G . i`

The i-th defining generator for G. The integer i must lie in the range $[-r, r]$, where r is the number of group G.

`Generators(G)`

A sequence containing the defining generators for G.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for G.

`Relations(G)`

A sequence containing the defining relations for G. The relations will be given between elements of the free group of which G is a quotient. In these relations the (image of the) left hand side (in G) will always be greater than the (image of the) right hand side (in G) in the ordering on words used to construct G.

`NumberOfRelations(G)`

`Nrels(G)`

The number of relations in G.

`Ordering(G)`

The ordering of G.

Example H74E4

We illustrate the access operations using the following presentation of $\mathbf{Z} \wr C_2$.

```
> FG<a,b,t> := FreeGroup(3);
> F := quo< FG | t^2=1, b*a=a*b, t*a*t=b>;
> G<x,y,z> := RWSGroup(F);
> G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1, t, t^-1 ]
Ordering = ShortLex.
The reduction machine has 6 states.
The rewrite relations are:
    a * a^-1 = Id(F)
    a^-1 * a = Id(F)
    b * b^-1 = Id(F)
    b^-1 * b = Id(F)
    t^2 = Id(F)
    b * a = a * b
    t * a = b * t
    b^-1 * a = a * b^-1
    t * b = a * t
```

```
    b * a^-1 = a^-1 * b
    t * a^-1 = b^-1 * t
    t^-1 = t
    b^-1 * a^-1 = a^-1 * b^-1
    t * b^-1 = a^-1 * t
> G.1;
x
> G.1*G.2;
x * y
> Generators(G);
[ x, y, z ]
> Ngens(G);
3
> Relations(G);
[ a * a^-1 = Id(F), a^-1 * a = Id(F), b * b^-1 = Id(F), b^-1 * b = Id(F), t^2 =
Id(F), b * a = a * b, t * a = b * t, b^-1 * a = a * b^-1, t * b = a * t, b *
a^-1 = a^-1 * b, t * a^-1 = b^-1 * t, t^-1 = t, b^-1 * a^-1 = a^-1 * b^-1, t *
b^-1 = a^-1 * t ]
> Nrels(G);
14
> Ordering(G);
ShortLex
```

74.3 Properties of a Rewrite Group

`IsConfluent(G)`

Returns `true` if G is confluent, `false` otherwise.

`IsFinite(G)`

Given a confluent group G return `true` if G has finite order and `false` otherwise. If G does have finite order also return the order of G.

`Order(G)`

`#G`

The order of the group G as an integer. If the order of G is known to be infinite, the symbol ∞ is returned.

Example H74E5

We construct the Weyl group E_8.

```
> FG<a,b,c,d,e,f,g,h> := FreeGroup(8);
> Q := quo< FG | a^2=1, b^2=1, c^2=1, d^2=1, e^2=1, f^2=1, g^2=1,
>     h^2=1, b*a*b=a*b*a, c*a=a*c, d*a=a*d, e*a=a*e, f*a=a*f,
>     g*a=a*g, h*a=a*h, c*b*c=b*c*b, d*b=b*d, e*b=b*e, f*b=b*f,
>     g*b=b*g, h*b=b*h, d*c*d=c*d*c, e*c*e=c*e*c, f*c=c*f,
>     g*c=c*g, h*c=c*h, e*d=d*e, f*d=d*f, g*d=d*g, h*d=d*h,
>     f*e*f=e*f*e, g*e=e*g, h*e=e*h, g*f*g=f*g*f, h*f=f*h,
>     h*g*h=g*h*g>;
> G := RWSGroup(Q);
> IsConfluent(G);
true
> IsFinite(G);
true 696729600
```

So the group is finite of order $696,729,600$.

Example H74E6

We construct a 2-generator 2-relator group and use the order function to show that the group is infinite. The symbol `Infinity`, returned by `Order`, indicates that the group has infinite order.

```
> G := Group< x, y | x^2 = y^2, x*y*x = y*x*y >;
> R := RWSGroup(G);
> print Order(G);
Infinity
```

74.4 Arithmetic with Words

74.4.1 Construction of a Word

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity word in G.

`G ! [ i`$_1$`, ..., i`$_s$` ]`

Given a rewrite group G defined on r generators and a sequence $[i_1, \cdots, i_s]$ of integers lying in the range $[-r, r]$, excluding 0, construct the word

$$G.|i_1|^{\epsilon_1} * G.|i_2|^{\epsilon_2} * \cdots * G.|i_s|^{\epsilon_s}$$

where ϵ_j is $+1$ if i_j is positive, and -1 if i_j is negative. The resulting word is reduced using the reduction machine associated with G.

`Parent(w)`

The parent group G for the word w.

Example H74E7

We construct the Fibonacci group $F(2,7)$, and its identity.

```
> FG<a,b,c,d,e,f,g> := FreeGroup(7);
> F := quo< FG | a*b=c, b*c=d, c*d=e, d*e=f, e*f=g, f*g=a, g*a=b>;
> G := RWSGroup(F : TidyInt := 1000);
> Id(G);
Id(G)
> G!1;
Id(G)
> G![1,2];
G.3
```

74.4.2 Element Operations

Having constructed a rewrite group G one can perform arithmetic with words in G. Assuming we have $u, v \in G$ then the product $u * v$ will be computed as follows:

(i) the product $w = u * v$ is formed as a product in the appropriate free group.

(ii) w is reduced using the reduction machine associated with G.

If G is confluent, then w will be the unique minimal word that represents $u * v$ under the ordering of G. If G is not confluent, then there are some pairs of words which are equal in G, but which reduce to distinct words, and hence w will not be a unique normal form. Note that:

(i) reduction of w can cause an increase in the length of w. At present there is an internal limit on the length of a word – if this limit is exceeded during reduction an error will be raised. Hence any word operation involving reduction can fail.

(ii) the implementation is designed more with speed of execution in mind than with minimizing space requirements; thus, the reduction machine is always used to carry out word reduction, which can be space-consuming, particularly when the number of generators is large.

`u * v`

Given words w and v belonging to a common group, return their product.

`u / v`

Given words w and v belonging to a common group, return the product of the word u by the inverse of the word v, i.e. the word $u * v^{-1}$.

`u ^ n`

The n-th power of the word w.

`u ^ v`

Given words w and v belonging to a common group, return the conjugate of the word u by the word v, i.e. the word $v^{-1} * u * v$.

`Inverse(w)`

The inverse of the word w.

`(u, v)`

Given words w and v belonging to a common group, return the commutator of the words u and v, i.e., the word $u^{-1}v^{-1}uv$.

`(u`$_1$`, ..., u`$_r$`)`

Given r words $u_1, \ldots, u_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

`u eq v`

Given words w and v belonging to the same group, return `true` if w and v reduce to the same normal form, `false` otherwise. If G is confluent this tests for equality. If G is non-confluent then two words which are the same may not reduce to the same normal form.

`u ne v`

Given words w and v belonging to the same group, return `false` if w and v reduce to the same normal form, `true` otherwise. If G is confluent this tests for non-equality. If G is non-confluent then two words which are the same may reduce to different normal forms.

`IsId(w)`
`IsIdentity(w)`

Returns `true` if the word w is the identity word.

`#u`

The length of the word w.

`ElementToSequence(u)`
`Eltseq(u)`

The sequence Q obtained by decomposing the element u of a rewrite group into its constituent generators and generator inverses. Suppose u is a word in the rewrite group G. Then, if $u = G.i_1^{e_1} \cdots G.i_m^{e_m}$, with each $e_i = \pm 1$, then $Q[j] = i_j$ if $e_j = +1$ and $Q[j] = -i_j$ if $e_j = -1$, for $j = 1, \ldots, m$.

Example H74E8

We illustrate the word operations by applying them to elements of the Fibonacci group $F(2,5)$.

```
> FG<a,b,c,d,e> := FreeGroup(5);
> F := quo< FG | a*b=c, b*c=d, c*d=e, d*e=a, e*a=b>;
> G<a,b,c,d,e> := RWSGroup(F);
> a*b^-1;
e^-1
> a/b;
e^-1
> (c*d)^4;
a
> a^b, b^-1*a*b;
a a
> a^-2,
> Inverse(a)^2;
d d
> c^-1*d^-1*c*d eq (c,d);
true
> IsIdentity(a*b*c^-1);
true
> #(c*d);
1
```

74.5 Operations on the Set of Group Elements

`Random(G, n)`

A random word of length at most n in the generators of G.

`Random(G)`

A random word (of length at most the order of G) in the generators of G.

`Representative(G)`

`Rep(G)`

An element chosen from G.

`Set(G, a, b)`

`Search` MonStgElt *Default :* "*DFS*"

Create the set of reduced words, w, in G with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Set(G)`

`Search` MonStgElt *Default :* "*DFS*"

Create the set of reduced words that is the carrier set of G. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Seq(G, a, b)`

`Search` MonStgElt *Default :* "*DFS*"

Create the sequence S of reduced words, w, in G with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

`Seq(G)`

`Search` MonStgElt *Default :* "*DFS*"

Create a sequence S of reduced words in the carrier set of G. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

Example H74E9

We construct the group D_{22}, together with a representative word from the group, a random word and a random word of length at most 5 from the group, and the set of elements of the group.

```
> FG<a,b,c,d,e,f> := FreeGroup(6);
> Q := quo< FG | a*c^-1*a^-1*d=1, b*f*b^-1*e^-1=1, c*e*c^-1*d^-1=1,
>               d*f^-1*d^-1*a=1, e*b*e^-1*a^-1=1, f*c^-1*f^-1*b^-1=1>;
```

```
> G<a,b,c,d,e,f> := RWSGroup(Q);
> Representative(G);
Id(G)
> Random(G);
b
> Random(G, 5);
a * d * b
> Set(G);
{ a * d * b, a * b, a * b * e, a * c, a * d, d * b, b * e, a * b * a,
  a * b * d, b * a, a * c * e, Id(G), b * d, c * e, e, f, a, a * e, b,
  c, a * f, d }
> Seq(G : Search := "DFS");
[ Id(G), a, a * b, a * b * a, a * b * d, a * b * e, a * c, a * c * e,
  a * d, a * d * b, a * e, a * f, b, b * a, b * d, b * e, c, c * e, d,
  d * b, e, f ]
```

74.6 Homomorphisms

For a general description of homomorphisms, we refer to chapter 16. This section describes some special aspects of homomorphisms whose domain or codomain is a rewrite group.

74.6.1 General Remarks

Groups in the category `GrpRWS` currently are accepted as codomains only in some special situations. The most important cases in which a rewrite group can be used as a codomain are group homomorphisms whose domain is in one of the categories `GrpFP`, `GrpGPC`, `GrpRWS` or `GrpAtc`.

74.6.2 Construction of Homomorphisms

`hom< R -> G | S >`

Returns the homomorphism from the rewrite group R to the group G defined by the expression S which can be the one of the following:

(i) A list, sequence or indexed set containing the images of the n generators $R.1, \ldots, R.n$ of R. Here, the i-th element of S is interpreted as the image of $R.i$, i.e. the order of the elements in S is important.

(ii) A list, sequence, enumerated set or indexed set, containing n tuples $< x_i, y_i >$ or arrow pairs $x_i \to y_i$, where x_i is a generator of R and $y_i \in G$ $(i = 1, \ldots, n)$ and the set $\{x_1, \ldots, x_n\}$ is the full set of generators of R. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

It is the user's responsibility to ensure that the provided generator images actually give rise to a well-defined homomorphism. No checking is performed by the constructor.

Note that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of R.

74.7 Conversion to a Finitely Presented Group

There is a standard way to convert a rewrite group into a finitely presented group using the functions `Relations` and `Simplify`. This is shown in the following example.

Example H74E10

We construct a two generator free abelian group as a rewrite group and then convert it into a finitely presented group.

```
> FG<a,b> := FreeGroup(2);
> F := quo< FG | b^-1*a*b=a >;
> G := RWSGroup(F);
> print G;
A confluent rewrite group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
Ordering = ShortLex.
The reduction machine has 5 states.
The rewrite relations are:
    a * a^-1 = Id(FG)
    a^-1 * a = Id(FG)
    b * b^-1 = Id(FG)
    b^-1 * b = Id(FG)
    b^-1 * a = a * b^-1
    b * a = a * b
    b^-1 * a^-1 = a^-1 * b^-1
    b * a^-1 = a^-1 * b
> P<x,y> := Simplify(quo< FG | Relations(G)>);
> print P;
Finitely presented group P on 2 generators
Generators as words
    x = $.1
    y = $.2
Relations
    (y, x^-1) = Id(P)
```

74.8 Bibliography

[**Hol97**] Derek Holt. *KBMAG – Knuth-Bendix in Monoids and Automatic Groups.* University of Warwick, 1997.

[**Sim94**] Charles C. Sims. *Computation with finitely presented groups.* Cambridge University Press, Cambridge, 1994.

75 AUTOMATIC GROUPS

Chapter 75
AUTOMATIC GROUPS

75.1 Introduction

Automatic groups provide a Magma level interface to Derek Holt's KBMAG programs, and specifically to KBMAG's automatic groups program *autgroup*. Much of the material in this chapter is based on the KBMAG documentation [Hol97]. Familiarity with the Knuth–Bendix completion procedure and the automata associated with a short-lex automatic group is assumed. Some familiarity with KBMAG would be beneficial.

75.1.1 Terminology

An automatic group G is a finitely presented group in which various group operations, notably equality between words of G and word enumeration, are decidable through the use of various automata. The words in the automatic group G that can be computed in MAGMA are ordered using the short-lex ordering on words (shorter words come before longer, and for words of equal length lexicographical ordering is used, based on the given ordering of the generators).

75.1.2 The Category of Automatic Groups

The family of all automatic groups forms a category. The objects are the automatic groups and the morphisms are group homomorphisms. The MAGMA designation for this category of groups is `GrpAtc`. Elements of a automatic group are designated as `GrpAtcElt`.

75.1.3 The Construction of an Automatic Group

An automatic group G is constructed in a three-step process:

(i) We construct a free group FG.

(ii) We construct a quotient F of FG.

(iii) We create a monoid presentation for F and then run procedures which attempt to construct the automata associated with G and to prove them correct.

These procedures may or may not succeed. Of course, if G is not an automatic group then they have no chance of succeeding.

75.2 Creation of Automatic Groups

75.2.1 Construction of an Automatic Group

`AutomaticGroup(F: parameters)`

`IsAutomaticGroup(F: parameters)`

Internally a monoid presentation P of the group F is constructed. By default the generators of P are taken to be $g_1, {g_1}^{-1}, \ldots, g_n, {g_n}^{-1}$ where $g_1, \ldots, g_n$ are the generators of F. The relations of P are taken to be the relations of F. The trivial relations between the generators and their inverses are also added. The word ordering is the short-lex ordering. The Knuth–Bendix completion procedure for monoids is now run on P to calculate the word difference automata corresponding to the generated equations, which are then used to calculate the finite state automata associated with a short-lex automatic group. In successful cases these automata are proved correct in the final step.

If the procedure succeeds the result will be an automatic group, G, containing four automata. These are the first and second word-difference machines, the word acceptor, and the word multiplier. The form `AutomaticGroup` returns an automatic group while the form `IsAutomaticGroup` returns the boolean value *true* and the automatic group. If the procedure fails, the first form does not return a value while the second returns the boolean value *false*.

For simple examples, the algorithms work quickly, and do not require much space. For more difficult examples, the algorithms are often capable of completing successfully, but they can sometimes be expensive in terms of time and space requirements. Another point to be borne in mind is that the algorithms sometimes produce temporary disk files which the user does not normally see (because they are automatically removed after use), but can occasionally be very large. These files are stored in the */tmp* directory. If you interrupt a running automatic group calculation you must remove these temporary files yourself.

As the Knuth–Bendix procedure will more often than not run forever, some conditions must be specified under which it will stop. These take the form of limits that are placed on certain variables, such as the number of reduction relations. If any of these limits are exceeded during a run of the completion procedure it will fail, returning a non-confluent automatic group. The optimal values for these limits varies from example to example. Some of these limits may be specified by setting parameters (see the next section). In particular, if a first attempt to compute the automatic structure of a group fails, it should be run again with the parameter `Large` (or `Huge`) set to `true`.

Example H75E1

We construct the automatic structure for the fundamental group of the torus. Since a generator ordering is not specified, the default generator ordering, $[a, a^{-1}, b, b^{-1}, c, c^{-1}, d, d^{-1}]$, is used.

```
> FG<a,b,c,d> := FreeGroup(4);
```

```
> F := quo< FG | a^-1*b^-1*a*b=d^-1*c^-1*d*c>;
> f, G := IsAutomaticGroup(F);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#Halting with 118 equations.
#First word-difference machine with 33 states computed.
#Second word-difference machine with 33 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 36 states computed.
#General multiplier with 104 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 220 states computed.
#Checking inverse and short relations.
#Checking relation:  _8*_6*_7*_5 = _2*_4*_1*_3
#Axiom checking succeeded.
> G;
An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1, c, c^-1, d, d^-1 ]
The second word difference machine has 33 states.
The word acceptor has 36 states.
```

75.2.2 Modifying Limits

In this section we describe the various parameters used to control the execution of the procedures employed to determine the automatic structure.

`AutomaticGroup(F: parameters)`

`IsAutomaticGroup(F: parameters)`

Attempt to construct an automatic structure for the finitely presented group F (see the main entry). We now present details of the various parameters used to control the execution of the procedures.

`Large` BoolElt *Default :* `false`

If `Large` is set to `true` large hash tables are used internally. Also the Knuth–Bendix algorithm is run with larger parameters, specifically `TidyInt` is set to 500, `MaxRelations` is set to 262144, `MaxStates` is set to unlimited, `HaltingFactor` is set to 100, `MinTime` is set to 20 and `ConfNum` is set to 0. It is advisable to use this option only after having first tried without it, since it will result in much longer execution times for easy examples.

`Huge` BoolElt *Default :* `false`

Setting `Huge` to `true` doubles the size of the hash tables and `MaxRelations` over the `Large` parameter. As with the `Large` parameter, it is advisable to use this option only after having first tried without it.

`MaxRelations` RngIntElt *Default :* 200

Limit the maximum number of reduction equations to `MaxRelations`.

`TidyInt` RngIntElt *Default :* 20

After finding n new reduction equations, the completion procedure interrupts the main process of looking for overlaps, to tidy up the existing set of equations. This will eliminate any redundant equations performing some reductions on their left and right hand sides to make the set as compact as possible. (The point is that equations discovered later often make older equations redundant or too long.) The word-differences arising from the equations are calculated after each such tidying and the number reported if verbose printing is on. The best strategy in general is to try a small value of `TidyInt` first and, if that is not successful, try increasing it. Large values such as 1000 work best in really difficult examples.

`GeneratorOrder` SeqEnum *Default :*

Give an ordering for the generators of P. This ordering affects the ordering of words in the alphabet. If not specified, the ordering defaults to $[g_1, {g_1}^{-1}, \ldots, g_n, {g_n}^{-1}]$ where $g_1, \ldots, g_n$ are the generators of F.

`MaxWordDiffs` RngIntElt *Default :*

Limit the maximum number of word differences to `MaxWordDiffs`. The default behaviour is to increase the number of allowed word differences dynamically as required, and so usually one does not need to set this option.

`HaltingFactor` RngIntElt *Default :* 100

`MinTime` RngIntElt *Default :* 5

These options are experimental halting options. `HaltingFactor` is a positive integer representing a percentage. After each tidying it is checked whether both the number of equations and the number of states have increased by more than `HaltingFactor` percent since the number of word-differences was last less than what it is now. If so the program halts. A sensible value seems to be 100, but occasionally a larger value is necessary. If the `MinTime` option is also set then halting only occurs if at least `MinTime` seconds of cpu-time have elapsed altogether. This is sometimes necessary to prevent very early premature halting. It is not very satisfactory, because of course the cpu-time depends heavily on the particular computer being used, but no reasonable alternative has been found yet.

`SetVerbose("KBMAG", v)`

Set the verbose printing level for the Knuth–Bendix completion algorithm. Setting this level allows a user to control how much extra information on the progress of the algorithm is printed. Currently the legal values for v are 0 to 3 inclusive. Setting v to 0 corresponds to the '-silent' option of KBMAG in which no extra output is

printed. Setting v to 2 corresponds to the '-v' (verbose) option of KBMAG in which a small amount of extra output is printed. Setting v to 3 corresponds to the '-vv' (very verbose) option of KBMAG in which a huge amount of diagnostic information is printed.

Example H75E2

We attempt to construct an automatic structure for one of Listing's knot groups.

```
> F := Group< d, f | f*d*f^-1*d*f*d^-1*f^-1*d*f^-1*d^-1=1>;
> b, G := IsAutomaticGroup(F);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#Maximum number of equations exceeded.
#Halting with 195 equations.
#First word-difference machine with 45 states computed.
#Second word-difference machine with 53 states computed.
> b;
false;
```

So this attempt has failed. We run the `IsAutomaticGroup` function again setting `Large` to `true`. This time we succeed.

```
> f, G := IsAutomaticGroup(F : Large := true);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 262144
MaxStates     = 0
TidyInt       = 500
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#Halting with 3055 equations.
#First word-difference machine with 49 states computed.
#Second word-difference machine with 61 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 101 states computed.
#General multiplier with 497 states computed.
#Multiplier incorrect with generator number 3.
#General multiplier with 509 states computed.
#Multiplier incorrect with generator number 3.
#General multiplier with 521 states computed.
#Multiplier incorrect with generator number 3.
#General multiplier with 525 states computed.
#Validity test on general multiplier succeeded.
```

```
#General length-2 multiplier with 835 states computed.
#Checking inverse and short relations.
#Checking relation:  _3*_1*_4*_1*_3 = _1*_3*_2*_3*_1
#Axiom checking succeeded.
> G;
An automatic group.
Generator Ordering = [ d, d^-1, f, f^-1 ]
The second word difference machine has 89 states.
The word acceptor has 101 states.
```

Example H75E3

We construct the automatic group corresponding to the fundamental group of the trefoil knot. A generator order is specified.

```
> F<a, b> := Group< a, b | b*a^-1*b=a^-1*b*a^-1>;
> f, G := IsAutomaticGroup(F: GeneratorOrder := [a,a^-1, b, b^-1]);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#Halting with 83 equations.
#First word-difference machine with 15 states computed.
#Second word-difference machine with 17 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 15 states computed.
#General multiplier with 67 states computed.
#Multiplier incorrect with generator number 4.
#General multiplier with 71 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 361 states computed.
#Checking inverse and short relations.
#Checking relation:  _3*_2*_3 = _2*_3*_2
#Axiom checking succeeded.
> G;
An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
The second word difference machine has 21 states.
The word acceptor has 15 states.
```

75.2.3 Accessing Group Information

The functions in this group provide access to basic information stored for an automatic group G.

`G . i`

The i-th defining generator for G. The integer i must lie in the range $[-r, r]$, where r is the number of group G.

`Generators(G)`

A sequence containing the defining generators for G.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of defining generators for G.

Example H75E4

We illustrate the access operations using the Von Dyck (2,3,5) group (isomorphic to A_5).

```
> F<a,b> := FreeGroup(2);
> Q := quo< F | a*a=1, b*b=b^-1, a*b^-1*a*b^-1*a=b*a*b*a*b>;
> f, G<a,b> := IsAutomaticGroup(Q);
> G;
An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1 ]
The second word difference machine has 33 states.
The word acceptor has 28 states.
> print G.1*G.2;
a * b
> print Generators(G);
[ a, b ]
> print Ngens(G);
2
> rels := Relations(G);
> print rels[1];
Q.2 * Q.2^-1 = Id(Q)
> print rels[2];
Q.2^-1 * Q.2 = Id(Q)
> print rels[3];
Q.1^2 = Id(Q)
> print rels[4];
Q.2^2 = Q.2^-1
> print Nrels(G);
18
> print Ordering(G);
ShortLex
```

`FPGroup(G)`

Returns the finitely presented group F used in the construction of G, and the isomorphism from F to G.

`WordAcceptor(G)`

A record describing the word acceptor automaton stored in G.

`WordAcceptorSize(G)`

The number of states of the word acceptor automaton stored in G, and the size of the alphabet of this automaton.

`WordDifferenceAutomaton(G)`

A record describing the word difference automaton stored in G.

`WordDifferenceSize(G)`

The number of states of the 2nd word difference automaton stored in G, and the size of the alphabet of this automaton.

`WordDifferences(G)`

The labels of the states of the word difference automaton stored in G. The result is a sequence of elements of the finitely presented group used in the construction of G.

`GeneratorOrder(G)`

The value of the `GeneratorOrder` parameter used in the construction of G. The result is a sequence of generators and their inverses from the finitely presented group used in the construction of G.

75.3 Properties of an Automatic Group

`IsFinite(G)`

Given an automatic group G return `true` if G has finite order and `false` otherwise. If G does have finite order also return the order of G.

`Order(G)`
`#G`

The order of the group G as an integer. If the order of G is known to be infinite, the symbol ∞ is returned.

Example H75E5

We construct the group $\mathbf{Z} \wr C_2$ and compute its order. The result of `Infinity` indicates that the group has infinite order.

```
> F<a,b,t> := FreeGroup(3);
> Q := quo< F | t^2=1, b*a=a*b, t*a*t=b>;
> f, G := IsAutomaticGroup(Q);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#System is confluent.
#Halting with 14 equations.
#First word-difference machine with 14 states computed.
#Second word-difference machine with 14 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 6 states computed.
#General multiplier with 27 states computed.
#Validity test on general multiplier succeeded.
#Checking inverse and short relations.
#Axiom checking succeeded.
> Order(G);
Infinity
```

Example H75E6

We construct a three fold cover of A_6 and check whether it has finite order.

```
> FG<a,b> := FreeGroup(2);
> F := quo< FG | a^3=1, b^3=1, (a*b)^4=1, (a*b^-1)^5=1>;
> f, G := IsAutomaticGroup(F : GeneratorOrder := [a,b,a^-1,b^-1]);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#System is confluent.
#Halting with 183 equations.
#First word-difference machine with 289 states computed.
#Second word-difference machine with 360 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 314 states computed.
#General multiplier with 1638 states computed.
```

```
#Multiplier incorrect with generator number 4.
#General multiplier with 1958 states computed.
#Multiplier incorrect with generator number 2.
#General multiplier with 2020 states computed.
#Multiplier incorrect with generator number 1.
#General multiplier with 2038 states computed.
#Validity test on general multiplier succeeded.
#General length-2 multiplier with 4252 states computed.
#Checking inverse and short relations.
#Checking relation:  _1*_2*_1*_2 = _4*_3*_4*_3
#Checking relation:  _1*_4*_1*_4*_1 = _2*_3*_2*_3*_2
#Axiom checking succeeded.
> IsFinite(G);
true 1080
> isf, ord := IsFinite(G);
> isf, ord;
true 1080
```

75.4 Arithmetic with Words

75.4.1 Construction of a Word

`G ! [ i`$_1$`, ..., i`$_s$` ]`

Given an automatic group G defined on r generators and a sequence $[i_1, \cdots, i_s]$ of integers lying in the range $[-r, r]$, excluding 0, construct the word

$$G.|i_1|^{\epsilon_1} * G.|i_2|^{\epsilon_2} * \cdots * G.|i_s|^{\epsilon_s}$$

where ϵ_j is $+1$ if i_j is positive, and -1 if i_j is negative. The word will be returned in reduced form.

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity word in the automatic group G.

`Parent(w)`

The parent group G for the word w.

Example H75E7

We construct some words in a two-generator two-relator group.

```
> F<a, b> := Group< a, b | a^2 = b^2, a*b*a = b*a*b >;
> f, G<a, b> := IsAutomaticGroup(F);
> G;
An automatic group.
Generator Ordering = [ $.1, $.1^-1, $.2, $.2^-1 ]
The second word difference machine has 11 states.
The word acceptor has 8 states.

> Id(G);
Id(G)
> print G!1;
Id(G)
> a*b*a*b^3;
a^4 * b * a
> G![1,2,1,2,2,2];
a^4 * b * a
```

75.4.2 Operations on Elements

Having constructed an automatic group G one can perform arithmetic with words in G. Assuming we have $u, v \in G$ then the product $u * v$ will be computed as follows:

(i) The product $w = u * v$ is formed as a product in the appropriate free group.

(ii) w is reduced using the second word difference machine associated with G.

Note that:

(i) Reduction of w can cause an increase in the length of w. At present there is an internal limit on the length of a word – if this limit is exceeded during reduction an error will be raised. Hence any word operation involving reduction can fail.

(ii) The implementation is designed more with speed of execution in mind than with minimizing space requirements; thus, the reduction machine is always used to carry out word reduction, which can be space-consuming, particularly when the number of generators is large.

`u * v`

Given words w and v belonging to a common group, return their product.

`u / v`

Given words w and v belonging to a common group, return the product of the word u by the inverse of the word v, i.e. the word $u * v^{-1}$.

`u ^ n`

The n-th power of the word w.

`u ^ v`

Given words w and v belonging to a common group, return the conjugate of the word u by the word v, i.e. the word $v^{-1} * u * v$.

`Inverse(w)`

The inverse of the word w.

`(u, v)`

Given words w and v belonging to a common group, return the commutator of the words u and v, i.e., the word $u^{-1}v^{-1}uv$.

`(u`$_1$`, ..., u`$_r$`)`

Given r words $u_1, \ldots, u_r$ belonging to a common group, return their commutator. Commutators are *left-normed*, so they are evaluated from left to right.

`u eq v`

Given words w and v belonging to the same group, return `true` if w and v reduce to the same normal form, `false` otherwise. If G is confluent this tests for equality. If G is non-confluent then two words which are the same may not reduce to the same normal form.

`u ne v`

Given words w and v belonging to the same group, return `false` if w and v reduce to the same normal form, `true` otherwise. If G is confluent this tests for non-equality. If G is non-confluent then two words which are the same may reduce to different normal forms.

`IsId(w)`
`IsIdentity(w)`

Returns `true` if the word w is the identity word.

`#u`

The length of the word w.

`ElementToSequence(u)`
`Eltseq(u)`

The sequence Q obtained by decomposing the element u of a rewrite group into its constituent generators and generator inverses. Suppose u is a word in the rewrite group G. Then, if $u = G.i_1^{e_1} \cdots G.i_m^{e_m}$, with each $e_i = \pm 1$, then $Q[j] = i_j$ if $e_j = +1$ and $Q[j] = -i_j$ if $e_j = -1$, for $j = 1, \ldots, m$.

Example H75E8

We illustrate the word operations by applying them to elements of the fundamental group of a 3-manifold.
We illustrate the word operations by applying them to elements of a free group of rank two (with lots of redundant generators).

```
> FG<a,b,c,d,e> := FreeGroup(5);
> F := quo< FG | a*d*d=1, b*d*d*d=1, c*d*d*d*d*d*e*e*e=1>;
> f, G<a,b,c,d,e> := IsAutomaticGroup(F);
> G;
An automatic group.
Generator Ordering = [ a, a^-1, b, b^-1, c, c^-1, d, d^-1, e, e^-1 ]
The second word difference machine has 41 states.
The word acceptor has 42 states.
> print a*d;
d^-1
> print a/(d^-1);
d^-1
> print c*d^5*e^2;
e^-1
> print a^b, b^-1*a*b;
a a
> print (a*d)^-2, Inverse(a*d)^2;
a^-1 a^-1
> print c^-1*d^-1*c*d eq (c,d);
true
> print IsIdentity(b*d^3);
true
> print #(c*d*d*d*d*d*e*e);
1
```

75.5 Homomorphisms

For a general description of homomorphisms, we refer to chapter 16. This section describes some special aspects of homomorphisms whose domain or codomain is an automatic group.

75.5.1 General Remarks

Groups in the category `GrpAtc` currently are accepted as codomains only in some special situations. The most important cases in which an automatic group can be used as a codomain are group homomorphisms whose domain is in one of the categories `GrpFP`, `GrpGPC`, `GrpRWS` or `GrpAtc`.

75.5.2 Construction of Homomorphisms

`hom< A -> G | S >`

Returns the homomorphism from the automatic group A to the group G defined by the expression S which can be the one of the following:

(i) A list, sequence or indexed set containing the images of the n generators $A.1, \ldots, A.n$ of A. Here, the i-th element of S is interpreted as the image of $A.i$, i.e. the order of the elements in S is important.

(ii) A list, sequence, enumerated set or indexed set, containing n tuples $< x_i, y_i >$ or arrow pairs $x_i -> y_i$, where x_i is a generator of A and $y_i \in G$ $(i = 1, \ldots, n)$ and the set $\{x_1, \ldots, x_n\}$ is the full set of generators of A. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

It is the user's responsibility to ensure that the provided generator images actually give rise to a well-defined homomorphism. No checking is performed by the constructor.

Note that it is currently not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of A.

75.6 Set Operations

In this section we describe functions which allow the user to enumerate various sets of elements of an automatic group G.

`Random(G, n)`

A random word of length at most n in the generators of G.

`Random(G)`

A random word (of length at most the order of G) in the generators of G.

`Representative(G)`

`Rep(G)`

An element chosen from G.

`Set(G, a, b)`

`Search` MonStgElt *Default : "DFS"*

Create the set of words, w, in G with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Set(G)`

`Search` MonStgElt *Default :* "*DFS*"

Create the set of words that is the carrier set of G. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Seq(G, a, b)`

`Search` MonStgElt *Default :* "*DFS*"

Create the sequence S of words, w, in G with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

`Seq(G)`

`Search` MonStgElt *Default :* "*DFS*"

Create a sequence S of words from the carrier set of G. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

Example H75E9

We construct the group D_{22}, together with a representative word from the group, a random word and a random word of length at most 5 from the group, and the set of elements of the group.

```
> FG<a,b,c,d,e,f> := FreeGroup(6);
> F := quo< FG | a*c^-1*a^-1*d=1, b*f*b^-1*e^-1=1,
>                c*e*c^-1*d^-1=1, d*f^-1*d^-1*a=1,
>                e*b*e^-1*a^-1=1, f*c^-1*f^-1*b^-1=1 >;
> f, G<a,b,c,d,e,f> := IsAutomaticGroup(F);
Running Knuth-Bendix with the following parameter values
MaxRelations  = 200
MaxStates     = 0
TidyInt       = 20
MaxWdiffs     = 512
HaltingFactor = 100
MinTime       = 5
#System is confluent.
#Halting with 41 equations.
#First word-difference machine with 16 states computed.
```

```
#Second word-difference machine with 17 states computed.
#System is confluent, or halting factor condition holds.
#Word-acceptor with 6 states computed.
#General multiplier with 58 states computed.
#Validity test on general multiplier succeeded.
#Checking inverse and short relations.
#Axiom checking succeeded.
> Representative(G);
Id(G)
> Random(G);
a*c;
> Random(G, 5);
a * b
> Set(G);
{ a * d * b, a * b, a * b * e, a * c, a * d, d * b, b * e,
  a * b * a, a * b * d, b * a, a * c * e, Id(G), b * d, c * e,
  e, f, a, a * e, b, c, a * f, d }
> Seq(G : Search := "BFS");
[ Id(G), a, b, c, d, e, f, a * b, a * c, a * d, a * e, a * f,
  b * a, b * d, b * e, c * e, d * b, a * b * a, a * b * d,
  a * b * e, a * c * e, a * d * b ]
```

75.7 The Growth Function

`GrowthFunction(G)`

`Primes` SeqEnum *Default :* []

Compute the growth function of the word acceptor automaton associated with G. The growth function of a DFA, A, is the quotient of two integral polynomials in a single variable x. The coefficient of x^n in the Taylor expansion (about 0) of this quotient is equal to the number of words of length n accepted by A. That is, the result is a closed form for this generating function.

The algorithm is by Derek Holt. The `Primes` parameter is no longer used, but is kept for backward compatibility. It may be removed in future releases.

Example H75E10

We construct a dihedral group of order 10 and compute the growth function of its word acceptor. As the group is finite, the result will be a polynomial. Note here that the `R!f` is only necessary to get pretty printing, specifically to ensure that f is printed in the variable x.

```
> R<x> := RationalFunctionField(Integers());
> FG<a,b> := FreeGroup(2);
> Q := quo< FG | a^5, b^2, a^b = a^-1>;
> G := AutomaticGroup(Q);
> f := GrowthFunction(G);
```

```
> R!f;
2*x^3 + 4*x^2 + 3*x + 1
```

Now we take as example an infinite dihedral group. The group is infinite, so the result cannot be polynomial. We then extract the coefficients of the growth function for word lengths 0 to 14.

```
> FG2<d,e> := FreeGroup(2);
> Q2 := quo<FG2| e^2, d^e = d^-1>;
> G2 := AutomaticGroup(Q2);
> f2 := GrowthFunction(G2);
> R!f2;
(-x^2 - 2*x - 1)/(x - 1)
> PSR := PowerSeriesRing(Integers():Precision := 15);
> Coefficients(PSR!f2);
[ 1, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
```

75.8 Bibliography

[**Hol97**] Derek Holt. *KBMAG – Knuth-Bendix in Monoids and Automatic Groups*. University of Warwick, 1997.

76 GROUPS OF STRAIGHT-LINE PROGRAMS

Chapter 76
GROUPS OF STRAIGHT-LINE PROGRAMS

76.1 Introduction

This Chapter describes the category of straight-line program groups (SLP-groups). A straight-line program is formally a sequence $[s_1, s_2, \ldots, s_n]$ such that each s_i is one of the following:

(i) A generator of the SLP-group;

(ii) A product $s_j s_k, \quad j < i, k < i$;

(iii) A power $s_j^n, \quad j < i$;

(iv) A conjugate $s_j^{s_k}, \quad j < i, k < i$.

Effectively, a straight-line program can be regarded as a word in the generators which is stored as an expression tree instead of a list of generator-exponent pairs.

The importance of such a category of groups is that storing a word as an expression tree allows much faster evaluation of homomorphisms given as the unique extension of a mapping of the generators into a group of any category, as common subexpressions may be computed once only, and powers or conjugates may be more efficiently computed in the target group than by a linear product of generators and their inverses.

The name in MAGMA for the category of SLP-groups is `GrpSLP`.

76.2 Construction of an SLP-Group and its Elements

76.2.1 Structure Constructors

`SLPGroup(n)`

Construct the free group F of straight-line programs on n generators, where n is a non-negative integer. The i-th generator may be referenced by the expression `F.i`, $i = 1, \ldots, n$.

Example H76E1

The statement

```
> F := SLPGroup(2);
```

creates the free group on two generators. Here the generators may be referenced using the standard names, `F.1` and `F.2`. Group operations on the elements will be stored as part of the result.

76.2.2 Construction of an Element

`Identity(G)`

`Id(G)`

`G ! 1`

Construct the identity element (the straight-line program [] of length 0) for the SLP-group G.

76.3 Arithmetic with Elements

`u * v`

Given straight-line programs $u = [u_1, \ldots, u_m]$ and $v = [v_1, \ldots, v_n]$ belonging to the same SLP-group G, return a straight-line program corresponding to the product of u and v. It is clear that the straight-line program $[u_1, \ldots, u_m, v_1, \ldots, v_n, u_m v_n]$ satisfies the formal definition. In practice, the u_i and v_i need not be distinct, so the resulting program may be shorter.

`u ^ m`

Given an integer m and a straight-line program u, return the straight-line program corresponding to the m-th power of u.

`u ^ v`

Given straight-line programs u and v, return the straight-line program corresponding to the conjugate of u by v.

`#u`

Given a straight-line program u, return the number of multiplication, power or conjugate operations required to evaluate a homomorphism on u.

76.3.1 Accessing the Defining Generators and Relations

The functions described here provide access to basic information stored for an SLP-group G.

`G . i`

The i-th generator for G.

`Generators(G)`

A set containing the generators for G.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of generators for B.

`Parent(u)`

The parent group G of the straight-line program u.

76.4 Addition of Extra Generators

It is often the case that a particular expression in the original generators is important in some homomorphic image of the group in the sense that it will very often form a common subexpression of subsequent expression. For this reason, MAGMA provides facilities to build related SLP-groups in which particular words acquire generator status themselves.

`AddRedundantGenerators(G, Q)`

An SLP-group H on $n + q$ generators (where G has n generators and Q has q elements). Furthermore, the identification of $G.i$ with $H.i$ ($i \leq n$) and of $Q[i]$ with $H.(n + i)$ is maintained, allowing coercion between G and H and also simple definition of homomorphisms.

76.5 Creating Homomorphisms

Because SLP-groups exist primarily to allow the user to write efficient code for evaluating words under a homomorphism, there are some extra features in the homomorphism constructor which rely on the user providing correct input.

Whne evaluating single words, it may not be desirable to explicitly construct the homomorphism. The `Evaluate` function uses the same evaluation mechanism as the homomorphisms and may a useful alternative.

`hom< G -> H | L:` *parameters* `>`

`CheckCodomain` BOOLELT *Default :* `true`

Return the group homomorphism $\phi : G \to H$ defined by the list L. The list may contain:

(i) Elements of the codomain. This form can only be used when all the preceding entries have given the image of the corresponding generator of G;

(ii) Generator-image pairs of the form `G.i -> x` or `<G.i, x>`;

(iii) A homomorphism ψ from an SLP-group B to H where G has been defined as a result of adding redundant generators to B. If this item appears, it must appear first. After the remaining generators have been processed, any images which are not yet assigned are computed from ψ. If the parameter `CheckCodomain` has the value `false`, then it is assumed that the generator images lie in the codomain.

`Evaluate(u, Q)`

`Evaluate(u, G)`

`Evaluate(v, Q)`

`Evaluate(v, G)`

Evaluate the word u using the elements of Q as images of the generators of the parent of u. The sequence Q must contains at least as many group elements as the parent of u has generators.

The second form evaluates all the words in v simultaneously, which is usually quicker than doing individual evaluations.

When the second argument is a group G, Q is taken as the sequence of generators of G.

Example H76E2

An illustration of the use of `AddRedundantGenerators` and the homomorphism constructing machinery.

```
> G := SLPGroup(2);
> M := GeneralLinearGroup(19, 7);
> P := RandomProcess(G);
> x := Random(P);
> #x;
74
```

We evaluate x im M using the `Evaluate` function.

```
> m := Evaluate(x, [M.1, M.2]);
> Order(m);
118392315154200
```

If we wish to evaluate several different words, we may be better off using a homomorphism.

```
> Q := [x^G.1, x^G.2, x^(G.1*G.2)];
> phi := hom<G -> M | M.1, M.2>;
> time R1 := phi(Q);
Time: 0.129
```

We note that x has become important since it is now a common sub-expression of several straight-line programs. We can build a homomorphism which will store the image of x by adding x as a redundant generator and defining the same homomorphism from the resulting group.

```
> H := AddRedundantGenerators(G, [x]);
> QQ := [H | x: x in Q];
```

We will define psi as the unique map on H which matches phi.

```
> psi := hom<H -> M | phi>;
> time R2 := psi(QQ);
Time: 0.000
> R1 eq R2;
true
```

In fact, if we had looked at the expression lengths of the straight-line programs involved, we would have found the following, which explains the significant speed up:

```
> [#x: x in Q];
[ 75, 75, 75 ]
> [#x: x in QQ];
[ 1, 1, 2 ]
```

76.6 Operations on Elements

76.6.1 Equality and Comparison

`u eq v`

Reutrns `true` if and only if the straight-line programs u and v are identical. Identical here means they are identical as expression trees, not that they will always evaluate to the same word in the generators.

`u ne v`

Returns `true` if and only if the straight-line programs u and v are not identical. Identical here means they are identical as expression trees, not that they will always evaluate to the same word in the generators.

76.7 Set-Theoretic Operations

76.7.1 Membership and Equality

`g in G`

Given a straight-line program g and an SLP-group G, return `true` if g is an element of G, `false` otherwise.

`g notin G`

Given an straight-line program g and an SLP-group G, return `true` if g is not an element of G, `false` otherwise.

`S subset G`

Given an group G and a set S of elements belonging to a group H, where G and H are related, return `true` if S is a subset of G, `false` otherwise.

`S notsubset G`

Given a group G and a set S of elements belonging to a group H, where G and H are related, return `true` if S is not a subset of G, `false` otherwise.

76.7.2 Set Operations

`RandomProcess(G)`

`Slots`	RngIntElt	*Default :* 10
`Scramble`	RngIntElt	*Default :* 100

Create a process to generate randomly chosen elements from the group G. The process is based on the product-replacement algorithm of [CLGM$^+$95], modified by the use of an accumulator. At all times, N elements are stored where N is the maximum of the specified value for `Slots` and `Ngens`$(G)+1$. Initially, these are just the generators of G. As well, one extra group element is stored, the accumulator. Initially, this is the identity. Random elements are now produced by successive calls to `Random(P)`, where P is the process created by this function. Each such call chooses one of the elements in the slots and multiplies it into the accumulator. The element in that slot is replaced by the product of it and another randomly chosen slot. The random value returned is the new accumulator value. Setting `Scramble := m` causes m such operations to be performed before the process is returned.

It should be noted that this process is not suitable for infinite groups, since all elements produced are products of the generators only and not their inverses. However, as long as the homomorphic image of G that is being worked with is finite, there is no problem.

`Random(P)`

Given a random element process P created by the function `RandomProcess(G)` for the SLP-group G, construct a random element of G by forming a random product over the expanded generating set stored as part of the process. The expanded generating set stored with the process is modified by replacing an existing generator by the element returned.

`Rep(G)`

A representative element of G.

Example H76E3

As an illustration of the efficiency of computing homomorphisms from SLP-groups, we set up the same random expression as both a straight-line program and a linear word.

```
> G := SLPGroup(3);
> F := FreeGroup(3);
> M := GeneralOrthogonalGroup(7, 3);
> gf := hom<G -> F | F.1, F.2, F.3>;
> gm := hom<G -> M | M.1, M.2, M.3>;
> fm := hom<F -> M | M.1, M.2, M.3>;
> P := RandomProcess(G);
> x := Random(P);
> #x;
```

```
85
```

The evaluation of the straight-line program will take 85 operations in M.

```
> w := gf(x);
> #w;
52307
```

The evaluation of the word will take 52306 multiplications in M.

```
> time h1 := gm(x);
Time: 0.020
> time h2 := fm(w);
Time: 1.640
> h1 eq h2;
true
```

76.7.3 Coercions Between Related Groups

`G ! g`

Given an element g belonging to an SLP-group H related to the group G, rewrite g as an element of G.

76.8 Bibliography

[**CLGM+95**] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien. Generating random elements of a finite group. *Comm. Algebra*, 23(13):4931–4948, 1995.

77 FINITELY PRESENTED SEMIGROUPS

Chapter 77
FINITELY PRESENTED SEMIGROUPS

77.1 Introduction

This Chapter presents the functions designed for computing with finitely-presented semigroups (fp-semigroups for short).

77.2 The Construction of Free Semigroups and their Elements

77.2.1 Structure Constructors

`FreeSemigroup(n)`

Construct the free semigroup F on n generators, where n is a positive integer. The i-th generator may be referenced by the expression `F.i`, $i = 1, \ldots, n$. Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of F. In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the **left hand side** of the assignment statement.

`FreeMonoid(n)`

Construct the free monoid F on n generators, where n is a positive integer. The i-th generator may be referenced by the expression `F.i`, $i = 1, \ldots, n$. Note that a special form of the assignment statement is provided which enables the user to assign names to the generators of F. In this form of assignment, the list of generator names is enclosed within angle brackets and appended to the variable name on the **left hand side** of the assignment statement.

Example H77E1

The statement

```
> F := FreeSemigroup(2);
```

creates the free semigroup on two generators. Here the generators may be referenced using the standard names, $F.1$ and $F.2$.
The statement

```
> F<x, y> := FreeSemigroup(2);
```

defines F to be the free semigroup on two generators and assigns the names x and y to the generators.

77.2.2 Element Constructors

Suppose S is an fp-semigroup, not necessarily free, for which generators have already been defined. A *word* is defined inductively as follows:

(i) A generator is a word;

(ii) The product uv of the words u and v is a word;

(iii) The power of a word, u^n, where u is a word and n is an integer, is a word.

An element (word) of S may be constructed as an expression in the generators as outlined below.

`S ! [i1, ... is]`

Given a semigroup S defined on r generators and a sequence $Q = [i_1, \cdots, i_s]$ of integers lying in the range $[1, r]$, construct the word $G.i_1G.i_2 \cdots G.i_s$.

`Id(M)`

`M ! 1`

Construct the identity element (empty word) for the fp-monoid M.

77.3 Elementary Operators for Words

77.3.1 Multiplication and Exponentiation

The word operations defined here may be applied either to the words of a free semigroup or the words of a semigroup with non-trivial relations.

`u * v`

Given words u and v belonging to the same fp-semigroup S, return the product of u and v.

`u ^ n`

The n-th power of the word u, where n is a positive integer.

`G ! Q`

Given a sequence Q of words belonging to the fp-semigroup G, return the product $Q[1]Q[2]\cdots Q[n]$ of the terms of Q as a word in G.

77.3.2 The Length of a Word

`#u`

The length of the word u.

77.3.3 Equality and Comparison

The words of an fp-semigroup S are ordered first by length and then lexicographically. The lexicographic ordering is determined by the following ordering on the generators:

$$S.1 < S.2 < S.3 < S.4 < \cdots$$

Here, u and v are words belonging to some common fp-semigroup.

`u eq v`

Returns `true` if the words u and v are identical (as elements of the appropriate free semigroup), `false` otherwise.

`u ne v`

Returns `true` if the words u and v are not identical (as elements of the appropriate free semigroup), `false` otherwise.

`u lt v`

Returns `true` if the word u precedes the word v, with respect to the ordering defined above for elements of an fp-semigroup, `false` otherwise.

`u le v`

Returns `true` if the word u either precedes, or is equal to, the word v, with respect to the ordering defined above for elements of an fp-semigroup, `false` otherwise.

`u ge v`

Returns `true` if the word u either follows, or is equal to, the word v, with respect to the ordering defined above for elements of an fp-semigroup, `false` otherwise.

`u gt v`

Returns `true` if the word u follows the word v, with respect to the ordering defined above for elements of an fp-semigroup.

`IsOne(u)`

Returns `true` if the word u, belonging to the monoid M, is the identity word, `false` otherwise.

77.4 Specification of a Presentation

77.4.1 Relations

`w1 = w2`

Given words w_1 and w_2 over the generators of an fp-semigroup S, create the relation $w_1 = w_2$. Note that this relation is not automatically added to the existing set of defining relations R for S. It may be added to R, for example, through use of the `quo`-constructor (see below).

`LHS(r)`

Given a relation r over the generators of S, return the left hand side of the relation r. The object returned is a word over the generators of S.

`RHS(r)`

Given a relation r over the generators of S, return the right hand side of the relation r. The object returned is a word over the generators of S.

77.4.2 Presentations

A semigroup with non-trivial relations is constructed as a quotient of an existing semigroup, possibly a free semigroup.

`Semigroup< generators | relations >`

Given a *generators* clause consisting of a list of variables $x_1, \cdots, x_r$, and a set of relations *relations* over these generators, first construct the free semigroup F on the generators $x_1, \cdots, x_r$ and then construct the quotient of F corresponding to the ideal of F defined by *relations*.

The syntax for the *relations* clause is the same as for the `quo`-constructor. The function returns:

(a) The quotient semigroup S;

(b) The natural homomorphism $\phi : F \to S$.

Thus, the statement

```
S< y1, ..., yr > := Semigroup< x1, ..., xr | w1, ..., ws >;
```

is an abbreviation for

```
F< x1, ..., xr > := FreeSemigroup(r);
S< y1, ..., yr > := quo< F | w1, ..., ws >;
```

`Monoid< generators | relations >`

Given a *generators* clause consisting of a list of variables $x_1, \cdots, x_r$, and a set of relations *relations* over these generators, first construct the free monoid F on the generators $x_1, \cdots, x_r$ and then construct the quotient of F corresponding to the ideal of F defined by *relations*.

The syntax for the *relations* clause is the same as for the `quo`-constructor. The function returns:

(a) The quotient monoid M;

(b) The natural homomorphism $\phi : F \to M$.

Thus, the statement

```
M< y1, ..., yr > := Monoid< x1, ..., xr | w1, ..., ws >;
```

is an abbreviation for

```
F< x1, ..., xr > := FreeMonoid(r);
M< y1, ..., yr > := quo< F | w1, ..., ws >;
```

Example H77E2

We create the monoid defined by the presentation $< x, y \mid x^2, y^2, (xy)^2 >$.

```
> M<x,y> := Monoid< x, y | x^2, y^2, (x*y)^2 >;
> M;
Finitely presented monoid
Relations:
    x^2 = Id(M)
    y^2 = Id(M)
    (x * y)^2 = Id(M)
```

77.4.3 Accessing the Defining Generators and Relations

The functions in this group provide access to basic information stored for a finitely-presented semigroup G.

`S . i`

The i-th defining generator for S.

`Generators(S)`

A set containing the generators for S.

`NumberOfGenerators(S)`

`Ngens(S)`

The number of generators for S.

`Parent(u)`

The parent semigroup S of the word u.

`Relations(S)`

A sequence containing the defining relations for S.

77.5 Subsemigroups, Ideals and Quotients

77.5.1 Subsemigroups and Ideals

`sub< S | L`$_1$`, ..., L`$_r$ `>`

Construct the subsemigroup R of the fp-semigroup S generated by the words specified by the terms of the *generator list* $L_1,\ldots, L_r$.
A term L_i of the generator list may consist of any of the following objects:

(a) A word;

(b) A set or sequence of words;

(c) A sequence of integers representing a word;

(d) A set or sequence of sequences of integers representing words;

(e) A subsemigroup of an fp-semigroup;

(f) A set or sequence of subsemigroups.

The collection of words and semigroups specified by the list must all belong to the semigroup S, and R will be constructed as a subgroup of S.

The generators of R consist of the words specified directly by terms L_i together with the stored generating words for any semigroups specified by terms of L_i. Repetitions of an element and occurrences of the identity element are removed (unless R is trivial).

`ideal< S | L`$_1$`, ..., L`$_r$ `>`

Construct the two-sided ideal I of the fp-semigroup S generated by the words specified by the terms of the *generator list* $L_1,\ldots, L_r$.

The possible forms of a term L_i of the generator list are the same as for the `sub`-constructor.

`lideal< G | L`$_1$`, ..., L`$_r$ `>`

Construct the left ideal I of the fp-semigroup S generated by the words specified by the terms of the *generator list* $L_1,\ldots, L_r$.

The possible forms of a term L_i of the generator list are the same as for the `sub`-constructor.

`rideal< G | L`$_1$`, ..., L`$_r$ `>`

Construct the right ideal I of the fp-semigroup S generated by the words specified by the terms of the *generator list* $L_1,\ldots, L_r$.

The possible forms of a term L_i of the generator list are the same as for the `sub`-constructor.

77.5.2 Quotients

`quo< F | relations >`

Given an fp-semigroup F, and a list of relations *relations* over the generators of F, construct the quotient of F by the ideal of F defined by *relations*.

The expression defining F may be either simply the name of a previously constructed semigroup, or an expression defining an fp-semigroup.

Each term of the list *relations* must be a *relation*, a *relation list* or, if S is a monoid, a *word*.

A *word* is interpreted as a relator if S is a monoid.

A *relation* consists of a pair of words, separated by '='. (See above).

A *relation list* consists of a list of words, where each pair of adjacent words is separated by '=': $w_1 = w_2 = \cdots = w_r$. This is interpreted as the relations $w_1 = w_r, \ldots, w_{r-1} = w_r$.

Note that the relation list construct is only meaningful in the context of the `fp semigroup`-constructor.

In the context of the `quo`-constructor, the identity element (empty word) of a monoid may be represented by the digit `1`.

Note that this function returns:

(a) The quotient semigroup S;

(b) The natural homomorphism $\phi : F \to S$.

77.6 Extensions

`DirectProduct(R, S)`

Given two fp-semigroups R and S, construct the direct product of R and S.

`FreeProduct(R, S)`

Given two fp-semigroups R and S, construct the free product of R and S.

77.7 Elementary Tietze Transformations

`AddRelation(S, r)`

`AddRelation(S, r, i)`

Given an fp-semigroup S and a relation r in the generators of S, create the quotient semigroup obtained by adding the relation r to the defining relations of S. If an integer i is specified as third argument, insert the new relation after the i-th relation of S. If the third argument is omitted, r is added to the end of the relations that are carried across from S.

`DeleteRelation(S, r)`

Given an fp-semigroup S and a relation r that occurs among the given defining relations for S, create the semigroup T, having the same generating set as S but with the relation r removed.

`DeleteRelation(S, i)`

Given an fp-semigroup S and an integer i, $1 \leq i \leq m$, where m is the number of defining relations for S, create the semigroup T having the same generating set as S but with the i-th relation omitted.

`ReplaceRelation(S, r1, r2)`

Given an fp-semigroup S and relations r_1 and r_2 in the generators of S, where r_1 is one of the given defining relations for S, create the semigroup T having the same generating set as S but with the relation r_1 replaced by the relation r_2.

`ReplaceRelation(S, i, r)`

Given an fp-semigroup S, an integer i, $1 \leq i \leq m$, where m is the number of defining relations for S, and a relation r in the generators of S, create the semigroup T having the same generating set as S but with the i-th relation of S replaced by the relation r.

`AddGenerator(S)`

Given an fp-semigroup S with presentation $< X \mid R >$, create the semigroup T with presentation $< X \cup \{y\} \mid R >$, where y denotes a new generator.

`AddGenerator(S, w)`

Given an fp-semigroup S with presentation $< X \mid R >$ and a word w in the generators of S, create the semigroup T with presentation $< X \cup \{y\} \mid R \cup \{y = w\} >$, where y denotes a new generator.

`DeleteGenerator(S, y)`

Given an fp-semigroup S with presentation $< X \mid R >$ and a generator y of S such that either S has no relations involving y, or a single relation r containing a single occurrence of y, create the semigroup T with presentation $< X - \{y\} \mid R - \{r\} >$.

77.8 String Operations on Words

`Eliminate(u, x, v)`

Given words u and v, and a generator x, belonging to a semigroup S, return the word obtained from u by replacing each occurrence of x by v.

`Match(u, v, f)`

Suppose u and v are words belonging to the same semigroup S, and that f is an integer such that $1 \leq f \leq \#u$. If v is a subword of u, the function returns true, as well as the least integer l such that:

(a) $l \geq f$; and,

(b) v appears as a subword of u, starting at the l-th letter of u.

If no such l is found, `Match` returns only false.

`Random(S, m, n)`

A random word of length l in the generators of the semigroup S, where $m \leq l \leq n$.

`RotateWord(u, n)`

The word obtained by cyclically permuting the word u by n places. If n is positive, the rotation is from left to right, while if n is negative the rotation is from right to left. In the case where n is zero, the function returns u.

`Substitute(u, f, n, v)`

Given words u and v belonging to a semigroup S, and non-negative integers f and n, this function replaces the substring of u of length n, starting at position f, by the word v. Thus, if $u = x_{i_1} \cdots x_{i_f} \cdots x_{i_{f+n-1}} \cdots x_{i_m}$ then the substring $x_{i_f} \cdots x_{i_{f+n-1}}$ is replaced by v. If u and v belong to a monoid M and the function is invoked with $v =$`Id(M)`, then the substring $x_{i_f} \cdots x_{i_{f+n-1}}$ of u is deleted.

`Subword(u, f, n)`

The subword of the word u comprising the n consecutive letters commencing at the f-th letter of u.

`ElementToSequence(u)`

`Eltseq(u)`

The sequence obtained by decomposing u into the indices of its constituent generators. Thus, if $u = x_{i_1} \ldots x_{i_m}$, then the sequence constructed by `ElementToSequence` is $[i_1, i_2, \ldots, i_m]$.

78 MONOIDS GIVEN BY REWRITE SYSTEMS

Chapter 78
MONOIDS GIVEN BY REWRITE SYSTEMS

78.1 Introduction

The category of monoids defined by finite sets of rewrite rules provide a Magma level interface to Derek Holt's KBMAG programs, and specifically to KBMAG's Knuth–Bendix completion procedure on monoids defined by a finite presentation. As such much of the documentation in this chapter is taken from the KBMAG documentation [Hol97]. Familiarity with the Knuth–Bendix completion procedure is assumed. Some familiarity with KBMAG would be beneficial.

78.1.1 Terminology

A rewrite monoid M is a finitely presented monoid in which equality between elements of M, called *words* or *strings*, is decidable via a sequence of rewriting equations, called *reduction relations, rules, or equations*. In the interests of efficiency the reduction rules are codified into a finite state automaton called a *reduction machine*. The words in a rewrite monoid M are ordered, as are the reduction relations of M. Several possible orderings of words are supported, namely short-lex, recursive, weighted short-lex and wreath-product orderings. A rewrite monoid can be *confluent* or *non-confluent*. If a rewrite monoid M is confluent its reduction relations, or more specifically its reduction machine, can be used to reduce words in M to their irreducible normal forms under the given ordering, and so the word problem for M can be efficiently solved.

78.1.2 The Category of Rewrite Monoids

The family of all rewrite monoids forms a category. The objects are the rewrite monoids and the morphisms are monoid homomorphisms. The MAGMA designation for this category of monoids is `MonRWS`. Elements of a rewrite monoid are designated as `MonRWSElt`.

78.1.3 The Construction of a Rewrite Monoid

A rewrite monoid M is constructed in a three-step process:

(i) A free monoid F of the appropriate rank is defined.

(ii) A quotient Q of F is created.

(iii) The Knuth–Bendix completion procedure is applied to the monoid Q to produce a monoid M defined by a rewrite system.

The Knuth–Bendix procedure may or may not succeed. If it fails the user may need to perform the above steps several times, manually adjusting parameters that control the execution of the Knuth–Bendix procedure. If it succeeds then the rewrite systems constructed will be confluent.

78.2 Construction of a Rewrite Monoid

`RWSMonoid(Q: parameters)`

The Knuth–Bendix completion procedure for monoids is run, with the relations of Q taken as the initial reduction rules for the procedure. Regardless of whether or not the completion procedure succeeds, the result will be a rewrite monoid, M, containing a reduction machine and a sequence of reduction relations. If the procedure succeeds M will be marked as confluent, and the word problem for M is therefore decidable. If, as is very likely, the procedure fails then M will be marked as non-confluent. In this case M will contain both the reduction relations and the reduction machine computed up to the point of failure.

As the Knuth–Bendix procedure will more often than not run forever, some conditions must be specified under which it will stop. These take the form of limits that are placed on certain variables, such as the number of reduction relations. If any of these limits are exceeded during a run of the completion procedure it will fail, returning a non-confluent rewrite monoid. The optimal values for these limits varies from example to example.

`MaxRelations`	RngIntElt	*Default :* 32767

Limit the maximum number of reduction equations to `MaxRelations`.

`GeneratorOrder`	SeqEnum	*Default :*

Give an ordering for the generators. This ordering affects the ordering of words in the alphabet. If not specified the ordering defaults to the order induced by Q's generators, that is $[g_1, \ldots, g_n]$ where $g_1, \ldots, g_n$ are the generators of Q.

`Ordering`	MonStgElt	*Default :* "*ShortLex*"
`Levels`	SeqEnum	*Default :*
`Weights`	SeqEnum	*Default :*

`Ordering := "ShortLex"`: Use the short-lex ordering on strings. Shorter words come before longer, and for words of equal length lexicographical ordering is used, using the given ordering of the generators.
`Ordering := "Recursive" | "RTRecursive"`: Use a recursive ordering on strings. There are various ways to define this. Perhaps the quickest is as follows. Let u and v be strings in the generators. If one of u and v, say v, is empty, then $u \geq v$. Otherwise, let $u = u'a$ and $v = v'b$, where a and b are generators. Then $u > v$ if and only if one of the following holds:

(i) $a = b$ and $u' > v'$;

(ii) $a > b$ and $u > v'$;

(iii) $b > a$ and $u' > v$.

The `RTRecursive` ordering is similar to the `Recursive` ordering, but with $u = au'$ and $v = bv'$. Occasionally one or the other runs significantly quicker, but usually they perform similarly.

`Ordering := "WTShortLex"`: Use a weighted-lex ordering. `Weights` should be a sequence of non-negative integers, with the i-th element of `Weights` giving the weight of the i-the generator. The length of `Weights` must equal the number of generators. The length of words in the generators is then computed by adding up the weights of the generators in the words. Otherwise, ordering is as for short-lex.
`Ordering := "Wreath"`: Use a wreath-product ordering. `Levels` should be a sequence of non-negative integers, with the i-th element of `Levels` giving the level of the i-the generator. The length of `Levels` must equal the number of generators. In this ordering, two strings involving generators of the same level are ordered using short-lex, but all strings in generators of a higher level are larger than those involving generators of a lower level. That is not a complete definition; one can be found in [Sim94, pp. 46–50]. Note that the recursive ordering is the special case in which the level of generator number i is i.

`TidyInt` RngIntElt *Default :* 100

After finding `TidyInt` new reduction equations, the completion procedure interrupts the main process of looking for overlaps, to tidy up the existing set of equations. This will eliminate any redundant equations performing some reductions on their left and right hand sides to make the set as compact as possible. (The point is that equations discovered later often make older equations redundant or too long.)

`RabinKarp` Tup *Default :*

Use the Rabin-Karp algorithm for word-reduction on words having length at least l, provided that there are at least n equations, where `RabinKarp := <l, n>`. This uses less space than the default reduction automaton, but it is distinctly slower, so it should only be used when seriously short of memory. Indeed this option is only really useful for examples in which collapse occurs - i.e. at some intermediate stage of the calculation there is a very large set of equations, which later reduces to a much smaller confluent set. Collapse is not uncommon when analysing pathological presentations of finite groups, and this is one situation where the performance of the Knuth–Bendix algorithm can be superior to that of Todd-Coxeter coset enumeration. The best setting for `RabinKarp` varies from example to example - generally speaking, the smaller l is, the slower things will be, so set it as high as possible subject to not running out of memory. The number of equations n should be set higher than the expected final number of equations.

`MaxStates` RngIntElt *Default :*

Limit the maximum number of states of the finite state automaton used for word reduction to `MaxStates`. By default there is no limit, and the space allocated is increased dynamically as required. The space needed for the reduction automaton can also be restricted by using the `RabinKarp` parameter. This limit is not usually needed.

`MaxReduceLen` RngIntElt *Default :* 32767

Limit the maximum allowed length that a word can reach during reduction to `MaxReduceLen`. It is only likely to be exceeded when using the recursive ordering

on words. This limit is usually not needed.

`ConfNum` RngIntElt *Default :* 500

If `ConfNum` overlaps are processed and no new equations are discovered, then the overlap searching process is interrupted, and a fast check for confluence performed on the existing set of equations. Doing this too often wastes time, but doing it at the right moment can also save a lot of time. If `ConfNum` = 0, then the fast confluence check is performed only when the search for overlaps is complete.

Warning: Changing the default setting for any of the following parameters may either cause the procedure to terminate without having found a confluent presentation or change the monoid.

`MaxStoredLen` Tup *Default :*

Only equations in which the left and right hand sides have lengths at most l and r, respectively, where `MaxStoredLen := <l, r>` are kept. Of course this may cause the overlap search to complete on a set of equations that is not confluent. In some examples, particularly those involving collapse (i.e. a large intermediate set of equations, which later simplifies to a small set), it can result in a confluent set being found much more quickly. It is most often useful when using a recursive ordering on words. Another danger with this option is that sometimes discarding equations can result in information being lost, with the effect of changing the monoid defined by the equations.

`MaxOverlapLen` RngIntElt *Default :*

Only overlaps of total length at most `MaxOverlapLen` are processed. Of course this may cause the overlap search to complete on a set of equations that is not confluent.

`Sort` BoolElt *Default :* `false`

`MaxOpLen` RngIntElt *Default :* 0

If `Sort` is set to `true` then the equations will be sorted in order of increasing length of their left hand sides, rather than the default, which is to leave them in the order in which they were found. `MaxOpLen` should be a non-negative integer. If `MaxOpLen` is positive, then only equations with left hand sides having length at most `MaxOpLen` are output. If `MaxOpLen` is zero, then all equations are sorted by length. Of course, if `MaxOpLen` is positive, there is a danger that the monoid defined by the output equations may be different from the original.

`SetVerbose("KBMAG", v)`

Set the verbose printing level for the Knuth-Bendix completion algorithm. Setting this level allows a user to control how much extra information on the progress of the algorithm is printed. Currently the legal values for v are 0 to 3 inclusive. Setting v to 0 corresponds to the '-silent' option of KBMAG in which no extra output is printed. Setting v to 2 corresponds to the '-v' (verbose) option of KBMAG in which a small amount of extra output is printed. Setting v to 3 corresponds to the '-vv' (very verbose) option of KBMAG in which a huge amount of diagnostic information is printed.

Example H78E1

Starting with a monoid presentation for the alternating group A_4, we construct a rewrite system. Since we don't specify an ordering the default `ShortLex` ordering is used. Since we don't specify a generator ordering, the default generator ordering, in this case that from Q, is used.

```
> FM<g10,g20,g30> := FreeMonoid(3);
> Q := quo< FM | g10^2=1, g20*g30=1, g30*g20=1,
>         g20*g20=g30, g30*g10*g30=g10*g20*g10>;
> M := RWSMonoid(Q);
> print M;
A confluent rewrite monoid.
Generator Ordering = [ g10, g20, g30 ]
Ordering = ShortLex.
The reduction machine has 12 states.
The rewrite relations are:
    g10^2 = Id(FM)
    g20 * g30 = Id(FM)
    g30 * g20 = Id(FM)
    g20^2 = g30
    g30 * g10 * g30 = g10 * g20 * g10
    g30^2 = g20
    g20 * g10 * g20 = g10 * g30 * g10
    g30 * g10 * g20 * g10 = g20 * g10 * g30
    g10 * g20 * g10 * g30 = g30 * g10 * g20
    g20 * g10 * g30 * g10 = g30 * g10 * g20
    g10 * g30 * g10 * g20 = g20 * g10 * g30
```

Example H78E2

We construct the second of Bernard Neumann's series of increasingly complicated presentations of the trivial monoid. The example runs best with a large value of `TidyInt`. Again the default `ShortLex` ordering is used.

```
> FM<x,X,y,Y,z,Z> := FreeMonoid(6);
> Q := quo< FM |
>         x*X=1, X*x=1, y*Y=1, Y*y=1, z*Z=1, Z*z=1,
>         y*y*X*Y*x*Y*z*y*Z*Z*X*y*x*Y*Y*z*z*Y*Z*y*z*z*Y*Z*y=1,
>         z*z*Y*Z*y*Z*x*z*X*X*Y*z*y*Z*Z*x*x*Z*X*z*x*x*Z*X*z=1,
>         x*x*Z*X*z*X*y*x*Y*Y*Z*x*z*X*X*y*y*X*Y*x*y*y*X*Y*x=1>;
> M := RWSMonoid(Q : TidyInt := 3000);
> print M;
A confluent rewrite monoid.
Generator Ordering = [ x, X, y, Y, z, Z ]
Ordering = ShortLex.
The reduction machine has 1 state.
The rewrite relations are:
    Z = Id(FM)
    Y = Id(FM)
```

```
    z = Id(FM)
    X = Id(FM)
    y = Id(FM)
    x = Id(FM)
```

Example H78E3

We construct a confluent presentation of a submonoid of a nilpotent group.

```
> FM<a,b,c> := FreeMonoid(6);
> Q := quo< FM | b*a=a*b*c, c*a=a*c, c*b=b*c >;
> M := RWSMonoid(Q:Ordering:="Recursive", GeneratorOrder:=[c,b,a]);
> M;
A confluent rewrite monoid.
Generator Ordering = [ c, b, a ]
Ordering = Recursive.
The reduction machine has 3 states.
    b * a = a * b * c
    c * a = a * c
    c * b = b * c
> Order(M);
Infinity
```

Example H78E4

We construct a monoid presentation corresponding to the Fibonacci group $F(2,7)$. This is a very difficult calculation unless the parameters of `RWSMonoid` are selected carefully. The best approach is to run the Knuth-Bendix once using a `Recursive` ordering with a limit on the lengths of equations stored (`MaxStoredLen := <15,15>` works well). This will halt returning a non-confluent rewrite monoid M, and give a warning message that the Knuth-Bendix procedure only partly succeeded. The original equations should then be appended to the relations of M, and the Knuth-Bendix re-run with no limits on lengths. It will then quickly complete with a confluent set. This is typical of a number of difficult examples, where good results can be obtained by running more than once.

```
> FM<a,b,c,d,e,f,g> := FreeMonoid(7);
> I := [a*b=c, b*c=d, c*d=e, d*e=f, e*f=g, f*g=a, g*a=b];
> Q := quo<FM | I>;
> M := RWSMonoid(Q: Ordering := "Recursive", MaxStoredLen := <15,15>);
Warning: Knuth Bendix only partly succeeded
> Q := quo< FM | Relations(M) cat I>;
> M := RWSMonoid(Q: Ordering := "Recursive");
> print M;
A confluent rewrite monoid.
Generator Ordering = [ a, b, c, d, e, f, g ]
Ordering = Recursive.
The reduction machine has 30 states.
The rewrite relations are:
```

```
    c = a^25
    d = a^20
    e = a^16
    f = a^7
    g = a^23
    b = a^24
    a^30 = a
> Order(M);
30
```

It turns out that the non-identity elements of this monoid form a submonoid isomorphic to the Fibonacci group $F(2,7)$ which is cyclic of order 29.

78.3 Basic Operations

78.3.1 Accessing Monoid Information

The functions in this section provide access to basic information stored for a rewrite monoid M.

`M . i`

The i-th defining generator for M.

`Generators(M)`

A sequence containing the defining generators for M.

`NumberOfGenerators(M)`

`Ngens(M)`

The number of defining generators for M.

`Relations(M)`

A sequence containing the defining relations for M. The relations will be given between elements of the free monoid of which M is a quotient. In these relations the (image of the) left hand side (in M) will always be greater than the (image of the) right hand side (in M) in the ordering on words used to construct M.

`NumberOfRelations(M)`

`Nrels(M)`

The number of relations in M.

`Ordering(M)`

The ordering of M.

`Parent(w)`

The parent monoid M for the word w.

Example H78E5

We illustrate the access operations using the following presentation of S_4.

```
> FM<a,b> := FreeMonoid(2);
> Q := quo< FM | a^2=1, b^3=1, (a*b)^4=1 >;
> M<x,y> := RWSMonoid(Q);
> print M;
A confluent rewrite monoid.
Generator Ordering = [ a, b ]
Ordering = ShortLex.
The reduction machine has 12 states.
    a^2 = Id(FM)
    b^3 = Id(FM)
    b * a * b * a * b = a * b^2 * a
    b^2 * a * b^2 = a * b * a * b * a
    b * a * b^2 * a * b * a = a * b * a * b^2 * a * b
> print Order(M);
24
> print M.1;
x
> print M.1*M.2;
x * y
> print Generators(M);
[ x, y ]
> print Ngens(M);
2
> print Relations(M);
[ a^2 = Id(FM), b^3 = Id(FM), b * a * b * a * b = a * b^2 * a, b^2 * a * b^2 = a
* b * a * b * a, b * a * b^2 * a * b * a = a * b * a * b^2 * a * b ]
> print Nrels(M);
5
> print Ordering(M);
ShortLex
```

78.3.2 Properties of a Rewrite Monoid

`IsConfluent(M)`

Returns `true` if M is confluent, `false` otherwise.

`IsFinite(M)`

Given a confluent monoid M return `true` if M has finite order and `false` otherwise. If M does have finite order also return the order of M.

`Order(M)`
`#M`

Given a monoid M defined by a confluent presentation, this function returns the cardinality of M. If the order of M is known to be infinite ∞ is returned.

Example H78E6

We construct a threefold cover of A_6.

```
> FM<a,b> := FreeMonoid(2);
> Q := quo< FM | a^3=1, b^3=1, (a*b)^4=1, (a*b^2)^5 = 1 >;
> M := RWSMonoid(Q);
> print Order(M);
1080
> IsConfluent(M);
true
```

Example H78E7

We construct the 2-generator free abelian group and compute its order. The result `Infinity` indicates that the group has infinite order.

```
> FM<a,A,b,B> := FreeMonoid(4);
> Q := quo< FM | a*A=1, A*a=1, b*B=1, B*b=1, B*a*b=a>;
> M := RWSMonoid(Q);
> Order(M);
Infinity
```

Example H78E8

We construct the Weyl group E_8 and test whether or not it has finite order.

```
> FM<a,b,c,d,e,f,g,h> := FreeMonoid(8);
> Q := quo< FM | a^2=1, b^2=1, c^2=1, d^2=1, e^2=1, f^2=1, g^2=1,
>         h^2=1, b*a*b=a*b*a, c*a=a*c, d*a=a*d, e*a=a*e, f*a=a*f,
>         g*a=a*g, h*a=a*h, c*b*c=b*c*b, d*b=b*d, e*b=b*e, f*b=b*f,
>         g*b=b*g, h*b=b*h, d*c*d=c*d*c, e*c*e=c*e*c, f*c=c*f,
>         g*c=c*g, h*c=c*h, e*d=d*e, f*d=d*f, g*d=d*g, h*d=d*h,
>         f*e*f=e*f*e, g*e=e*g, h*e=e*h, g*f*g=f*g*f, h*f=f*h,
>         h*g*h=g*h*g>;
> M := RWSMonoid(Q);
> print IsFinite(M);
true
> isf, ord := IsFinite(M);
> print isf, ord;
true 696729600
```

78.3.3 Construction of a Word

`Identity(M)`

`Id(M)`

`M ! 1`

Construct the identity word in M.

`M ! [ i`$_1$`, ..., i`$_s$` ]`

Given a rewrite monoid M defined on r generators and a sequence $[i_1, \cdots, i_s]$ of integers lying in the range $[1, r]$, construct the word $M.i_1 * M.i_2 * \cdots * M.i_s$.

Example H78E9

We construct the Fibonacci group $F(2, 7)$, and it's identity.

```
> FM<a,A,b,B,c,C,d,D,e,E,f,F,g,G> := FreeMonoid(14);
> Q := quo< FM | a*A=1, A*a=1, b*B=1, B*b=1, c*C=1, C*c=1,
>                d*D=1, D*d=1, e*E=1, E*e=1, f*F=1, F*f=1, g*G=1, G*g=1,
>                a*b=c, b*c=d, c*d=e, d*e=f, e*f=g, f*g=a, g*a=b>;
> M := RWSMonoid(Q : TidyInt := 1000);
> print Id(M);
Id(M)
> print M!1;
Id(M)
> Order(M);
29
```

78.3.4 Arithmetic with Words

Having constructed a rewrite monoid M one can perform arithmetic with words in M. Assuming we have $u, v \in M$ then the product $u * v$ will be computed as follows:

(i) The product $w = u * v$ is formed as a product in the appropriate free monoid.

(ii) The word w is reduced using the reduction machine associated with M.

If M is confluent, then w will be the unique minimal word that represents $u * v$ under the ordering of M. If M is not confluent, then there are some pairs of words which are equal in M, but which reduce to distinct words, and hence w will not be a unique normal form. Note that:

(i) Reduction of w can cause an increase in the length of w. At present there is an internal limit on the length of a word – if this limit is exceeded during reduction an error will be raised. Hence any word operation involving reduction can fail.

(ii) The implementation is designed more with speed of execution in mind than with minimizing space requirements; thus, the reduction machine is always used to carry out word reduction, which can be space-consuming, particularly when the number of generators is large.

`u * v`

Product of the words w and v.

`u ^ n`

The n-th power of the word w, where n is a positive or zero integer.

`u eq v`

Given words w and v belonging to the same monoid, return true if w and v reduce to the same normal form, false otherwise. If M is confluent this tests for equality. If M is non-confluent then two words which are the same may not reduce to the same normal form.

`u ne v`

Given words w and v belonging to the same monoid, return false if w and v reduce to the same normal form, true otherwise. If M is confluent this tests for non-equality. If M is non-confluent then two words which are the same may reduce to different normal forms.

`IsId(w)`

`IsIdentity(w)`

Returns `true` if the word w is the identity word.

`#u`

The length of the word w.

`ElementToSequence(u)`

`Eltseq(u)`

The sequence Q obtained by decomposing the element u of a rewrite monoid into its constituent generators. Suppose u is a word in the rewrite monoid M. If $u = M.i_1 \cdots M.i_m$, then $Q[j] = i_j$, for $j = 1, \ldots, m$.

Example H78E10

We illustrate the word operations by applying them to elements of the Fibonacci monoid $FM(2, 5)$.

```
> FM<a,b,c,d,e> := FreeMonoid(5);
> Q:=quo< FM | a*b=c, b*c=d, c*d=e, d*e=a, e*a=b >;
> M<a,b,c,d,e> := RWSMonoid(Q);
> a*b*c*d;
b^2
> (c*d)^4 eq a;
true
> IsIdentity(a^0);
true
> IsIdentity(b^2*e);
false
```

78.4 Homomorphisms

For a general description of homomorphisms, we refer to Chapter 16. This section describes some special aspects of homomorphisms whose domain is a rewrite monoid.

78.4.1 General Remarks

Monoids in the category MonRWS currently are accepted as codomains only for monoid homomorphisms, whose codomain is a rewrite monoid as well.

78.4.2 Construction of Homomorphisms

`hom< M -> N | S >`

Returns the homomorphism from the rewrite group M to the monoid N defined by the expression S which must be the one of the following:

(i) A list, sequence or indexed set containing the images of the n generators $M.1, \ldots, M.n$ of M. Here, the i-th element of S is interpreted as the image of $M.i$, i.e. the order of the elements in S is important.

(ii) A list, sequence, enumerated set or indexed set, containing n tuples $< x_i, y_i >$ or arrow pairs $x_i -> y_i$, where x_i is a generator of M and $y_i \in N$ $(i = 1, \ldots, n)$ and the set $\{x_1, \ldots, x_n\}$ is the full set of generators of M. In this case, y_i is assigned as the image of x_i, hence the order of the elements in S is not important.

It is the user's responsibility to ensure that the provided generator images actually give rise to a well-defined homomorphism. No checking is performed by the constructor. Presently, N must be either a rewrite monoid or a group, and it is not possible to define a homomorphism by assigning images to the elements of an arbitrary generating set of M.

78.5 Set Operations

`Random(M, n)`

A random word of length at most n in the generators of M.

`Random(M)`

A random word (of length at most the order of M) in the generators of M.

`Representative(M)`

`Rep(M)`

An element chosen from M.

`Set(M, a, b)`

`Search` MonStgElt *Default :* "*DFS*"

Create the set of words, w, in M with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Set(M)`

`Search` MonStgElt *Default :* "*DFS*"

Create the set of words that is the carrier set of M. If `Search` is set to `"DFS"` (depth-first search) then words are enumerated in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words are enumerated in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker. Since the result is a set the words may not appear in the resultant set in the search order specified (although internally they will be enumerated in this order).

`Seq(M, a, b)`

`Search` MonStgElt *Default :* "*DFS*"

Create the sequence S of words, w, in M with $a \leq \text{length}(w) \leq b$. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

`Seq(M)`

`Search` MonStgElt *Default :* "*DFS*"

Create a sequence S of words from the carrier set of M. If `Search` is set to `"DFS"` (depth-first search) then words will appear in S in lexicographical order. If `Search` is set to `"BFS"` (breadth-first-search) then words will appear in S in lexicographical order for each individual length (i.e. in short-lex order). Depth-first-search is marginally quicker.

Example H78E11

We construct the group D_{22}, together with a representative word from the group, a random word and a random word of length at most 5 from the group, and the set of elements of the group.

```
> FM<a,b,c,d,e,f> := FreeMonoid(6);
> Q := quo< FM | a^2=1, f^2=1,
>                d*a=a*c, e*b=b*f, d*c=c*e, d*f=a*d, a*e=e*b, b*f*c=f >;
> M<a,b,c,d,e,f> := RWSMonoid(Q);
```

```
> print Order(M);
22
> print Representative(M);
Id(M)
> print Random(M);
c * e
> print Random(M, 5);
d
> Set(M);
{ a * c, e, a * d, f, a * e, a * f, Id(M), a * c * e, b * a,
  b * d, a * d * b, b * e, c * e, a * b * a, d * b, a * b * d,
  a, a * b * e, b, c, a * b, d }
> Seq(M : Search := "BFS");
[ Id(M), a, b, c, d, e, f, a * b, a * c, a * d, a * e, a * f,
  b * a, b * d, b * e, c * e, d * b, a * b * a, a * b * d,
  a * b * e, a * c * e, a * d * b ]
```

78.6 Conversion to a Finitely Presented Monoid

There is a standard way to convert a rewrite monoid into a finitely presented monoid using the function `Relations`. This is shown in the following example.

Example H78E12

We construct the Fibonacci monoid $FM(2,4)$ as a rewrite monoid, and then convert it into a finitely presented monoid.

```
> FM<a,b,c,d> := FreeMonoid(4);
> Q := quo< FM | a*b=c, b*c=d, c*d=a, d*a=b >;
> M := RWSMonoid(Q);
> Order(M);
11
> P<w,x,y,z> := quo < FM | Relations(M) >;
> P;
Finitely presented monoid
Relations
    w * x = y
    x * y = z
    y * z = w
    z * w = x
    y^2 = w * z
    z^2 = x * w
    z * y = x^2
    y * x = w^2
    x * w * z = x^2
    w^3 = w * z
    x * w^2 = x * z
```

```
x^2 * z = x
w^2 * y = w
x^3 = x * w
x^2 * w = z
z * x = x * z
y * w = w * y
w^2 * z = y
x * w * y = x
```

78.7 Bibliography

[**Hol97**] Derek Holt. *KBMAG – Knuth-Bendix in Monoids and Automatic Groups.* University of Warwick, 1997.

[**Sim94**] Charles C. Sims. *Computation with finitely presented groups.* Cambridge University Press, Cambridge, 1994.

PART XI
ALGEBRAS

79 ALGEBRAS

Chapter 79
ALGEBRAS

79.1 Introduction

Algebras are viewed as free modules over a ring R with an additional multiplication. There are no a priori conditions imposed on the ring except that it must be unital, but some functions may require that an echelonization algorithm is available for modules over R and sometimes it is also required that R is a field. For example, quotients of algebras can only be constructed over fields, since otherwise the quotient module is not necessarily a free module over R.

The most general way to define an algebra is by structure constants, but for special types of algebras MAGMA uses more efficient representations.

79.1.1 The Categories of Algebras

At present, MAGMA contains seven main categories of algebras:

(1) General algebras represented by structure constants: category `AlgGen`;

(2) Associative algebras represented by structure constants: category `AlgAss`;

(3) Quaternion algebras as special types of associative algebras; category `AlgQuat`;

(4) Lie algebras represented by structure constants: category `AlgLie`;

(5) Group algebras: category `AlgGrp` with a special type `AlgGrpSub` for subalgebras of group algebras;

(6) Matrix algebras: category `AlgMat`;

(7) Finitely presented algebras: category `AlgFP`.

The hierarchy of these categories is such that `AlgGen` is on the top level and `AlgAss` and `AlgLie` are on the next level inheriting the functions available for `AlgGen`. The categories `AlgQuat`, `AlgGrp` and `AlgMat` are on a third level inheriting the functions available for `AlgAss`. Finitely presented algebras are independent of the other categories.

79.2 Construction of General Algebras and their Elements

79.2.1 Construction of a General Algebra

The construction of an algebra depends on its category. The chapters on the individual algebra categories describe this in detail. Here only an overview is given.

`Algebra< R, n | Q >`

Let R be ring, n an integer and Q a sequence of n^3 elements of R. This function creates an algebra A of dimension n over R with basis $e_1, \ldots, e_n$ such that Q contains the structure constants of A, i.e. $e_i * e_j = \sum a_{ij}^k e_k$, where a_{ij}^k is the element in position $(i-1) * n^2 + (j-1) * n + k$ of Q.

`AssociativeAlgebra< R, n | Q >`

`Check` BoolElt *Default :* `true`

This function creates the associative structure constant algebra A as returned by `Algebra< R, n | Q >`. By default, the algebra is checked on associativity, but this can be avoided by setting `Check := false`. The returned algebra is of type `AlgAss`.

`QuaternionAlgebra< K | a, b >`

This function creates the quaternion algebra A over the field K on generators x and y with relations $x^2 = a$, $y^2 = b$, and $xy = -yx$.

`LieAlgebra< R, n | Q >`

`Check` BoolElt *Default :* `true`

This function creates the Lie structure constant algebra A as returned by `Algebra< R, n | Q >`. By default, the algebra is checked to be a Lie algebra, but this can be avoided by setting `Check := false`. The returned algebra is of type `AlgLie`.

`LieAlgebra(A)`

Given an associative algebra A, create the Lie algebra generated by the elements in L using the induced Lie product $(x, y) \to x * y - y * x$.

`GroupAlgebra(R, G)`

Given a ring R and a group G construct the group algebra $R[G]$ of dimension $|G|$ over R.

`MatrixAlgebra(R, n)`

Given a positive integer n and a ring R, create the full matrix algebra $M_n(R)$ of dimension n^2 over R.

79.2.2 Construction of an Element of a General Algebra

The construction of a generic element of an algebra varies for the different types of algebras and is therefore explained in the corresponding chapters.

`Zero(A)`
`A ! 0`

Create the zero element of the algebra A.

`One(A)`
`A ! 1`

If it exists, create the identity element of the algebra A; otherwise an error occurs.

`Random(A)`

Given an algebra A defined over a finite ring, return a random element.

79.3 Construction of Subalgebras, Ideals and Quotient Algebras

79.3.1 Subalgebras and Ideals

If the coefficient ring R of an algebra A is a Euclidean domain then one may construct submodules and ideals of A in MAGMA.

`sub< A | L >`

Create the subalgebra S of the algebra A that is generated by the elements defined by L, where L is a list of one or more items of the following types:

(a) An element of A;

(b) A set or sequence of elements of A;

(c) A subalgebra or ideal of A;

(d) A set or sequence of subalgebras or ideals of A.

The constructor returns the subalgebra as an algebra of the same type as A. An exception are group algebras, where the subalgebra is either of type `AlgAss` or of the special type `AlgGrpSub`. As well as the subalgebra S itself, the constructor returns the inclusion homomorphism $f : S \to A$.

`lideal< A | L >`

Create the left ideal I of the algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the left ideal as an algebra of the same type as A with the same exception for group algebras as for the `sub` constructor. As well as the left ideal I itself, the constructor returns the inclusion homomorphism $f : I \to A$.

`rideal< A | L >`

Create the right ideal I of the algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the right ideal as an algebra of the same type as A with the same exception for group algebras as for the `sub` constructor. As well as the right ideal I itself, the constructor returns the inclusion homomorphism $f : I \to A$.

`ideal< A | L >`

Create the (two-sided) ideal I of the algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the right ideal as an algebra of the same type as A with the same exception for group algebras as for the `sub` constructor. As well as the ideal I itself, the constructor returns the inclusion homomorphism $f : I \to A$.

79.3.2 Quotient Algebras

If the coefficient ring R of an algebra A is a field, then quotient algebras of A may also be constructed.

`quo< A | L >`

Create the quotient algebra $Q = A/I$, where I is the two-sided ideal of A generated by the elements of A specified by the list L, which should satisfy the same conditions as for the `sub` constructor above.

The constructor returns the quotient as a structure constant algebra with degree equal to its dimension. If A is known to be associative, then Q is of type `AlgAss`, otherwise Q is of type `AlgGen`. As well as the quotient Q itself, the constructor returns the natural homomorphism $f : A \to Q$.

`A / S`

The quotient of the algebra A by the (two-sided) ideal closure of its subalgebra S.

79.4 Operations on Algebras and Subalgebras

79.4.1 Invariants of an Algebra

`CoefficientRing(A)`

`BaseRing(A)`

The coefficient ring (or base ring) over which the algebra A is defined.

`Dimension(A)`

The dimension of the algebra A.

`#A`

The cardinality of the algebra A if both R and the dimension of A are finite. Note that this cannot be computed if the dimension of A is too large.

79.4.2 Changing Rings

`ChangeRing(A, S)`

Given an algebra A with base ring R, together with a ring S, construct the algebra B with base ring S obtained by coercing the coefficients of elements of A into S, together with the homomorphism from A to B.

This function can not be applied if A is of type `AlgGrpSub`, as the parent structure of elements of A is the full group algebra of which A is a subalgebra.

`ChangeRing(A, S, f)`

Given an algebra A with base ring R, together with a ring S and a map $f : R \to S$, construct the algebra B with base ring S obtained by mapping the coefficients of elements of A into S via f, together with the homomorphism from A to B.

As above, this function can not be applied if A is of type `AlgGrpSub`.

79.4.3 Bases

In general, every algebra comes with a basis, corresponding to its underlying module structure. The only exception for that are group algebras in the `"Terms"` representation, where the dimension of the algebra may be too large to create vectors of that degree.

`BasisElement(A, i)`

`A . i`

The i-th basis element of the algebra A.

`Basis(A)`

The basis of the algebra A, as a sequence of elements of A.

Note that if A is of type `AlgGrpSub` the returned elements will be elements of the full group algebra of which A is a subalgebra.

`IsIndependent(Q)`

Given a sequence Q of elements of the R-algebra A, this functions returns `true` if these elements are linearly independent over R; otherwise `false`.

`ExtendBasis(S, A)`

`ExtendBasis(Q, A)`

Given an algebra A and either a subalgebra S of dimension m of A or a sequence Q of m linearly independent elements of A, return a sequence containing a basis of A such that the first m elements are the basis of S resp. the elements in Q.

79.4.4 Decomposition of an Algebra

An algebra A can be regarded as a (left- or right-) module for itself. If A is defined over a finite field, the machinery to decompose modules over finite fields can be used to investigate the structure of the algebra A.

`CompositionSeries(A)`

Compute a composition series for the algebra A. The function has three return values:

(a) a sequence containing the composition series as an ascending chain of subalgebras such that the successive quotients are irreducible A-modules;

(b) a sequence containing the composition factors as structure constant algebras;

(c) a transformation matrix to a basis compatible with the composition series, that is, the first basis elements form a basis of the first term of the composition series, the next extend these to a basis for the second term etc.

`CompositionFactors(A)`

Compute the composition factors of a composition series for the algebra A. This function returns the same as the second return value of `CompositionSeries` above, but will often be very much quicker.

`MinimalLeftIdeals(A :` *parameters*`)`

`MinimalRightIdeals(A :` *parameters*`)`

`MinimalIdeals(A :` *parameters*`)`

`Limit` RngIntElt *Default :* ∞

Return the minimal left/right/two-sided ideals of A (in non-decreasing size). If `Limit` is set to n, at most n ideals are calculated and the second return value indicates whether all of the ideals were computed.

`MaximalLeftIdeals(A :` *parameters*`)`

`MaximalRightIdeals(A :` *parameters*`)`

`MaximalIdeals(A :` *parameters*`)`

`Limit` RngIntElt *Default :* ∞

Return the maximal left/right/two-sided ideals of A (in non-decreasing size). If `Limit` is set to n, at most n ideals are calculated and the second return value indicates whether all of the ideals were computed.

`JacobsonRadical(A)`

Construct the Jacobson (or nilpotent) radical of A, that is, the intersection of the maximal ideals of A (which is equal to the intersection of the maximal left or right ideals).

`IsSemisimple(A)`

Return `true` if the Jacobson radical of A is trivial; otherwise `false`.

`IsSimple(A)`

Return `true` if A has no non-trivial composition factor; otherwise `false`.

Example H79E1

We create a division algebra of dimension 4 over the rational field.

```
> Q := MatrixAlgebra< Rationals(), 4 |
>    [0,1,0,0, -1,0,0,0, 0,0,0,-1, 0,0,1,0],
>    [0,0,1,0, 0,0,0,1, -1,0,0,0, 0,-1,0,0]>;
> i := Q.1;
> j := Q.2;
> k := i*j;
> Dimension(Q);
4
> MinimalPolynomial( (1+i+j+k)/2 );
$.1^2 - $.1 + 1
```

Hence, the element `(1+i+j+k)/2` is integral. In fact, together with `1`, `i` and `j` it forms a $\mathbf{Z}$-basis of a maximal order in `Q`. We create this maximal order as a structure constant algebra over the integers.

```
> a := [ Q!1, i, j, (1+i+j+k)/2 ];
> T := MatrixAlgebra(Rationals(),4) ! &cat[ Coordinates(Q,a[i]) : i in [1..4] ];
> V := RSpace(Rationals(), 4);
> C := [ V ! Coordinates(Q, a[i]*a[j]) * T^-1 : j in [1..4], i in [1..4] ];
> A := ChangeRing( Algebra< V | C >, Integers() );
> IsAssociative(A);
true
> AA := AssociativeAlgebra(A);
> AA;
Associative Algebra of dimension 4 with base ring Integer Ring
> MinimalPolynomial(AA.4);
$.1^2 - $.1 + 1
```

The so constructed maximal order is ramified at 2 and ∞, hence it should be simple after reducing at odd primes.

```
> for p in [ i : i in [1..100] | IsPrime(i) ] do
>    if not IsSimple( ChangeRing( AA, GF(p) ) ) then
>       print p;
>    end if;
> end for;
2
> CS, CF, T := CompositionSeries( ChangeRing( AA, GF(2) ) );
> T;
[1 0 1 0]
```

```
[0 1 1 0]
[0 0 1 0]
[0 0 0 1]
```

A glance at the preimages of the basis of the irreducible submodule shows the ramification of AA at the prime 2.

```
> MinimalPolynomial(AA.1 + AA.3);
$.1^2 - 2*$.1 + 2
> MinimalPolynomial(AA.2 + AA.3);
$.1^2 + 2
```

79.4.5 Operations on Subalgebras

`IsZero(A)`

Returns `true` if the algebra A is trivial; otherwise `false`.

`A eq B`

Returns `true` if the algebras A and B (having a common superalgebra) are equal; otherwise `false`.

`A ne B`

Returns `true` if the algebras A and B are not equal; otherwise `false`.

`A subset B`

Returns `true` if A is a subalgebra of the algebra B; otherwise `false`.

`A notsubset B`

Returns `true` if A is not a subalgebra of the algebra B; otherwise `false`.

`A meet B`

The intersection of the algebras A and B, which must have a common superalgebra.

`A * B`

The algebra product $A * B$ of the algebras A and B, which must have a common superalgebra.

`A ^ n`

The (left-normed) n-th power of the algebra A, i.e. $((\ldots(A * A) * \ldots) * A)$.

`Morphism(A, B)`

The map giving the morphism from A to B. Either A is a subalgebra of B, in which case the embedding of A into B is returned, or B is a quotient algebra of A, in which case the natural epimorphism from A onto B is returned.

79.5 Operations on Elements of an Algebra

79.5.1 Operations on Elements

`a + b`

The sum of the elements a and b of an algebra A.

`-a`

The negation of the algebra element a.

`a - b`

The difference of the elements a and b of an algebra A.

`a * b`

The product of the elements a and b of an algebra A.

`a * r`
`r * a`

The product of the element a of the algebra A and the ring element $r \in R$, where R is the coefficient ring of A.

`a / r`

The product of the element a of the R-algebra A and the ring element $1/r \in R$, where R is a field.

`a ^ n`

If n is positive, form the (left-normed) n-th power $((\ldots((a * a) * a)\ldots) * a)$ of a; if the parent algebra A of a has an identity and n is zero, return the identity; if $n < 0$ and a has an inverse a^{-1} in A, form the n-th power of a^{-1}.

`MinimalPolynomial(a)`

If R is a field or the integer ring, return the minimal polynomial of the algebra element a.

`Parent(a)`

For an element a in an algebra A return A.

79.5.2 Comparisons and Membership

`a eq b`

Returns `true` if the elements a and b of an algebra A are equal; otherwise `false`.

`a ne b`

Returns `true` if the elements a and b of an algebra A are not equal; otherwise `false`.

`a in A`

Returns `true` if a is in the algebra A; otherwise `false`.

`a notin A`

Returns `true` if a is not in the algebra A; otherwise `false`.

79.5.3 Predicates on Elements

`IsZero(a)`

Returns `true` if the algebra element a is zero; otherwise `false`.

`IsOne(a)`

Returns `true` if the algebra element a is the identity; otherwise `false`.

`IsMinusOne(a)`

Returns `true` if the algebra element a is the negative of the identity element; otherwise `false`.

`IsUnit(a)`

Returns `true` if the element a is a unit, plus the inverse; otherwise `false`.

`IsRegular(a)`

Returns `true` if the element a is regular, that is, is not a zero divisor; otherwise `false`.

`IsZeroDivisor(a)`

Returns `true` if the algebra element a is a divisor of zero; otherwise `false`.

`IsIdempotent(a)`

Returns `true` if the element a is an idempotent, i.e. if $a^2 = a$; otherwise `false`.

`IsNilpotent(a)`

Returns `true` if the element a is nilpotent, i.e. if $a^n = 0$ for some $n \geq 0$; otherwise `false`. If `true`, the minimal n such that $a^n = 0$ is returned as a second value.

80 STRUCTURE CONSTANT ALGEBRAS

Chapter 80
STRUCTURE CONSTANT ALGEBRAS

80.1 Introduction

A structure constant algebra A of dimension n over a ring R can be defined in MAGMA by giving the n^3 *structure constants* $a_{ij}^k \in R (1 \leq i, j, k \leq n)$ such that, if $e_1, e_2, \ldots, e_n$ is the basis of A, $e_i * e_j = \sum_{k=1}^{n} a_{ij}^k * e_k$. Structure constant algebras may be defined over any unital ring R. However, many operations require that R be a Euclidean domain or even a field.

80.2 Construction of Structure Constant Algebras and Elements

80.2.1 Construction of a Structure Constant Algebra

There are three ways in MAGMA to specify the structure constants for a structure constant algebra A of dimension n. The first is to give n^3 ring elements, the second to identify A with the module $M = R^n$ and give the products $e_i * e_j$ as elements of M and the third to specify only the non-zero structure constants.

`Algebra< R, n | Q : parameters >`

`Algebra< M | Q : parameters >`

`Rep` MONSTGELT *Default : "Dense"*

This function creates the structure constant algebra A over the free module $M = R^n$, with standard basis $e_1, e_2, \ldots, e_n$, and with the structure constants a_{ij}^k being given by the sequence Q. The sequence Q can be of any of the following three forms. Note that in all cases the actual ordering of the structure constants is the same: it is only their division that varies.

(i) A sequence of n sequences of n sequences of length n. The j-th element of the i-th sequence is the sequence $[a_{ij}^1, \ldots, a_{ij}^n]$, or the element $(a_{ij}^1, \ldots, a_{ij}^n)$ of M, giving the coefficients of the product $e_i * e_j$.

(ii) A sequence of n^2 sequences of length n, or n^2 elements of M. Here the coefficients of $e_i * e_j$ are given by position $(i-1) * n + j$ of Q.

(iii) A sequence of n^3 elements of the ring R. Here the sequence elements are the structure constants themselves, with the ordering $a_{11}^1, a_{11}^2, \ldots, a_{11}^n, a_{12}^1, a_{12}^2, \ldots, a_{nn}^n$. So a_{ij}^k lies in position $(i-1) * n^2 + (j-1) * n + k$ of Q.

The optional parameter `Rep` can be used to select the internal representation of the structure constants. The possible values for `Rep` are `"Dense"`, `"Sparse"` and `"Partial"`, with the default being `"Dense"`. In the dense format, the n^3 structure

constants are stored as n^2 vectors of length n, similarly to (ii) above. This is the best representation if most of the structure constants are non-zero. The sparse format, intended for use when most structure constants are zero, stores the positions and values of the non-zero structure constants. The partial format stores the vectors, but records for efficiency the positions of the non-zero structure constants.

`Algebra< R, n | T : parameters >`

`Rep` MonStgElt *Default : "Sparse"*

This function creates the structure constant algebra A with standard basis $e_1, e_2, \ldots, e_n$ over R. The sequence T contains quadruples $< i, j, k, a_{ij}^k >$ giving the non-zero structure constants. All other structure constants are defined to be 0.

As above, the optional parameter `Rep` can be used to select the internal representation of the structure constants.

`ChangeBasis(A, B)`

`ChangeBasis(A, B)`

`ChangeBasis(A, B)`

`Rep` MonStgElt *Default : "Dense"*

Create a new structure constant algebra A', isomorphic to A, by recomputing the structure constants with respect to the basis B. The basis B can be specified as a set or sequence of elements of A, a set or sequence of vectors, or a matrix. The second returned value is the isomorphism from A to A'.

As above, the optional parameter `Rep` can be used to select the internal representation of the structure constants. Note that the default is dense representation, regardless of the representation used by A.

80.2.2 Construction of Elements of a Structure Constant Algebra

`elt< A | r_1, r_2, ..., r_n >`

Given a structure constant algebra A of dimension n over a ring R, and ring elements $r_1, r_2, \ldots, r_n \in R$ construct the element $r_1 * e_1 + r_2 * e_2 + \ldots + r_n * e_n$ of A.

`A ! Q`

Given a structure constant algebra A of dimension n and a sequence $Q = [r_1, r_2, \ldots, r_n]$ of elements of the base ring R of A, construct the element $r_1 * e_1 + r_2 * e_2 + \ldots + r_n * e_n$ of A.

`BasisProduct(A, i, j)`

Return the product of the i-th and j-th basis element of the algebra A.

`BasisProducts(A)`

`Rep` MonStgElt *Default : "Dense"*

Return the products of all basis elements of the algebra A.

The optional parameter `Rep` may be used to specify the format of the result. If `Rep` is set to "Dense", the products are returned as a sequence Q of n sequences of n elements of A, where n is the dimension of A. The element $Q[i][j]$ is the product of the i-th and j-th basis elements.

If `Rep` is set to "Sparse", the products are returned as a sequence Q containing quadruples (i, j, k, a_{ijk}) signifying that the product of the i-th and j-th basis elements is $\sum_{k=1}^{n} a_{ijk} b_k$, where b_k is the k-th basis element and $n = \dim(A)$.

80.3 Operations on Structure Constant Algebras and Elements

80.3.1 Operations on Structure Constant Algebras

`IsCommutative(A)`

Returns `true` if the algebra A is commutative; otherwise `false`.

`IsAssociative(A)`

Returns `true` if the algebra A is associative; otherwise `false`.

Note that for a structure constant algebra of dimension n this requires up to n^3 tests.

`IsLie(A)`

Returns `true` if the algebra A is a Lie algebra; otherwise `false`.

Note that for a structure constant algebra of dimension n this requires about $n^3/3$ tests of the Jacobi identity.

`DirectSum(A, B)`

Construct a structure constant algebra of dimension $n + m$ where n and m are the dimensions of the algebras A and B, respectively. The basis of the new algebra is the concatenation of the bases of A and B and the products $a * b$ where $a \in A$ and $b \in B$ are defined to be 0.

Example H80E1

We define a structure constant algebra which is a Jordan algebra.

```
> M := MatrixAlgebra( GF(3), 2 );
> B := Basis(M);
> C := &cat[Coordinates(M,(B[i]*B[j]+B[j]*B[i])/2) : j in [1..#B], i in [1..#B]];
> A := Algebra< GF(3), #B | C >;
> #A;
81
> IsAssociative(A);
false
> IsLie(A);
false
> IsCommutative(A);
true
```

This is a good start, as one of the defining properties of Jordan algebras is that they are commutative. The other property is that the identity $(x^2 * y) * x = x^2 * (y * x)$ holds for all $x, y \in A$. We check this on a random pair.

```
> x := Random(A); y := Random(A); print (x^2*y)*x - x^2*(y*x);
(0 0 0 0)
```

The algebra is small enough to check this identity on all elements.

```
> forall{<x, y>: x, y in A | (x^2*y)*x eq x^2*(y*x)};
true
```

So the algebra is in fact a Jordan algebra (which was clear by construction). We finally have a look at the structure constants.

```
> BasisProducts(A);
[
    [ (1 0 0 0), (0 2 0 0), (0 0 2 0), (0 0 0 0) ],
    [ (0 2 0 0), (0 0 0 0), (2 0 0 2), (0 2 0 0) ],
    [ (0 0 2 0), (2 0 0 2), (0 0 0 0), (0 0 2 0) ],
    [ (0 0 0 0), (0 2 0 0), (0 0 2 0), (0 0 0 1) ]
]
```

80.3.2 Indexing Elements

`a[i]`

If a is an element of a structure constant algebra A of dimension n and $1 \leq i \leq n$ is a positive integer, then the i-th component of the element a is returned (as an element of the base ring R of A).

`a[i] := r`

Given an element a belonging to a structure constant algebra of dimension n over R, a positive integer $1 \leq i \leq n$ and an element $r \in R$, the i-th component of the element a is redefined to be r.

80.3.3 The Module Structure of a Structure Constant Algebra

`Module(A)`

The module R^n underlying the structure constant algebra A.

`Degree(A)`

The degree (= dimension) of the module underlying the algebra A.

`Degree(a)`

Given an element belonging to the structure constant algebra A of dimension n, return n.

`ElementToSequence(a)`
`Eltseq(a)`

The sequence of coefficients of the structure constant algebra element a.

`Coordinates(S, a)`

Let a be an element of a structure constant algebra A and let S be a subalgebra of A containing a. This function returns the coefficients of a with respect to the basis of S.

`InnerProduct(a, b)`

The (Euclidean) inner product of the coefficient vectors of a and b, where a and b are elements of some structure constant algebra A.

`Support(a)`

The support of the structure constant algebra element a; i.e. the set of indices of the non-zero components of a.

80.3.4 Homomorphisms

`hom< A -> B | Q >`

Given a structure constant algebra A of dimension n over R and either a structure constant algebra B over R or a module B over R, construct the homomorphism from A to B specified by Q. The sequence Q may be of the form $[b_1, \ldots, b_n]$, $b_i \in B$, indicating that the i-th basis element of A is mapped to b_1 or of the form $[< a_1, b_1 >, \ldots, < a_n, b_n >]$ indicating that a_i maps to b_i, where the $a_i (1 \leq i \leq n)$ must form a basis of A.

Note that this is in general only a module homomorphism, it is not checked whether it is an algebra homomorphism.

Example H80E2

We construct the real Cayley algebra, which is a non-associative algebra of dimension 8, containing 7 quaternion algebras. If the basis elements are labelled $1, \ldots, 8$ and 1 corresponds to the identity, these quaternion algebras are spanned by $\{1, (n+1) \bmod 7+2, (n+2) \bmod 7+2, (n+4) \bmod 7+4\}$, where $0 \leq n \leq 6$. We first define a function, which, given three indices i, j, k constructs a sequence with the structure constants for the quaternion algebra spanned by $1, i, j, k$ in the quadruple notation.

```
> quat := func<i,j,k | [<1,1,1, 1>, <i,i,1, -1>, <j,j,1, -1>, <k,k,1, -1>,
>   <1,i,i, 1>, <i,1,i, 1>, <1,j,j, 1>, <j,1,j, 1>, <1,k,k, 1>, <k,1,k, 1>,
>   <i,j,k, 1>, <j,i,k, -1>, <j,k,i, 1>, <k,j,i, -1>, <k,i,j, 1>, <i,k,j, -1>]>;
```

We now define the sequence of non-zero structure constants for the Cayley algebra using the function `quat`. Some structure constants are defined more than once and we have to get rid of these when defining the algebra.

```
> con := &cat[quat((n+1) mod 7 +2, (n+2) mod 7 +2, (n+4) mod 7 +2):n in [0..6]];
> C := Algebra< Rationals(), 8 | Setseq(Set(con)) >;
> C;
Algebra of dimension 8 with base ring Rational Field
> IsAssociative(C);
false
> IsAssociative( sub< C | C.1, C.2, C.3, C.5 > );
true
```

The integral elements in this algebra are those where either all coefficients are integral or exactly 4 coefficients lie in $1/2 + \mathbf{Z}$ in positions i_1, i_2, i_3, i_4, such that i_1, i_2, i_3, i_4 are a basis of one of the 7 quaternion algebras or a complement of such a basis. These elements are called the integral Cayley numbers and form a $\mathbf{Z}$-algebra. The units in this algebra are the elements with either one entry ± 1 and the others 0 or with 4 entries $\pm 1/2$ and 4 entries 0, where the non-zero entries are in the positions as described above. This gives 240 units and they form (after rescaling with $\sqrt{2}$) the roots in the root lattice of type E_8.

```
> a := (C.1 - C.2 + C.3 - C.5) / 2;
> MinimalPolynomial(a);
$.1^2 - $.1 + 1
> MinimalPolynomial(a^-1);
$.1^2 - $.1 + 1
> MinimalPolynomial(C.2+C.3);
$.1^2 + 2
> MinimalPolynomial((C.2+C.3)^-1);
$.1^2 + 1/2
```

Tensoring the integral Cayley algebra with a finite field gives a finite Cayley algebra. As the $\mathbf{Z}$-algebra generated by the chosen basis for C has index 2^4 in the full integral Cayley algebra, we can get the finite Cayley algebras by applying the `ChangeRing` function for finite fields of odd characteristic. The Cayley algebra over $GF(q)$ has the simple group $G_2(q)$ as its automorphism group. Since the identity has to be fixed, every automorphism is determined by its image on the remaining 7 basis elements. Each of these has minimal polynomial $x^2 + 1$, hence one obtains a

permutation representation of $G_2(q)$ on the elements with this minimal polynomial. As $\pm$-pairs have to be preserved, this number can be divided by 2.

```
> C3 := ChangeRing( C, GF(3) );
> f := MinimalPolynomial(C3.2);
> f;
$.1^2 + 1
> #C3;
6561
> time Im := [ c : c in C3 | MinimalPolynomial(c) eq f ];
Time: 3.099
> #Im;
702
> C5 := ChangeRing( C, GF(5) );
> f := MinimalPolynomial(C5.2);
> f;
$.1^2 + 1
> #C5;
390625
> time Im := [ c : c in C5 | MinimalPolynomial(c) eq f ];
Time: 238.620
> #Im;
15750
```

In the case of the Cayley algebra over $GF(3)$ we obtain a permutation representation of degree 351, which is in fact the smallest possible degree (corresponding to the representation on the cosets of the largest maximal subgroup $U_3(3) : 2$). Over $GF(5)$, the permutation representation is of degree 7875, corresponding to the maximal subgroup $L_3(5) : 2$, the smallest possible degree being 3906.

81 ASSOCIATIVE ALGEBRAS

Chapter 81
ASSOCIATIVE ALGEBRAS

81.1 Introduction

Defining an algebra by structure constants gives a very general set-up, but many structural concepts are restricted to associative algebras. Therefore, MAGMA provides a special type for structure constant algebras which are known to be associative.

81.2 Construction of Associative Algebras

81.2.1 Construction of an Associative Structure Constant Algebra

The construction of an associative structure constant algebra is identical to that of a general structure constant algebra, with the exception that an additional parameter is provided which may be used to avoid checking that the algebra is associative.

`AssociativeAlgebra< R, n | Q : parameters >`

`AssociativeAlgebra< M | Q : parameters >`

`Check`	BOOLELT	*Default :* `true`
`Rep`	MONSTGELT	*Default :* "*Dense*"

This function creates the associative structure constant algebra A over the free module $M = R^n$, with standard basis $e_1, e_2, \ldots, e_n$, and with the structure constants a_{ij}^k being given by the sequence Q. The sequence Q can be of any of the following three forms. Note that in all cases the actual ordering of the structure constants is the same: it is only their division that varies.

(i) A sequence of n sequences of n sequences of length n. The j-th element of the i-th sequence is the sequence $[a_{ij}^1, \ldots, a_{ij}^n]$, or the element $(a_{ij}^1, \ldots, a_{ij}^n)$ of M, giving the coefficients of the product $e_i * e_j$.

(ii) A sequence of n^2 sequences of length n, or n^2 elements of M. Here the coefficients of $e_i * e_j$ are given by position $(i-1) * n + j$ of Q.

(iii) A sequence of n^3 elements of the ring R. Here the sequence elements are the structure constants themselves, $a_{11}^1, a_{11}^2, \ldots, a_{11}^n, a_{12}^1, a_{12}^2, \ldots, a_{nn}^n$. So a_{ij}^k lies in position $(i-1) * n^2 + (j-1) * n + k$ of Q.

By default the algebra is checked to be associative; this can be overruled by setting the parameter `Check` to `false`.

The optional parameter `Rep` can be used to select the internal representation of the structure constants. The possible values for `Rep` are `"Dense"`, `"Sparse"` and `"Partial"`, with the default being `"Dense"`. In the dense format, the n^3 structure

constants are stored as n^2 vectors of length n, similarly to (ii) above. This is the best representation if most of the structure constants are non-zero. The sparse format, intended for use when most structure constants are zero, stores the positions and values of the non-zero structure constants. The partial format stores the vectors, but records for efficiency the positions of the non-zero structure constants.

`AssociativeAlgebra< R, n | T : parameters >`

`Check`	BoolElt	*Default :* `true`
`Rep`	MonStgElt	*Default :* "*Sparse*"

This function creates the associative structure constant algebra A with standard basis $e_1, e_2, \ldots, e_n$ over R. The sequence T contains quadruples $< i, j, k, a_{ij}^k >$ giving the non-zero structure constants. All other structure constants are defined to be 0.

The optional parameters are as above.

`AssociativeAlgebra(A)`

Given a structure constant algebra A of type `AlgGen`, construct an isomorphic associative structure constant algebra of type `AlgAss`. If it is not known whether or not A is associative, this will be checked and an error occurs if it is not. The elements of the resulting algebra can be coerced into A and vice versa.

`ChangeBasis(A, B)`

`ChangeBasis(A, B)`

`ChangeBasis(A, B)`

`Rep`	MonStgElt	*Default :* "*Dense*"

Create a new associative structure constant algebra A', isomorphic to A, by recomputing the structure constants with respect to the basis B. The basis B can be specified as a set or sequence of elements of A, a set or sequence of vectors, or a matrix. The second returned value is the isomorphism from A to A'.

As above, the optional parameter `Rep` can be used to select the internal representation of the structure constants. Note that the default is dense representation, regardless of the representation used by A.

81.2.2 Associative Structure Constant Algebras from other Algebras

`Algebra(A)`

If A is either a group algebra of type `AlgGrp` given in vector representation or a matrix algebra of type `AlgMat`, construct the associative structure constant algebra B isomorphic to A together with the isomorphism $A \to B$.

`Algebra(F, E)`

Let E and F be either finite fields or algebraic number fields such that E is a subfield of F. This function returns the associative algebra A of dimension $[F : E]$ over E which is isomorphic to F, together with the isomorphism from F to A such that the $(i-1)$-th power of the generator of F over E is mapped to the i-th basis vector of A.

`AlgebraOverCenter(A)`

Given a simple algebra A of type `AlgMat` or `AlgAss` with center K, this function returns a K-algebra B which is K-isomorphic to A as well as an isomorphism from A to B.

81.3 Operations on Algebras and their Elements

81.3.1 Operations on Algebras

`Centre(A)`

The centre of the associative algebra A.

`Centralizer(A, S)`

`Centraliser(A, S)`

The centralizer of the subalgebra S of the associative algebra A, that is, the subalgebra of A commuting elementwise with S.

`Idealizer(A, B: parameters)`

`Idealiser(A, B: parameters)`

`Side` MonStgElt *Default : "Both"*

Given an associative algebra A and a subalgebra B of A, compute the idealizer of B in A, that is, the largest subalgebra of A in which B is an ideal. By default the two-sided idealizer, that is, the largest subalgebra in which B is a two-sided ideal, is found; the left- or right-idealizer can be found by setting the parameter `Side` to `"Left"` or `"Right"` respectively.

`LieAlgebra(A)`

For an associative structure constant algebra A, return the structure constant algebra L with product given by the Lie bracket $(a, b) \mapsto a * b - b * a$. As a second value the map identifying the elements of A and L is returned.

`CommutatorModule(A, B)`

Let A and B be subalgebras of an associative algebra with underlying module M. This function returns the submodule of M which is spanned by the elements $[a, b] = a * b - b * a$, $a \in A, b \in B$.

`CommutatorIdeal(A, B)`

For two subalgebras A and B of an associative algebra, return the ideal generated by all $[a, b] = a * b - b * a$, $a \in A, b \in B$.

`LeftAnnihilator(A, B)`

For two subalgebras A and B of an associative algebra, return the left annihilator of B in A; that is, the subalgebra of A consisting of all elements a such that $a * b = 0$ for all $b \in B$.

`RightAnnihilator(A, B)`

For two subalgebras A and B of an associative algebra, return the right annihilator of B in A; that is, the subalgebra of A consisting of all elements a such that $b * a = 0$ for all $b \in B$.

Example H81E1

We create the Lie algebra $sl_3(\mathbf{Q})$ as a structure constant algebra. First, we construct $gl_3(\mathbf{Q})$ from the full matrix algebra $M_3(\mathbf{Q})$ and get $sl_3(\mathbf{Q})$ as the derived algebra of $gl_3(\mathbf{Q})$.

```
> gl3 := LieAlgebra(Algebra(MatrixRing(Rationals(), 3)));
> sl3 := gl3 * gl3;
> sl3;
Lie Algebra of dimension 8 with base ring Rational Field
```

Let's see how the first basis element acts.

```
> for i in [1..8] do
>     print sl3.i * sl3.1;
> end for;
(0 0 0 0 0 0 0 0)
( 0 -1  0  0  0  0  0  0)
( 0  0 -2  0  0  0  0  0)
(0 0 0 1 0 0 0 0)
(0 0 0 0 0 0 0 0)
( 0  0  0  0  0 -1  0  0)
(0 0 0 0 0 0 2 0)
(0 0 0 0 0 0 0 1)
```

Since it acts diagonally, this element lies in a Cartan subalgebra. The next candidate seems to be the fifth basis element.

```
> for i in [1..8] do
>     print sl3.i * sl3.5;
> end for;
(0 0 0 0 0 0 0 0)
(0 1 0 0 0 0 0 0)
( 0  0 -1  0  0  0  0  0)
( 0  0  0 -1  0  0  0  0)
(0 0 0 0 0 0 0 0)
```

```
( 0  0  0  0  0 -2  0  0)
(0 0 0 0 0 0 1 0)
(0 0 0 0 0 0 0 2)
```

This also acts diagonally and commutes with `sl3.1`, hence we have luckily found a full Cartan algebra in $sl_3(\mathbf{Q})$. We can now easily work out the root system. Obviously the root spaces correspond to the pairs `(sl3.2, sl3.4)`, `(sl3.3, sl3.7)` and `(sl3.6, sl3.8)`. The product of a positive root with its negative should lie in the Cartan algebra.

```
> sl3.2*sl3.4;
( 1  0  0  0 -1  0  0  0)
> sl3.3*sl3.7;
(1 0 0 0 0 0 0 0)
> sl3.6*sl3.8;
(0 0 0 0 1 0 0 0)
```

Clearly some choices have to be made and we fix `sl3.3` as the element e_α corresponding to the first fundamental root α, `sl3.7` as $e_{-\alpha}$ and get `sl3.1` as $h_\alpha = e_\alpha * e_{-\alpha}$. For the other fundamental root β we have to find an element e_β such that $e_\alpha * e_\beta$ is non-zero.

```
> sl3.3*sl3.2;
(0 0 0 0 0 0 0 0)
> sl3.3*sl3.4;
( 0  0  0  0  0 -1  0  0)
> sl3.3*sl3.6;
(0 0 0 0 0 0 0 0)
> sl3.3*sl3.8;
(0 1 0 0 0 0 0 0)
```

We choose `sl3.8` as e_β, `sl3.6` as $e_{-\beta}$ and consequently `-sl3.5` as h_β. This now determines $e_{\alpha+\beta}$ to be `sl3.2` and $e_{-\alpha-\beta}$ to be `sl3.4`.

81.3.2 Operations on Elements

`Centralizer(A, s)`

`Centraliser(A, s)`

The centralizer of the element s of the associative algebra A, that is, the subalgebra of A commuting with s.

`LieBracket(a, b)`

`(a, b)`

The Lie bracket $a * b - b * a$ of a and b, where a and b are elements of an associative algebra A.

`IsScalar(a)`

Returns `true` (and a coerced to F) iff a belongs to the base ring F of its parent algebra.

`RepresentationMatrix(a, M : parameters)`

`Side` MonStgElt *Default :* "*Right*"

Returns the matrix representation of `Side`-multiplication by the element a in the associative algebra A (which must have 1) on the A-module M.

81.3.3 Representations

`MatrixAlgebra(A)`

For an associative algebra A of dimension n return an isomorphic matrix algebra. If A contains the identity-element, the matrix algebra will be of degree n, otherwise it will be of degree $n + 1$.

`MatrixAlgebra(A, M : parameters)`

`Side` MonStgElt *Default :* "*Right*"

Given a finite-dimensional R-algebra A and a `Side` A-module M (both free as R-modules), return the matrix algebra of A-endomorphisms of M, and the R-algebra homomorphism from A into this endomorphism ring.

`RegularRepresentation(A : parameters)`

`Side` MonStgElt *Default :* "*Right*"

For an associative algebra A of dimension n over R return its regular representation. If $B = (e_1, e_2, \ldots, e_n)$ is the stored basis for A, an element $a \in A$ is mapped to the matrix in $R^{n\times n}$ which has as its i-th row the coordinates of $e_i * a$ with respect to B. As a second map, the homomorphism of A onto the regular representation is returned.

By default, the right-regular representation is computed. This can be changed to the left-regular representation (in which the i-th row of the image of a contains the coordinates of $a * e_i$) by setting the parameter `Side` to `"Left"`.

81.3.4 Decomposition of an Algebra

This section describes a few functions that can be used to obtain information on the structure of a finite-dimensional associative algebra.

`JacobsonRadical(A)`

`Al` MonStgElt *Default :* "*Default*"

This returns the largest nilpotent ideal of A. This function works for finite-dimensional associative algebras defined over a field of characteristic 0, or over a finite field.

The algorithm used by default is taken from [CIW97]. The meataxe algorithm can be used by setting `Al := "Meataxe"`.

Example H81E2

We compute the Jacobson radical of the group algebra over the field of three elements of a 3-group. In that case it is equal to the augmentation ideal.

```
> G:= SmallGroup( 27, 5 );
> A:= GroupAlgebra( GF(3), G );
> JacobsonRadical( A );
Ideal of dimension 26 of the group algebra A
```

`DirectSumDecomposition(A)`

`IndecomposableSummands(A)`

Given an associative algebra A, return the direct sum decomposition of L as a sequence of ideals of L whose sum is L and each of which cannot be further decomposed into a direct sum of ideals. The second sequence return contains the corresponding primitive central idempotents.

For a description of the algorithm we refer to [EG96].

`CentralIdempotents(A)`

Let Z be the centre of the associative algebra A, and let $J(Z)$ denote its Jacobson radical. This function returns a sequence of primitive orthogonal idempotents in Z such that their images in $Z/J(Z)$ span $J(Z)$. Each such idempotent generates a two-sided ideal in A. The second return value is the sequence of these ideals.

In particular, if A is a semisimple algebra, then this function returns a sequence of primitive orthogonal idempotents spanning Z. Furthermore, the ideals in the second sequence returned are simple algebras, and their direct sum equals A.

For a description of the algorithm we refer to [EG96].

Example H81E3

We compute the direct sum decomposition of a group algebra.

```
> G:= SmallGroup( 10, 2 );
> A:= GroupAlgebra( Rationals(), G );
> ee, II:= CentralIdempotents( A );
> ee[1];
1/10*Id(G) + 1/10*G.2 + 1/10*G.2^2 + 1/10*G.2^3 + 1/10*G.2^4 + 1/10*G.1 +
1/10*G.1 * G.2 + 1/10*G.1 * G.2^2 + 1/10*G.1 * G.2^3 + 1/10*G.1 * G.2^4
> II;
[
    Ideal of dimension 1 of the group algebra A
    Basis:
        Id(G) + G.2 + G.2^2 + G.2^3 + G.2^4 + G.1 + G.1 * G.2 + G.1 * G.2^2 +
        G.1 * G.2^3 + G.1 * G.2^4,
    Ideal of dimension 1 of the group algebra A
    Basis:
```

```
        Id(G) + G.2 + G.2^2 + G.2^3 + G.2^4 - G.1 - G.1 * G.2 - G.1 * G.2^2 -
        G.1 * G.2^3 - G.1 * G.2^4,
    Ideal of dimension 4 of the group algebra A
    Basis:
        Id(G) - G.2^4 + G.1 - G.1 * G.2^4
        G.2 - G.2^4 + G.1 * G.2 - G.1 * G.2^4
        G.2^2 - G.2^4 + G.1 * G.2^2 - G.1 * G.2^4
        G.2^3 - G.2^4 + G.1 * G.2^3 - G.1 * G.2^4,
    Ideal of dimension 4 of the group algebra A
    Basis:
        Id(G) - G.2^4 - G.1 + G.1 * G.2^4
        G.2 - G.2^4 - G.1 * G.2 + G.1 * G.2^4
        G.2^2 - G.2^4 - G.1 * G.2^2 + G.1 * G.2^4
        G.2^3 - G.2^4 - G.1 * G.2^3 + G.1 * G.2^4
]
```

We see that here the group algebra is the direct sum of two 1-dimensional and two 4-dimensional ideals. The first idempotent is the sum over all group elements divided by the group order.

81.4 Orders

Let F be a number field with ring of integers R, and let A be a associative algebra over F (finite-dimensional, with 1). An *associative order* O of A is a subring $O \subset A$ which is a projective R-module such that $O \cdot F = A$. We will also refer to an associative order simply as an *order*.

In MAGMA, associative orders have the type `AlgAssVOrd`, and may be declared for any associative algebra of type `AlgAssV`, namely, `AlgAss`, `AlgMat`, `AlgQuat` and `AlgGrp`. Orders have ideals of type `AlgAssVOrdIdl`, and elements of type `AlgAssVOrdElt`. In the special case where A is a quaternion algebra over the rationals, A has type `AlgQuat` and orders in A have type `AlgQuatOrd`.

Orders, like modules over Dedekind domains, are represented by a pseudobasis, see Section 55.10. Currently, only basic arithmetic functions and procedures are available for general associative orders. Most of the nontrivial functionality currently available is designed for orders in quaternion algebras (over the rationals or number fields). The specialised functions for quaternionic orders are described in Chapter 86.

IMPORTANT WARNING for algebras over the rationals: In MAGMA, the rationals are *not* considered to be a number field (the type `FldRat` is not a subtype of `FldNum`). Currently, much of the functionality here is designed primarily for algebras whose base field is a `FldNum` (while some of it, but not all, also works for algebras over the `FldRat`). To compute with algebras over $\mathbf{Q}$, in many cases the best solution is to create $\mathbf{Q}$ as a number field at the outset, using `RationalsAsNumberField()`, and create the algebra over this field instead of `Rationals()`.

81.4.1 Creation of Orders

`Order(R, S)`

Given a ring R and sequence S of elements of an associative algebra A, returns the order of A generated freely over R by the sequence S. The ring R must be a number ring or $\mathbf{Z}$.

`Order(S)`

Given a sequence of elements S of an associative algebra A, returns the order of A generated by the sequence S. The algebra A must be defined over a number field F.

`Order(S, I)`

Given a sequence of elements S of an associative algebra A and a sequence I of ideals of a number ring R, returns the order of A generated by the sequence S with coefficient ideals I. The algebra A must be defined over a number field F and have ring of integers R.

`Order(A, m, I)`

Given an associative algebra A, a matrix m, and a sequence I of ideals of a number ring R, returns the order of A generated by the sequence of elements specified by the rows of m in the basis of A with coefficient ideals I. The algebra A must be defined over a number field F and have ring of integers R.

`Order(A, pm)`

Given an associative algebra A and a pseudomatrix pm, returns the order of A specified by the pseudomatrix pm. The basis of the order is specified by the rows of pm which have coefficients with respect to the basis of A. The algebra A must be defined over a number field with ring of integers R which is the base ring of the pseudomatrix pm.

Example H81E4

We begin by illustrating three methods for creating an associative order.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> A<alpha,beta,alphabeta> := QuaternionAlgebra<F | -3,b>;
```

First type of constructor takes an algebra, a matrix representing the basis elements, and coefficient ideals.

```
> M := MatrixAlgebra(F,4) ! 1;
> I := [ideal<Z_F | 1> : i in [1..4]];
> O := Order(A, M, I);
> O;
Order of Quaternion Algebra with base ring Field of Fractions of Z_F
```

```
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
```

The second type takes an algebra and a pseudomatrix.

```
> P := PseudoMatrix(I, M);
> O := Order(A, P);
> O;
Order of Quaternion Algebra with base ring Field of Fractions of Z_F
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
```

The third takes simply a sequence of elements.

```
> O := Order([alpha,beta]);
> O;
Order of Quaternion Algebra with base ring Field of Fractions of Z_F
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
```

Example H81E5

Here we give two other examples of order creation.

```
> F<w> := CyclotomicField(3);
> A := FPAlgebra<F, x,y | x^3-3, y^3+5, y*x-w*x*y>;
> Aass, f := Algebra(A);
> Aass;
Associative Algebra of dimension 9 with base ring F
> f;
Mapping from: AlgFP: A to AlgAss: Aass
> S := [f(A.i) : i in [1..2]];
> S;
[ (0 0 1 0 0 0 0 0 0), (0 1 0 0 0 0 0 0 0) ]
> O := Order(S);
> O;
Order of Associative Algebra of dimension 9 with base ring Field of Fractions of R
with coefficient ring Maximal Equation Order with defining polynomial x^2 + x +
    1 over its ground order
>
> A := GroupAlgebra(F, DihedralGroup(6));
> Aass := Algebra(A);
> O := Order([g : g in Generators(Aass)]);
> O;
Order of Associative Algebra of dimension 12 with base ring Field of Fractions
of R with coefficient ring Maximal Equation Order with defining polynomial
x^2 + x + 1 over its ground order
```

`MaximalOrder(A)`

Computes a maximal **Z**-order in the semisimple associative algebra A, which must be defined over the rational numbers. The algorithm can be found in [Fri00], §3.5. We refer to [IR93] for a very similar approach.

Example H81E6

First we define two 9×9-matrices that generate a 9-dimensional associative algebra. We check that its Jacobson radical is zero, and then we compute a maximal order.

```
> a1 := Matrix( [
> [-184174/80137, -325/80137, 71/2163699, 0, 0, 0, 0, 0, 0],
> [17713719/80137, 92087/80137, -325/80137, 0, 0, 0, 0, 0, 0],
> [-2189265975/80137, -16429806/80137, 92087/80137, 0, 0, 0, 0, 0, 0],
> [0, 0, 0, 64850/80137, 1472/240411, -25/2163699, 0, 0, 0],
> [0, 0, 0, -6237225/80137, -32425/80137, 1472/240411, 0, 0, 0],
> [0, 0, 0, 3305230272/80137, 45310743/80137, -32425/80137, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, 119324/80137, -497/240411, -46/2163699],
> [0, 0, 0, 0, 0, 0, -11476494/80137, -59662/80137, -497/240411],
> [0, 0, 0, 0, 0, 0, -1115964297/80137, -28880937/80137, -59662/80137] ] );
> a2:= Matrix( [
> [0, 0, 0, 282469/240411, 956/2163699, -26/2163699, 0, 0, 0],
> [0, 0, 0, -6486714/80137, -21029/240411, 956/2163699, 0, 0, 0],
> [0, 0, 0, 238511484/80137, -2766918/80137, -21029/240411, 0, 0, 0],
> [0, 0, 0, 0, 0, 0, -85879/240411, -4894/2163699, 64/6491097],
> [0, 0, 0, 0, 0, 0, 5322432/80137, 163145/240411, -4894/2163699],
> [0, 0, 0, 0, 0, 0, -1220999166/80137, -13720122/80137, 163145/240411],
> [-1183167/80137, -11814/80137, -14/80137, 0, 0, 0, 0, 0, 0],
> [-94306842/80137, -2653965/80137, -11814/80137, 0, 0, 0, 0, 0, 0],
> [-79581502242/80137, -1335450240/80137, -2653965/80137, 0, 0, 0, 0, 0, 0] ] );
> M := MatrixAlgebra( Rationals(), 9 );
> A := sub< M | [ a1, a2 ] >;
> Dimension(A);
9
> JacobsonRadical( A );
Matrix Algebra [ideal of A] of degree 9 and dimension 0 with 0 generators over
Rational Field
> O := MaximalOrder( A );
> Discriminant( O );
1
> T :=MultiplicationTable(O);
> T[3][7];
[ -16583482050411285785256, 5672389828626293786946, 1059868937213366777403,
5524536812663273356 1175, -41598423838438078787076, 1726223870812049536260,
6669449115981910248 9072, 7637318120140121751 7416, -11492818965549086607 1212 ]
```

81.4.2 Attributes

`BaseRing(O)`

`CoefficientRing(O)`

The base ring of the associative order O.

`Algebra(O)`

The container algebra of the associative order O.

`Degree(O)`

`Dimension(O)`

Returns the dimension (or degree) of the order O, equivalently the dimension of its parent algebra as a vector space over its ground field.

`Discriminant(O)`

Returns the discriminant of the order O. If O is a quaternion order, returns the reduced discriminant which is the square root of the usual discriminant.

`FactoredDiscriminant(O)`

Returns the factorization of the discriminant of the order O. If O is a quaternion order, returns the factorization of the reduced discriminant which is the square root of the usual discriminant.

`MultiplicationTable(O)`

Returns the multiplication table of the maximal order O. This is a three dimensional table of structure constants. If T denotes this table, then $T[i][j]$ is a sequence of integers containing the coefficients of the product of the i-th and j-th basis elements with respect to the basis of the order.

`Module(O)`

Return the pseudo matrix describing the basis of the associative order O over a number ring.

`TraceZeroSubspace(O)`

Given an order O in a quaternion algebra, this computes the submodule of elements with trace 0. A basis or a pseudo-basis for this submodule is returned, depending whether the base field of the quaternion algebra is $\mathbf{Q}$ or a number field. (The base ring of O is, respectively, either $\mathbf{Z}$ or an order in that number field.)

81.4.3 Bases of Orders

`Basis(O)`

Returns a basis of the order O. All other elements of the order are integral linear combinations of elements of this basis. Note that the elements of a basis will only be elements of the parent algebra A and may not be elements of O because of the existence of coefficient ideals.

`PseudoBasis(O)`

Returns the pseudobasis of the associative order O over a number ring.

`PseudoMatrix(O)`

Returns the pseudomatrix describing the pseudobasis of the associative order O over a number ring.

`ZBasis(O)`

Returns a $\mathbf{Z}$-basis for the order O.

`Generators(O)`

Returns a sequence of generators of O as a module over its base ring.

Example H81E7

We compute an order and show how a basis and a pseudobasis can differ.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> A := QuaternionAlgebra<F | -3,b>;
> O := Order([1/3*A.1, A.2], [ideal<Z_F | b^2+b+1>, ideal<Z_F | 1>]);
> O;
Order of Quaternion Algebra with base ring Field of Fractions of Z_F
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
> Basis(O);
[ Z_F.1, i, j, k ]
> PseudoBasis(O);
[
    <Principal Ideal of Z_F
    Generator:
        Z_F.1, Z_F.1>,
    <Fractional Principal Ideal of Z_F
    Generator:
        1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3, i>,
    <Principal Ideal of Z_F
    Generator:
        Z_F.1, j>,
    <Fractional Principal Ideal of Z_F
```

```
    Generator:
        1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3, k>
]
> PseudoMatrix(O);
Pseudo-matrix over Maximal Equation Order with defining polynomial x^3 - 3*x
     - 1 over its ground order
Principal Ideal of Z_F
Generator:
    Z_F.1 * ( Z_F.1 0 0 0 )
Fractional Principal Ideal of Z_F
Generator:
    1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3 * ( 0 Z_F.1 0 0 )
Principal Ideal of Z_F
Generator:
    Z_F.1 * ( 0 0 Z_F.1 0 )
Fractional Principal Ideal of Z_F
Generator:
    1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3 * ( 0 0 0 Z_F.1 )
> ZBasis(O);
[ Z_F.1, Z_F.2, Z_F.3, (1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3)*i, (1/3*Z_F.1 +
4/3*Z_F.2 + 1/3*Z_F.3)*i, (1/3*Z_F.1 + 4/3*Z_F.2 + 4/3*Z_F.3)*i, j, Z_F.2*j,
Z_F.3*j, (1/3*Z_F.1 + 1/3*Z_F.2 + 1/3*Z_F.3)*k, (1/3*Z_F.1 + 4/3*Z_F.2 +
1/3*Z_F.3)*k, (1/3*Z_F.1 + 4/3*Z_F.2 + 4/3*Z_F.3)*k ]
```

Note that the basis of O does not generate O—one needs to include the coefficient ideals.

81.4.4 Predicates

`O1 eq O2`

Return `true` if and only if the orders O_1 and O_2 are equal as subrings of the same algebra.

`x in O`

`x notin O`

Return `true` (respectively, `false`) if the element x of an associative algebra is in the associative order O.

81.4.5 Operations with Orders

`Adjoin(O, x)`

`Adjoin(O, x, I)`

Returns the order obtained by adjoining the element x to the order O, optionally with coefficient ideal I.

`O1 + O2`

Returns the sum of the orders O_1 and O_2.

`O1 meet O2`

Returns the intersection of the orders O_1 and O_2.

`O ^ x`

Returns the conjugate order $x^{-1}Ox$.

Example H81E8

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> F := FieldOfFractions(Z_F);
> A<alpha,beta,alphabeta> := QuaternionAlgebra<F | -3,b>;
> O := Order([alpha,beta]);
> O1 := Order([1/3*alpha,beta], [ideal<Z_F | b^2+b+1>, ideal<Z_F | 1>]);
> Discriminant(O1);
Principal Ideal of Z_F
Generator:
    4/1*F.1 + 12/1*F.2 + 8/1*F.3
> xi := (1 + alpha + (7+5*b+6*b^2)*beta + (3+b+6*b^2)*alphabeta)/2;
> zeta := (-6-25*b-5*b^2)*alpha - 3*beta;
> O2 := Adjoin(O, xi);
> O := O1+O2;
> Discriminant(O);
Ideal of Z_F
Basis:
[2 0 4]
[0 2 4]
[0 0 6]
```

81.5 Elements of Orders

81.5.1 Creation of Elements

`O ! 0`
`Zero(O)`

The zero element of the associative order O.

`O ! 1`
`One(O)`

The identity element of the associative order O.

`O . i`

Given an associative order O and an integer i, returns the ith basis element as an order over the base ring. Note that the element 1 may or may not be the first element of a basis. These basis elements are returned as elements of the algebra of O not as elements of O itself.

`O ! x`

Return an element of the associative order O described by x, where x may be a sequence, an element of an associative order, an element coercible into the coefficient ring of O or into the algebra of O.

`Random(O)`

Returns a "random" element of the associative order O with small coefficients.

81.5.2 Arithmetic of Elements

`x + y`

The sum of elements x and y of an order of an associative algebra.

`x - y`

The difference of elements x and y of an order of an associative algebra.

`-x`

The negation of element x of an order of an associative algebra.

`x * y`

The product of elements x and y of an associative algebra.

`u * c`
`c * u`

The product of the element u of an associative order by the scalar c.

`x / y`

The quotient of x by the unit y in the parent algebra.

`x div y`

The exact division of x by y in the order containing them.

`x ^ n`

The product of the element x of an associative order with itself n times.

81.5.3 Predicates on Elements

`x eq y`

Returns `true` if and only if the elements x and y are equal.

`x ne y`

Returns `true` if and only if the elements x and y are not equal.

`IsZero(x)`

Return `true` if the element x of an associative order is the zero element.

`IsUnit(a)`

Return `true` if the element x of an associative order is a unit in that order.

`IsScalar(x)`

Returns `true` if and only if x is an element of the base ring of the order containing it, and if so returns the coerced element.

81.5.4 Other Operations with Elements

`ElementToSequence(x)`

`Eltseq(x)`

Given an element x of an associative order O, returns the sequence of coordinates of x in terms of the basis of O.

`Norm(x)`

The norm of the element x of an order as an element of its parent algebra.

`Trace(x)`

The trace of the element x of an order as an element of its parent algebra.

`LeftRepresentationMatrix(e)`

`RightRepresentationMatrix(e)`

The representation matrix describing left (right) multiplication by the element e of an associative order.

`RepresentationMatrix(a)`

`Side` MonStgElt *Default : "Left"*

The representation matrix of the element a of an associative order. This describes left multiplication unless the parameter `Side` is set to `"Right"`.

`CharacteristicPolynomial(x)`

The characteristic polynomial of the element x of an order as an element of its parent algebra.

`MinimalPolynomial(x)`

The minimal polynomial of the element x of an order as an element of its parent algebra.

81.6 Ideals of Orders

81.6.1 Creation of Ideals

`lideal< O | E >`
`rideal< O | E >`
`ideal< O | E >`

For an associative order O, this constructs the left, right or two sided O-ideal generated by the elements in the given sequence E (these elements should be coercible into O).

`lideal< O | M >`
`rideal< O | M >`
`ideal< O | M >`

Constructs a left, right or two sided ideal of the associative order O whose basis is given by M, which may be either a matrix or a pseudo matrix.

`O * e`
`e * O`

The principal left (right) ideal of the associative order O generated by the element e.

`RandomRightIdeal(O)`

Returns a "random" right ideal of the order O, generated by elements with small coefficients.

81.6.2 Attributes of Ideals

`Algebra(I)`

The container algebra of the associative ideal I.

`Order(I)`

The associative order the associative ideal I was created as an ideal of.

`LeftOrder(I)`

`RightOrder(I)`

The order which maps the associative ideal I to itself under left (right) multiplication.

`Basis(I)`

`Basis(I, R)`

The basis of the associative ideal I. This will be returned as elements of the order or algebra R if this second argument is given, otherwise as elements of the algebra of I.

`BasisMatrix(I)`

`BasisMatrix(I, R)`

The basis matrix of the associative ideal I. This will be with respect to the basis of the order or algebra R if this second argument is given, otherwise with respect to the basis of the order I was created as an ideal of.

`PseudoBasis(I)`

`PseudoBasis(I, R)`

Return a sequence of tuples of the coefficient ideals and the basis elements of the associative ideal I. If a second argument is given, an order or algebra R, then the basis elements will be in R, otherwise the algebra of I.

`PseudoMatrix(I)`

`PseudoMatrix(I, R)`

Return a pseudo matrix describing the basis of the associative ideal I. If a second argument is given, an order or algebra R, then the basis matrix will be with respect to the basis of R, otherwise the order I was created as an ideal of.

`ZBasis(I)`

Returns a $\mathbf{Z}$-basis for the ideal I.

`Generators(I)`

Returns a sequence of generators for the ideal I as a module over its base ring.

`Denominator(I)`

Return the denominator of the ideal I. This is the minimal element d of the coefficient ring of O such that $d * I \subseteq O$ where O is the order I was created as an ideal of.

81.6.3 Arithmetic for Ideals

`I + J`

The sum of the ideals I and J, which are ideals which share a side in equal orders.

`I * J`

The product of the ideals I and J, where I is a right ideal and J is a left ideal of the same order O. Returns the product given the structure of left and right ideal.

`a * I`
`I * a`

Returns the product of a and I as an ideal.

`Colon(J, I)`

If I, J are left ideals, returns the colon $(J : I) = \{x \in A : xI \subset J\}$, similarly defined if I, J are right ideals.

`MultiplicatorRing(I)`

Returns the colon $(I : I)$ of the ideal I, the set of all elements which multiply I into I.

81.6.4 Predicates on Ideals

`IsLeftIdeal(I)`
`IsRightIdeal(I)`
`IsTwoSidedIdeal(I)`

Return `true` if the associative ideal I is a left, right or two sided ideal (respectively).

`I eq J`

Return `true` if the associative ideals I and J are equal.

`I subset J`

Returns `true` if and only if the ideal I is contained in the ideal J.

`a in I`
`a notin I`

Return `true` (`false`) if the element a of an associative algebra is contained in the associative ideal I.

81.6.5 Other Operations on Ideals

`Norm(I)`

Returns the norm of the ideal I, the ideal of the base number ring of I generated by the norms of the elements in I.

Example H81E9

```
> F<w> := CyclotomicField(3);
> R := MaximalOrder(F);
> A := Algebra(FPAlgebra<F, x, y | x^3-3, y^3+5, y*x-w*x*y>);
> O := Order([A.i : i in [1..9]]);
> MinimalPolynomial(O.2);
$.1^3 + 5/1*R.1
> I := rideal<O | O.2>;
> IsLeftIdeal(I), IsRightIdeal(I), IsTwoSidedIdeal(I);
false true false
> MultiplicatorRing(I) eq O;
true
> PseudoBasis(I);
[
    <Principal Ideal of R
    Generator:
        R.1, (0 R.1 0 0 0 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 R.1 0 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 0 -R.1 - R.2 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (-5/1*R.1 0 0 0 0 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 0 0 0 -R.1 - R.2 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 0 0 0 0 R.2 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 5/1*R.1 + 5/1*R.2 0 0 0 0 0 0)>,
    <Principal Ideal of R
    Generator:
        R.1, (0 0 0 0 0 0 0 0 R.2)>,
    <Principal Ideal of R
    Generator:
```

```
        R.1, (0 0 0 0 0 -5/1*R.2 0 0 0)>
]
> ZBasis(I);
[ [0 R.1 0 0 0 0 0 0 0], [0 R.2 0 0 0 0 0 0 0], [0 0 0 R.1 0 0 0 0 0], [0 0 0
    R.2 0 0 0 0 0], [0 0 0 0 -R.1 - R.2 0 0 0 0], [0 0 0 0 R.1 0 0 0 0],
    [-5/1*R.1 0 0 0 0 0 0 0 0], [-5/1*R.2 0 0 0 0 0 0 0 0] ]
> Norm(I);
Principal Ideal of R
Generator:
    15625/1*R.1
> J := rideal<O | O.3>;
> Norm(J);
Principal Ideal of R
Generator:
    729/1*R.1
> A!1 in I+J;
false
> Denominator(1/6*I);
[1, 0]
> Colon(J,I);
Pseudo-matrix over Maximal Equation Order with defining polynomial x^2 + x + 1
over its ground order
Principal Ideal of R
Generator:
    3/1*R.1 * ( R.1 0 0 0 0 0 0 0 0 )
Principal Ideal of R
Generator:
    3/1*R.1 * ( 0 R.1 0 0 0 0 0 0 0 )
Principal Ideal of R
Generator:
    R.1 * ( 0 0 R.1 0 0 0 0 0 0 )
Fractional Principal Ideal of R
Generator:
    3/5*R.1 * ( 0 0 0 R.1 0 0 0 0 0 )
Principal Ideal of R
Generator:
    R.1 * ( 0 0 0 0 R.1 0 0 0 0 )
Principal Ideal of R
Generator:
    R.1 * ( 0 0 0 0 0 R.1 0 0 0 )
Fractional Principal Ideal of R
Generator:
    -1/5*R.1 * ( 0 0 0 0 0 0 R.1 0 0 )
Principal Ideal of R
Generator:
    R.1 * ( 0 0 0 0 0 0 0 R.1 0 )
Fractional Principal Ideal of R
Generator:
```

```
1/5*R.1 * ( 0 0 0 0 0 0 0 0 R.1 )
```

81.7 Quaternionic Orders

The following intrinsics which take an argument of type `AlgAssVOrd`, `AlgAssVOrdElt`, or `AlgAssVOrdIdl`, apply only to associative orders of quaternion algebras and are documented in that chapter, Chapter 86.

`MaximalOrder(O)` `pMaximalOrder(O, p)`
`IsMaximal(O)` `IspMaximal(O, p)`
`pMatrixRing(O, p)`
`Embed(Oc, O)`
`LeftIdealClasses(S)` `RightIdealClasses(S)`
`TwoSidedIdealClasses(S)`
`TwoSidedIdealClassGroup(S)`
`OptimizedRepresentation(O)` `OptimisedRepresentation(O)`
`Units(S)` `MultiplicativeGroup(S)` `UnitGroup(S)`
`Conjugate(x)`
`Enumerate(O, A, B)` `Enumerate(O, B)` `Enumerate(I, B)`
`ReducedBasis(O)` `ReducedBasis(I)`
`IsIsomorphic(I, J)` `IsLeftIsomorphic(I, J)`
`IsRightIsomorphic(I, J)` `IsPrincipal(I)`

81.8 Bibliography

[**CIW97**] Arjeh M. Cohen, Gábor Ivanyos, and David B. Wales. Finding the radical of an algebra of linear transformations. *J. Pure Appl. Algebra*, 117/118:177–193, 1997. Algorithms for algebra (Eindhoven, 1996).

[**EG96**] W. Eberly and M. Giesbrecht. Efficient decomposition of associative algebras. In Y. N. Lakshman, editor, *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation: ISSAC'96*, pages 170–178, New York, 1996. ACM.

[**Fri00**] Carsten Friedrichs. *Berechnung von Maximalordnungen über Dedekindringen.* Dissertation, Technische Universität Berlin, 2000.
URL:http://www.math.tu-berlin.de/∼kant/publications/diss/diss_fried.pdf.gz.

[**IR93**] Gábor Ivanyos and Lajos Rónyai. Finding maximal orders in semisimple algebras over **Q**. *Comput. Complexity*, 3(3):245–261, 1993.

82 FINITELY PRESENTED ALGEBRAS

Chapter 82
FINITELY PRESENTED ALGEBRAS

82.1 Introduction

This chapter describes finitely presented algebras (FPAs) in MAGMA. An FPA is a quotient of a free associative algebra by an ideal of relations. To compute with these ideals, one constructs *noncommutative* Gröbner bases (GBs), which have many parallels with the standard commutative GBs, discussed in Chapter 105. At the heart of the theory is a noncommutative version of the *Buchberger algorithm* which computes a GB of an ideal of an algebra starting from an arbitrary basis (generating set) of the ideal. One significant difference with the commutative case is that a noncommutative GB may not be finite for a finitely-generated ideal. For overviews of the theory and the basic algorithms, see [Mor94, Li02].

MAGMA also contains an implementation of a noncommutative generalization of the Faugere F_4 algorithm (due to Allan Steel), based on sparse linear algebra techniques, which usually performs dramatically better than the Buchberger algorithm, and so this is used by MAGMA by default.

82.2 Representation and Monomial Orders

Let A be the free algebra $K\langle x_1, \ldots, x_n\rangle$ of rank n over a field K. A word in the underlying monoid of A is simply an associative product of the letters (or variables) of A. For consistency with the commutative case, we will call these monoid words *monomials*. Elements of A, called noncommutative polynomials, are finite sums of terms, where a term is the product of a coefficient from K and a monomial. The terms are sorted with respect to an admissible order $<$, which satisfies, for monomials p, q, r, the following conditions:

(a) If $p < q$, then $pr < qr$ and $sp < sq$.

(b) If $p = qr$ then $p > q$ and $p > r$.

Currently MAGMA only supports the noncommutative graded-lexicographical order (*glex*), which first compares degrees and then uses a left-lexicographical comparison for degree-ties. There is no admissible lexicographic order in the noncommutative case.

82.3 Exterior Algebras

Since V2.15 (December 2008), MAGMA has a special type for *exterior algebras*. Such an algebra is skew-commutative and is a quotient of the free algebra $K\langle x_1, \ldots, x_n\rangle$ by the relations $x_i^2 = 0$ and $x_i x_j = -x_j x_i$ for $1 \leq i, j \leq n$, $i \neq j$. Because of these relations, elements of the algebra can be written in terms of commutative monomials in the variables (via a collection algorithm), and the associated algorithms are much more efficient than for the general noncommutative case. Also, a Gröbner basis of an ideal of an exterior algebra is always finite (in fact, the whole exterior algebra has dimension 2^n as a K-vector space).

Exterior algebras may be constructed with the `ExteriorAlgebra` function below, and all operations applicable to general FP algebras are also applicable to them (so will not be duplicated here). Furthermore, modules over exterior algebras are also allowed: see Chapter 109 for details.

82.4 Creation of Free Algebras and Elements

82.4.1 Creation of Free Algebras

Currently algebras may only be created over fields. Free algebras are objects of type `AlgFr` with elements of type `AlgFrElt`.

`FreeAlgebra(K, n)`

Create a free algebra in $n > 0$ variables over the field K. The angle bracket notation can be used to assign names to the indeterminates; e.g., `F<a,b,c> := FreeAlgebra(GF(2), 3);`.

`ExteriorAlgebra(K, n)`

Create an exterior algebra in $n > 0$ variables over the field K. The angle bracket notation can be used to assign names to the indeterminates; The angle bracket notation can be used to assign names to the indeterminates; e.g., `F<a,b,c> := ExteriorAlgebra(GF(2), 3);`.

82.4.2 Print Names

The `AssignNames` and `Name` functions can be used to associate names with the indeterminates of free algebras after creation.

`AssignNames(~F, s)`

Procedure to change the name of the indeterminates of a free algebra F. The i-th indeterminate will be given the name of the i-th element of the sequence of strings s (for $1 \leq i \leq \#s$); the sequence may have length less than the number of indeterminates of F, in which case the remaining indeterminate names remain unchanged.

This procedure only changes the name used in printing the elements of F. It does *not* assign to identifiers corresponding to the strings the indeterminates in F; to do this, use an assignment statement, or use angle brackets when creating the free algebra.

`Name(F, i)`

Given a free algebra F, return the i-th indeterminate of F (as an element of F).

82.4.3 Creation of Polynomials

The easiest way to create (noncommutative) polynomials in a given algebra is to use the angle bracket construction to attach variables to the indeterminates, and then to use these variables to create polynomials (see the examples). Below we list other options.

`F . i`

Return the i-th indeterminate for the free algebra F in n variables ($1 \leq i \leq n$) as an element of F.

`elt< R | a >`
`R ! s`
`elt< R | s >`

This element constructor can only be used for trivial purposes in noncommutative free algebras: given a free algebra $F = R[x_1, \ldots, x_n]$ and an element a that can be coerced into the coefficient ring R, the constant polynomial a is returned; if a is in F already it will be returned unchanged.

`One(F)` `Identity(F)` `Zero(F)` `Representative(F)`

82.5 Structure Operations

82.5.1 Related Structures

The main structure related to a free algebra is its coefficient ring. Multivariate free algebras belong to the MAGMA category `AlgFr`.

`BaseRing(F)`
`CoefficientRing(F)`

Return the coefficient ring of the free algebra F.

`Category(F)` `Parent(F)` `PrimeRing(F)`

82.5.2 Numerical Invariants

Note that the `#` operator only returns a value for finite (quotients of) free algebras.

`Rank(F)`

Return the number of indeterminates of free algebra F over its coefficient ring.

`Characteristic(F)` `# F`

82.5.3 Homomorphisms

In its most general form, a homomorphism taking a free algebra $K\langle x_1, \ldots, x_n\rangle$ as domain requires $n + 1$ pieces of information, namely, a map (homomorphism) telling how to map the coefficient ring K together with the images of the n indeterminates. The map for the coefficient ring is optional.

```
hom<  F -> S | f, y1, ..., yn  >
```

```
hom<  F -> S | y1, ..., yn  >
```

Given a free algebra $F = K\langle x_1, \ldots, x_n\rangle$, a ring or associative algebra S (including another FP-algebra or a matrix algebra), and a map $f : K \to S$ and n elements $y_1, \ldots, y_n \in S$, create the homomorphism $g : F \to S$ by applying the rules that $g(rx_1^{a_1} \cdots x_n^{a_n}) = f(r)y_1^{a_1} \cdots y_n^{a_n}$ for monomials and linearity, that is, $g(M + N) = g(M) + g(N)$.

The coefficient ring map may be omitted, in which case the coefficients are mapped into S by the coercion map.

No attempt is made to check whether the map defines a genuine homomorphism.

Example H82E1

In this example we map an algebra F first into F itself, and then into a matrix algebra.

```
> K := RationalField();
> F<x,y,z> := FreeAlgebra(K, 3);
> h := hom<F -> F | x*y, y*x, z*x>;
> h(x);
x*y
> h(y);
y*x
> h(x*y);
x*y^2*x
> h(x + y + z);
x*y + y*x + z*x
> A := MatrixAlgebra(K, 2);
> M := [A | [1,1,-1,1], [-1,3,4,1], [11,7,-7,8]];
> M;
[
    [ 1  1]
    [-1  1],

    [-1  3]
    [ 4  1],

    [11  7]
    [-7  8]
]
> h := hom<F -> A | M>;
> h(x);
```

```
[ 1  1]
[-1  1]
> h(y);
[-1  3]
[ 4  1]
> h(x*y - y*z);
[ 35 -13]
[-32 -38]
```

82.6 Element Operations

82.6.1 Arithmetic Operators

The usual unary and binary ring operations are available for noncommutative polynomials, noting that multiplication is associative but noncommutative, of course.

`+ a` `- a`

`a + b` `a - b` `a * b` `a ^ k` `a / b` `a div b`

`a +:= b` `a -:= b` `a *:= b` `a div:= b`

82.6.2 Equality and Membership

`a eq b` `a ne b`

`a in R` `a notin R`

82.6.3 Predicates on Algebra Elements

`IsZero(f)` `IsOne(f)` `IsMinusOne(f)`

`IsNilpotent(f)` `IsIdempotent(f)`

`IsUnit(f)` `IsZeroDivisor(f)` `IsRegular(f)`

`IsIrreducible(f)` `IsPrime(f)`

82.6.4 Coefficients, Monomials, Terms and Degree

The functions in this subsection allow one to access noncommutative polynomials.

`Coefficients(f)`

Given a noncommutative polynomial f with coefficients in R, this function returns a sequence of 'base' coefficients, that is, a sequence of elements of R occurring as coefficients of the monomials in f. Note that the monomials are ordered, and that the sequence of coefficients corresponds exactly to the sequence of monomials returned by `Monomials(f)`.

`LeadingCoefficient(f)`

Given a noncommutative polynomial f with coefficients in R, this function returns the leading coefficient of f as an element of R; this is the coefficient of the leading monomial of f, that is, the first among the monomials occurring in f with respect to the ordering of monomials used in F.

`TrailingCoefficient(f)`

Given a noncommutative polynomial f with coefficients in R, this function returns the trailing coefficient of f as an element of R; this is the coefficient of the trailing monomial of f, that is, the last among the monomials occurring in f with respect to the ordering of monomials used in F.

`MonomialCoefficient(f, m)`

Given a noncommutative polynomial f and a monomial m, this function returns the coefficient with which m occurs in f as an element of R.

`Monomials(f)`

Given a noncommutative polynomial $f \in F$, this function returns a sequence of the monomials (monoid words) occurring in f. Note that the monomials in F are ordered, and that the sequence of monomials corresponds exactly to the sequence of coefficients returned by `Coefficients(f)`.

`LeadingMonomial(f)`

Given a noncommutative polynomial $f \in F$ this function returns the leading monomial of f, that is, the first monomial element of F that occurs in f, with respect to the ordering of monomials used in F.

`Terms(f)`

Given a noncommutative polynomial $f \in F$, this function returns the sequence of (non-zero) terms of f as elements of F. The terms are ordered according to the ordering on the monomials in F. Consequently the i-th element of this sequence of terms will be equal to the product of the i-th element of the sequence of coefficients and the i-th element of the sequence of monomials.

`LeadingTerm(f)`

Given a noncommutative polynomial $f \in F$, this function returns the leading term of f as an element of F; this is the product of the leading monomial and the leading coefficient that is, the first among the monomial terms occurring in f with respect to the ordering of monomials used in F.

`TrailingTerm(f)`

Given a noncommutative polynomial $f \in F$, this function returns the trailing term of f as an element of F; this is the last among the monomial terms occurring in f with respect to the ordering of monomials used in F.

`Length(m)`

Given a noncommutative monomial (word) m, return the length of m, i.e., the number of letters of m. Note that this differs from the commutative case, where the number of terms in a polynomial is returned.

`m[i]`

Given a noncommutative monomial (word) m of length l, and an integer i with $1 \leq i \leq l$, return the i-th letter of m.

`TotalDegree(f)`

Given a noncommutative polynomial f, this function returns the total degree of f, which is the maximum of the lengths of all monomials that occur in f. If f is the zero polynomial, the return value is -1.

`LeadingTotalDegree(f)`

Given a noncommutative polynomial, this function returns the leading total degree of f, which is the length of the leading monomial of f.

Example H82E2

In this example we illustrate the above access functions.

```
> K := RationalField();
> F<x,y,z> := FreeAlgebra(K, 3);
> f := (3*x*y - 2*y*z)*(4*x - 7*z*y) + 23*x*y*z;
> f;
-21*x*y*z*y + 14*y*z^2*y + 12*x*y*x + 23*x*y*z - 8*y*z*x
> TotalDegree(f);
4
> Coefficients(f);
[ -21, 14, 12, 23, -8 ]
> Monomials(f);
[
    x*y*z*y,
    y*z^2*y,
```

```
    x*y*x,
    x*y*z,
    y*z*x
]
> Terms(f);
[
    -21*x*y*z*y,
    14*y*z^2*y,
    12*x*y*x,
    23*x*y*z,
    -8*y*z*x
]
> MonomialCoefficient(f, x*y*z);
23
> LeadingTerm(f);
-21*x*y*z*y
> LeadingCoefficient(f);
-21
> m := Monomials(f)[1];
> m;
x*y*z*y
> Length(m);
4
> m[1];
x
> m[2];
y
```

82.6.5 Evaluation

`Evaluate(f, s)`

Given an element f of a free algebra $F = R\langle x_1, \ldots, x_n \rangle$ and a sequence or tuple s of ring or algebra elements of length n, return the value of f at s, that is, the value obtained by substituting $x_i = s[i]$. This behaves in the same way as the `hom` constructor above.

If the elements of s lie in a ring and can be lifted into the coefficient ring R, then the result will be an element of R. If the elements of s cannot be lifted to the coefficient ring, then an attempt is made to do a generic evaluation of f at s. In this case, the result will be of the same type as the elements of s.

Example H82E3

In this example we illustrate the above access functions.

```
> K := RationalField();
> F<x,y,z> := FreeAlgebra(K, 3);
> g := x*y + y*z;
> g;
x*y + y*z
> Evaluate(g, [1,2,3]);
8
> Parent($1);
Rational Field
> Evaluate(g, [y,x,z]);
x*z + y*x
> Parent($1);
Finitely presented algebra of rank 3 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y, z
```

82.7 Ideals and Gröbner Bases

MAGMA supports left-sided, right-sided, and two-sided ideals of free algebras. In general, there are not many operations applicable to single-sided ideals: quotients are supported only in the case of two-sided ideals.

Within the general context of fp-algebras, the term "basis" will refer to an *ordered* sequence of polynomials which generate an ideal. (Thus a basis may contain duplicates and zero elements so it is dissimilar to a basis of a vector space.)

82.7.1 Creation of Ideals

`ideal< A | L >`

`lideal< A | L >`

`rideal< A | L >`

Given a free algebra A over a field K, return the two-sided (`ideal`), left-sided (`lideal`), or right-sided (`rideal`) of A generated by the elements of A specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of A;

(b) A set or sequence of elements of A;

(c) An ideal of A;

(d) A set or sequence of ideals of A.

`Basis(I)`

Given an ideal I, return the current basis of I. If a Gröbner basis of I has been computed, that is returned.

`BasisElement(I, i)`

Given an ideal I together with an integer i, return the i-th element of the current basis of I. This the same as `Basis(I)[i]`.

82.7.2 Gröbner Bases

Gröbner bases (GBs) may be computed for any kind of ideal (left-, right-, or two-sided), but for single-sided ideals, the GBs are generally weak (i.e., they rarely differ much from the original generators of the ideals).

Unfortunately, the GB of a given ideal may not be finite. Thus the Buchberger or F_4 algorithms below will run forever in such cases. One can interrupt any GB computation by pressing `Ctrl-C`. Alternatively, the function `GroebnerBasis(S,d)` below, which creates a truncated degree-d Gröbner basis, can be used to set a limit on the degrees of the pairs considered, so the computation will always terminate.

As in the commutative case, when MAGMA constructs a GB G of an ideal I, then G is always the unique sorted minimal reduced GB of I. Before this happens, an ideal will usually possess a basis which is not a Gröbner basis, but that will be changed into the unique Gröbner basis when the GB is computed. Thus the original basis will be discarded. See the procedure `Groebner` below for details on the algorithms available.

The unique Gröbner basis will be computed automatically when necessary; the `Groebner` procedure below simply allows control of the algorithms used to compute the Gröbner basis.

`Groebner(I: parameters)`

(Procedure.) Explicitly force a Gröbner basis (GB) for I to be constructed. This procedure is normally not necessary, as MAGMA will automatically compute the GB when needed, but it does allow one to control how the GB is constructed.

`Faugere` BOOLELT *Default :* `true`

MAGMA has two algorithms for computing noncommutative GBs:

(1) A noncommutative generalization (due to Allan Steel) of the Faugère F_4 algorithm [Fau99], which works by specialized sparse linear algebra and is applicable to two-sided ideals defined over a finite field or the rational field;

(2) The noncommutative Buchberger algorithm [CLO96, Chap. 2, §7] for ideals defined over any field.

If the parameter `Faugere` is set to `true`, then the Faugère F_4 algorithm will be used (if the field is a finite field or the rational field); otherwise the Buchberger algorithm is used.

The current implementation of the Faugère algorithm is usually very much faster than the Buchberger algorithm and usually does not take much more memory, so that it is why it is selected by default. However, there may be examples for which it

may be more desirable to use the Buchberger algorithm (particularly to save some memory).

`GroebnerBasis(I: parameters)`

Given an ideal I, force the Gröbner basis of I to be computed, and then return that. The parameters are the same as those for the procedure `Groebner`.

`GroebnerBasis(S: parameters)`

Given a set or sequence S of polynomials, return the unique Gröbner basis of the two-sided ideal generated by S as a sorted sequence. This function is useful for computing Gröbner bases without the need to construct ideals. The parameters are the same as those for the procedure `Groebner`. See also the function `GroebnerBasis(S,d)` below, which creates a truncated degree-d Gröbner basis.

`GroebnerBasis(S, d: parameters)`

Given a set or sequence S of polynomials, return the degree-d Gröbner basis of the ideal generated by S, which is the truncated Gröbner basis obtained by ignoring S-polynomial pairs whose total degree is greater than d.

If the ideal is homogeneous, then it is guaranteed that the result is equal to the set of all polynomials in the full Gröbner basis of the ideal whose total degree is less than or equal to d, and a polynomial whose total degree is less than or equal to d is in the ideal if and only if its normal form with respect to this truncated basis is zero. But if the ideal is not homogeneous, these last properties may not hold, but it may still be useful to construct the truncated basis.

The parameters are the same as those for the procedure `Groebner`.

82.7.3 Verbosity

This subsection describes the verbose flags available for the Gröbner basis algorithms. There are separate verbose flags for each algorithm (`Buchberger`, etc.), but the all-encompassing verbose flag `Groebner` includes all these flags implicitly.

For each procedure provided for setting one of these flags, the value `false` is equivalent to level 0 (nothing), and `true` is equivalent to level 1 (minimal verbosity). For each `Set-` procedure, there is also a corresponding `Get-` function to return the value of the corresponding flag.

`SetVerbose("Groebner", v)`

(Procedure.) Change the verbose printing level for all Gröbner basis algorithms to be v. This includes all of the algorithms whose verbosity is controlled by flags subsequently listed, as well as some other minor related algorithms. Currently the legal levels are 0, 1, 2, 3, or 4. One would normally set this flag to 1 for minimal verbosity for Gröbner basis-type computations, and possibly also set one or more of the following flags to levels higher than 1 for more verbosity.

`SetVerbose("Buchberger", v)`

(Procedure.) Change the verbose printing level for the Buchberger algorithm to be v. Currently the legal levels are 0, 1, 2, 3, or 4. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

`SetVerbose("Faugere", v)`

(Procedure.) Change the verbose printing level for the Faugère algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

82.7.4 Related Functions

The following functions and procedures perform operations related to Gröbner bases.

`MarkGroebner(I)`

(Procedure.) Given an ideal I, mark the current basis of I to be *the* Gröbner basis of the ideal with respect to the monomial order of the ideal. Note that the current basis must exactly equal the unique (reverse) sorted minimal reduced Gröbner basis for the ideal, as returned by the function `GroebnerBasis`. This procedure is useful when one creates an ideal with a basis known to be the Gröbner basis of the ideal from a previous computation or for other reasons. If the basis is not the unique Gröbner basis, the results are unpredictable.

`Reduce(S)`

Given a set or sequence S of polynomials, return the sequence consisting of the reduction of S. The reduction is obtained by reducing to normal form each element of S with respect to the other elements and sorting the resulting non-zero elements left. Note that all Gröbner bases returned by MAGMA are automatically reduced so that this function would usually only be used just to simplify a set or sequence of polynomials which is not a Gröbner basis.

Example H82E4

For a certain sequence B of noncommutative polynomials, we create the left-, right- and two-sided ideals generated by B. We note that for the first two cases, the GB is no different from B, but for the two-sided case, the GB contains several more elements.

```
> K := RationalField();
> F<x,y,z> := FreeAlgebra(K, 3);
> B := [x^2 - y*z, x*y - y*z, y*x - z^2, y^3 - x*z];
> I := lideal<F | B>;
> I;
Left ideal of Finitely presented algebra of rank 3 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y, z
Basis:
[
```

```
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2,
    y^3 - x*z
]
> GroebnerBasis(I);
[
    y^3 - x*z,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> I := rideal<F | B>;
> GroebnerBasis(I);
[
    y^3 - x*z,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> I := ideal<F | B>;
> Groebner(I);
> I;
Two-sided ideal of Finitely presented algebra of rank 3 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y, z
Groebner basis:
[
    y*z^2*y - y*z^2,
    y*z^3 - y*z^2,
    z*y*z^2 - y*z^2,
    z^2*y^2 - y*z^2,
    z^2*y*z - y*z^2,
    z^3*y - y*z^2,
    z^4 - y*z^2,
    x*z*x - y*z^2,
    x*z*y - z^3,
    x*z^2 - y*z^2,
    y^3 - x*z,
    y^2*z - z^2*y,
    y*z*x - y*z^2,
    y*z*y - y*z^2,
    z^2*x - z^2*y,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> NormalForm(x*y, I);
```

```
y*z
> NormalForm(y*x, I);
z^2
```

Finally, we compute some truncated bases of the two-sided ideal. For degree 2, the truncated GB has no new polynomials while for degree 3, some are added. Only at degree 5 do we obtain the full GB.

```
> GroebnerBasis(B, 2);
[
    y^3 - x*z,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> GroebnerBasis(B, 3);
[
    x*z^2 - y*z^2,
    y^3 - x*z,
    y^2*z - z^2*y,
    y*z*x - y*z^2,
    y*z*y - y*z^2,
    z^2*x - z^2*y,
    x^2 - y*z,
    x*y - y*z,
    y*x - z^2
]
> #GroebnerBasis(I);
18
> #GroebnerBasis(B, 4);
16
> #GroebnerBasis(B, 5);
18
> GroebnerBasis(B, 5) eq GroebnerBasis(I);
true
```

82.8 Basic Operations on Ideals

In the following, note that the free algebra F itself is a valid ideal (the ideal containing 1).

82.8.1 Construction of New Ideals

`I + J`

Given ideals I and J belonging to the same algebra F, return the sum of I and J, which is the ideal generated by the union of the generators of I and J.

`I * J`

Given ideals I and J belonging to the same algebra A, return the product of I and J, which is the ideal generated by the products of the generators of I with those of J.

`F / J`

Given an algebra F over a field and an ideal J of F, return the fp-algebra F/J (see below).

`Generic(I)`

Given an ideal I of a generic algebra A, return A.

82.8.2 Ideal Predicates

`I eq J`

Given two ideals I and J belonging to the same algebra F, return whether I and J are equal.

`I ne J`

Given two ideals I and J belonging to the same algebra F, return whether I and J are not equal.

`I notsubset J`

Given two ideals I and J belonging to the same algebra F return whether I is not contained in J.

`I subset J`

Given two ideals I and J belonging to the same algebra F return whether I is contained in J.

`IsZero(I)`

Given an ideal I of the algebra F, return whether I is the zero ideal (contains zero alone).

82.8.3 Operations on Elements of Ideals

`f in I`

Given a polynomial f from an algebra F, together with an ideal I of F, return whether f is in I.

`NormalForm(f, I)`

Given a polynomial f from an algebra F, together with an ideal I of F, return the unique normal form of f with respect to (the Gröbner basis of) I. The normal form of f is zero if and only if f is in I.

`NormalForm(f, S)`

Given a polynomial f from an algebra F, together with a set or sequence S of polynomials from F, return a normal form of f with respect to S. This is not unique in general. If the normal form of f is zero then f is in the ideal generated by S, but the converse is false in general. In fact, the normal form is unique if and only if S forms a Groëbner basis.

`f notin I`

Given a polynomial f from an algebra F, together with an ideal I of F, return whether f is not in I.

Example H82E5

We demonstrate the element operations with respect to an ideal of $\mathbf{Q}[x, y, z]$.

```
> F<x,y,z> := FreeAlgebra(RationalField(), 3);
> I := ideal<F | (x + y)^3, (y - z)^2, y^2*z + z>;
> NormalForm(y^2*z + z, I);
0
> NormalForm(x^3, I);
-x^2*y - x*y*x - x*y*z - x*z*y + x*z^2 - y*x^2 - y*x*y - y*z*x -
    y*z*y - z*y*x - z*y*z + z^2*x + z^3
> NormalForm(z^4 + y^2, I);
z^4 + y*z + z*y - z^2
> x + y in I;
false
```

82.9 Changing Coefficient Ring

The `ChangeRing` function enables the changing of the coefficient ring of an algebra or ideal.

`ChangeRing(I, S)`

Given an ideal I of an algebra $F = R[x_1, \ldots, x_n]$ of rank n with coefficient ring R, together with a ring S, construct the ideal J of the algebra $Q = S[x_1, \ldots, x_n]$ obtained by coercing the coefficients of the elements of the basis of I into S. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. If R and S are fields and R is known to be a subfield of S and the current basis of I is a Gröbner basis, then the basis of J is marked automatically to be a Gröbner basis of J.

82.10 Finitely Presented Algebras

A finitely presented algebra (fp-algebra) in MAGMA is simply the quotient ring of a free algebra $F = R\langle x_1, \ldots, x_n \rangle$ by an ideal J of F. It is an object of type `AlgFP` with elements of type `AlgFPElt`.

The elements of fp-algebras are simply noncommutative polynomials which are always kept reduced to normal form modulo the ideal J of "relations". Practically all operations which are applicable to noncommutative polynomials are also applicable in MAGMA to elements of fp-algebras (when meaningful).

If an fp-algebra A has finite dimension, considered as a vector space over its coefficient field, then extra special operations are available for A and its elements.

82.11 Creation of FP-Algebras

One can create an fp-algebra simply by forming the quotient of a free algebra by an ideal (`quo` constructor or `/` function). A special constructor `FPAlgebra` is also provided to remove the need to create the free algebra.

NOTE: When one creates an fp-algebra, the ideal of relations is left unchanged, but as soon as one does just about any operation with elements of the algebra (such as printing or multiplying), then the Gröbner basis of the underlying ideal will have to be computed to enable MAGMA to compute a unique form of each element of the algebra.

`quo< F | J >`

`quo< F | a`$_1$`, ..., a`$_r$ `>`

Given a free algebra F and a two-sided ideal J of F, return the fp-algebra (quotient algebra) F/J. The ideal J may either be specified as an ideal or by a list a_1, a_2, ..., a_r, of generators which all lie in F. The angle bracket notation can be used to assign names to the indeterminates: `A<q, r> := quo< I | I.1 + I.2, I.2^2 - 2, I.3^2 + I.4 >;`.

`F / J`

Given a free algebra F and an ideal J of F, return the fp-algebra F/J.

`FPAlgebra< K, X | L >`

Given a field K, a list X of n identifiers, and a list L of noncommutative polynomials (relations) in the n variables X, create the fp-algebra of rank n with base ring K with given quotient relations; i.e., return $K[X]/\langle L\rangle$. The angle bracket notation can be used to assign names to the indeterminates.

Example H82E6

We illustrate equivalent ways of creating FP-algebras.

```
> K := RationalField();
> A<x,y> := FPAlgebra<K, x, y | x^2*y - y*x, x*y^3 - y*x>;
> A;
Finitely Presented Algebra of rank 2 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y
Quotient relations:
[
    x^2*y - y*x,
    x*y^3 - y*x
]
> x;
x
> x*y;
x*y
> x^2*y;
y*x
> A;
Finitely Presented Algebra of rank 2 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y
Quotient relations:
[
    x*y*x^2 - y*x^2,
    x*y*x*y - y^2*x,
    x*y^2*x - y^2*x,
    x*y^3 - y*x,
    y*x^3 - y*x^2,
    y*x*y*x - y^2*x,
    y*x*y^2 - x*y*x,
    y^2*x^2 - y^2*x,
    y^2*x*y - y*x^2,
    y^3*x - y*x^2,
    x^2*y - y*x
```

```
]
```

The following is equivalent.

```
> K := RationalField();
> F<x,y> := FreeAlgebra(K, 2);
> A<x,y> := quo<F | x^2*y - y*x, x*y^3 - y*x>;
```

82.12 Operations on FP-Algebras

This section describes operations on fp-algebras. Most of the operations are very similar to those for noncommutative free algebras; such operations are done by mapping the computation to the preimage ideal and then by mapping the result back into the fp-algebra. See the corresponding functions for the noncommutative free algebras for details.

`A . i`

Given an fp-algebra A, return the i-th indeterminate of A as an element of A.

`CoefficientRing(A)`

Return the coefficient ring of the fp-algebra A.

`Rank(A)`

Return the rank of the fp-algebra A (the number of indeterminates of A).

`DivisorIdeal(I)`

Given an ideal I of an fp-algebra A which is the quotient ring F/J, where F is a free algebra and J an ideal of F, return the ideal J.

`PreimageIdeal(I)`

Given an ideal I of an fp-algebra A which is the quotient ring F/J, where F is a free algebra and J an ideal of F, return the ideal I' of F such that the image of I' under the natural epimorphism $F \to A$ is I.

`PreimageRing(A)`

Given an fp-algebra A which is the quotient ring F/J, where F is a free algebra and J an ideal of F, return the free algebra F.

`OriginalRing(A)`

Return the generic free algebra F such that A is F/J for some ideal J of F.

`IsCommutative(A)`

Return whether the algebra A is commutative.

`I eq J`

Given two ideals I and J of the same fp-algebra A, return `true` if and only if I and J are equal.

`I subset J`

Given two ideals I and J of the same fp-algebra A, return `true` if and only if I is contained in J.

`I + J`

Given two ideals I and J of the same fp-algebra A, return the sum $I + J$.

`I * J`

Given two ideals I and J of the same fp-algebra A, return the product $I * J$.

`IsProper(I)`

Given an ideal I of the fp-algebra A, return whether I is proper; that is, whether I is strictly contained in A.

`IsZero(I)`

Given an ideal I of the fp-algebra A, return whether I is the zero ideal. Note that this is equivalent to whether the preimage ideal of I is the divisor ideal of A.

82.13 Finite Dimensional FP-Algebras

If an fp-algebra A has finite dimension, considered as a vector space over its coefficient field, then extra special operations are available for A and the elements of A.

`Dimension(A)`

Given a finite dimensional fp-algebra A, return the dimension of A.

`VectorSpace(A)`

Given a finite dimensional fp-algebra A, construct the vector space V isomorphic to A, and return V together with the isomorphism f from A onto V.

`MatrixAlgebra(A)`

Given a finite dimensional fp-algebra A, construct the matrix algebra M isomorphic to A, and return M together with the isomorphism f from A onto M.

`Algebra(A)`

Given a finite dimensional fp-algebra A, construct the associative structure-constant algebra S isomorphic to A, and return S together with the isomorphism f from A onto S.

`RepresentationMatrix(f)`

Given an element f of a finite dimensional fp-algebra A, return the representation matrix of f, which is a d by d matrix over the coefficient field of A which represents f (where d is the dimension of A).

`IsUnit(f)`

Given an element f of a finite dimensional fp-algebra A defined over a field, return whether f is a unit.

`IsNilpotent(f)`

Given an element f of a finite dimensional fp-algebra A defined over a field, return whether f is nilpotent, and if so, return also the smallest q such that $f^q = 0$.

`MinimalPolynomial(f)`

Given an element f of a finite dimensional fp-algebra A defined over a field, return the minimal polynomial of f as a univariate polynomial over the coefficient field of A.

Example H82E7

We demonstrate the functions available for finite-dimensional fp-algebras.

```
> K := RationalField();
> A<x,y,z> := FPAlgebra<K, x,y,z |
>           x^2 - y*z*y + z, y^2 - y*x*y + 1, z^2 - y*x*y - x*z*x>;
> A;
Finitely Presented Algebra of rank 3 over Rational Field
Non-commutative Graded Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    y^4 - 7/2*z^2*x + z^3 - 1/2*x^2 + 2*y^2 + z*x + z^2 + x + 1,
    z*y*x^2 + y^3 - z^2*y + y,
    z^2*y*x - 1/2*z*y*z - z^2*y - 1/2*y*x,
    z^2*y*z - 3/2*y*x^2 - 3/4*z*y*z - 3/2*z^2*y - 3/4*y*x - y*z - z*y,
    z^3*x - 3/2*z^3 - 1/2*z*x,
    z^3*y - 3/2*y*x^2 - 3/4*z*y*z - 3/2*z^2*y - 3/4*y*x - y*z - z*y,
    z^4 - 2*z^2*x - 9/4*z^3 + 3/2*y^2 - 3/4*z*x - 1/2*z^2 + x + 3/2,
    x^3 + 3/2*z^2*x - z^3 + 1/2*x^2 + z*x,
    y^2*x - y^2 - 1,
    y^2*z + 3/2*z^2*x - z^3 + 1/2*x^2 + z,
    y*z*x - z*y*x,
    y*z*y - x^2 - z,
    y*z^2 - z^2*y,
    z*x^2 + y^2 - z^2 + 1,
    z*y^2 + 3/2*z^2*x - z^3 + 1/2*x^2 + z,
    x*y - y*x,
```

```
    x*z - z*x
]
> IsCommutative(A);
false
> y*z;
y*z
> z*y;
z*y
> U<u> := PolynomialRing(K);
> MinimalPolynomial(x);
u^8 - 7/2*u^7 + 4*u^6 - 3/2*u^5 - 1/2*u^4 + 2*u^3 - 2*u^2
> MinimalPolynomial(y);
u^16 - u^12 + 3*u^10 + u^8 - 10*u^6 - 15*u^4 - 9*u^2 - 2
> Dimension(A);
18
> V, Vf := VectorSpace(A);
> V;
Full Vector space of degree 18 over Rational Field
> Vf(x);
(0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> [V.i@@Vf: i in [1 .. Dimension(V)]];
[
    1,
    z,
    y,
    x,
    z^2,
    z*y,
    z*x,
    y*z,
    y^2,
    y*x,
    x^2,
    z^3,
    z^2*y,
    z^2*x,
    z*y*z,
    z*y*x,
    y^3,
    y*x^2
]
> M, Mf := MatrixAlgebra(A);
> M;
Matrix Algebra of degree 18 with 4 generators over Rational Field
> M.1;
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
```

```
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[-1 0 0 0 1 0 0 0 -1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 -1 0 0 0 -1/2 1 0 -3/2 0 0 0 0]
[0 0 0 0 0 0 1/2 0 0 0 0 3/2 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1/2 0 0 1 0 1/2 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1/2 0 0 3/2 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1/2 0 0 1 0 1/2 0 0 0]
[0 0 -1 0 0 0 0 0 0 0 0 0 1 0 0 0 -1 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 -1 0 1]
> M.1 eq RepresentationMatrix(x);
true
> MinimalPolynomial(M.1);
u^8 - 7/2*u^7 + 4*u^6 - 3/2*u^5 - 1/2*u^4 + 2*u^3 - 2*u^2

> FactoredMinimalPolynomial(M.1);
[
    <u, 2>,
    <u^6 - 7/2*u^5 + 4*u^4 - 3/2*u^3 - 1/2*u^2 + 2*u - 2, 1>
]
> N := Kernel(M.1);
> N;
Vector space of degree 18, dimension 4 over Rational Field
Echelonized basis:
(0 1 0 0 0 0 0 0 0 0 3/4 -1/2 0 3/4 0 0 0 0)
(0 0 0 1 3 0 0 0 0 0 0 0 0 -2 0 0 0 0)
(0 0 0 0 0 1 0 -1 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 1 0 -1 0 0 0)
> a := N.1 @@ Vf;
> a;
3/4*z^2*x - 1/2*z^3 + 3/4*x^2 + z
> IsNilpotent(a);
true 3
> a^3;
0
> S, Sf := Algebra(A);
> S;
Associative Algebra of dimension 18 with base ring Rational Field
> Centre(S);
Associative Algebra of dimension 15 with base ring Rational Field
> J := JacobsonRadical(S);
> J;
Associative Algebra of dimension 4 with base ring Rational Field
```

```
> L := [J.i@@Sf: i in [1 .. 4]];
> L;
[
    3/4*z^2*x - 1/2*z^3 + 3/4*x^2 + z,
    -2*z^2*x + 3*z^2 + x,
    -y*z + z*y,
    -z*y*z + z^2*y
]
> [MinimalPolynomial(x): x in L];
[
    u^3,
    u^2,
    u^3,
    u^2
]
```

82.14 Vector Enumeration

Vector enumeration (originally misnamed module enumeration) is an algorithm for converting a finitely-presented module for a finitely-presented algebra into a concrete vector space on which the algebra has explicit matrix action. The algebra may be the group algebra of an fp-group, in which case the resulting module will be a matrix representation of the group, or it might be a more general fp-algebra, such as a Hecke algebra or a quotient of a polynomial ring.

82.14.1 Finitely Presented Modules

For a ring R, let M be an R-module M, generated as an R-module by s elements $\{m_1, \ldots, m_s\}$. There is an R-module epimorphism $\psi : R^s \mapsto M$, given by

$$(r_1, \ldots, r_s) \mapsto m_1 r_1 + \cdots + m_s r_s.$$

This shows that M is isomorphic to $A^s / \ker \psi$.

If $\ker \psi$ is generated as an R-module by a finite set L then we will say that M is presented by s generators and the relators L.

82.14.2 S-algebras

Suppose there is another ring S, equipped with a ring homomorphism $\phi : S \mapsto R$, such that $\phi(S)$ is central in R. In this situation any R-module can be described as an S-module, on which R acts as a ring of S-module endomorphisms. We say that R is an S-algebra. In particular, when S is a field k, any R-module is a k-vector space. If such a module V has finite k-dimension n, then V is characterised by this dimension and R will act on it as a subring of $M_n(k)$.

82.14.3 Finitely Presented Algebras

Given a finite set X, and a ring S, we can define the free S-algebra A generated by X. This can be seen either as the monoid algebra of the free monoid of words in X, or as all expressions in X and k, combined by addition and multiplication.

Given a finite set $R \subset A$ we can define

$$P = \frac{A}{\langle ARA \rangle}.$$

We say that P is a *finitely-presented* S-algebra, with generators X and relators R, and write it

$$P = \langle X \mid R \rangle .$$

The monoid algebra of any finitely-presented monoid (or group, of course) is finitely-presented, since

$$k \langle X \mid l_1 = r_1, \ldots, l_k = r_k \rangle_{\text{monoid}} = \langle X \mid l_1 - r_1, \ldots, l_k - r_k \rangle_{k-\text{algebra}} .$$

Furthermore, any quotient of a finitely-presented algebra by a finitely-generated two-sided ideal is finitely-presented. This gives us the general form of a finitely-presented algebra in Magma, as the quotient of the monoid algebra of an fp-monoid, by the two-sided ideal generated by some additional relators.

82.14.4 Vector Enumeration

The vector enumeration algorithm explicitly reconciles these two descriptions of an R-module, in the case where R is a finitely presented k-algebra for a field k, and M is a finitely presented R-module, which also has finite k-dimension.

Given the presentation of R as k-algebra, and that of M as R-module, it computes the k-dimension of M and the matrices giving the action of the generators of R on M. If M has infinite k-dimension the algorithm will fail to terminate.

Example H82E8

We consider some examples in the abstract. Below we will see how these and other calculations may be performed in Magma.

(1) **A permutation module**

In practice, it is always better to use the classical Todd-Coxeter algorithm to construct permutation representations of groups.

We know that the group with presentation

$$\left\langle a, b \mid a^4 = b^2 = (ab)^2 = 1 \right\rangle$$

is the dihedral group of order 8. Its group algebra over any field k is thus the fp-k-algebra

$$\left\langle a, b, a', b' \mid aa' - 1, a'a - 1, bb' - 1, b'b - 1, a^4 - 1, b^2 - 1, (ab)^2 - 1 \right\rangle .$$

The permutation module of degree 4 of this algebra is presented by one generator (as it is transitive) and the submodule generator $b-1$.
Applying the algorithm to this case we obtain the matrices

$$a = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad b = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and their inverses for a' and b'.
(2) **A quotient of a permutation module**
Like all permutation modules, this one fixes the all-ones vector $(1,1,1,1)$, which we can see to be an image of $1+a'(1+b+a')$. We can construct the quotient of the permutation module by this 1-dimensional submodule by adding that word to the submodule generators. This gives

$$a = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \qquad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

(3) **A non-cyclic module**
A permutation module of a group-ring is cyclic (that is, generated as a module by one element) just when the permutation representation is transitive. An intransitive permutation representation can be easily constructed from its transitive components, but in general it is not so easy to construct an arbitrary module from cyclic submodules. Accordingly it can be worthwhile to construct non-cyclic modules directly.
As an example we take two copies of the representation constructed in example one, and fuse their one-dimensional submodules. The generators and relators are as before, and now $s=2$ and the submodule generators are $a^2 = \{(b-1,0),(0,b-1),(1+a'(1+a'+b),-1-a'(1+a'+b))\}$. We obtain a representation of degree seven, given by

$$a = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & 1 & 1 & -1 & -1 \end{pmatrix} \qquad b = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

82.14.5 The Isomorphism

In determining the k-dimension of M, and giving matrices representing the action of R on M, the algorithm is, in effect, constructing a k-vector space, on which R has matrix action, and which is R-module isomorphic to M, which is formally a quotient module of R^s. The algorithm also computes this isomorphism, giving images in the vector space for the s standard generators of R^s and pre-images in R^s of the basis of the vector space.

In example (1) above, we find that the module generator has image $(1,0,0,0)$, while the basis vectors have pre-images 1, a', a'^2 and b.

In example (3) the images of the two module generators are $(1,0,0,0,0,0,0)$ and $(0,1,0,0,0,0,0)$ while the basis vectors are images of $(1,0)$, $(0,1)$, $(a',0)$,$(a,0)$, $(a^2,0)$, $(0,a')$ and $(0,a)$.

82.14.6 Sketch of the Algorithm

The algorithm is based on the Haselgrove-Leech-Trotter (HLT) version of the Todd-Coxeter algorithm, which we consider as a means of constructing the permutation representation of a finitely-presented group on the cosets of a finitely generated subgroup.

The algorithm proceeds by manipulating a partial action of the free algebra on a vector space. This is represented as a table, with columns indexed by the generators of the algebra, and rows indexed by the basis of the space. Each entry is either a vector or $\perp$, signifying that no action is given.

We can extend this to a "complete action with side-effects", by adding a new row to the table whenever needed. We call this *the action with defining*.

The action of the fp-algebra P on M gives an action for the free algebra A, and our objective is to modify out partial action until it becomes this action. We know certain things about this action, which drive our modification process:

1. It is a complete action.
2. The relators of P annihilate every vector.
3. The images in the space of the relators of M are zero.
4. The space contains images of the R-module generators of M.

The algorithm begins by applying the fourth fact and creating s linearly independent vectors. It then applies the third, computing the action with defining of the relators of M (the set called L above). The fact that these images should be zero gives a linear dependence among our basis vectors (called a coincidence), which we use to reduce their number. As we do this, we have to take care not to lose the information already in the table, which may give rise to further coincidences.

We now start to exploit the second fact about M, constructing the action (with defining) of the relators of P, on the basis vectors, and applying the resulting coincidences. This process may not terminate, as we are defining new basis vectors on the one hand, and removing them through coincidences on the other. It is possible to prove, however, that if M is in fact finite-dimensional then the process will terminate.

The first fact is applied by adding the relators $x - x$ for each $x \in X$ to the relators of P, so that the image of every basis vector under each x is sure to be defined.

82.14.7 Weights

The above sketch leaves open the question of the order in which the relators are applied to the basis vectors. The proof that the algorithm completes when M is finite-dimensional imposes some rather loose constraints, but within these constraints there is considerable choice.

The implementation in MAGMA uses a system of weights. Each relator r is assigned a weight w_r by the user, and each basis vector e has a weight w_e. There is a current weight w, which increases as the computation progresses, and at weight w all basis vector, relator pairs (b, r) such that $w_b + w_r \leq w$, which have not been processed already, are processed. New basis vectors defined during this process are assigned weight w. The initial basis vectors and those defined during processing of the submodule generators are assigned weight 1. See below for details of how to set the weights, and their default values.

82.14.8 Setup Functions

For V2.11, the following functions create the old representation for fp-algebras which are necessary for the Vector Enumeration algorithm. These are compatible with older versions of Magma. See the examples below for examples of how to use these functions.

```
FreeAlgebra(R, M)
```

```
FreeAlgebra(R, G)
```

Construct the special fp-algebra over the ring R and the monoid M or the group G, for use in the Vector Enumeration algorithm.

82.14.9 The Quotient Module Function

```
QuotientModule(A, S)
```

Given an fp-k-algebra A, for a field k, with r generators, and a submodule N of the free A-module of rank s specified by S, construct an A-module isomorphic to the quotient module A^s/N together with the isomorphism.

The three values returned are:

M a sequence of r $n \times n$ matrices with entries in k;

I a sequence of s vectors of length n with entries in k;

P a sequence of n elements of the free A-module A^s.

The matrices M specify a homomorphism from A to $\mathrm{End}_k(k^n)$, under which k^n becomes an A-module isomorphic to A^s/N. That isomorphism is given in one direction by the vectors I, which are the images in k^n of the s generators $(1, 0, \ldots, 0)$, $(0, 1, 0, \ldots, 0)$, $\ldots$, $(0, \ldots, 0, 1)$ of A^s. In the other direction, the elements P give representatives for the images in A^s/N of the images of the n standard basis elements of k^n.

The submodule N may be specified by the parameter S in three ways:

(1) S may simply be N, a finitely generated submodule of a free A-module (except that such an object cannot currently be created).

(2) S may be a finitely generated right ideal of A, in which case $s = 1$ and N is S considered as a submodule of A considered as a right module for itself.

(3) S may be a sequence of elements of a free A-module A^s, in which case N is the submodule that they generate (this is a stand in for 1 and could be removed when 1 is implemented).

82.14.10 Structuring Presentations

The relations used by the vector enumeration algorithm come from three sources:

(1) The relations of the fp-group or fp-monoid underlying A.

(2) The relations of A itself.

(3) The generators of N.

The third group play the same role as subgroup generators in the Todd-Coxeter algorithm, and are treated specially, while the first two groups are logically equivalent, forming the relations of A in the variety of free finitely-generated associative algebras. However, when the underlying monoid of A is actually an fp-group G, the vector enumeration algorithm can use a more efficient technique to process the relations of G, and can take advantage of the fact that the generators of G are known to be invertible.

This greatly improves the performance of the algorithm, and so users are recommended to ensure as far as possible that:

(1) If the underlying monoid of an fp-algebra is in fact an fp-group then it should be presented to MAGMA as such.

(2) Relations which can be written as equations between monomials should be given as relations of the underlying monoid, rather than as relations of the algebra.

82.14.11 Options and Controls

The `QuotientModule` function supports a large selection of optional arguments.

82.14.12 Weights

The processing of the relations of A by the vector enumeration algorithm depends on weights which are assigned to those relations (see above). The higher the weight of a relation the later it will be processed. By default, all relations are given weight 3, except those arising from relators of an fp-group, which are given weight equal to half the length of the relator.

Separate weights are used in lookahead mode, with the same default values.

`QuotientModule(A, S)`

`MonomialWeights` [RNGINTELT] *Default :*

`MonWts` [RNGINTELT] *Default :*

This option sets the sequence of weights for the relations derived from the relations of the underlying monoid of A. The weights w_1, w_2, etc. are applied to the relations in the order in which they appear. If there are fewer weights than relations the remaining relations are assigned the default weight; if there are more weights than relations the extra weights are silently discarded.

Unless the `MonomialLookaheadweights` or `MonLWts` parameters are present, these weights are also used in lookahead mode.

`MonomialLookaheadWeights`

[RNGINTELT] *Default :*

`MonLWts` [RNGINTELT] *Default :*

This option sets the sequence of weights for the relations derived from the relations of the underlying monoid of A in lookahead mode only. It is otherwise similar to `MonomialWeights`.

`AlgebraWeights` [RNGINTELT] *Default :*

`AlgWts` [RngIntElt] *Default :*

This option sets the sequence of weights for the relations given explicitly as relations of the algebra A. It is otherwise similar to `MonomialWeights`.

`AlgebraLookaheadWeights`

[RngIntElt] *Default :*

`AlgLWts` [RngIntElt] *Default :*

This option sets the sequence of weights for the relations given explicitly as relations of the algebra A in lookahead mode only. It is otherwise similar to `AlgebraWeights`.

82.14.13 Limits

`QuotientModule(A, S)`

Options in this group set limits on the progress of the algorithm. If the calculation cannot be performed under these constraints the value `undef` is returned, unless the `ErrorOnFail` option was set, in which case a run-time error is generated.

`MaximumDimension` RngIntElt *Default :* ∞

`MaxDim` [RngIntElt] *Default :* ∞

This sets a limit of n on the dimension of the vector-space constructed, and on the dimension of the intermediate spaces used in the construction.

By default there is no limit, except for available memory.

`MaximumTime` FldReElt *Default :* ∞

`MaxTime` FldReElt *Default :* ∞

This sets a limit on the CPU time available for the vector enumeration. The limit is given as a real number t and is measured in seconds.

This limit is only checked at certain points in the calculation, so it is possible for a vector enumeration to over-run, possibly by a significant amount.

By default, there is no limit.

`MaximumWeight` RngIntElt *Default :* 100

`MaxWt` RngIntElt *Default :* 100

This sets a limit on the maximum weight of (basis vector, relation) pairs that will be used by the algorithm.

The weight of a basis vector is the weight of the pair that was being processed when it was defined. The weight of a pair is the weight of the basis vector plus the weight of the relation (see above).

The default limit is 100.

82.14.14 Logging

`QuotientModule(A, S)`

There are a number of options to control the level of detail provided in the informational message from the vector enumerator. When multiple contradictory options are given the first one given takes precedence.

`NoLogging`	BoolElt	*Default :* `false`
`NoLog`	BoolElt	*Default :* `false`
`Silent`	BoolElt	*Default :* `false`

This option turns off all the informational messages produced by the vector enumerator.

`MaximumLogging`	BoolElt	*Default :* `false`
`MaxLog`	BoolElt	*Default :* `false`

This option turns on the highest possible level of detail in the informational messages. This is *too detailed* for almost all purposes except debugging.

`LogActions`	RngIntElt	*Default :* 0
`LogAct`	RngIntElt	*Default :* 0

This option sets the level of messages about the computation of the action of the algebra on the vector space under construction. At level 0 (the default) no messages are produced. All other levels produce copious output, with all levels above 2 being equivalent.

`LogCoincidences`	RngIntElt	*Default :* 0
`LogCoin`	RngIntElt	*Default :* 0

This option sets the level of messages about the coincidences discovered and the processing of them. At level 0 (the default) no messages are produced. At level 1 every coincidence and deduction is recorded when it is discovered and when it is processed. At level 2 or higher the operation of finding the undeleted image of a vector is also recorded.

`LogInitialization`	RngIntElt	*Default :* 0
`LogInitialisation`	RngIntElt	*Default :* 0
`LogInit`	RngIntElt	*Default :* 0

This option sets the level of messages about the initialisation of new basis vectors. At level 0 (the default) no messages are produced. All other levels produce a message whenever a new basis vector is defined.

`LogPacking`	RngIntElt	*Default :* 1
`LogPack`	RngIntElt	*Default :* 1

This option sets the level of messages about the reclamation of free space in the tables used by the algorithm. At level 0 no messages are produced. At level 1 (the

default) it produces a message each time the pack routine is called. At level 2 or higher it records the exact renaming used to reclaim the space.

`LogPushes`	RngIntElt	*Default :* 0
`LogPush`	RngIntElt	*Default :* 0

This option sets the level of messages about the pushing (or tracing) of (basis vector, relation) pairs. At level 0 (the default) it produces no messages. At level 1 a message is produced for each push that is started. At level 2 or higher the outcome of the push is also recorded.

`LogProgress`	RngIntElt	*Default :* 0
`LogStages`	RngIntElt	*Default :* 0

This option sets the level of messages about the overall progress of the algorithm. At level 0 no messages are produced. At level 1, simple messages are printed as the algorithm passes through its major stages. At level 2 the relations are printed as they are read in, and the complete action on the final module is printed. At level 3 or higher the action is also printed after the processing of submodule generators is complete.

`LogWeightChanges`	RngIntElt	*Default :* 1
`LogWt`	RngIntElt	*Default :* 1

This option sets the level of messages about changes in the current weight (ie the weight of (basis vector, relation pairs) currently being pushed. At level 0 no such messages are produced. At level 1 (the default) or higher a message giving the new weight and the current dimension is printed.

82.14.15 Miscellaneous

`QuotientModule(A, S)`

`Lookahead`	BoolElt	*Default :* `true`

This option controls whether, and to what extent, lookahead is used. If x is `false` then lookahead is not used. If x is `true`, the default, the lookahead by the default amount (two weights) is used. If x is a positive integer n then lookahead n weights is used. A sufficiently large value of n is equivalent to complete lookahead. Lookahead is commenced approximately every time the dimension doubles.

`EarlyClosing`	BoolElt	*Default :* `false`
`Early`	BoolElt	*Default :* `false`

This option permits the algorithm to stop as soon as the table represents a complete action (but see below), without checking to see whether the action satisfies all the relations. In practice this action is usually correct. The default behaviour is to continue and check all the relations.

`EarlyClosingMinimum`	RngIntElt	*Default :*
`ECMin`	RngIntElt	*Default :*

This option sets a minimum dimension at which the algorithm may stop without checking all the relators. It implies `EarlyClosing`.

`EarlyClosingMaximum`	RngIntElt	*Default :*
`ECMax`	RngIntElt	*Default :*

This option sets a maximum dimension at which the algorithm aamy stop without checking all the relators. It implies `EarlyClosing`.

`ConstructMorphism`	BoolElt	*Default :* `true`
`Morphism`	BoolElt	*Default :* `true`

This option controls whether the third return value of the `QuotientModule` function is in fact computed. A small overhead of time and space is required to compute it, and many applications do not need it, so this option is provided. When b is `true` (the default) the third return value is computed, when b is `false` it is not.

`ErrorOnFail`	BoolElt	*Default :*
`ErrFail`	BoolElt	*Default :*

This option controls the behaviour of the program if there is insufficient time or space to complete the calculation, or if the calculation has not been completed when the maximum weight is reached. If it is present a run-time error is generated, otherwise the value `undef` is returned.

Example H82E9

First we repeat the examples above, in Magma. The permutation action of D_8

```
> d8<a,b> := Group<a,b | a^4 = b^2 = (a*b)^2 = 1>;
> q := RationalField();
> a1<a,b> := FreeAlgebra(q,d8);
> i1 := rideal<a1 | b-1 >;
> mats, im, preim := QuotientModule(a1,i1);
Read Input
Done submodule generators
Starting weight 2 in define mode, 1 alive out of 2
Starting weight 3 in define mode, 1 alive out of 2
Looking ahead ...
Starting weight 4 in lookahead mode, 4 alive out of 5
Starting weight 5 in lookahead mode, 4 alive out of 5
    ...done
Packing 5 to 4
Starting weight 4 in define mode, 4 alive out of 4
Starting weight 5 in define mode, 4 alive out of 4
Starting weight 6 in define mode, 4 alive out of 4
Starting weight 7 in define mode, 5 alive out of 6
Starting weight 8 in define mode, 6 alive out of 7
Starting weight 9 in define mode, 4 alive out of 7
Starting weight 10 in define mode, 4 alive out of 7
Closed, 7 rows defined
```

```
Packing 7 to 4
4 live dimensions
Successful
> mats;
[
    [0 0 1 0]
    [1 0 0 0]
    [0 0 0 1]
    [0 1 0 0],

    [1 0 0 0]
    [0 0 1 0]
    [0 1 0 0]
    [0 0 0 1]
]
> im;
[
    (1 0 0 0)
]
> preim;
[ Id(), a^-1, a^-1 * b, a^-2 ]
>
```

Example H82E10

A quotient of that module.
We continue from the last example and set:

```
> d8<a,b> := Group<a,b | a^4 = b^2 = (a*b)^2 = 1>;
> q := RationalField();
> a1<a,b> := FreeAlgebra(q,d8);
> i2 := rideal<a1 | b-1, 1+a^3+a^3*b+a^2>;
> mats, im, preim := QuotientModule(a1,i2);
Read Input
Done submodule generators
Starting weight 2 in define mode, 4 alive out of 6
Starting weight 3 in define mode, 5 alive out of 7
Starting weight 4 in define mode, 3 alive out of 7
Starting weight 5 in define mode, 3 alive out of 7
Closed, 7 rows defined
Packing 7 to 3
3 live dimensions
Successful
> mats;
[
    [ 0  1  0]
    [ 0  0  1]
    [-1 -1 -1],
```

```
    [ 1  0  0]
    [-1 -1 -1]
    [ 0  0  1]
]
```

82.15 Bibliography

[**CLO96**] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms.* Undergraduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2nd edition, 1996.

[**Fau99**] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139 (1-3):61–88, 1999.

[**Li02**] Huishi Li. *Noncommutative Gröbner Bases and Filtered-Graded Transfer*, volume 1795 of *Lecture Notes in Math.* Springer-Verlag, Berlin–Heidelberg–New York, 2002.

[**Mor94**] Teo Mora. An introduction to commutative and noncommutative Gröbner bases. *Theoretical Computer Science*, 134:134–173, 1994.

83 MATRIX ALGEBRAS

Chapter 83
MATRIX ALGEBRAS

83.1 Introduction

Matrix algebras (or matrix rings) may be defined over any ring S. We shall regard such a matrix algebra as an *S-algebra*. Let us denote the complete algebra of $n \times n$ matrices over S by $M_n(S)$. It will often be convenient to regard $M_n(S)$ as the endomorphism ring of the free S-module $S^{(n)}$. We will then speak of $S^{(n)}$ as the natural S-module associated with $M_n(S)$.

Matrix algebras have type `AlgMat` and their elements have type `AlgMatElt`. These types inherit from `AlgMatV` and `AlgMatVElt` respectively (which cover both ordinary matrix algebras and matrix Lie algebras).

83.2 Construction of Matrix Algebras and their Elements

83.2.1 Construction of the Complete Matrix Algebra

`MatrixAlgebra(S, n)`

`MatrixRing(S, n)`

Given a positive integer n and a ring S, create the complete matrix algebra $M_n(S)$, consisting of all $n \times n$ matrices with coefficients in the ring S.

83.2.2 Construction of a Matrix

`elt< R | L >`

Given a matrix algebra defined as a subalgebra of $M_n(S)$, create the element of R defined by the list L of n^2 elements from S.

`R ! Q`

Given a matrix algebra R defined as a subalgebra of $M_n(S)$ and a sequence $Q = [a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{n1}, \ldots, a_{nn}]$ of n^2 elements of S, return the matrix

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{pmatrix}$$

as an element of R. Note that the algebra R must exist before an attempt is made to create matrices.

`CambridgeMatrix(t, K, n, Q)`

This function creates a n by n matrix over the finite field K of cardinality q specified in a "Cambridge" format in the general matrix algebra of degree n over K. The parameter t specifies the type of the format. If t is 1, then q is assumed to be less than 10 and the sequence Q must consist of n strings which give the n rows—each string must have length n and contain the entries of that row (each entry is a digit in the range $[0,\ q-1]$). If t is 3 then Q must consist of n^2 integers in the range $[0,\ q-1]$ which give the entries in row-major order. In either format, if $q = p^e$, where p is prime and $e > 1$, then an entry x is written as a vector using the base-p representation of length e of x and the corresponding element in K is used (see the Finite Fields chapter for details). This function is principally provided for the reading in of large matrices.

`CompanionMatrix(p)`

Given a monic polynomial p of degree n over a ring R, create the companion matrix C for p as an element of $M_n(R)$. The minimal and characteristic polynomial of C is then p.

`DiagonalMatrix(R, Q)`

If R is a subalgebra of $M_n(S)$ and Q is a sequence of n elements of S, create the diagonal matrix $diag(Q[1], Q[2], \ldots, Q[n])$.

`MatrixUnit(R, i, j)`

Create the matrix unit $E(i, j)$ in the matrix algebra R, i.e. the matrix having the one of the coefficient ring of R in position (i, j) and zeros elsewhere.

`Random(R)`

Create a random matrix of the matrix algebra R.

`ScalarMatrix(R, t)`

If R is a subalgebra of $M_n(S)$ and t is an element of the ring S, create the scalar matrix $t * I$ in R.

`R ! 1`

Create the identity matrix I_n of the matrix algebra R.

`R ! 0`

Create the zero matrix of the matrix algebra R.

`R ! t`

Create the scalar matrix $t * I$ of the matrix algebra R.

83.2.3 Constructing a General Matrix Algebra

```
MatrixAlgebra< S, n | L >
```

```
MatrixRing< S, n | L >
```

Given a commutative ring S and a positive integer n, create the S-algebra R consisting of the $n \times n$ matrices over the ring S generated by the elements defined in the list L. Let F denote the algebra $M_n(S)$. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n^2 elements of S defining an element of F.

(b) A set or sequence whose terms are sequences of type (a).

(c) An element of F.

(d) A set or sequence whose terms are elements of F.

(e) The null list.

The generators stored for R consist of the elements specified by terms L_i together with the stored generators for subalgebras specified by terms of L_i. Repetitions of an element and occurrences of scalar matrices are removed.

Example H83E1

We demonstrate the use of the matrix algebra constructor by creating an algebra of 3×3 lower-triangular matrices over the rational field.

```
> Q := RationalField();
> A := MatrixAlgebra< Q, 3 | [ 1/3,0,0, 3/2,3,0, -1/2,4,3],
>        [ 3,0,0, 1/2,-5,0, 8,-1/2,4] >;
> A:Maximal;
Matrix Algebra of degree 3 with 2 generators over Rational Field
Generators:
[ 1/3    0    0]
[ 3/2    3    0]
[-1/2    4    3]

[   3    0    0]
[ 1/2   -5    0]
[   8 -1/2    4]
> Dimension(A);
6
```

Example H83E2

We construct a 4 by 4 matrix over the finite field with 5 elements using the CambridgeMatrix function.

```
> K := FiniteField(5);
> x := CambridgeMatrix(1, K, 4, [ "1234", "0111", "4321", "1211" ]);
```

```
> x;
[1 2 3 4]
[0 1 1 1]
[4 3 2 1]
[1 2 1 1]
```

83.2.4 The Invariants of a Matrix Algebra

`R . i`

The i-th defining generator for the matrix algebra R.

`BaseRing(R)`

`CoefficientRing(R)`

The coefficient ring S for the matrix algebra R.

`Degree(R)`

Given a matrix algebra R, return the degree n of R.

`Generators(R)`

The set consisting of the defining generators for the matrix algebra R.

`Generic(R)`

The complete matrix algebra $M_n(S)$ in which the matrix algebra R is naturally embedded.

`BaseModule(R)`

If R is a subring of the matrix algebra $M_n(S)$, then R is considered to act on the free S-module of rank n, consisting of n-tuples over S. The function `BaseModule` returns this S-module.

`NumberOfGenerators(R)`

`Ngens(R)`

The number of defining generators for the matrix algebra R.

`Parent(a)`

Given an element a belonging to the matrix algebra R, return R, i.e. the parent structure for a.

Example H83E3

We illustrate the use of these functions by applying them to the algebra of 3×3 lower-triangular matrices over the rational field constructed above.

```
> Q := RationalField();
> A := MatrixAlgebra< Q, 3 | [ 1/3,0,0, 3/2,3,0, -1/2,4,3],
>        [ 3,0,0, 1/2,-5,0, 8,-1/2,4] >;
> CoefficientRing(A);
Rational Field
> Degree(A);
3
> Ngens(A);
2
> Generators(A);
{
    [ 1/3    0    0]
    [ 3/2    3    0]
    [-1/2    4    3],

    [   3    0    0]
    [ 1/2   -5    0]
    [   8 -1/2    4]
}
> Generic(A);
Full Matrix Algebra of degree 3 over Rational Field
> Dimension(A);
6
```

83.3 Construction of Subalgebras, Ideals and Quotient Rings

`sub< R | L >`

Given the matrix algebra R, defined as a subring of $M_n(S)$, construct the subring T of R generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) A sequence of n^2 elements of S defining an element of R;

(b) An element of R;

(c) A set or sequence of elements of R;

(d) A subring of R;

(e) A set or sequence of subrings of R.

Each element or subalgebra specified by the list must belong to the *same* complete matrix algebra. The subalgebra T will be constructed as a subalgebra of some

matrix algebra which contains each of the elements and subalgebras specified in the list.

The generators of T consist of the elements specified by the terms of the list L together with the stored generators for subalgebras specified by terms of the list. Repetitions of an element and occurrences of the identity element are removed (unless T is trivial).

The constructor returns the subalgebra T and the inclusion homomorphism $f : T \to R$.

`ideal< R | L >`

Given the matrix algebra R, construct the two-sided ideal I of R generated by the elements of R specified by the list L, where the possibilities for L are the same as for the `sub`-constructor.

`lideal< R | L >`

Given the matrix algebra R, construct the left ideal I of R generated by the elements of R specified by the list L, where the possibilities for L are the same as for the `sub`-constructor.

`rideal< R | L >`

Given the matrix algebra R, construct the right ideal I of R generated by the elements of R specified in the list L, where the possibilities for L are the same as for the `sub`-constructor.

Example H83E4

We construct the subalgebra of the matrix algebra A (defined above) that is generated by the first generator.

```
> Q := RationalField();
> A := MatrixAlgebra< Q, 3 | [ 1/3,0,0, 3/2,3,0, -1/2,4,3],
>        [ 3,0,0, 1/2,-5,0, 8,-1/2,4] >;
> B := sub< A | A.1 >;
> Dimension(B);
3
> B: Maximal;
Matrix Algebra of degree 3 and dimension 3 with 1 generator
over Rational Field
Generators:
[ 1/3    0    0]
[ 3/2    3    0]
[-1/2    4    3]

Basis:

[1 0 0]
[0 1 0]
```

```
[0 0 1]

[    0     0     0]
[    1  16/9     0]
[    0 88/27  16/9]

[   0    0    0]
[   0    0    0]
[   1 16/9    0]
```

83.4 The Construction of Extensions and their Elements

83.4.1 The Construction of Direct Sums and Tensor Products

`DirectSum(R, T)`

Given two matrix algebras R and T, where R and T have the same coefficient ring S, return the direct sum D of R and T (with the action given by the direct sum of the action of R and the action of T).

`TensorProduct(A, B)`

Given two unital matrix algebras A and B, where A and B have the same coefficient ring S, construct the tensor product of A and B.

Example H83E5

We construct the direct product and tensor product of the matrix algebra A (defined above) with itself.

```
> Q := RationalField();
> A := MatrixAlgebra< Q, 3 | [ 1/3,0,0, 3/2,3,0, -1/2,4,3],
>        [ 3,0,0, 1/2,-5,0, 8,-1/2,4] >;
> AplusA := DirectSum(A, A);
> AplusA: Maximal;
Matrix Algebra of degree 6 with 4 generators over
    Rational Field
Generators:
[ 1/3    0    0    0    0    0]
[ 3/2    3    0    0    0    0]
[-1/2    4    3    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]

[   3    0    0    0    0    0]
[ 1/2   -5    0    0    0    0]
[   8 -1/2    4    0    0    0]
```

```
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]

[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0  1/3    0    0]
[   0    0    0  3/2    3    0]
[   0    0    0 -1/2    4    3]

[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    0    0    0]
[   0    0    0    3    0    0]
[   0    0    0  1/2   -5    0]
[   0    0    0    8 -1/2    4]
> AtimesA := TensorProduct(A, A);
> AtimesA: Maximal;
Matrix Algebra of degree 9 with 4 generators over
    Rational Field
Generators:
[ 1/3    0    0    0    0    0    0    0    0]
[   0  1/3    0    0    0    0    0    0    0]
[   0    0  1/3    0    0    0    0    0    0]
[ 3/2    0    0    3    0    0    0    0    0]
[   0  3/2    0    0    3    0    0    0    0]
[   0    0  3/2    0    0    3    0    0    0]
[-1/2    0    0    4    0    0    3    0    0]
[   0 -1/2    0    0    4    0    0    3    0]
[   0    0 -1/2    0    0    4    0    0    3]

[   3    0    0    0    0    0    0    0    0]
[   0    3    0    0    0    0    0    0    0]
[   0    0    3    0    0    0    0    0    0]
[ 1/2    0    0   -5    0    0    0    0    0]
[   0  1/2    0    0   -5    0    0    0    0]
[   0    0  1/2    0    0   -5    0    0    0]
[   8    0    0 -1/2    0    0    4    0    0]
[   0    8    0    0 -1/2    0    0    4    0]
[   0    0    8    0    0 -1/2    0    0    4]

[ 1/3    0    0    0    0    0    0    0    0]
[ 3/2    3    0    0    0    0    0    0    0]
[-1/2    4    3    0    0    0    0    0    0]
[   0    0    0  1/3    0    0    0    0    0]
[   0    0    0  3/2    3    0    0    0    0]
[   0    0    0 -1/2    4    3    0    0    0]
```

```
[   0    0    0    0    0    0  1/3    0    0]
[   0    0    0    0    0    0  3/2    3    0]
[   0    0    0    0    0    0 -1/2    4    3]

[   3    0    0    0    0    0    0    0    0]
[ 1/2   -5    0    0    0    0    0    0    0]
[   8 -1/2    4    0    0    0    0    0    0]
[   0    0    0    3    0    0    0    0    0]
[   0    0    0  1/2   -5    0    0    0    0]
[   0    0    0    8 -1/2    4    0    0    0]
[   0    0    0    0    0    0    3    0    0]
[   0    0    0    0    0    0  1/2   -5    0]
[   0    0    0    0    0    0    8 -1/2    4]
```

83.4.2 Construction of Direct Sums and Tensor Products of Elements

`DirectSum(a, b)`

Given an element a of the matrix algebra Q and an element b of the matrix algebra R, form the direct sum of matrices a and b. The square is returned as an element of the matrix algebra T, which must be the direct sum of the parent of a and the parent of b.

`ExteriorSquare(a)`

Given an element a of the matrix algebra $M_n(S)$, form the exterior square of a as an element of $M_m(S)$, where $m = n(n-1)/2$.

`ExteriorPower(a,r)`

Given an element a of the matrix algebra $M_n(S)$, form the rth exterior power of a as an element of $M_m(S)$, where $\binom{m=n}{r}$.

`SymmetricSquare(a)`

Given an element a of the matrix algebra $M_n(S)$, form the symmetric square of a as an element of $M_m(S)$, where $m = n(n+1)/2$.

`SymmetricPower(a,r)`

Given an element a of the matrix algebra $M_n(S)$, form the rth symmetric power of a as an element of $M_m(S)$, for the appropriate m.

`TensorProduct(a, b)`

Given an element a belonging to a subalgebra of $M_{n_1}(S)$ and an element b belonging to a subalgebra of $M_{n_2}(S)$, construct the tensor product of a and b as an element of the matrix algebra $M_n(S)$, where $n = n_1 * n_2$.

83.5 Operations on Matrix Algebras

`Centre(A)`

Given a matrix algebra A whose base ring is a field, return the centre of A.

`Centralizer(A, S)`

Given a matrix algebra A whose base ring is a field, together with a subalgebra S of A return the centralizer of S in A.

83.6 Changing Rings

`ChangeRing(A, S)`

Given a matrix algebra A with base ring R, together with a ring S, construct the matrix algebra B with base ring S obtained by coercing the components of elements of A into S, together with the homomorphism from A to B.

`ChangeRing(A, S, f)`

Given a matrix algebra A with base ring R, together with a ring S and a homomorphism $f : R- > S$, construct the matrix algebra B with base ring S obtained by mapping the components of elements of R into S by f, together with the homomorphism from A to B.

`hom< A -> B | f >`

Given *full* matrix algebras A and B, together with a homomorphism f from the base ring R of A to the base ring S of B, create the homomorphism from A to B which, given a matrix in A, applies f to the entries of the matrix.

83.7 Elementary Operations on Elements

83.7.1 Arithmetic

`a + b`

Sum of the matrices a and b, where a and b belong to a common matrix algebra R.

`a + t`
`t + a`

Sum of the matrix a and the scalar matrix $t * I$.

`-a`

Negation of the matrix a.

`a - b`

Difference of the matrices a and b, where a and b belong to the same matrix algebra R.

`a - t`
`t - a`

Difference of the matrix a and the scalar matrix $t * I$.

`a * b`

Product of the matrices a and b, where a and b belong to the same matrix algebra R.

`a * b`

Given a matrix a belonging to a subalgebra of $M_n(S)$ and an element b of a submodule of $\mathrm{Hom}(R^{(n)}, R^{(m)})$, construct the product of a and b as an element of $\mathrm{Hom}(R^{(n)}, R^{(m)})$.

`a * b`

Given a matrix a belonging to a submodule of $\mathrm{Hom}(R^{(n)}, R^{(m)})$ and an element b of a subalgebra of $M_m(S)$, construct the product of a and b as an element of $\mathrm{Hom}(R^{(n)}, R^{(m)})$.

`t * a`
`a * t`

Given an element a of the matrix algebra R, and an element t belonging to the coefficient ring S of R, form their scalar product.

`u * a`

Given an element u belonging to the S-module $S^{(n)}$ and an element a belonging to a subalgebra of $M_n(S)$, form the element $u * a$ of S^n.

`a ^ n`

If n is positive, form the n-th power of a; if n is zero, form the identity matrix; if n is negative, form the $(-n)$-th power of the inverse of a.

`NumberOfColumns(a)`
`Ncols(a)`

The number of columns in the matrix a.

`NumberOfRows(a)`
`Nrows(a)`

The number of rows in the matrix a.

83.7.2 Predicates

83.7.2.1 Comparison

`a eq b`

Returns `true` if the matrix a is equal to the matrix b, where a and b are elements of a common matrix algebra R.

`a ne b`

Returns `true` if the matrix a is not equal to the matrix b, where a and b are elements of a common matrix algebra R.

83.7.2.2 Properties of Elements

The functions given here test properties of matrices. See also the section in the Lattices chapter for a description of the function `IsPositiveDefinite` and related functions.

`IsDiagonal(a)`

Returns `true` iff the element a belonging to the matrix algebra R is a diagonal matrix; i.e. the only non-zero entries are on the diagonal.

`IsMinusOne(a)`

Returns `true` iff the element a belonging to the matrix algebra R is the negation of the identity element for R.

`IsOne(a)`

Returns `true` iff the element a belonging to the matrix algebra R is the identity element for R.

`IsScalar(a)`

Returns `true` iff the element a belonging to the matrix algebra R is a scalar matrix.

`IsSymmetric(a)`

Returns `true` iff the element a belonging to the matrix algebra R is a symmetric matrix; i.e. the transpose of a equals a.

`IsUnit(a)`

Returns `true` iff the matrix a belonging to the matrix algebra R is a unit.

`IsZero(a)`

Returns `true` iff the element a belonging to the matrix algebra R is the zero element for R.

`IsNilpotent(a)`

Return `true` if some power of the matrix a belonging to a matrix algebra is the zero of the matrix algebra. Also returns the minimum exponent n such that $a^n = 0$.

`IsUnipotent(a)`

Return `true` if the matrix a belonging to a matrix algebra is the identity of that algebra plus a nilpotent matrix. Also returns the index of nilpotence of $a - I$.

`Rank(a)`

Return the rank of the element a belonging to the matrix algebra R.

`Determinant(A)`

`MonteCarloLevel`	RngIntElt	*Default :* 0
`Proof`	BoolElt	*Default :* `true`
`pAdic`	BoolElt	*Default :* `true`
`Divisor`	RngIntElt	*Default :* 0

Given a square matrix A over the ring R, return the determinant of A as an element of R. R may be any commutative ring. The determinant of the 0×0 matrix over R is defined to be `R!1`.

If the coefficient ring is the integer ring $\mathbf{Z}$ or the rational field $\mathbf{Q}$ then a modular algorithm based on that of Abbott et al. [ABM99] is used, which first computes a divisor d of the determinant D using a fast p-adic nullspace computation, and then computes the quotient D/d by computing the determinant D modulo enough small primes to cover the Hadamard bound divided by d. This always yields a correct answer.

If the parameter `MonteCarloLevel` is set to a small positive integer s, then a probabilistic Monte-Carlo modular technique is used. Rather than using sufficient primes to cover the Hadamard bound divided by the divisor d, this version of the algorithm terminates when the constructed residue remains constant for s steps. The probability of this being wrong is non-zero but extremely small, even if s is only 1 or 2. If the level is set to 0, then the normal deterministic algorithm is used. Setting the parameter `Proof` to `false`is equivalent to setting `MonteCarloLevel` to 2.

If the coefficient ring is $\mathbf{Z}$ and the parameter `Divisor` is set to an integer d, then d must be a known exact divisor of the determinant (the sign does not matter), and the algorithm may be sped up because of this knowledge.

`Trace(a)`

Given an element a of a subalgebra of $M_n(S)$, return the trace of a as an element of S.

`Transpose(a)`

Given an element a of a subalgebra of $M_n(S)$, return the transpose of a as an element of $M_n(S)$.

`Order(a)`

Given an invertible matrix a over any commutative ring, determine the order of a. If a has infinite order, the function may become stuck indefinitely since it cannot prove such.

`FactoredOrder(a)`

Given an invertible matrix a over a finite field, return the order of a in factored form.

`ProjectiveOrder(a)`

Given an invertible matrix a over a finite field, return the projective order o of a and a scalar s such that $a^o = sI$.

`FactoredProjectiveOrder(a)`

Given an invertible matrix a over a finite field, return the projective order o of a in factored form and a scalar s such that $a^o = sI$.

`CharacteristicPolynomial(a:` *parameters*`)`

`Al`	MonStgElt	*Default :* "*Modular*"
`Proof`	BoolElt	*Default :* `true`

The characteristic polynomial of the element a belonging to the algebra $M_n(R)$, where R can be any commutative ring. The parameter `Al` may be used to specify the algorithm used. The algorithm `Modular` (the default) can be used for matrices over Z and Q—in such a case the parameter `Proof` can also be used to suppress proof of correctness. The algorithm `Hessenberg`, allowed for matrices over fields, works by first reducing the matrix to Hessenberg form. The algorithm `Interpolation`, allowed for matrices over Z and Q, works by evaluating the characteristic matrix of a at various points and then interpolating. The algorithm `Trace`, allowed for matrices over fields, works by calculating the traces of powers of a.

`MinimalPolynomial(a)`

The minimal polynomial of the element a belonging to the module $M_n(R)$, where R is a field or Z.

`HessenbergForm(a)`

The Hessenberg form for the matrix a belonging to the algebra $M_n(K)$, where the coefficient ring K must be a field. The form has zero entries above the super-diagonal. (This form is used in one of the characteristic polynomial algorithms.)

`Adjoint(a)`

The adjoint of the matrix a belonging to the algebra $M_n(K)$, where the coefficient ring K must be a ring with exact division whose characteristic must be zero or greater than the degree of a.

`Eigenvalues(a)`

The eigenvalues of the matrix a returned as a set of pairs, each of which gives the value of a distinct eigenvalue and its multiplicity. The coefficient ring must have a polynomial roots algorithm.

`Eigenspace(a, e)`

The eigenspace of the matrix a, corresponding to the eigenvalue e, returned as a submodule of the base module for the parent algebra of a (i.e. the kernel of $a - eI$). If the ring element e is not a eigenvalue for the matrix a then the trivial space is returned.

83.8 Elements of M_n as Homomorphisms

The matrix algebra $M_n(S)$ may also be viewed as the module $\mathrm{Hom}(S^{(n)}, S^{(n)})$. At present this will not happen automatically so that in order to treat elements of $M_n(S)$ as homomorphisms, it is necessary to explicitly coerce the matrix into $\mathrm{Hom}(S^{(n)}, S^{(n)})$. However, two fundamental homomorphism-type operators are provided for elements of $M_n(S)$.

`Image(a)`
`RowSpace(a)`

Given an element of $M_n(S)$, return the image of the module $S^{(n)}$ under the homomorphism represented by the matrix a (as an element of $S^{(n)}$).

`Kernel(a)`
`NullSpace(a)`

`Al` MonStgElt *Default :* "*Default*"

Given an element of $M_n(S)$, return the kernel of the homomorphism represented by the matrix a (as an element of $S^{(n)}$).

`RowNullSpace(a)`
`NullspaceOfTranspose(a)`

Given an element of $M_n(S)$, return the row nullspace of the homomorphism represented by the matrix a (as an element of $S^{(n)}$). This is equal to the kernel of the transpose of a.

83.9 Elementary Operations on Subalgebras and Ideals

83.9.1 Bases

The functions described here assume that the matrix algebra R is defined over a ring S with a matrix echelonization algorithm. MAGMA computes a basis for R considered as a S-module when necessary so then operations like membership testing can be performed. The following functions allow one to access this basis.

`Dimension(R)`

Assuming that R is a subalgebra of $M_n(S)$, return the dimension of R, considered as a S-module.

`Basis(R)`

Assuming that R is a subalgebra of $M_n(S)$, return the S-basis of R, considered as a S-module. The basis is returned as a sequence of matrices of R.

`BasisElement(R, i)`

Given R a subalgebra of $M_n(S)$, return the i-th element of the S-basis of R, where i must be between 1 and the dimension of R.

`Coordinates(R, X)`

Assuming that R is a subalgebra of $M_n(S)$, and given an element X of R, return the coordinates of X with respect to the basis of R. If R has dimension k over its coefficient ring S, and R has basis $U_1, \ldots, U_k$, the coordinates are returned as the unique sequence $[a_1, \ldots, a_k]$ of elements of S such that $X = a_1U_1 + \ldots + a_rU_r$.

83.9.2 Intersection of Subalgebras

`R meet T`

Given algebras R and S that are subalgebras of the same complete algebra $M_n(S)$, where S is a PIR, this operator constructs their intersection.

83.9.3 Membership and Equality

The operations described here assume that the matrix algebra is defined over a principal ideal ring.

`x in R`
`X subset R`
`T subset R`

Given a matrix x (set of matrices X, matrix algebra T) and a matrix algebra R all belonging to a common matrix algebra defined over a PIR, return `true` if x (X, T, respectively) is contained in R, `false` otherwise.

`x notin R`
`X notsubset R`
`T notsubset R`

Given a matrix x (set of matrices X, matrix algebra T) and a matrix algebra R all belonging to a common matrix algebra defined over a PIR, return `true` if x (X, T, respectively) is not contained in R, `false` otherwise.

`R eq T`

Given a matrix algebra R, and a matrix algebra T, return `true` if R is equal to T, `false` otherwise.

`R ne T`

Given a matrix algebra R and a matrix algebra T, return `true` if R is not equal to T, `false` otherwise.

83.10 Accessing and Modifying a Matrix

83.10.1 Indexing

`a[i]`

Given an element a belonging to the matrix algebra R over the ring S, return the i-th row of a as an element of the natural S-module associated with R.

`a[i] := u`

Given an element a belonging to the matrix algebra R over the ring S, an integer i in the range $[1, n]$ and an element u of the natural S-module associated with R, replace the i-th row of a by the vector u.

`a[i, j]`

Given an element a belonging to the matrix algebra R over the ring S, return the (i, j)-th entry of a as an element of S.

`a[i, j] := t`

Given an element a belonging to the matrix algebra R over the ring S, integers i and j in the range $[1, n]$, and an element t of S, replace the (i, j)-th entry of a by t.

`ElementToSequence(a)`
`Eltseq(a)`

Given an element a of the matrix algebra R over S, where $a = (a_{ij})$, $1 \leq i, j \leq n$, return a as the sequence of elements of S:

$$[a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2n}, \ldots, a_{n1}, \ldots, a_{nn}].$$

83.10.2 Extracting and Inserting Blocks

`Submatrix(a, i, j, p, q)`

`ExtractBlock(a, i, j, p, q)`

Given a matrix a belonging to a subalgebra of $M_n(S)$ and integers i, j, p and q satisfying the conditions, $1 \leq i + p \leq m$, $1 \leq j + q \leq n$, create the matrix b consisting of the $p \times q$ submatrix of a whose first entry is the (i, j)-th entry of a. If $p \neq q$, the matrix b is created as an element of $\mathrm{Hom}(P, Q)$, where $\mathrm{Rank}(P) = p$, $\mathrm{Rank}(Q) = q$. Otherwise it is created as an element of $M_p(S)$.

`InsertBlock(~a, b, i, j)`

(Procedure.) Given that the matrix a belongs to a subalgebra of $M_n(S)$ and the $p \times q$ matrix b is also over S, the integers i, j, p and q must satisfy the conditions, $1 \leq i + p \leq m$, $1 \leq j + q \leq n$. This procedure modifies a so that the $p \times q$ block beginning at the (i, j)-th entry of a is replaced by b.

83.10.3 Joining Matrices

`HorizontalJoin(X, Y)`

Given matrices X with r rows and c columns, and Y with r rows and d columns, both over the same coefficient ring R, return the matrix over R with r rows and $(c + d)$ columns obtained by joining X and Y horizontally (placing Y to the right of X).

`HorizontalJoin(Q)`

Given a sequence Q of matrices, each having the same number of rows and being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q horizontally in order.

`VerticalJoin(X, Y)`

Given matrices X with r rows and c columns and Y with s rows and c columns, both over the same coefficient ring R, return the matrix with $(r + s)$ rows and c columns over R obtained by joining X and Y vertically (placing Y underneath X).

`VerticalJoin(Q)`

Given a sequence Q of matrices, each having the same number of columns and being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q vertically in order.

`DiagonalJoin(X, Y)`

Given matrices X with a rows and b columns and Y with c rows and d columns, both over the same coefficient ring R, return the matrix with $(a+c)$ rows and $(b+d)$ columns over R obtained by joining X and Y diagonally (placing Y diagonally to the right of and underneath X, with zero blocks above and below the diagonal).

`DiagonalJoin(Q)`

Given a sequence Q of matrices, each being over the same coefficient ring R, return the matrix over R obtained by joining the elements of Q diagonally in order.

83.10.4 Row and Column Operations

For the following operations, a is an element of a subring of the matrix algebra $M_n(S)$, u is a non-zero element of S, and i and j are integers in the range $[1, n]$. Each of the operations described here acts on the matrix in place, and is therefore implemented as a procedure.

`SwapRows(~a, i, j)`

Mutate the matrix a by interchanging rows i and j.

`MultiplyRow(~a, u, j)`

Mutate the matrix a by multiplying row j by the scalar u.

`AddRow(~a, u, i, j)`

Mutate the matrix a by adding u times row i to row j.

`SwapColumns(~a, i, j)`

Mutate the matrix a by interchanging columns i and j.

`MultiplyColumn(~a, u, i)`

Mutate the matrix a by multiplying column i by the scalar u.

`AddColumn(~a, u, i, j)`

Mutate the matrix a by adding u times column i to column j.

83.11 Canonical Forms

83.11.1 Canonical Forms for Matrices over Euclidean Domains

The functions given here apply to matrices defined over Euclidean Domains. See also the section on Reduction in the Lattices chapter for a description of the function `LLL` and related functions.

`EchelonForm(a)`

The (row) echelon form of matrix a belonging to a submodule of the module $M_n(S)$. This function returns two values:

(a) The (row) echelon form e of a; and

(b) A matrix b such that $b * a = e$, i.e. b is a product of elementary matrices that transforms a into echelon form.

`ElementaryDivisors(a)`

The elementary divisors of the matrix a belonging to a submodule of the module $M_n(S)$. The divisors are returned as a sequence $[e_1, \ldots, e_d]$, $e_i | e_{i+1}$ $(i = 1, \ldots, d-1)$ of d elements of R (which may include ones), where d is the rank of a. If R is a field, the result is always the sequence of r ones, where r is the rank of a.

`HermiteForm(X)`

`Al`	MonStg	*Default :* "*LLL*"
`Optimize`	BoolElt	*Default :* `true`
`Integral`	BoolElt	*Default :* `true`

The row Hermite normal form of an matrix X belonging to the matrix algebra $M_n(R)$. The coefficient ring R must be an Euclidean domain. This function returns two values:

(a) The Hermite normal form H of X; and

(b) A unimodular matrix T such that $T \cdot X = H$, i.e., T is the product of elementary matrices which transforms X into Hermite normal form.

If R is the ring of integers $\mathbf{Z}$ and the matrix T is requested (i.e., if an assignment statement is used with two variables on the left side), then the LLL algorithm will be used by default to improve T (using the kernel of X) so that the size of its entries are very small. If the parameter `Optimize` is set to `false`, then this will not happen (which will be faster but the entries of T will not be as small). If the parameter `Integral` is set to `true`, then the integral (de Weger) LLL method will be used in the LLL step, instead of the default floating point method. The integral method will often be faster if the rank of the kernel of X is very large (say 200 or more).

If R is the ring of integers $\mathbf{Z}$ and the parameter `Al` is set to the string `"Sort"`, then the sorting-gcd algorithm will be used. However, the new algorithm will practically always perform better than the sorting-gcd algorithm.

`SmithForm(a)`

The Smith normal form for the matrix a belonging to a submodule of the module $M_n(S)$, where S is a Euclidean Domain. This function returns three values:

(a) The Smith normal form s of a; and

(b) Unimodular matrices b and c such that $b * a * c = s$, i.e. b and c are matrices that transform a into Smith normal form.

Example H83E6

We illustrate some of these operations in the context of the algebra $M_4(K)$, where K is the field $\mathbf{F}_8$.

```
> K<w> := FiniteField(8);
> M := MatrixAlgebra(K, 4);
> A := M ! [1, w, w^5, 0,  w^3, w^4, w, 1,  w^6, w^3, 1, w^4, 1, w, 1, w ];
```

```
> A;
[  1   w w^5   0]
[w^3 w^4   w   1]
[w^6 w^3   1 w^4]
[  1   w   1   w]
> EchelonForm(A);
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

83.11.2 Canonical Forms for Matrices over a Field

The functions in this group apply to elements of matrix algebras whose coefficient rings are fields which allow factorization of univariate polynomials over them.

`PrimaryRationalForm(a)`

The primary rational canonical form of a matrix a belonging to $M_n(K)$, where the coefficient ring K must be a field allowing factorization of univariate polynomials over it. Each block corresponds to a power of an irreducible polynomial. This function returns three values:

(a) The primary rational canonical form p of a;

(b) A non-singular matrix t such that $t * a * t^{-1} = p$;

(c) A sequence of pairs corresponding to the blocks of p where each pair consists of the irreducible polynomial and multiplicity making up the block.

`JordanForm(a)`

The (generalized) Jordan canonical form for the matrix a belonging to the algebra $M_n(K)$, where the coefficient ring K must be a field allowing factorization of univariate polynomials over it. This function returns three values:

(a) The (generalized) Jordan canonical form j of a;

(b) A non-singular matrix t such that $t * a * t^{-1} = j$;

(c) A sequence of pairs corresponding to the blocks of j where each pair consists of the irreducible polynomial and multiplicity making up the block.

`RationalForm(a)`

The rational canonical form of a matrix a belonging to $M_n(K)$, where the coefficient ring K must be a field allowing factorization of univariate polynomials over it. For each block before the last block, the polynomial corresponding to that block divides the polynomial corresponding to the next block. This function returns three values:

(a) The rational canonical form f of a;

(b) A non-singular matrix t such that $t * a * t^{-1} = f$;

(c) A sequence containing the polynomials corresponding to each block (each non-last one dividing the next).

`PrimaryInvariantFactors(a)`

The primary invariant factors of the matrix a. This is the same as the third return value of `PrimaryRationalForm(a)` or `JordanForm(a)`. The coefficient ring must be a field allowing factorization of univariate polynomials over it.

`InvariantFactors(a)`

The invariant factors of the matrix a. This is the same as the third return value of `RationalForm(a)`. The coefficient ring must be a field allowing factorization of univariate polynomials over it.

`IsSimilar(a, b)`

Returns `true` iff the matrix a is similar to the matrix b. If a is similar to b, a transformation matrix t is also returned with $t * a * t^{-1} = b$. The coefficient ring must be a field allowing factorization of univariate polynomials over it.

Example H83E7

We consider the algebra $M_5(P)$, where P is the polynomial ring in indeterminate x over the field $\mathbf{F}_5$. We take the matrix having $x^i + x^j$ in its (i, j)-th position.

```
> K := GaloisField(5);
> P<x> := PolynomialAlgebra(K);
> M := MatrixAlgebra(P, 5);
> a := M ! [x^i + x^j: i, j in [1..5]];
> a;
[      2*x   x^2 + x   x^3 + x   x^4 + x   x^5 + x]
[  x^2 + x     2*x^2 x^3 + x^2 x^4 + x^2 x^5 + x^2]
[  x^3 + x x^3 + x^2     2*x^3 x^4 + x^3 x^5 + x^3]
[  x^4 + x x^4 + x^2 x^4 + x^3     2*x^4 x^5 + x^4]
[  x^5 + x x^5 + x^2 x^5 + x^3 x^5 + x^4     2*x^5]
> ElementaryDivisors(a);
[
    x,
    x^3 + 3*x^2 + x
]
> Rank(a);
```

2

Example H83E8

We construct a 5 by 5 matrix over the finite field with 5 elements and then calculate various canonical forms. We verify the correctness of the polynomial invariant factors corresponding to the rational form by calculating the Smith form of the characteristic matrix of the original matrix.

```
> K := GF(5);
> P<x> := PolynomialRing(K);
> A := MatrixAlgebra(K, 5);
> a := A !
> [
>     0, 2, 4, 2, 0,
>     2, 2, 2, 3, 3,
>     3, 4, 4, 1, 3,
>     0, 0, 0, 0, 1,
>     0, 0, 0, 1, 0
> ];
> a;
[0 2 4 2 0]
[2 2 2 3 3]
[3 4 4 1 3]
[0 0 0 0 1]
[0 0 0 1 0]
> PrimaryInvariantFactors(a);
[
    <x + 1, 1>,
    <x + 1, 1>,
    <x + 4, 1>,
    <x + 4, 1>,
    <x + 4, 1>
]
> r, t, f := RationalForm(a);
> r;
[1 0 0 0 0]
[0 0 1 0 0]
[0 1 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
> t;
[1 3 0 2 1]
[2 1 2 2 0]
[3 4 3 4 1]
[1 0 0 0 0]
[0 2 4 2 0]
> f;
[
```

```
    x + 4,
    x^2 + 4,
    x^2 + 4
]
> PA := MatrixAlgebra(P, 5);
> ax := PA ! x - PA ! a;
> ax;
[    x     3     1     3     0]
[    3 x + 3     3     2     2]
[    2     1 x + 1     4     2]
[    0     0     0     x     4]
[    0     0     0     4     x]
> SmithForm(ax);
[      1       0       0       0       0]
[      0       1       0       0       0]
[      0       0   x + 4       0       0]
[      0       0       0 x^2 + 4       0]
[      0       0       0       0 x^2 + 4]
> ElementaryDivisors(ax);
[
    1,
    1,
    x + 4,
    x^2 + 4,
    x^2 + 4
]
```

83.12 Diagonalising Commutative Algebras over a Field

The functions in this group apply to (elements of) matrix algebras whose coefficient rings are fields which allow the construction of splitting fields of univariate polynomials over them. Specifically, the base field must be the rationals, a number field, a finite field, or an algebraically closed field.

`CommonEigenspaces(Q)`

The common eigenspaces of the sequence Q of pairwise-commuting matrices. If the first sequence returned is V and the second is E, then $E[i][j]$ is the eigenvalue of $Q[i]$ corresponding to the common eigenspace $V[j]$.

`CommonEigenspaces(A)`

The common eigenspaces of the commutative matrix algebra A. If the first sequence returned is V and the second is E, then $E[i][j]$ is the eigenvalue of $A.i$ corresponding to the common eigenspace $V[j]$.

`Diagonalisation(Q)`

`Diagonalization(Q)`

The diagonalisation of the sequence Q of pairwise-commuting matrices. That is, a sequence of diagonal matrices of the form $[P * Q[1] * P^{-1}, P * Q[2] * P^{-1}, \ldots]$. The second value returned is the matrix P. Note that the returned values may have a larger base field than the input.

`Diagonalisation(A)`

`Diagonalization(A)`

The diagonalisation of the commutative matrix algebra A. That is, an algebra with diagonal generators $[P * A.1 * P^{-1}, P * A.2 * P^{-1}, \ldots]$. The second value returned is the matrix P.

Example H83E9

```
> M := MatrixAlgebra(Rationals(),2);
> x := M![0,1,-2,0];
> y := M![0,3,-6,0];
> CommonEigenspaces([x,y]);
[*
    Vector space of degree 2, dimension 1 over Number Field with defining
    polynomial $.1^2 + 2 over the Rational Field
    Generators:
    (      1 1/2*r.1)
    Echelonized basis:
    (      1 1/2*r.1),
    Vector space of degree 2, dimension 1 over Number Field with defining
    polynomial $.1^2 + 2 over the Rational Field
    Generators:
    (       1 -1/2*r.1)
    Echelonized basis:
    (       1 -1/2*r.1)
*] [
    [
        -r.1,
        -3*r.1
    ],
    [
        r.1,
        3*r.1
    ]
]
> Diagonalisation(sub<M|x,y>);
Matrix Algebra of degree 2 with 2 generators over Number Field with defining
polynomial $.1^2 + 2 over the Rational Field
```

```
[       1  1/2*r.1]
[       1 -1/2*r.1]
```

83.13 Solutions of Systems of Linear Equations

`IsConsistent(A, w)`

Given a matrix A belonging to $M_n(R)$ and a vector w belonging to the tuple module $R^{(n)}$, return `true` iff the system of linear equations $v * A = w$ is consistent. If the system is consistent, then the function will also return:

(a) A particular solution v;

(b) The kernel K of A so that $(v + k) * A = w$ for $k \in K$.

`IsConsistent(A, W)`

Given a matrix A belonging to $M_n(R)$ and a sequence W of vectors belonging to the tuple module $R^{(m)}$, return `true` iff the system of linear equations $V[i] * A = W[i]$ for each i is consistent. If the systems are all consistent, then the function will also return:

(a) A solution sequence V;

(b) The kernel K of A so that $(V[i] + k) * A = W[i]$ for $k \in K$.

`Solution(A, w)`

Given a matrix A belonging to $M_n(R)$ and a vector w belonging to the tuple module $R^{(n)}$, solve the system of linear equations $v*A = w$. The function returns two values:

(a) A particular solution v;

(b) The kernel K of A so that $(v + k) * A = w$ for $k \in K$.

`Solution(A, W)`

Given a matrix A belonging to $M_n(R)$ and a sequence W of vectors belonging to the tuple module $R^{(n)}$, solve the system of linear equations $V[i] * A = W[i]$ for each i. The function returns two values:

(a) A solution sequence V;

(b) The kernel K of A so that $(V[i] + k) * A = W[i]$ for $k \in K$.

83.14 Presentations for Matrix Algebras

Magma has the capability of constructing a presentation for an algebra A generated by a collection $\alpha_1, \ldots, \alpha_t$ of $n \times n$ matrices over a finite field k. The presentation has the form

$$A \cong P/I$$

where P is a free algebra in noncommuting variable and I is a two-sided ideal in P. The presentation is obtained by computing a set of primitive idempotents for the algebra and extracting generators for the radical of the algebra. For such an algebra there is a sequence

$$0 \longrightarrow \mathrm{Rad}(A) \longrightarrow A \longrightarrow A/\mathrm{Rad}(A) \longrightarrow 0$$

which is split in the sense that $A/\mathrm{Rad}(A)$ is a subalgebra A. Moreover, $A/\mathrm{Rad}(A)$ is a direct sum of complete matrix algebras A_i over extensions K_i of the field k. Each complete matrix algebra A_i is generated by two elements b_i and t_i. The element b_i has minimal rank and $b_i^{n_i-1} = e_i$ is a primitive idempotent where n_i is the number of elements in the field K_i. That is, b_i is a generator for the multiplicative group of nonzero elements of $e_i A_i e_i \cong K_i$. The element t_i is conjugate to a permutation matrix of degree n_i in A_i. The elements $z_1, \ldots, z_s$ are generators of the radical of A. These elements are computed so as to lie in the condensed algebra eAe where $e = \sum e_i$.

The calculation produces also a set of generators and relations for the condensed algebra eAe. This algebra is Morita equivalent to A, and hence shares many of the same homological properties of the algebra A.

In the course of obtaining the presentation, several aspects of the algebra are computed.

83.14.1 Quotients and Idempotents

`NaturalFreeAlgebraCover(A)`

Returns the map of a free algebra onto the matrix algebra A, such that the variables of the free algebra go to the generators of A.

`SimpleQuotientAlgebras(A)`

The simple quotient algebras of the matrix algebra A. The output is a record having the following fields:

(a) The actual simple algebras that are the quotients of the algebra A. The function returns the sequence of mappings from the natural free-algebra cover of A to the quotients. The variable corresponding to a generator of A is mapped to the corresponding generator of the quotient. (field name `SimpleQuotients`)

(b) The degrees of the quotients as matrix algebras over their centers (field name `DegreesOverCenters`).

(c) The degrees of the extension of center of the quotient algebra over the base field of A (field name `DegreesOfCenters`).

(d) The number of elements in the center of the quotient algebra (field name `OrdersOfCenters`).

`PrimitiveIdempotentData(A)`

The initial data for a decomposition of the matrix algebra A. The output is a sequence of records, one for each simple quotient algebra of A, each consisting of the following fields.

(a) `AlgebraIdempotent` : An idempotent whose image in the simple quotient is the identity matrix.

(b) `PrimitiveIdempotent` : A primitive idempotents whose image in the quotient is primitive.

(c) `PrimitiveIdempotentOnQuotient` : The image of the primitive idempotent in the quotient algebra.

(d) `FieldGenerator` : A multiple of the primitive idempotent which is a field generator for the center of the algebra.

(e) `FieldGeneratorOnQuotient` : The image of the field generator in the quotient algebra.

(f) `GeneratingPolForCenter` : The minimal polynomial for the matrix of the field generator.

`PrimitiveIdempotents(A)`

A list of primitive idempotent for the matrix algebra A, one idempotent for each irreducible module.

`RanksOfPrimitiveIdempotents(A)`

The sequence of ranks of the primitive idempotents for the matrix algebra A.

`NaturalFreeAlgebraCover(A)`

Returns the map of a free algebra onto the matrix algebra A, such that the variables of the free algebra go the generators of the algebra.

`CondensedAlgebra(A)`

Returns the algebra eAe where e is a sum of primitive idempotents, one for each simple A-module.

Example H83E10

We form a matrix algebra over the field with three elements generated by two elements. The algebra is block upper triangular, where the upper left block is a field extension of degree three and the lower block is a filed extension of degree 2.

```
> a1 := KMatrixSpace(GF(3),3,3)![0,1,0,0,0,1,-1,0,1];
> a2 := KMatrixSpace(GF(3),2,2)![0,1,-1,0];
> z1 := KMatrixSpace(GF(3),5,5)!0;
> z2 := InsertBlock(z1,a1,1,1);
> z2;
[0 1 0 0 0]
```

```
[0 0 1 0 0]
[2 0 1 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
> z3 := InsertBlock(z1,a2,4,4);
```

$z2$ and $z3$ are the matrices of the field extensions. Next, add an entry to $z3$ that is an extension class between the two algebras.

```
> z3[1][4] := 1;
> z3;
[0 0 0 1 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 1]
[0 0 0 2 0]
> A := MatrixAlgebra<GF(3),5|z2,z3>;
```

We can check to see if the quotients came out as expected.

```
> SimpleQuotientAlgebras(A);
rec<recformat<SimpleQuotients: SeqEnum, DegreesOverCenters:
SeqEnum, DegreesOfCenters: SeqEnum, OrdersOfCenters: SeqEnum> |
    SimpleQuotients := [
        Mapping from: Free associative algebra of rank 2 over
                GF(3) to Matrix Algebra of degree 3 with 2
            generators over GF(3),
        Mapping from: Free associative algebra of rank 2 over
                GF(3) to Matrix Algebra of degree 2 with 2
            generators over GF(3)
    ],
    DegreesOverCenters := [ 1, 1 ],
    DegreesOfCenters := [ 3, 2 ],
    OrdersOfCenters := [ 27, 9 ]
```

Here are the idempotents.

```
> PrimitiveIdempotents(A);
[
    [1 0 0 0 0]
    [0 1 0 0 0]
    [0 0 1 0 2]
    [0 0 0 0 0]
    [0 0 0 0 0],
    [0 0 0 0 0]
    [0 0 0 0 0]
    [0 0 0 0 1]
    [0 0 0 1 0]
    [0 0 0 0 1]
```

```
]
```

Finally we have the Cartan matrix.

```
> CartanMatrix(A);
[1 3]
[0 1]
```

83.14.2 Generators and Presentations

We use the idempotents as the first step in the presentation process. The generators of the algebra consist of the field generators, each of which is a generator of the center of a simple quotient algebra multiplied by a corresponding primitive idempotent, permutation matrices, one for each simple quotient algebra and generators for the radical, all of which can be taken to be in the subalgebra eAe where e is the sum of one primitive idempotent for each simple quotient algebra.

For display purposes the variables for the free algebra of the presentations are given names. The variables $b.i$ represent field generators, while the $t.i$'s are permutation matrices and the $z.i$'s are the generators of the radical. The reader should be warned that these names are not suitable for input into other function.

`SemisimpleGeneratorData(A)`

The data on the semisimple generators of the algebra A, that is the generators in A of $A/Rad(A)$. Some of the output is intended for use in other functions. The return is a sequence of records, one for each simple quotient algebra. Each record consists of the following fields.

(a) `PowersOfFieldGenerators` : A record consisting of look-up tables for the powers of the field generators as elements both of the algebra A and the simple quotient algebra.

(b) `Permutation` : The permutation matrix of the quotient algebra as an element of A.

(c) `PermutationOnQuotient` : The permutation matrix on the quotient.

(d) `FieldGenerator` : The matrix of the field generator as an element of A.

(e) `FieldGeneratorOnQuotient` : The matrix of the field generator as an element of the quotient algebra.

(f) `PrimitiveIdempotent` : The matrix of the primitive idempotent on A.

(g) `PrimitiveIdempotentOnQuotient` : The matrix of the primitive idempotent on the quotient algebra.

(h) `GeneratingPolForCenter` : The Galois polynomial for the extension of the center of the quotient algebra over the base ring.

`AlgebraGenerators(A)`

The standard generators of the matrix algebra A. The output is a record consisting of the following fields.

(a) `FieldGenerators` : The sequence of matrices of the field generators, one for each simple quotient algebra.

(b) `PermutationMatrices` : The sequence of permutation matrices for the quotient algebras as elements of A, one for each quotient algebra.

(c) `PrimitiveIdempotents` : The sequence of primitive idempotents for A, one for each quotient algebra.

(d) `RadicalGenerators` : A set of generators for the radical of A arranged as a list of lists of generators of the radical of e_iAe_j for e_i and e_j primitive idempotents.

(e) `SequenceRadicalGenerators` : A sequence of minimal generators of radical of A.

(f) `GeneratingPolynomialsForCenters` : The galois polynomials of the centers over the base field.

(g) `StandardFormConjugationMatrices` : The matrices which conjugate the element of A into the standard form relative to the computed primitive idempotents for A.

`AlgebraStructure(A)`

The accumulated structure of the matrix algebra A. The return is a record with the following fields. Some of this information is saved to be used in other calculations.

(a) `FreeAlgebra` : The free algebra in the newly computed variables for the algebra.

(b) `RelationsIdeal` : The ideal of relations among the new computed generators.

(c) `StandardFreeAlgebraCover` : The map from the free algebra to A.

(d) `FieldGenerators` : The matrices in A of the generators of the centers of the simple quotient algebras of A.

(e) `PermutationMatrices` : The permutation matrices each of which together with the corresponding field generator, generates a simple quotient algebra as a subalgebra of A.

(f) `PrimitiveIdempotents` : The primitive idempotents, each of which is a power of the corresponding field generator.

(g) `RadicalGenerators` : A list of lists giving the generators of the radical of A which are in e_iAe_j where $\{e_i\}$ are the primitive idempotents.

(h) `CondensedRadicalBasis` : The condensed matrices in eAe of a basis for the radical of the algebra. The output is a list of lists giving the basis for e_iAe_j. Each entry in the list of lists is a tuple consisting of a matrix in the condensed algebra and the monomial which expresses this matrix as a product of the generators.

(i) `CondensedFieldGenerators` : The condensed matrices in eAe of the field generators.

(j) `FieldPolynomials` : The sequence of minimal polynomials of the field generators.

(k) `DegreesOfSimpleModules` : The dimensions of the Simple modules of A.

(l) `DegreeOfFieldExtensions` : The degrees of the centers of simple quotient algebras of A over the base field of A.

(m) `SimpleQuotientAlgebras` : The simple quotient algebras of A.

(n) `StandardFormConjugationMatrices` : The matrices which conjugate the algebra A into standard form with respect to the computed system of primitive idempotents.

`Presentation(A)`

The presentation in generators and relations of the matrix algebra A. The function returns the free algebra and the relations ideal calculated in the algebra structure program, as well as the map from the free algebra to A.

`StandardFormConjugationMatrices(A)`

Returns the pair (M and M^{-1}) of matrices that conjugate the matrix algebra A into standard form with respect to a chosen set of primitive idempotents.

`CondensationMatrices(A)`

The matrices, conjugating by which, gives the condensation of A.

`SequenceOfRadicalGenerators(A)`

The sequence of matrices of elements that generate the radical of A.

`CartanMatrix(A)`

The Cartan Matrix of the algebra A.

Example H83E11

In this example we form the permutation module M of the symmetric group G on seven letters by the Young subgroup that is the direct product H of two copies of the symmetric group on three letters. The algebra A is the image of the group algebra in the ring of endomorphisms of M.

```
> G := Sym(7);
> H := sub<G| G!(1,2,3),G!(1,2),G!(4,5,6), G!(4,5)>;
> M := PermutationModule(G, H, GF(5));
> M;
GModule M of dimension 140 over GF(5)
> A := Action(M);
```

We see that A has seven simple quotients.

```
> SimpleQuotientAlgebras(A);
```

```
rec<recformat<SimpleQuotients: SeqEnum, DegreesOverCenters:
SeqEnum, DegreesOfCenters: SeqEnum, OrdersOfCenters: SeqEnum> |
    SimpleQuotients := [
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 35 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 15 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 13 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 8 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 8 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 6 with 2
            generators over GF(5),
        Mapping from: Free associative algebra of rank 2 over
                GF(5) to Matrix Algebra of degree 1 with 2
            generators over GF(5)
    ],
    DegreesOverCenters := [ 35, 15, 13, 8, 8, 6, 1 ],
    DegreesOfCenters := [ 1, 1, 1, 1, 1, 1, 1 ],
    OrdersOfCenters := [ 5, 5, 5, 5, 5, 5, 5 ]
    >
> RanksOfPrimitiveIdempotents(A);
[ 1, 1, 3, 2, 1, 4, 3 ]
```

The condensed algbra of A is a much smaller object.

```
> B := CondensedAlgebra(A);
> B;
Matrix Algebra of degree 15 with 13 generators over GF(5)
> CartanMatrix(A);
[1 0 0 0 0 0 0]
[0 1 0 0 0 0 0]
[0 0 2 0 1 0 1]
[0 0 0 1 0 1 0]
[0 0 1 0 1 0 0]
[0 0 0 1 0 2 0]
[0 0 1 0 0 0 2]
```

From the Cartan Matrix we can see that A has four blocks. The first and second simple subalgebras are in blocks by themselves. The subalgebras numbers 3, 5, and 7 form another block as do the

subalgebras 4 and 6. Note that B, being Morita equivalent to A, has the same Cartan Matrix and the same block structure.

83.14.3 Solving the Word Problem

The presentation machinery also gives a test for membership in a matrix algebra. If A is a subalgebra of the $n \times n$ matrices generated by some collection of matrices, MAGMA can tell if any $n \times n$ matrix is an element of A. If the element is in A then MAGMA can write the element as a polynomial in the polynomial ring of the presentation.

`WordProblemData(A)`

The data needed for the solution to the word problem. The output is a list of lists of basis elements for the radical together with the corresponding monomials in the generators of the free algebra.

`WordProblem(A, x)`

Returns `true` if the matrix x is in the subalgebra A, and if true returns also an expression of the element x as a polynomial in the presentation of A.

Example H83E12

In this example we form the permutation module for the symmetric group G acting on the coset space of the normalizer H of Sylow 3-subgroup of G. The coefficients are in the field with two elements. The algebra A is the image of the group algebra in the endomorphism ring of the permutation module.

```
> G := Sym(5);
> H := Normalizer(G,Sylow(G,3));
> M := PermutationModule(G,H, GF(2));
> M;
GModule M of dimension 10 over GF(2)
> A := Action(M);
```

Here we get the presentation of A.

```
> P, I, mu  := Presentation(A);
> Dimension(P/I);
42
```

Thus the dimension of A is 42.

```
> CartanMatrix(A);
[1 0 1]
[0 1 0]
[1 0 2]
> B := CondensedAlgebra(A);
> Q, J, theta := Presentation(B);
```

The presentation of B has the form:

```
> J;
```

```
Two-sided ideal of Free associative algebra of rank 5 over GF(2)
Non-commutative Graded Lexicographical Order
Variables: b_1, b_2, b_3, z_1, z_2
Groebner basis:
[
    b_2^2 + b_2,
    b_2*b_3,
    b_2*z_1,
    b_2*z_2,
    b_3*b_2,
    b_3^2 + b_3,
    b_3*z_1,
    b_3*z_2 + z_2,
    z_1*b_2,
    z_1*b_3 + z_1,
    z_1^2,
    z_1*z_2,
    z_2*b_2,
    z_2*b_3,
    z_2^2,
    b_1 + b_2 + b_3 + 1
]
```

The matrix $A.1$ is the first of the original generators for A. We can check that $A.1$ is an element of the algebra generated by the computed generators of A.

```
> boo, y := WordProblem(A,A.1);
> boo;
true
```

The element y is the expression of $A.1$ as a polynomial in the free algebra P in the new computed generators for A.

```
> y;
t_1^4*b_1*t_1^3 + t_2^4*b_2*t_2^3 + t_1^4*b_1*t_1^2 +
    t_2^3*b_2*t_2^3 + t_1^2*b_1*t_1^3 + t_1^4*b_1*t_1 +
    t_2^2*b_2*t_2^3 + t_2^3*b_2*t_2^2 + t_2^4*b_2*t_2 +
    t_1^2*b_1*t_1^2 + t_1^3*b_1*t_1 + t_1^4*b_1 + t_1^4*z_1 +
    t_2*b_2*t_2^3 + t_2^2*b_2*t_2^2 + t_2^3*b_2*t_2 +
    t_1*b_1*t_1^2 + t_1^3*b_1 + t_3*z_2*t_1^2 + t_1^2*b_1 +
    t_2*b_2*t_2 + t_2^2*b_2 + t_1*b_1 + t_2*b_2 + t_3*b_3 +
    t_3*z_2
```

The mapping mu is the function from the free algebra P into A.

```
> mu(y);
[0 1 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0]
```

```
[0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
```

The ultimate check of the accuracy of the computation is that the polynomial expressions give back the original generators.

```
> mu(y) eq A.1;
true
```

Finally we can check whether a random 10×10 matrix is an element of A.

```
> b := Random(Generic(A));
> WordProblem(A,b);
false
```

83.15 Bibliography

[**ABM99**] John Abbott, Manuel Bronstein, and Thom Mulders. Fast Deterministic Computation of Determinants of Dense Matrices. In Sam Dooley, editor, *Proceedings ISSAC'99*, pages 197–204, New York, 1999. ACM Press.

84 GROUP ALGEBRAS

Chapter 84
GROUP ALGEBRAS

84.1 Introduction

Group algebras (or group rings) may be defined over any unital ring R and any group G in which elements have a canonical form (for example permutation groups, matrix groups or polycyclic groups). Basic operations, such as simple arithmetic of elements, may be applied in any such group algebra. Certain functions, however, place stricter requirements on R and G, such as requiring that R have a matrix echelon algorithm in MAGMA.

84.2 Construction of Group Algebras and their Elements

84.2.1 Construction of a Group Algebra

There are two different representations of group algebra elements used in MAGMA. Which is most suitable depends on the group G and on the operations required, and must be decided upon when the algebra is created.

The first representation, which requires that G is not too large, is to choose (once and for all) an ordering $(g_1, g_2, \ldots, g_n)$ of the elements of G (where $n = |G|$), and then to store elements of the group algebra $A = R[G]$ as coefficient vectors relative to the basis $(g_1, g_2, \ldots, g_n)$ of A. So an element $a = a_1 * g_1 + a_2 * g_2 + \ldots + a_n * g_n$ $(a_i \in R)$ of A will be stored as the vector $(a_1, a_2, \ldots, a_n)$. This makes for fast arithmetic and allows the use of matrix echelonization for dealing with subalgebras and ideals (if R has a matrix echelon algorithm).

The alternative representation, necessary when dealing with large groups, stores an element $a \in A = R[G]$ as a pair of parallel arrays giving the terms of the element. One array contains the nonzero coefficients of the element and the other contains their associated group elements. This representation allows group algebras to be defined over any group in which the elements have a canonical form, including potentially infinite matrix groups over a ring of characteristic 0 or even free groups. Note however, that operations in such algebras are limited, as the length of the representing arrays may grow exponentially with the number of multiplications performed.

`GroupAlgebra( R, G: parameters)`

`GroupAlgebra< R, G: parameters >`

`Rep`	MONSTGELT	*Default :*
`Table`	BOOLELT	*Default :*

Given a unital ring R and a group G (which is not a finitely presented group), create the group algebra $R[G]$. There are two optional arguments associated with this creation function.

The optional parameter `Rep` can be used to specify which representation should be chosen for elements of the algebra. Its possible values are `"Vector"` and `"Terms"`. If `Rep` is not assigned, then MAGMA chooses the vector representation if $|G| \leq 1000$ and the term representation otherwise. If `Rep` is set to `"Vector"`, MAGMA has to compute the order of G. If it does not succeed in that or the order is too large to construct vectors of this size, then an error message is displayed.

The second optional parameter `Table` can be used to specify whether or not the multiplication table of the group should be computed and stored upon creation of the algebra. Storing the multiplication table makes multiplication of algebra elements (and all other group algebra operations using this) much faster. Building this table does, however, increase the time needed to create the algebra. The table can only be stored if the vector representation of elements is used. If the `Table` parameter is not set, then MAGMA stores the multiplication table only if $G \leq 200$.

Example H84E1

We first construct the default group algebra $A = R[G]$ where R is the ring of integers and G is the symmetric group on three points.

```
> G := SymmetricGroup(3);
> R := Integers();
> A := GroupAlgebra( R, G );
> A;
Group algebra with vector representation and stored multiplication table
Coefficient ring: Integer Ring
Group: Permutation group G acting on a set of cardinality 3
    Order = 6 = 2 * 3
        (1, 2, 3)
        (1, 2)
```

Next we construct the group algebra $A = R[G]$ where $R = GF(5)$ and G is the dihedral group of order 100, given as a polycyclic group. This time we specify that the term representation should be used for elements.

```
> G := PCGroup( DihedralGroup(50) );
> A := GroupAlgebra( GF(5), G: Rep := "Terms" );
> A;
Group algebra with terms representation
Coefficient ring: Finite field of size 5
Group: GrpPC : G of order 100 = 2^2 * 5^2
    PC-Relations:
        G.1^2 = Id(G),
        G.2^2 = Id(G),
        G.3^5 = G.4,
        G.4^5 = Id(G),
        G.3^G.1 = G.3^4 * G.4^4,
        G.4^G.1 = G.4^4
```

84.2.2 Construction of a Group Algebra Element

`elt< A | r, g >`

Given a group algebra $A = R[G]$, a ring element $r \in R$ and a group element $g \in G$, create the element $r * g$ of A.

`A ! g`

Given a group element $g \in G$ create the element $1_R * g$ of the group algebra $A = R[G]$.

`A ! r`

Given a ring element $r \in R$, create the element $r * 1_G$ of the group algebra $A = R[G]$.

`A ! [`$c_1, \ldots, c_n$`]`

Given a group algebra $A = R[G]$ in vector representation and a sequence $[c_1, \ldots, c_n]$ of $n = |G|$ elements of R, create the element $c_1 * g_1 + \ldots + c_n * g_n$, where $(g_1, \ldots, g_n)$ is the fixed basis of A.

`Eta(A)`

For a group algebra $A = R[G]$ in vector representation, create the element $\sum_{g \in G} 1_R * g$ of A.

Example H84E2

We check that $Eta(A)/|G|$ is an idempotent in A (provided the characteristic of R does not divide the order of G).

```
> G := Alt(6);
> A := GroupAlgebra( GF(7), G );
> e := Eta(A) / #G;
> e^2 - e;
0
```

84.3 Construction of Subalgebras, Ideals and Quotient Algebras

Let $A = R[G]$ be a group algebra defined in MAGMA, where R is a unital ring with a matrix echelon algorithm and G is a group. If A was constructed to use the vector representation for its elements, then subalgebras and left, right and two-sided ideals of A can be constructed. If R is a field, then quotient algebras of A are also possible.

A subalgebra or ideal of a group algebra A is not in general a group algebra; hence, a different type is needed for these objects. Two possibilities are provided in MAGMA. The first is to construct the subalgebra or ideal to have type `AlgGrpSub`: a subalgebra (which may be an ideal) of a group algebra. Its elements are elements of the group algebra and it remains closely connected to its defining group algebra. By default, ideals of group algebras will be of this type.

The alternative is to create the subalgebra or ideal as an associative algebra given by structure constants (type `AlgAss`). The required structure constants are easily computable from the group algebra. This is the default MAGMA uses when creating subalgebras of a group algebra.

Regardless of the type given to a subalgebra or ideal, the inclusion map back into the group algebra is also provided.

```
sub< cat : A | L >
```

`cat` CAT *Default :* `AlgGrpSub`

Create the subalgebra S of the group algebra A generated by the elements defined by L, where L is a list of one or more items of the following types:

(a) an element of A;

(b) a set or sequence of elements of A;

(c) a subalgebra of A;

(d) a set or sequence of subalgebras of A;

(e) a sequence of ring elements specifying an element of A, as returned when the `ElementToSequence` (or `Eltseq`) function is applied to the element.

The constructor returns the subalgebra as an element of the category `cat`, where `cat` is either `AlgGrpSub` (the default) or `AlgAss`. As well as the subalgebra itself, the constructor returns the inclusion homomorphism $f : S \to A$.

```
lideal< cat : A | L >
```

`cat` CAT *Default :* `AlgGrpSub`

Create the left ideal of the group algebra A, generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the ideal as an element of the category `cat`, where `cat` is either `AlgGrpSub` (the default) or `AlgAss`. As well as the ideal itself, the constructor returns the inclusion homomorphism $f : I \to A$.

`rideal< cat : A | L >`

`cat` CAT *Default :* `AlgGrpSub`

Create the right ideal of the group algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the ideal as an element of the category `cat`, where `cat` is either `AlgGrpSub` (the default) or `AlgAss`. As well as the ideal itself, the constructor returns the inclusion homomorphism $f : I \to A$.

`ideal< cat : A | L >`

`cat` CAT *Default :* `AlgGrpSub`

Create the (two-sided) ideal I of the group algebra A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the ideal as an element of the category `cat`, where `cat` is either `AlgGrpSub` (the default) or `AlgAss`. As well as the ideal itself, the constructor returns the inclusion homomorphism $f : I \to A$.

The above operations can also be applied with a subalgebra (type `AlgGrpSub`) S in place of the group algebra A. In this case the subalgebra and ideal closures are taken in S.

`a * I`

For a right ideal I of the group algebra A construct the right ideal $\{a * b : b \in I\}$.

`I * a`

For a left ideal I of the group algebra A construct the left ideal $\{b * a : b \in I\}$.

`quo< A | L >`

Create the quotient of the group algebra A by the ideal of A generated by the elements defined by L, where L is a list as for the `sub` constructor above.

The constructor returns the quotient as an associative algebra. As well as the quotient Q itself, the constructor returns the natural homomorphism $f : A \to Q$.

`A / S`

The quotient of the algebra A by the ideal closure of its subalgebra S.

Example H84E3

We demonstrate how the degrees of the absolutely irreducible characters of a group can be found by computing the Wedderburn decomposition of the group algebra over a suitable finite field. Of course, this is neither the smartest nor the quickest way to get this information. In the example we deal with an extraspecial group of order 3^3.

```
> G := ExtraSpecialGroup(3, 1);
> Exponent(G);
```

```
3
```

We have to choose a finite field that contains the roots of unity that may be required by the absolutely irreducible characters. This is the case for $GF(q)$ if the exponent of G divides $q-1$. In our example, $q = 4$ or $q = 7$ are possible choices.

```
> FG := GroupAlgebra(GF(4), G);
> FG;
Group algebra with vector representation and stored multiplication table
Coefficient ring: Finite field of size 2^2
Group: Permutation group G acting on a set of cardinality 27
    Order = 27 = 3^3
        (1, 19, 10)(2, 20, 11)(3, 21, 12)(4, 22, 13)(5, 23, 14)(6, 24, 15)(7,
            25, 16)(8, 26, 17)(9, 27, 18)
        (1, 7, 4)(2, 8, 5)(3, 9, 6)(10, 18, 14)(11, 16, 15)(12, 17, 13)(19, 26,
            24)(20, 27, 22)(21, 25, 23)
        (1, 3, 2)(4, 6, 5)(7, 9, 8)(10, 12, 11)(13, 15, 14)(16, 18, 17)(19, 21,
            20)(22, 24, 23)(25, 27, 26)
> MI := MinimalIdeals(FG);
> #MI;
11
```

The minimal ideals are the simple Wedderburn components of the group algebra, corresponding to the absolutely irreducible characters of the group, and their dimensions are the squares of the character degrees.

```
> [ Isqrt(Dimension(I)) : I in MI ];
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3 ]
```

Hence, there are 9 linear characters and two of degree 3 (which we might have concluded immediately from the fact that the derived quotient of G has order 9).

84.4 Operations on Group Algebras and their Subalgebras

84.4.1 Operations on Group Algebras

The operations in this section can only be applied to a full group algebra. Functions accepting also a subalgebra of type `AlgGrpSub` are dealt with in the next section.

`Algebra(A)`

For a group algebra A given in vector representation, construct the associative structure constant algebra B isomorphic to A together with the isomorphism $A \to B$.

`AugmentationMap(A)`

The augmentation map of A. That is, the map $A \to R : \sum_{g \in G} r_g * g \to \sum_{g \in G} r_g$.

`AugmentationIdeal(A)`

The augmentation ideal of the group algebra A given in vector representation. This is defined as the kernel of the augmentation map.

`RepresentationType(A)`

Given a group algebra A, return either `"Vector"` or `"Terms"` depending on which representation is used for the elements of A.

`ChangeRepresentationType(A, Rep)`

Given a group algebra A, construct an isomorphic group algebra B in which the elements are represented as specified by `Rep` which may be `"Vector"` or `"Terms"`, together with the homomorphism from A to B.

`ConstructTable(A)`

Procedure which, given a group algebra $A = R[G]$ in vector representation, constructs the multiplication table for the group G to speed up multiplication in A. If the multiplication table already exists, nothing is done.

`CoefficientRing(A)`

`BaseRing(A)`

The coefficient ring (base ring) of A.

84.4.2 Operations on Subalgebras of Group Algebras

The functions in this section can be applied to group algebras and their subalgebras of type `AlgGrpSub`.

`S ! 1`

Create the identity element of the group algebra (subalgebra) S. Note that for a proper subalgebra of the full group algebra this may be different from the identity element of the group.

`Group(S)`

The group G for the group algebra (subalgebra) S.

`GroupAlgebra(S)`

The group algebra of which S is a subalgebra.

`Module(S)`

For a subalgebra S of the group algebra $A = R[G]$, return the submodule of the module underlying A which corresponds to S. This is an R-module of dimension `Dimension(S)` and degree `Dimension(A)`. Also returns (as a second return value) the natural map from the subalgebra to the module.

`CoefficientRing(A)`

`BaseRing(A)`

The coefficient ring (base ring) of A.

`BasisMatrix(S)`

For a subalgebra S of the group algebra $A = R[G]$ return the coefficient matrix of the basis of S with respect to the basis of A. If S has dimension m this is an $m \times |G|$-matrix over R where the i-th row are the coefficients of the i-th basis vector of S with respect to the fixed basis of A.

`Coordinates(S, a)`

Given an element a which lies in the subalgebra S, return a sequence giving the coordinates of a with respect to the basis of S.

`IsLeftIdeal(S)`

Returns `true` if S is a left ideal of its group algebra; otherwise `false`.

`IsRightIdeal(S)`

Returns `true` if S is a right ideal of its group algebra; otherwise `false`.

`IsIdeal(S)`

Returns `true` if S is a (two-sided) ideal of its group algebra; otherwise `false`.

`Centraliser(S)`

`Centralizer(S)`

The centralizer of the subalgebra S of a group algebra A (in A).

`Idealiser(S)`

`Idealizer(S)`

The largest subalgebra T of A such that S is an ideal in T.

`LeftAnnihilator(S)`

For a subalgebra S of the group algebra A construct the left annihilator of S, that is, the subalgebra of A consisting of all elements a such that $a * s = 0$ for all $s \in S$.

`RightAnnihilator(S)`

For a subalgebra S of the group algebra A construct the right annihilator of S, that is, the subalgebra of A consisting of all elements a such that $s * a = 0$ for all $s \in S$.

Example H84E4

We construct the group algebra of an elementary abelian group over $GF(2)$ and get its Jacobson ideal.

```
> A := AbelianGroup([2,2,2,2,2]);
> FG := GroupAlgebra(GF(2), A);
> J := JacobsonRadical(FG);
> J;
Ideal of dimension 31 of the group algebra FG
```

We now check that the Jacobson radical is nilpotent and get its nilpotency class.

```
> JPow := [ J ];
> I := J;
> while Dimension(I) ne 0 do
>     I := I*J;
>     Append(~JPow, I);
> end while;
> [ Dimension(I) : I in JPow ];
[ 31, 26, 16, 6, 1, 0 ]
```

Thus, J is nilpotent of class 6. However, every non-zero element of J is of course nilpotent of class 2.

```
> IsNilpotent(Random(J));
true 2
```

84.5 Operations on Elements

The operations in this section can be applied to elements of either a group algebra or of a group algebra subalgebra of type `AlgGrpSub`. Only those operations are listed, which are additional to those available for general algebras.

`a + r`
`r + a`

The sum of the group algebra element $a \in R[G]$ and the scalar $r \in R$.

`a + g`
`g + a`

The sum of the group algebra element $a \in R[G]$ and the group element $g \in G$.

`a - r`
`r - a`

The difference of the group algebra element $a \in R[G]$ and the scalar $r \in R$.

`a - g`
`g - a`

The difference of the group algebra element $a \in R[G]$ and the group element $g \in G$.

`a * r`
`r * a`

The product of the group algebra element $a \in R[G]$ and the scalar $r \in R$.

`g * a`
`a * g`

The product of the group algebra element $a \in R[G]$ and the group element $g \in G$.

`Support(a)`

The support of a; that is, the sequence of group elements whose coefficients in a are non-zero.

`Trace(a)`

The trace of a; that is, the coefficient of 1_G in a.

`Augmentation(a)`

The augmentation of the group algebra element a; that is, $\sum_{g \in G} r_g$ where $a = \sum_{g \in G} r_g * g$.

`Involution(a)`

If $a = \sum_{g \in G} r_g * g$, returns $\sum_{g \in G} r_g * g^{-1}$.

`Coefficient(a, g)`
`a[g]`

The coefficient of $g \in G$ in $a \in R[G]$.

`ElementToSequence(a)`
`Eltseq(a)`

If a is an element from a group algebra A given in vector representation, this returns the sequence of coefficients with respect to the fixed basis of A. If A is given in terms representation, this returns a sequence of tuples, where the second entry is a group element and the first is the coefficient of that group element in a.

`Coefficients(a)`

For an element a from a group algebra A given in vector representation, this returns the sequence of coefficients with respect to the fixed basis of A.

Centraliser(a)
Centralizer(a)

The centralizer in the group algebra A of the element a of A.

Centraliser(S, a)
Centralizer(S, a)

The centralizer of the element a (of a group algebra A) in the subalgebra S of A.

Example H84E5

We use the group algebra to determine the diameter of the Cayley graph of a group.

```
> G := Alt(6);
> QG := GroupAlgebra( Rationals(), G );
> e := QG!1 + &+[ QG!g : g in Generators(G) ];
> e;
Id(G) + (1, 2)(3, 4, 5, 6) + (1, 2, 3)
```

The group elements that can be expressed as words of length at most n in the generators of G have non-zero coefficient in e^n. The following function returns for a group algebra element e a sequence with the cardinalities of the supports of e^n and breaks when the group order is reached.

```
> wordcount := function(e)
>     f := e;
>     count := [ #Support(f) ];
>     while count[#count] lt #Group(Parent(e)) do
>         f *:= e;
>         Append(~count, #Support(f));
>     end while;
>     return count;
> end function;
```

Now apply this function to the above defined element:

```
> wordcount( e );
[ 3, 7, 14, 26, 47, 83, 140, 219, 293, 345, 360 ]
```

Thus, every element in A_6 can be expressed as a word of length at most 11 in the generators $(1,2)(3,4,5,6)$ and $(1,2,3)$. A better 2-generator set is for example $(1,2,3,4,5)$ and $(1,5,3,6,4)$, where all elements can be expressed as words of length at most 10 and this is in fact optimal. A worst 2-generator set is given by $(1,2)(3,4)$ and $(1,5,3,2)(4,6)$.

```
> wordcount( QG!1 + G!(1,2,3,4,5) + G!(1,5,3,6,4) );
[ 3, 7, 15, 31, 60, 109, 183, 274, 350, 360 ]
> wordcount( QG!1 + G!(1,2)(3,4) + G!(1,5,3,2)(4,6) );
[ 3, 6, 11, 18, 28, 43, 63, 88, 119, 158, 206, 255, 297, 329, 352, 360 ]
```

Example H84E6

The group algebra can also be used to investigate the random distribution of words of a certain length in the generators of the group.

```
> M11 := sub< Sym(11) | (1,11,9,10,4,3,7,2,6,5,8), (1,5,6,3,4,2,7,11,9,10,8) >;
> A := GroupAlgebra(RealField(16), M11 : Rep := "Vector");
> A;
Group algebra with vector representation
Coefficient ring: Real Field of precision 16
Group: Permutation group M11 acting on a set of cardinality 11
    Order = 7920 = 2^4 * 3^2 * 5 * 11
        (1, 11, 9, 10, 4, 3, 7, 2, 6, 5, 8)
        (1, 5, 6, 3, 4, 2, 7, 11, 9, 10, 8)
> e := (A!M11.1 + A!M11.2) / 2.0;
> eta := Eta(A) / #M11;
```

For growing n, the words of length n in the generators of `M11` converge towards a random distribution iff $\texttt{e}^n$ converges towards `eta`. We look at the quadratic differences of the coefficients of $\texttt{e}^n$`-eta` for $n = 10, 20, 30, 40, 50$.

```
> e10 := e^10;
> f := A!1;
> for i in [1..5] do
>     f *:= e10;
>     print &+[ c^2 : c in Eltseq(f - eta) ];
> end for;
0.0012050667195213
1.289719354694155e-5
5.9390965208879e-7
3.394099291966e-8
2.19432454574986e-9
```

85 BASIC ALGEBRAS

Chapter 85
BASIC ALGEBRAS

85.1 Introduction

A basic algebra is a finite dimensional algebra A over a field, all of whose simple modules have dimension one. In the literature such an algebra is known as a "split" basic algebra. Every algebra is Morita equivalent to a basic algebra, though a field extension may be necessary to obtain the split basic algebra. MAGMA has several functions that create the basic algebras corresponding to algebras of different types.

The type `AlgBas` in MAGMA is optimized for the purposes of doing homological calculations. A basic algebra A is generated by elements $a_1, a_2, \ldots, a_t$ where $a_1, \ldots, a_s$ are the primitive idempotent generators and $a_{s+1}, \ldots, a_t$ are the nonidempotent generators. Each nonidempotent generator, a_k must have the property that $a_i * a_k * a_j = a_k$ for specific idempotent generators a_i and a_j. The projective indecomposable modules have the form $P_i = a_i \cdot A$ for $i = 1, \ldots, s$ and the simple modules have the form $S_i = P_i/\mathrm{Rad}(P_i)$, where $\mathrm{Rad}(P_i)$ is the radical of P_i.

85.2 Basic Algebras

In the MAGMA implementation the algebra is given as the sequence of projective modules $P_1, \ldots, P_s$ together with a path tree for each projective module. A projective module consists of a matrix for each generator $a_1, a_2, \ldots, a_t$ giving the action of the generator on the vector space of the module. The basis $b_1, b_2, \ldots, b_n$ for the vector space of P_i is chosen so that each basis element is the image of a basis element of lower index under multiplication by a nonidempotent generator of A. The structure of the basis is recorded in the path tree which is a sequence $[< 1, i >, < j, k >, \ldots]$ of 2-tuples of length $n =$ `Dimension`(P_i). The first entry $< 1, i >$ indicates that $b_1 = b_1 * a_i$ where a_i is the primitive idempotent in the algebra A such that $P_i = A \cdot a_i$. Similarly, if entry number k in the path tree is $\langle u, v \rangle$ then $b_k = b_u * a_v$ where $v > s$ if $k > 1$.

85.2.1 Creation

The first function for creating a basic algebra is the most basic, in which the user supplies the projective modules and the path trees directly.

`BasicAlgebra(Q)`

Given a sequence $[Q_i, \ldots, Q_s]$ of 2-tuples such that each $Q_i =< M_i, T_i >$ consisting of a module for a matrix algebra M_i and a path tree T_i for M_i, the function creates the basic algebra whose projective modules are the first entries $M_1, \ldots, M_s$ and the path trees are the corresponding second entries.

The next two functions create a basic algebra from generators and relations. The user must, additionally, specify the quiver with relations, giving the number of idempotents and the beginning and end points of the arrows in the quiver.

`BasicAlgebra(F,R,s,P)`

Creates the basic algebra given by the presentation. Here F is a free algebra and R is the sequence of relation for the nonidempotent generators of the algebra. If the free algebra F is generated by elements $a_1, \ldots, a_t$, the function assumes that $a_1, \ldots, a_s$ are the mutually orthogonal primitive idempotents and it creates all of the appropriate relations including $a_1 + \ldots + a_s = 1$. The nonidempotent generators are then $a_{s+1}, \ldots, a_t$. So $\ell_k = \langle i, j \rangle$ for $i, j \leq s$ means that $a_{s+k} = a_i * a_{s+k} * a_j$. Each of the relations in R is given as a linear combination of words in the nonidempotent generators $a_{s+1}, \ldots, a_t \in F$. The sequence P is a sequence of 2-tuples, one for each nonidempotent generator, giving the beginning and ending nodes of the generator. That is, each tuple is the pair of indices of the idempotents which multiply as the identity on the nonidempotent generator on the left and on the right.

`BasicAlgebra(F,R)`

Creates the basic algebra of a local algebra from the presentation of the algebra. Here F is a free algebra whose variable represent the nonidempotent generators and R is the sequence of relations among those variables.

`TensorProduct(A, B)`

The tensor product of the basic algebras A and B.

The modular group algebra of a p-group is naturally a basic algebra. The next function create the basic algebra from the group information.

`BasicAlgebra(G, k)`

`BasicAlgebra(G, k)`

`BasicAlgebra(G)`

Given a finite p-group G and a finite field k of characteristic p, returns the group algebra kG in the form of a basic algebra. If no field k is supplied then the prime field of characteristic p is assumed to be the field of coefficients.

85.2.2 Special Basic Algebras

There are several functions that create basic algebras of special interest. The most basic of these creates the basic algebra that is Morita equivalent to a matrix algebra defined over a finite field by first condensing the algebra and then splitting the irreducible modules as needed.

Included among these constructions are the basic algebras of Schur algebras and Hecke algebras over finite fields. The Schur algebras arise in the representation theory of symmetric groups. The Schur algebras have finite global dimension and hence their ext-algebras have finite dimension. By a Hecke algebra, we mean the algebra of endomorphisms of a permutation module of a finite group.

`BasicAlgebra(A)`

`BasicAlgebraOfMatrixAlgebra(A)`

This function creates the split basic algebra of the matrix algebra A. The function first produces a presentation and condensed algebra for A. In the event that the field of coefficients k of A is not a splitting field, then the returned basic algebra is defined over the minimal extension of k that is a splitting field for A.

`BasicAlgebraOfEndomorphismAlgebra(M)`

Returns the split basic algebra of the endomorphism ring of the module M. In the event that the field of coefficients of M is not a splitting field for M, then the field is extended and the basic algebra is defined over the minimal extension needed to split M.

`BasicAlgebraOfHeckeAlgebra(G, H, F)`

Returns the basic algebra of the Hecke algebra which is the algebra of endomorphisms of the permutation module over G with point stabilizer H and field of coefficients F. In the event that F is not a splitting field then the field F is extended to a splitting field which is the field of coefficients of the returned basic algebra.

`BasicAlgebraOfSchurAlgebra(n, r, F)`

Creates the basic algebra of the Schur algebra $S(n, r)$ over the field F. The Schur algebra is the algebra of endomorphisms of the module over the symmetric group $Sym(r)$ that is the tensor product of r copies of a vector space V over F of dimension n, the group $Sym(r)$ acting by permuting the r copies.

`BasicAlgebraOfGroupAlgebra(G,F)`

`BasicAlgebraOfGroupAlgebra(G,F)`

`BasicAlgebraOfGroupAlgebra(G,F)`

The function returns the basic algebra of the group algebra of G with coefficients in the field F. The function requires the creation of the projective indecomposable FG-modules. In the event that the field F is not a splitting field for the irreducible FG-modules then the base ring of the returned algebra is the minimal extension of F that is necessary to get a splitting field.

`BasicAlgebra(S)`

Returns the basic algebra of the action algebra on the module which is the direct sum of the modules in the sequence S. In the event that the irreducible composition factors of the modules in S are not absolutely irreducible, then the returned basic algebra is defined over the splitting field for the irreducible modules.

`BasicAlgebraOfBlockAlgebra(S)`

Returns the basic algebra of the block algebra, the projective modules of which are given in the sequence S. In the event that the irreducible composition factors of the modules in S are not absolutely irreducible, then the returned basic algebra is defined over the splitting field for the irreducible modules.

`BasicAlgebraOfPrincipalBlock(G,k)`

Returns the basic algebra of the principal block of the group algebra of G. If the simple modules in the principal kG block are all absolutely simple, then the ordering of the projective modules for the returned basic algebra is exactly the same as the ordering of the projectives modules in the principal block returned by the function `IndecomposableProjectives`. Otherwise, the base ring of the returned basic algebra is the least extension of k necessary to split the simple modules in the principal block and the simple modules of the returned algebra are ordered by increasing dimension of the corresponding simple modules of kG.

`BasicAlgebraOfExtAlgebra(A)`

The function returns the basic algebra from a computed ext-algebra which is $\mathrm{Ext}^*_A(S,S)$, where S is the direct sum of the irreducible A-modules, of the basic algebra A. If no ext-algebra for A has been computed or if the exalgebra is not verified to be finite dimensional then an error is returned.

`BasicAlgebraOfExtAlgebra(A)`

The function returns the basic algebra for the ext-algebra, which is $\mathrm{Ext}^*_A(S,S)$, where S is the direct sum of the irreducible A-modules, of A computed to n steps. If no ext-algebra for A to n steps has been computed then it will be computed. If the ext-algebra is not verified to be finite dimensional by the computation, then an error is returned.

`BasicAlgebraOfExtAlgebra(A)`

The function creates the basic algebra from a computed ext-algebra. The input *ext* is the output of the `ExtAlgebra` function. If the ext-algebra is not verified to be finite dimensional by the computation, then an error is returned.

`OppositeAlgebra(B)`

The opposite algebra of the basic algebra B. The opposite algebra of B is the algebra with the same set of elements and the same addition but with multiplication $*$ given by $a * b = ba$ for a and b in A.

Example H85E1

We form the basic algebra of the group algebra of the alternating group on 6 letters in characteristic 2.

```
> G := AlternatingGroup(6);
> A := BasicAlgebraOfGroupAlgebra(G, GF(2));
> A;
Basic algebra of dimension 36 over GF(2^2)
Number of projective modules: 5
Number of generators: 9
```

Note that the field of coefficients has been extended in order to split the irreducible G-modules. A great deal of information on the nature of the group algebra and its basic algebra can be obtained from analysis such as the following.

```
> [Dimension(ProjectiveModule(A,i)): i in [1 .. 5]];
[ 9, 9, 16, 1, 1 ]
> prj := CompactProjectiveResolutionsOfAllSimpleModules(A,8);
> [x`BettiNumbers:x in prj];
[
    [
        [ 1, 0, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 1, 0, 0, 0, 0 ],
        [ 1, 0, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 1, 0, 0, 0, 0 ],
        [ 1, 0, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 1, 0, 0, 0, 0 ]
    ],
    [
        [ 0, 1, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 0, 1, 0, 0, 0 ],
        [ 0, 1, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 0, 1, 0, 0, 0 ],
        [ 0, 1, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ],
        [ 0, 1, 0, 0, 0 ]
    ],
    [
        [ 0, 0, 3, 0, 0 ],
        [ 1, 1, 2, 0, 0 ],
        [ 0, 0, 3, 0, 0 ],
        [ 0, 0, 2, 0, 0 ],
        [ 1, 1, 1, 0, 0 ],
        [ 0, 0, 2, 0, 0 ],
```

```
        [ 0, 0, 1, 0, 0 ],
        [ 1, 1, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0 ]
    ],
    [
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 1, 0 ]
    ],
    [
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 1 ]
    ]
]
```

So we see that the last two simple modules are projective and hence in the group algebra FG, they represent blocks of defect 0. From the information on the Betti numbers we observe that the first two simple modules have periodic projective resolutions. Thus the third simple module for the basic algebra corresponds to the trivial FG-module, which we know is neither projective nor periodic.

Example H85E2

We construct the Schur algebra $S(3,7)$ with coefficients in a field of characteristic 2.

```
> A := BasicAlgebraOfSchurAlgebra(3,6,GF(2));
> A;
Basic algebra of dimension 58 over GF(2)
Number of projective modules: 7
Number of generators: 21
```

It is known that A has finite global dimension, hence its ext-algebra has finite dimension.

```
> B := BasicAlgebraOfExtAlgebra(A,10);
> B;
Basic algebra of dimension 56 over GF(2)
Number of projective modules: 7
```

```
Number of generators: 25
```

We check to see if the ext-algebra of B might have finite dimension.

```
> SumOfBettiNumbersOfSimpleModules(B,5);
600
> SumOfBettiNumbersOfSimpleModules(B,6);
1334
> SumOfBettiNumbersOfSimpleModules(B,7);
3008
```

The sum of the Betti number would be the dimension of the ext-algebra computed to the indicated degree. It would appear that the ext-algebra is infinite dimensional. Just to check, we look at a particular simple module, chosen randomly.

```
> CompactProjectiveResolution(SimpleModule(B,4),10)‘BettiNumbers;
[
    [ 8, 0, 81, 3, 61, 69, 55 ],
    [ 3, 0, 36, 1, 26, 30, 24 ],
    [ 1, 0, 16, 1, 11, 13, 10 ],
    [ 1, 0, 7, 1, 5, 5, 4 ],
    [ 1, 0, 3, 1, 2, 2, 1 ],
    [ 1, 0, 1, 0, 1, 1, 0 ],
    [ 0, 0, 0, 0, 1, 1, 0 ],
    [ 0, 0, 0, 0, 1, 0, 1 ],
    [ 0, 0, 0, 1, 0, 1, 0 ],
    [ 1, 0, 0, 0, 0, 0, 1 ],
    [ 0, 0, 0, 1, 0, 0, 0 ]
]
```

Thus we have strong evidence that the fourth simple module has infinite projective dimension. In that case, the ext-algebra of B is not finite dimensional.
Now we consider the same example except in characteristic 3.

```
> A := BasicAlgebraOfSchurAlgebra(3,6,GF(3));
> A;
Basic algebra of dimension 48 over GF(3)
Number of projective modules: 7
Number of generators: 21
> B := BasicAlgebraOfExtAlgebra(A,10);
> B;
Basic algebra of dimension 98 over GF(3)
Number of projective modules: 7
Number of generators: 21
> SumOfBettiNumbersOfSimpleModules(B,5);
48
> SumOfBettiNumbersOfSimpleModules(B,6);
48
```

So we see that the algebra B has global dimension at most 5. We can compute its ext-algebra.

```
> C := BasicAlgebraOfExtAlgebra(B,10);
```

```
> C;
Basic algebra of dimension 48 over GF(3)
Number of projective modules: 7
Number of generators: 21
> D := BasicAlgebraOfExtAlgebra(C,10);
> D;
Basic algebra of dimension 98 over GF(3)
Number of projective modules: 7
Number of generators: 21
```

This provides evidence that A is isomorphic to its double ext-algebra. This would suggest that A might be a Koszul algebra.

85.2.3 Access Functions

These functions return basic information, underlying structures and elements of the given basic algebras.

`B . i`

The i^{th} element in the standard basis for the underlying vector space of the algebra B.

`BaseRing(B)`

`CoefficientRing(B)`

Given an algebra B over a field k the function returns k.

`VectorSpace(B)`

`KSpace(B)`

The underlying k-vector space of the algebra B. The space is the direct sum of the underlying vector space of the indecomposable projective modules.

`Dimension(B)`

The dimension of the underlying vector space of the algebra B.

`Basis(B)`

A basis of the underlying vector space of the algebra B.

`Generators(B)`

The generators of the algebra B as a sequence of elements in the underlying vector space of the algebra B.

`IdempotentGenerators(B)`

The sequence of mutually orthogonal idempotent generators of the basic algebra B as elements in the underlying vector space of B.

`IdempotentPositions(B)`

The sequence $N = [n_1, \ldots, n_s]$ such that $B.n_1, \ldots, B.n_s$ are the mutually orthogonal idempotent generators of the algebra B.

`NonIdempotentGenerators(B)`

The sequence of nonidempotent generators of the basic algebra B as elements in the underlying vector space of the algebra B.

`Random(B)`

A random element of the algebra B as an element of the underlying vector space of B.

`NumberOfProjectives(B)`

The number of nonisomorphic indecomposable projective modules in the basic algebra B.

`NumberOfGenerators(B)`

`Ngens(B)`

The number of generators (idempotent and nonidempotent) of the basic algebra B.

`DimensionsOfProjectiveModules(B)`

The sequence of the dimensions of the projective modules of the basic algebra B.

`DimensionsOfInjectiveModules(B)`

The sequence of the dimensions of the injective modules of the basic algebra B.

85.2.4 Elementary Operations

`a + b`

The sum of the two elements a and b.

`a * b`

The product of the two elements a and b.

`a ^ n`

The n^{th} power of the element a.

Example H85E3

We create the basic algebra of the quiver with three nodes and three arrows over the field with 7 elements. The first arrow (a) goes from node 1 to node 2, the second (b) from node 2 to node 1, and (c) from node 2 to node 3. The arrows satisfy the relation that $(\mathtt{a} * \mathtt{b})^3 = 0$.

```
> ff := GF(7);
> FA<e1,e2,e3,a,b,c> := FreeAlgebra(ff,6);
> rrr := [a*b*a*b*a*b];
> D := BasicAlgebra(FA,rrr,3,[<1,2>,<2,1>,<2,3>]);
> D;
Basic algebra of dimension 21 over GF(7)
Number of projective modules: 3
Number of generators: 6
> DimensionsOfProjectiveModules(D);
[ 9, 11, 1 ]
> DimensionsOfInjectiveModules(D);
[ 6, 7, 8 ]
```

So we can see that the algebra is not self-injective.
Now we can check the nilpotence degree of the radical of D. The radical of D is generated by the nonidempotent generators.

```
> S := NonIdempotentGenerators(D);
> S;
[ (0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0), (0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0), (0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0) ]
> S2 := [x*y:x in S, y in S|x*y ne 0];
> #S2;
3
> S3 := [x*y:x in S2, y in S|x*y ne 0];
> #S3;
3
> S4 := [x*y:x in S3, y in S|x*y ne 0];
> #S4;
3
> S5 := [x*y:x in S4, y in S|x*y ne 0];
> #S5;
3
> S6 := [x*y:x in S5, y in S|x*y ne 0];
> #S6;
2
> S7 := [x*y:x in S6, y in S|x*y ne 0];
> #S7;
1
> S8 := [x*y:x in S7, y in S|x*y ne 0];
> #S8;
0
```

Example H85E4

First we create the basic algebra for the symmetric group S_3 over the field $GF(3)$

```
> FA<e1,e2,a,b> := FreeAlgebra(GF(3),4);
> MM:= [a*b*a, b*a*b];
> BS3 := BasicAlgebra(FA, MM, 2, [<1,2>,<2,1>]);
> BS3;
Basic algebra of dimension 6 over GF(3)
Number of projective modules: 2
Number of generators: 4
> DimensionsOfProjectiveModules(BS3);
[ 3, 3 ]
```

Next we create the basic algebra for the cyclic group of order 3.

```
> gg := CyclicGroup(3);
> BC3 := BasicAlgebra(gg,GF(3));
> BC3;
Basic algebra of dimension 3 over GF(3)
Number of projective modules: 1
Number of generators: 2
```

We create the basic algebra for the direct product $C_3 \times S_3$.

```
> A := TensorProduct(BS3,BC3);
> A;
Basic algebra of dimension 18 over GF(3)
Number of projective modules: 2
Number of generators: 6
> DimensionsOfProjectiveModules(A);
[ 9, 9 ]
```

Example H85E5

We create the basic algebra for A_4 over a field with 2 elements. The group algebra has two nonisomorphic projective modules. We define the basic algebra by constructing the matrix algebra for the projective modules and the path trees and entering this data into the `BasicAlgebra` function.
Note that the matrices are sparse so we will define them by specifying the nonzero rows.

```
> ff := GF(2);
> MM6 := MatrixAlgebra(ff,6);
> e11 := MM6!0;
> e12 := MM6!0;
> VV6 := VectorSpace(GF(2),6);
> BB6 := Basis(VV6);
> e11[1]  := BB6[1];
> e11[3]  := BB6[3];
> e11[4]  := BB6[4];
> e11[6]  := BB6[6];
```

```
> e12[2]  := BB6[2];
> e12[5]  := BB6[5];
> a1 := MM6!0;
> b1 := MM6!0;
> c1 := MM6!0;
> d1 := MM6!0;
> a1[1] := BB6[2];
> b1[1] := BB6[3];
> c1[2] := BB6[4];
> a1[3] := BB6[5];
> b1[4] := BB6[6];
> c1[5] := BB6[6];
> A1 := sub< MM6 | [e11, e12, a1, b1, c1, d1] >;
> T1 := [ <1,1>, <1,3>, <1,4>, <2,5>, <3,3>, <4,4>];
>
> VV5 := VectorSpace(ff,5);
> BB5 := Basis(VV5);
> MM5 := MatrixAlgebra(ff,5);
> e21 := MM5!0;
> e22 := MM5!0;
> e22[1]  := BB5[1];
> e22[3]  := BB5[3];
> e22[5]  := BB5[5];
> e21[2]  := BB5[2];
> e21[4]  := BB5[4];
> a2 := MM5!0;
> b2 := MM5!0;
> c2 := MM5!0;
> d2 := MM5!0;
> f2 := MM5!0;
> g2 := MM5!0;
> c2[1] := BB5[2];
> d2[1] := BB5[3];
> b2[2] := BB5[4];
> d2[3] := BB5[5];
> a2[4] := BB5[5];
> A2 := sub< MM5 | [e21, e22, a2, b2, c2, d2] >;
> T2 := [<1,2>, <1,5>, <1,6>, <2,4>, <3,6>];
>
> C := BasicAlgebra( [<A1, T1>, <A2, T2>] );
> C;
Basic algebra of dimension 11 over GF(2)
Number of projective modules: 2
Number of generators: 6
> DimensionsOfProjectiveModules(C);
[ 6, 5 ]
> DimensionsOfInjectiveModules(C);
```

```
[ 6, 5 ]
```

85.2.5 Boolean Functions

A basic algebra in MAGMA is a sequence of matrix algebras, each with a path tree. When a basic algebra is created by entering such a sequence, MAGMA does not check to see if all of the properties of a basic algebra are satisfied, as this is an expensive operation. The following two function check to see if the properties of a basic algebra are satisfied.

`IsDimensionCompatible(B)`

Returns `true` if the dimension of a basic algebra is the same as the dimension of the matrix algebra of its action on itself. If false then the algebra is not a basic algebra.

`IsPathTree(B)`

Returns `true` if the basis elements of the projective modules in the basic algebra are determined by the path tree. If false, then the algebra is not a true basic algebra.

Two other functions check commutativity.

`IsCommutative(A)`

Returns `true` if the basic algebra A is commutative.

`IsCentral(A,x)`

Returns `true` if the element x is in the center of the basic algebra A.

85.3 Homomorphisms

MAGMA has the capability of creating homomorphisms of basic algebras. A homomorphism of a basic algebra has the type Map. The matrix of the homomorphism ϕ is accessed by entering `Matrix(phi)`. The kernel of a homomorphism is returned as a subspace of the vector space of the domain of the algebra. This is a two-sided ideal. There is no special type for ideals. They are subspaces of the underlying vector space of the algebra. They can be generated from any given set of elements of the algebra

The image of a homomorphism is returned as a basic algebra, together with the embedding homomorphism.

A homomorphism from a basic algebra A to a basic algebra B is normally created from a matrix having dimension of A rows and dimension of B columns. Note that MAGMA does not automatically check to see if the created map is an algebra homomorphism.

`hom< A - >`

The algebra homomorphism from basic algebra A to basic algebra B, whose matrix is the matrix S. Given a map ϕ, a homomorphism of basic algebra, the matrix of that map is recalled with the command `Matrix(phi)`.

`Kernel(phi)`

The kernel of the map ϕ.

`Image(phi)`

The image of the homomorphism ϕ together with the embedding homomorphism of the image into the codomain of ϕ.

`IsAlgebraHomomorphism(A, B, psi)`

Return `true` if the matrix ψ represents a homomorphism from basic algebra A to basic algebra B.

`X * Y`

The composition of the maps X and Y.

`IsAlgebraHomomorphism(A, B, psi)`
`IsAlgebraHomomorphism(A, B, psi)`
`IsAlgebraHomomorphism(A, B, psi)`
`IsAlgebraHomomorphism(A, B, psi)`

Returns `true`, if the map ψ is a homomorphism of basic algebras

`IsAlgebraHomomorphism(psi)`

Returns `true` if the map ψ is a homomorphism of basic algebras.

85.4 Subalgebras and Quotient Algebras

A subalgebra is also returned with the embedding homomorphism, and a quotient algebra is returned with the natural quotient map. These are needed for creating some standard subalgebras such as the centre of the algebra.

85.4.1 Subalgebras and their Constructions

`sub< A | S >`

The subalgebra of A generated by the elements of the sequence S, together with the inclusion map of the subalgebra into A. The subalgebra contains the idempotent of minimal rank in A that acts as a multiplicative identity on the elements of S.

`SubalgebraFromBasis(A, V)`

Given a basic algebra A and the basis V of a subspace of A, the function returns the basic algebra which is the subalgebra spanned by the subspace and the inclusion matrix of the homomorphism embedding the subalgebra into A. Note that the space V might not contain the identity element of V and in that case the minimal possible identity element is added to the returned subalgebra.

`MaximalIdempotent(A, S)`

Given a basic algebra A, a subspace S of the vector space of A that is closed under multiplication, this function returns an idempotent in A which has maximal rank among all idempotents contained in S.

`MinimalIdentity(A, S)`

Returns the idempotent of smallest rank that is a two sided identity for the elements in the set S.

`Centre(A)`

The centre of the basic algebra as a basic algebra together with the inclusion homomorphism.

`Centralizer(A,S)`

Returns the centralizer in the basic algebra A of the elements in the sequence S, along with the homomorphism embedding the centralizer into A.

`MaximalCommutativeSubalgebra(A,S)`

Returns a maximal commutative subalgebra of the basic algebra A that contains the elements of the sequence S. An error occurs if the elements of S do not commute.

85.4.2 Ideals and their Construction

`ideal< A | S >`
`ideal< A | S >`

Returns the subspace of the vector space of the algebra A that is the ideal of the A generated by the given sequence of elements S.

`LeftAnnihilator(A, S)`

Returns a basis for the left annihilator of the sequence S of elements in the basic algebra A.

`RightAnnihilator(A, S)`

Returns a basis for the right annihilator of the sequence S of elements in the basic algebra A.

`Annihilator(A,S)`

Returns a basis for the two-sided annihilator of the sequence of elements S of the basic algebra A.

`IsIdeal(A, S)`

Returns `true` if the subspace spanned by the elements of S is a two-sided ideal of the basic algebra A.

`IsLeftIdeal(A,S)`

Returns `true` if the subspace spanned by the elements of S is a left ideal of the basic algebra A.

`IsRightIdeal(A, S)`

Returns `true` if the subspace spanned by the elements of S is a two-sided ideal of the basic algebra A.

`RandomIdealGeneratedBy(A, n)`

Returns the ideal generated by n randomly selected elements in the Jacobson radical of the basic algebra A.

85.4.3 Quotient Algebras

`quo< A | S >`

Returns the quotient algebra of A by the ideal S, which is a subspace of the vector space of A, together with the quotient map.

`CoverAlgebra(A)`

Constructs the maximal extension B as in $[0 \to K \to B \to A \to 0]$ such that B acts trivially on K and B is an algebra with exactly the same minimal number of generators as A. Returns B and the algebra homomorphism of B onto A.

`GradedCoverAlgebra(A)`

This assumes that we are given the truncated algebra of a graded algebra. It creates the basic algebra of the natural cover of A and also returns the matrix of the cover onto A.

`TruncatedAlgebra(A,n)`

The quotient of the algebra by the n^{th} power of the radical of A. Returns also the quotient map.

85.5 Minimal Forms and Gradings

There are some standard ways of rewriting an algebra to an isomorphic form. We can also constructed the associated graded algebra of a basic algebra. Also included here is a function for rearranging the orders of idempotents in a basic algebra.

The first intrinsic is used extensively in the programs for computing automorphisms and isomorphisms.

`MinimalGeneratorForm(A)`

Returns a record consisting of an isomorphic basic algebra having the property that it is generated by a minimal number of elements and the projective modules are filtered by radical layers. The record consists of the following fields:

(a) The algebra in minimal form (field `algebra`).

(b) The map from the minimal generator form algebra to A (field `Homomorphism`).

(c) The inverse map from A to the minimal generator form algebra (field `InverseHomomorphism`).

(d) The dimensions of the radical layer (field `RadicalDimensions`).

(e) The dimensions of the filtration of the top radical layer by the socle layer (field `FilterDimensions`).

`MinimalGeneratorFormAlgebra(A)`

Returns an isomorphic algebra having minimal generator form.

`AssociatedGradedAlgebra(A)`

Returns the basic algebra that is isomorphic to the associated graded algebra of A.

`GradedCapHomomorphism(A)`

Returns the matrix of the map from $A/Rad(A)$ to $X/Rad(X)$ where X is the associated graded algebra of A.

`GradedCapHomomorphism(A, B, mu)`

Given an algebra homomorphism $mu : A \to B$, returns the induced homomorphism $A/\operatorname{Rad}^2(A) \to B/\operatorname{Rad}^2(B)$, where Rad^2 is the second power of the Jacobson radical.

`BuildHomomorphismFromGradedCap(A, B, phi)`

Returns the graded homomorphism from the associated graded algebra X of A to the associated graded algebra Y of B, whose cap is ϕ. That is phi is the matrix of the induced homomorphism of $X/Rad^2(X)$ to $Y/Rad^2(Y)$.

`ChangeIdempotents(A, S)`

`ChangeIdempotents(A, S)`

Returns the basic algebra isomorphic to A, obtained by permuting the order of the idempotents by the permutation S. The permutation S can be given as an element of the symmetric group or as the sequence of images of the permutation.

Example H85E6

In this examples we investigate properties of the basic algebra of a Schur algebra.

```
> A := BasicAlgebraOfSchurAlgebra(3,6,GF(3));
> A;
Basic algebra of dimension 48 over GF(3)
Number of projective modules: 7
Number of generators: 21
> B := BasicAlgebraOfExtAlgebra(A,10);
> B;
Basic algebra of dimension 98 over GF(3)
Number of projective modules: 7
Number of generators: 21
> C := BasicAlgebraOfExtAlgebra(B,10);
> C;
Basic algebra of dimension 48 over GF(3)
Number of projective modules: 7
Number of generators: 21
> boo,mat := IsIsomorphic(A,C);
> boo;
true
> IsAlgebraHomomorphism(mat);
true
```

So we see that A is isomorphic to its double ext-algebra, and it is graded since the double ext-algebra is graded. Thus we know that the basic algebra A is a Koszul algebra.

Example H85E7

Here we see how to rearrange an algebra by changing the ordering on the primitive idempotents.

```
> A := BasicAlgebraOfSchurAlgebra(3,5,GF(3));
> A;
Basic algebra of dimension 11 over GF(3)
Number of projective modules: 5
Number of generators: 9
> B, uu  := ChangeIdempotents(A,[2,4,5,1,3]);
> B;
Basic algebra of dimension 11 over GF(3)
Number of projective modules: 5
Number of generators: 9
> DimensionsOfProjectiveModules(A);
[ 2, 3, 2, 1, 3 ]
> DimensionsOfProjectiveModules(B);
[ 3, 1, 3, 2, 2 ]
> IsAlgebraHomomorphism(A,B,uu);
true
> uu;
[0 0 0 0 0 0 0 1 0 0 0]
```

```
[0 0 0 0 0 0 0 0 1 0 0]
[1 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0]
```

We see that uu is a permutation matrix.

85.6 Automorphisms and Isomorphisms

MAGMA has the capability of computing automorphisms and isomorphisms of basic algebras. Because the automorphism groups tend to be rather large, the functions work best on small examples.

`GradedAutomorphismGroupMatchingIdempotents(A)`

Returns the group of graded automorphisms of the basic algebra A that preserve the idempotents of A. Returns also the graded caps (matrices of homomorphisms from $X/Rad^2(X)$ to itself, where X is the Associated Graded algebra of A) of the generators of the automorphism group in two sequenced, nonunipotent generators and unipotent generators.

`GradedAutomorphismGroup(A)`

Returns the group of graded automorphisms of the associated graded algebra X of the basic algebra A. The function returns also the graded caps of the generators of the graded automorphism group. These are the induced automorphisms of $X/Rad^2(X)$ to itself, and they are returned as two lists of nonunipotent and unipotent generators that preserve the idempotents of X and a list of generators that permute the idempotents.

`IsGradedIsomorphic(A, B)`

Returns `true` if the associated graded algebras of A and B are isomorphic, in which case the isomorphism is returned.

`AutomorphismGroupMatchingIdempotents(A)`

Returns the group of all automorphism of the basic algebra A that preserve the basic idempotent structure. That is, any element of this group induces the identity automorphism on the quotient $A/Rad(A)$ of A by its Jacobson radical.

`AutomorphismGroup(A)`

Returns the automorphism group of the basic Algebra A, together with the sequences of nonnilpotent generators preserving idempotents, nilpotent generators preserving idempotents and generators that permute the idempotents.

`IsIsomorphic(A, B)`

Returns `true` if the basic algebra A is isomorphic to the basic algebra B and, if so, the function also returns an isomorphism.

Example H85E8

We compute the automorphism group of a group algebra.

```
> A := BasicAlgebra(SmallGroup(81, 7));
> time ba := AutomorphismGroup(A);
Time: 6.260
> #ba;
34358501382134088735764075317708084846807168 1334
> Factorization(#ba);
[ <2, 1>, <3, 99> ]
```

As expected the automorphism group has a very large unipotent 3-subgroup.

Example H85E9

We can see if small changes in the presentation of an algebra affect the isomorphism type. We define the algebras by generators and relations.

```
> F<e1,e2,z,y,x> := FreeAlgebra(GF(5),5);
> RR:= [(y*z)^3,x^4,x*y*z];
> A := BasicAlgebra(F,RR,2,[<1,2>,<2,1>,<2,2>]);
> A;
Basic algebra of dimension 49 over GF(5)
Number of projective modules: 2
Number of generators: 5
> RS:= [(y*z)^3-x^4,x^5,x*y*z,(z*y)^3];
> B := BasicAlgebra(F,RS,2,[<1,2>,<2,1>,<2,2>]);
> B;
Basic algebra of dimension 49 over GF(5)
Number of projective modules: 2
Number of generators: 5
> RT:= [(y*z)^3-2*x^4,x^5,x*y*z,(z*y)^3];
> C := BasicAlgebra(F,RS,2,[<1,2>,<2,1>,<2,2>]);
> C;
Basic algebra of dimension 49 over GF(5)
Number of projective modules: 2
Number of generators: 5
> time ab, x := IsIsomorphic(A,B);
Time: 0.100
```

```
> ab;
false
> time ac, x := IsIsomorphic(A,C);
Time: 0.050
> print ac;
false
> time bc,x := IsIsomorphic(B,C);
Time: 2.350
> print bc;
true
```

Example H85E10

We can see which groups of order 32 have mod 2 group algebras that are graded. The algebra is graded if and only if it is isomorphic to its associated graded algebra. So we make a list of those algebras whose group algebras are graded. Note that there are 51 isomorphism classes of groups of order 32. The last one (number 51) is elementary abelian and we know that its group algebra is graded.

```
> graded := [];
> for i := 1 to 50 do
>     A := BasicAlgebra(SmallGroup(32,i));
>     B := AssociatedGradedAlgebra(A);
>     boo, map := IsIsomorphic(A, B);
>     if boo then Append(~graded, i); end if;
> end for;
> graded;
[ 1, 2, 3, 5, 9, 12, 14, 16, 18, 21, 22, 25, 36, 39, 45, 46 ]
```

85.7 Modules over Basic Algebras

A module M over a basic algebra B is presented as a sequence of matrices, one for each generator of the algebra.

85.7.1 Indecomposable Projective Modules

The indecomposable projective modules are defined from the structure of the algebra and have associated path trees that solve the homomorphism lifting problem.

`ProjectiveModule(B, i)`

The i^{th} projective module of the basic algebra B.

`PathTree(B, i)`

The path tree of the i^{th} projective module of the basic algebra B.

`ActionGenerator(B, i)`

The sequence of matrices for the generators of the basic algebra B acting on the i^{th} projective module of B.

`IdempotentActionGenerators(B, i)`

The sequence of matrices for the idempotent generators of the basic algebra B acting on the i^{th} projective module of B.

`NonIdempotentActionGenerators(B, i)`

The sequence of matrices for the nonidempotent generators of the basic algebra B acting on the i^{th} projective module of B.

`Injection(B, i, v)`

Given a vector v in in the i^{th} projective module of the basic algebra B, the function returns the image of inclusion of v into B.

85.7.2 Creation

`AModule(B, Q)`

Given a basic algebra B and a sequence Q of elements in a matrix algebra the function returns the B-module M on which the generators of B act by multiplication by the corresponding elements of Q.

`ProjectiveModule(B, S)`

Given a sequence $S = [s_1, s_2, ...]$, the function returns a projective module which is the direct sum of s_1 copies of the first projective of the algebra B, s_2 copies of the second, etc. It also returns the sequence of inclusions and projections from and to the indecomposable projective modules.

`IrreducibleModule(B, i)`
`SimpleModule(B, i)`

The i^{th} irreducible module of the algebra B. The module is the quotient of the i^{th} projective module by its radical.

`ZeroModule(B)`

The zero B-module.

`RightRegularModule(B)`

The algebra B as a right module over itself. The module is the direct sum of the projectives modules of B.

`RegularRepresentation(v)`

If v is an element of a basic algebra given as a vector in the underlying space, then the function computes the matrix of the action by right multiplication of the element on the algebra.

`Restriction(M, B, xi)`

If B is a subalgebra of the basic algebra A, ξ is the embedding of B into A, and M is an A-module, then the function returns the restriction of M to a B-module.

`ChangeAlgebra(M, B, xi)`

`ChangeAlgebra(M, B, xi)`

Given a module M over an algebra A and an algebra homomorphism ξ from B to A, the function returns the module M as a B-module.

`JacobsonRadical(M)`

The Jacobson radical of the module M.

`Socle(M)`

The socle of the module M. The sum of the simple submodules of M.

85.7.3 Access Functions

`Algebra(M)`

Given a module M over a basic algebra B, the function returns B.

`Dimension(M)`

The dimension of the module M over its base ring.

`Action(M)`

The matrix algebra of the action of the algebra of M on M.

`IsomorphismTypesOfRadicalLayers(M)`

Given a module M over a basic algebra, returns the sequence of isomorphism types of simple composition factors in each layer of the radical filtration of M.

`IsomorphismTypesOfSocleLayers(M)`

Given a module M over a basic algebra, returns a sequence of isomorphism types of simple composition factors in each socle layer with reversed order, *i. e.* isomorphism types of the socle of M will appear last.

`IsomorphismTypesOfBasicAlgebraSequence(S)`

Given a sequence of irreducible modules S for a basic algebra A, return a sequence of isomorphism types comparing with the simple modules of A.

Example H85E11

We show the restriction of a module over an algebra A to a subalgebra of A.

```
> G := SmallGroup(32,7);
> A := BasicAlgebra(G);
> C, mu := Center(A);
> X := RightRegularModule(A);
> Z := JacobsonRadical(X);
> L := Restriction(Z,C,mu);
> L;
AModule L of dimension 31 over GF(2)
> A eq Algebra(L);
True
> IndecomposableSummands(L);
[
    AModule of dimension 1 over GF(2),
    AModule of dimension 30 over GF(2)
]
> Dimension(Socle(L));
16
```

Next we show how to pull back modules along a quotient map. We use the same algebra A.

```
> U := ideal<A|[A.13 +A.17]>;
> Q, theta := quo<A|U>;
> X := ProjectiveModule(Q,1);
> Y := ChangeAlgebras(X,A,theta);
> Y;
AModule Y of dimension 16 over GF(2)
```

Example H85E12

Here is another example of pulling back a module along a quotient map. This one involves algebras with more than one idempotent.

```
> load m11;
Loading "/usr/local/dmagma/libs/pergps/m11"
M11 - Mathieu group on 11 letters - degree 11
Order 7 920 = 2^4 * 3^2 * 5 * 11;  Base 1,2,3,4
Group: G
> A:= BasicAlgebraOfPrincipalBlock(G,GF(2));
> A;
Basic algebra of dimension 22 over GF(2)
Number of projective modules: 3
Number of generators: 9
> DimensionsOfProjectiveModules(A);
[ 8, 8, 6 ]
> I := ideal<A|[A.9]>;
> B, mu := quo<A|I>;
```

```
> B;
Basic algebra of dimension 6 over GF(2)
Number of projective modules: 2
Number of generators: 5
> P := ProjectiveModule(B,1);
> P;
AModule P of dimension 3 over GF(2)
> Q := ChangeAlgebras(P,A,mu);
> Algebra(Q) eq A;
true
```

Example H85E13

In this example, we investigate the structure of the projective modules of a basic algebra.

```
> G := PSL(3,3);
> N := Normalizer(G,Sylow(G,2));
> A := BasicAlgebraOfHeckeAlgebra(G,N,GF(2));
> DimensionsOfProjectiveModules(A);
[ 1, 2, 3, 9, 9, 1, 1, 1, 1 ]
> IsomorphismTypesOfRadicalLayers(ProjectiveModule(A,4));
[
    [ 4 ],
    [ 2, 3, 4, 5 ],
    [ 4, 4, 5 ],
    [ 5 ]
]
> IsomorphismTypesOfSocleLayers(ProjectiveModule(A,4));
[
    [ 4 ],
    [ 3, 4, 5 ],
    [ 2, 4, 5 ],
    [ 4, 5 ]
]
```

So we see that, unlike a group algebra, a Hecke algebra can have indecomposable projective modules whose socles are not simple.

85.7.4 Predicates

The following functions return a boolean value.

`IsSemisimple(M)`

Returns `true` if the module M is a semisimple module and `false` otherwise. If `true`, then the function also returns a list of the ranks of the primitive idempotents of the algebra. This is also a list of the multiplicities of the simple modules of the algebra as composition factors in a composition series for the module.

`IsProjective(M)`

Returns `true` if the module M is projective. The function also returns a sequence of multiplicities of the standard projective modules as direct summands of the projective cover of M.

`IsInjective(M)`

Returns `true` if the module M is injective. The function also returns a sequence of multiplicities of the standard injective modules as direct summands of the injective hull of M.

85.7.5 Elementary Operations

`m * b`

Given an element b in a basic algebra B and an element m in a module M over B, $m * b$ is the product.

Example H85E14

We obtain the dimensions of the radical layers of the group algebra of an extra special group of order 243 over a field of characteristic 3.

```
> G := ExtraSpecialGroup(3,2);
> G;
Permutation group G acting on a set of cardinality 243
> ff := GF(3);
> A := BasicAlgebra(G,ff);
> A;
Basic algebra of dimension 243 over GF(3)
Number of projective modules: 1
Number of generators: 6
> P := ProjectiveModule(A,1);
> P;
AModule P of dimension 243 over GF(3)
> R := JacobsonRadical(P);
> R;
AModule R of dimension 242 over GF(3)
> while Dimension(R) ne 0 do
>     T := JacobsonRadical(R);
>     print Dimension(R) - Dimension(T);
>     R := T;
> end while;
4
11
20
30
36
39
```

```
36
30
20
11
4
1
```

Example H85E15

We consider the mod-2 group algebra of an extraspecial group of order 128, and construct the module induced from the trivial module on the subgroups of order 4 generated by the first generator of the group.

```
> G := ExtraSpecialGroup(2,3);
> G;
Permutation group G acting on a set of cardinality 128
> F := GF(2);
> A := BasicAlgebra(G,F);
> A;
Basic algebra of dimension 128 over GF(2)
Number of projective modules: 1
Number of generators: 8
> A.1;
(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0)
```

Note that `A.1` is the unique idempotent (identity element) in the group algebra, whereas `A.2` is `G.1-1` where `G.1` is the first generator of the group.

```
> A.2;
(0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0)
> g := A.1+A.2;
```

Now we check the order of `g`.

```
> g^2;
(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0)
> g^4;
(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0)
```

```
> g^4 eq A!1;
true
```

So g has order 4.

```
> P := ProjectiveModule(A,1);
> P;
AModule P of dimension 128 over GF(2)
```

Note that P is generated by P.1 which corresponds to the identity element of A if we think of P as the algebra A as a module over itself. Now we create the induced module as the submodule generated by $(\mathtt{g}-1)^3$, since $(\mathtt{g}-1)^4 = 0$.

```
> U := sub<P|P.1*A.6>;
> U;
AModule U of dimension 32 over GF(2)
```

Because the dimension is a quarter of the order of the group we can be sure that we have the right thing by just checking that U is generated by a g fixed point.

```
> U.1*g eq U.1;
true
```

85.8 Homomorphisms of Modules

A homomorphism from module M to module N is simply a matrix that commutes with the action of the algebra on M and N.

85.8.1 Creation

`AHom(M, N)`

The space of homomorphisms from module M to module N.

`PHom(M,N)`

The space of projective homomorphisms from module M to module N. That is, the space of all homomorphisms that factor through a projective module.

`ZeroMap(M, N)`

The zero homomorphism from module M to module N.

`LiftHomomorphism(x, n)`

Given an element x in a module over a basic algebra and a natural number n, the function returns the homomorphism from the n^{th} projective module for the algebra to the module with the property that the idempotent e of the projective module maps to $x * e$.

`LiftHomomorphism(X, N)`

Given a sequence $X = [x_1, \ldots, x_t]$ of elements in a module M over a basic algebra and a sequence $N = [n_1, \ldots, n_s]$ of nonnegative integers, such that $n_1 + \ldots + n_s = t$, the function returns the homomorphism $P \to M$ from the projective module $P = \sum_{j=1}^{s} P_j^{n_i}$ to M that takes the idempotent e for the i^{th} summand in P to the element $x_i * e$ in M. Here P_j denotes the j^{th} projective module for the algebra.

`Pushout(M, f1, N1, f2, N2)`

The pushout of the diagram

$$\begin{array}{ccc} M & \xrightarrow{f_1} & N_1 \\ \downarrow{\scriptstyle f_2} & & \\ N_2 & & \end{array}$$

The function returns the module $L = (N_1 \oplus N_2)/\{(f_1(m), -f_2(m)) | m \in M\}$ and the homomorphisms $g_1 : N_1 \longrightarrow L$ and $g_2 : N_2 \longrightarrow L$ such that $f_1 g_1 = f_2 g_2$.

`Pullback(f1, M1, f2, M2, N)`

The pullback of the diagram

$$\begin{array}{ccc} & & M_2 \\ & & \downarrow{\scriptstyle f_2} \\ M_1 & \xrightarrow{f_1} & N \end{array}$$

The function returns the module $L = \{(m_1, m_2) \in M_1 \oplus M_2 | f_1(m_1) = f_2(m_2)\}$ and the homomorphisms $g_1 : L \longrightarrow M_1$ and $g_2 : L \longrightarrow M_2$ such that $g_1 f_1 = g_2 f_2$.

85.8.2 Access Functions

`IsModuleHomomorphism(f)`

Returns `true` if the map f is a homomorphism of modules over the algebra.

`Domain(f)`

The domain of f.

`Codomain(f)`

The codomain of f.

`Kernel(f)`

The kernel of f and the inclusion of the kernel in `Domain(f)`.

`Cokernel(f)`

The cokernel of f and the quotient map from `Codomain(f)` onto the cokernel.

85.8.3 Projective Covers and Resolutions

A projective cover of a module M is a projective module P and a surjective homomorphism $\phi : P \longrightarrow M$ such that P is minimal with respect to the property of having such a surjective homomorphism to M. A projective resolution to n steps of an A-module M is a pair consisting of a complex

$$P_n \xrightarrow{\partial_n} P_{n-1} \longrightarrow \ldots \longrightarrow P_1 \xrightarrow{\partial_1} P_0$$

which is exact except at the ends, and an augmentation homomorphism $\epsilon : P_0 \to M$ that is a projective cover of M. In addition, the image of ∂_1 must equal the kernel of ϵ. The resolution is minimal if each P_i is a projective cover of its image in P_{i-1}. In this case the i^{th} syzygy module is the image of ∂_i.

In the implementation the main function is `CompactProjectiveResolution.` This function computes a minimal projective resolution of a given module and stores the minimal amount of information that is necessary to create the boundary maps and the terms of the resolution. It runs relatively fast because it avoids the computation of the terms of the projective resolution as modules over the algebra. Instead the terms in the compact resolution are only vector spaces together with a sequence of types for the projective modules. The other information that is recorded is the sequence of images of the generators for the indecomposable projective modules. That is, for the boundary map

$$P_n \xrightarrow{\partial_n} P_{n-1}$$

the module $P_n \cong \oplus_{i=1}^{m} Q_i$ where each Q_i is an indecomposable projective module generated by an element a_i corresponding to the appropriate idempotent in the basic algebra. The function records the images $\partial_n(a_i)$ as a sequence of vectors in the vector space of the module P_{n-1}.

`ProjectiveCover(M)`

The projective cover of the module M given as the projective module P, the surjective homomorphism of P onto M, the sequences of inclusion and projection homomorphism of P from and to its indecomposable direct summands and the isomorphism type of P in the form of a list of the number of copies of the projective modules of the algebra of each type that make up P.

`ProjectiveResolution(M, n)`

The complex giving the minimal projective resolution of the module M out to n steps together with the augmentation homomorphism from the projective cover of M into M. Note that homomorphisms go from left to right so that last term of the complex (in degree 0) is the projective cover of M and the cokernel of the last homomorphism in the complex is M. The complex is constructed from the compact projective resolution of M. The function creates the compact projective resolution if it has not already been computed.

`CompactProjectiveResolution(M, n)`

Returns a minimal projective resolution for the module M out to n steps in compact form together with the augmentation map ($P_0 \to M$). The compact form of the resolution is a list of the minimal pieces of information needed to reconstruct the boundary maps in the resolution. That is the boundary map ($P_i \xrightarrow{\partial_i} P_{i-1}$) is recorded as a matrix whose entries are the images of the generators for indecomposable projective modules making up P_i in the indecomposable projective modules making up P_{i-1}. If a compact projective resolution has been previously computed to degree m and $m < n$ then the function extends that resolution by $n - m$ steps. If $m \geq n$ the function returns the previously computed compact projective resolution. The function returns a record with the fields:

(a) The list of isomorphism types of the projective modules in the resolution, each given as a sequence of integers giving the number of direct summands of each indecomposable projective in the module (field name `BettiNumbers`).

(b) The record of the boundary maps (field name `ResolutionRecord`).

(c) The module M (field name `Module`).

(d) The augmentation map (field name `AugmentationMap`).

(e) The type of the resolution, whether projective or injective (field name `Typ`).

`CompactProjectiveResolutionsOfSimpleModules(A,n)`

Returns a sequence of the compact projective resolutions of the simple A-modules computed to degree n.

`SyzygyModule(M, n)`

The n^{th} syzygy module of the module M. The module is constructed from the compact projective resolution of M. The compact resolution is constructed if it does not already exist.

`SimpleHomologyDimensions(M)`

The sequence of sequences of dimensions of the homology groups $\mathrm{Tor}_j(S_i, M)$ for simple modules S_i and module M, to the extent that they have been computed.

Example H85E16

We consider the basic algebra of a quiver with relation. The quiver has four nodes and 5 non-idempotent generators (a, b, c, d, e). The first goes from node 1 to node 2, the second from 2 to 3, the third from 3 to 4, the fourth from 3 to 2 and the last from 4 to 1. They satisfy the relations $bcfadbd = (abcf)^5ab = (bd)^2b = 0$.

```
> ff := GF(8);
> FA<e1,e2,e3,e4,a,b,c,d,f> := FreeAlgebra(ff,9);
> rrr := [b*c*f*a*b*d*b,a*b*c*f*a*b*c*f*a*b*c*f*a*b*c*f*a*b*c*f*a*b,
>     b*d*b*d*b];
> BA := BasicAlgebra(FA,rrr,4,[<1,2>,<2,3>,<3,4>,<3,2>,<4,1>]);
```

```
> BA;
Basic algebra of dimension 296 over GF(2^3)
Number of projective modules: 4
Number of generators: 9
```

Now we take the projective resolutions of the simple modules out to 5 steps. We print the type of the projective module at each stage.

```
> for i := 1 to 4 do
>     S := SimpleModule(BA,i);
>     prj := CompactProjectiveResolution(S, 5);
>     SimpleHomologyDimensions(S);
> end for;
[
    [ 0, 0, 10, 0 ],
    [ 0, 0, 5, 0 ],
    [ 0, 0, 2, 0 ],
    [ 0, 0, 1, 0 ],
    [ 0, 1, 0, 0 ],
    [ 1, 0, 0, 0 ]
]
[
    [ 0, 0, 16, 0 ],
    [ 0, 0, 8, 0 ],
    [ 0, 0, 4, 0 ],
    [ 0, 0, 2, 0 ],
    [ 0, 0, 1, 0 ],
    [ 0, 1, 0, 0 ]
]
[
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 1, 0, 1 ],
    [ 0, 0, 1, 0 ]
]
[
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 1, 0, 0, 0 ],
    [ 0, 0, 0, 1 ]
]
```

So we see that the third and fourth simple modules have finite projective dimension. The projective resolutions of the first and second simple modules appear to have exponential rates of growth but

the terms after the second term are all direct sums of copies of the third projective module.

```
> for i := 1 to 4 do
>     Dimension(Socle(ProjectiveModule(BA,i)));
> end for;
12
13
25
12
```

Notice that the socles of the projective modules have very large dimensions so the injective resolutions are probably going to grow at a very rapid rate.

Example H85E17

We create the quotient of the group algebra of a p-group by the ideal generated by a central element. In particular, we choose the group algebra of an extra special 3-group and factor out the ideal generated by $(z-1)^2$ where z is a central element of order 3. Then we compare the size of the projective resolution of the trivial module for the new algebra with that of the antecedent group algebra.

```
> G := ExtraSpecialGroup(3,1);
> F := GF(3);
> B := BasicAlgebra(G,F);
> B;
Basic algebra of dimension 27 over GF(3)
Number of projective modules: 1
Number of generators: 4
> s := NonIdempotentGenerators(B)[3];
```

Now check that `s` is in the center.

```
> [s*x eq x*s: x in Generators(B)];
[ true, true, true, true ]
> P := ProjectiveModule(B,1);
> Q := quo<P|P.1*s^2>;
> Q;
AModule Q of dimension 18 over GF(3)
```

We need to create the path tree for the projective module of the matrix algebra of the action on `Q`. In this case it is an easy exercise because the last 9 element of the basis of the projective module for `B` span the submodule that we are factoring out. This can actually be seen from the path tree for the projective module of `B`.

```
> PathTree(B,1);
[ <1, 1>, <1, 2>, <2, 2>, <1, 3>, <2, 3>, <3, 3>, <4, 3>,
<5, 3>, <6, 3>, <1, 4>, <2, 4>, <3, 4>, <4, 4>, <5, 4>,
<6, 4>, <7, 4>, <8, 4>, <9, 4>, <10, 4>, <11, 4>, <12, 4>,
<13, 4>, <14, 4>, <15, 4>, <16, 4>, <17, 4>, <18, 4> ]
```

So we get the path tree for the new module by truncation.

```
> PT := [PathTree(B,1)[j]: j in [1 .. 18]];
```

```
> PT;
[ <1, 1>, <1, 2>, <2, 2>, <1, 3>, <2, 3>, <3, 3>,
<4, 3>, <5, 3>, <6, 3>, <1, 4>, <2, 4>, <3, 4>,
<4, 4>, <5, 4>, <6, 4>, <7, 4>, <8, 4>, <9, 4> ]
```

Now form the new basic algebra.

```
> C := BasicAlgebra([<Action(Q),PT>]);
> C;
Basic algebra of dimension 18 over GF(3)
Number of projective modules: 1
Number of generators: 4
> S := SimpleModule(C,1);
> prj := CompactProjectiveResolution(S, 15);
> SimpleHomologyDimensions(S);
[ 92, 77, 70, 57, 51, 40, 35, 26, 22, 15, 12, 7, 5, 2, 1 ]
```

Now compare this with the projective resolution for the group algebra.

```
> T := SimpleModule(B,1);
> pj2 := CompactProjectiveResolution(T,15);
> SimpleHomologyDimensions(T);
[ 20, 18, 17, 16, 15, 14, 12, 10, 9, 8, 7, 6, 4, 2, 1 ]
```

85.9 Duals and Injectives

If k is the base ring for the algebra A then the k-dual $\mathrm{Hom}_k(M, k)$ for a right module M over A is a right module over the opposite algebra OA of A. Furthermore, the dual of a projective OA-module is an injective A-module and the dual of a projective OA-resolution of a module M is an A-injective resolution of the dual of M.

`Dual(M)`

Given a module M defined over a basic algebra M, this function returns the dual of M as a module over the opposite of the algebra of M. Note that the opposite of the algebra of M is created if it does not already exist.

`BaseChangeMatrix(A)`

Given a basic algebra A that has opposite algebra O, the function creates the change of basis matrix B from the vector space of A to the vector space of O, so that if x, y are in A then $(xy)B$ is the same as $(yB)(xB) \in O$.

85.9.1 Injective Modules

Injective hulls, and injective resolutions of a module are computed by taking the projective cover or projective resolution of the dual module over the opposite algebra and then again taking the dual to retrieve modules or complexes over the original algebra. If the opposite algebra of the module has not been computed then it will be created in the evaluation of any of the injective module functions.

`InjectiveModule(B, i)`

The i^{th} injective module of the algebra B.

`InjectiveHull(M)`

The injective hull of the module M is the injective module I of minimal dimension such that there is an inclusion $\iota : M \to I$. The function returns I, ι, the sequences of inclusions and projections from and to the indecomposable injective summands of I, and the type of I as a sequence $T := [t_1 \ldots, t_s]$ where I has t_1 summands of type 1, t_2 of type 2, etc.

`InjectiveResolution(M, n)`

The complex giving the minimal injective resolution of the module M together with the inclusion homomorphism from M into its injective hull. Note that homomorphisms go from left to right so that the kernel of the first homomorphism in the complex is M. The function computes the compact injective resolution and creates the complex of the injective resolution from that.

`CompactInjectiveResolution(M, n)`

A minimal injective resolution for the module M out to n steps in compact form together with the coaugmentation map ($M \to I_0$). The compact form of the resolution is a list of the minimal pieces of information needed to reconstruct the boundary maps in the resolution. That is, the boundary map ($I_{i-1} \xrightarrow{\partial_i} I_i$) is recorded as a matrix whose entries are the images of the generators for indecomposable injective modules making up I_{i-1} in the indecomposable projective modules making up I_i. The actual return of the function is the compact projective resolution of the dual module of M over the opposite algebra of the algebra of M. The return is a record with the fields:

(a) The list of isomorphism types of the injective modules in the resolution, each given as a sequence of integers giving the number of direct summands of each indecomposable injective in the module (field name `BettiNumbers`).

(b) The record of the boundary maps (field name `ResolutionRecord`).

(c) The module M (field name `Module`).

(d) The coaugmentation map (field name `CoaugmentationMap`).

(e) The type of the resolution, whether projective or injective (field name `Typ`).

`InjectiveSyzygyModule(M, n)`

The n^{th} injective-syzygy module of M. The module is constructed from the compact injective resolution of M. The compact resolution is constructed if it does not already exist.

`SimpleCohomologyDimensions(M)`

The sequence of sequences of dimensions of the cohomology groups $\mathrm{Ext}^j(S_i, M)$ for simple modules S_i and module M, to the extent that they have been computed.

Example H85E18

We create the basic algebra of a quiver over a field with 8 elements. The quiver has two nodes and three arrows, going from node 1 to node 2, from 2 to 1 and from 1 to 1. The relations are given in the sequence `rrr`.

```
> ff := GF(8);
> FA<e1,e2,a,b,c> := FreeAlgebra(ff,5);
> rrr := [a*b*a*b*a, c*c*c*c, a*b*c - c*a*b];
> B := BasicAlgebra(FA,rrr,2,[<1,2>,<2,1>,<1,1>]);
> B;
Basic algebra of dimension 41 over GF(2^3)
Number of projective modules: 2
Number of generators: 5
> DimensionsOfProjectiveModules(B);
[ 20, 21 ]
> DimensionsOfInjectiveModules(B);
[ 24, 17 ]
> P1 := ProjectiveModule(B,1);
> Socle(P1);
AModule of dimension 1 over GF(2^3)
```

We consider the injective resolution of the first projective module.

```
> time in1 := CompactInjectiveResolution(P1,10);
reverse trees
Time: 3.850
```

Note that part of the time was required to create the opposite algebra of `B`.

```
> SimpleCohomologyDimensions(P1);
[
    [ 1, 0 ],
    [ 0, 4 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ],
    [ 4, 0 ],
```

```
    [ 4, 0 ],
    [ 4, 0 ]
]
```

The injective resolution appears to be periodic. Now we look at a module constructed from the resolution.

```
> M := InjectiveSyzygyModule(P1,6);
> M;
AModule M of dimension 64 over GF(2^3)
```

Consider the space of endomorphisms of M.

```
> hh := AHom(M,M);
> hh;
KMatrixSpace of 64 by 64 matrices and dimension 128 over GF(2^3)
> [Rank(hh.i): i in [1 .. Dimension(hh)]];
[ 16, 16, 16, 16, 12, 12, 12, 12, 8, 8, 8, 8, 8, 8, 8, 8, 6, 6, 6, 6, 4, 4, 4,
4, 4, 4, 4, 4, 2, 2, 2, 2, 4, 8, 12, 16, 16, 12, 8, 4, 8, 4, 12, 16, 4, 8, 12,
16, 16, 16, 16, 16, 2, 4, 6, 8, 8, 6, 4, 2, 4, 2, 6, 8, 2, 4, 6, 8, 12, 12, 12,
12, 8, 8, 8, 8, 8, 8, 8, 8, 6, 6, 6, 6, 4, 4, 4, 4, 4, 4, 4, 4, 2, 2, 2, 2, 16,
16, 16, 16, 12, 12, 12, 12, 8, 8, 8, 8, 8, 8, 8, 8, 6, 6, 6, 6, 4, 4, 4, 4, 4,
4, 4, 4, 2, 2, 2, 2 ]
```

Note that no generator of the endomorphism ring has rank more than 16. This would indicate that the module is decomposable since the identity map must be a sum of homomorphisms of smaller rank.
How can we produce a decomposition? One method is the following.

```
> vv := Random(hh);
> Rank(vv);
64
> vv*vv eq vv;
false
> [Rank(vv*vv-u*vv):u in ff];
[ 60, 64, 64, 64, 64, 64, 64, 64 ]
> [u:u in ff];
[ 1, ff.1, ff.1^2, ff.1^3, ff.1^4, ff.1^5, ff.1^6, 0 ]
> Rank(vv*vv - vv);
60
> U := vv*vv - vv;
> Rank(U);
60
> Rank(U*U);
56
> Rank(U*U*U);
52
> Rank(U*U*U*U);
48
> Rank(U*U*U*U*U);
48
```

```
> Rank(U*U*U*U*U*U);
48
> T := U*U*U*U;
> N1 := Kernel(T);
> N2 := Image(T);
> Dimension(N1);
16
> Dimension(N2);
48
> Dimension(N1+N2);
64
```

So the sum of `N1` and `N2` must be all of `M` and by counting dimensions, it must be a direct sum.

85.10 Cohomology

`CohomologyRingGenerators(P)`

Given a compact projective resolution P for a simple module S over a basic algebra A, the function returns the chain maps in compact form of a minimal set of generators for the cohomology $\mathrm{Ext}^*_A(S,S)$, as well as some other information. The record that is returned has the following fields:

(a) The list of maps in compact form for the chain map of the generators (field name `ChainMapRecord`).

(b) The sequence of degrees of cohomology generators (field name `ChainDegrees`).

(c) The tops of the chain maps (maps on modules modulo radicals) for the purposes of computing products (field name `TopsOfCohomologyGenerators`).

(d) The tops of the chain maps representing monomials in the generators (field name `TopsOfCohomologyChainMaps`).

(e) The original compact projective resolution (field name `ProjectiveResolution`).

`CohomologyRightModuleGenerators(P, Q, CQ)`

Given projective resolutions P and Q for simple modules S and T over a basic algebra A and the cohomology generators CQ for T associated to the resolution Q, the function returns the chain maps in compact form of the minimal generators for the cohomology $\mathrm{Ext}^*_A(S,T)$ as a right module over the cohomology ring $\mathrm{Ext}^*_A(T,T)$. The function returns a record consisting of the following fields.

(a) The list of maps in compact form for the chain map of each cohomology generator (field name `ChainMapRecord`).

(b) The sequence of degrees of cohomology generators (field name `ChainDegrees`).

(c) The tops of the chain maps (maps on modules module radicals) for the purposes of computing products (field name `TopsOfCohomologyGenerators`).

`CohomologyLeftModuleGenerators(P, CP, Q)`

Given projective resolutions P and Q for simple modules S and T over a basic algebra A and the cohomology generators CP for T associated to the resolution Q, the function returns the chain maps in compact form of the minimal generators for the cohomology $\mathrm{Ext}^*_A(S,T)$ as a left module over the cohomology ring $\mathrm{Ext}^*_A(S,S)$. The function returns a record consisting of the following fields.

(a) The list of maps in compact form for the chain map of each cohomology generator (field name `ChainMapRecord`).

(b) The sequence of degrees of cohomology generators (field name `ChainDegrees`).

(c) The tops of the chain maps (maps on modules module radicals) for the purposes of computing products (field name `TopsOfCohomologyGenerators`).

`DegreesOfCohomologyGenerators(C)`

Given the generators C for cohomology, as either module generators or as ring generators, the function returns the list of degrees of the minimal generators.

`CohomologyGeneratorToChainMap(P, Q, C, n)`

Given the projective resolutions P and Q of two modules M and N and the cohomology generators C of the cohomology module, $\mathrm{Ext}^*_B(M,N)$, the function returns the chain map from P to Q that lifts the n^{th} generator of the cohomology module and has degree equal to the degree of that generator.

`CohomologyGeneratorToChainMap(P, C, n)`

Given the projective resolution P of a module and the cohomology generators C of the cohomology ring of that module, the function returns the chain map from P to P that lifts the n^{th} generator of the cohomology ring and has degree equal to the degree of that generator.

Example H85E19

We create the Basic algebra for the principal block of the sporadic simple group M_{11} in characteristic 2. The block algebra has three simple modules of dimension 1, 44, and 10. The basic algebra has dimension 22.

```
> ff := GF(2);
> VV8 := VectorSpace(ff,8);
> BB8 := Basis(VV8);
> MM8 := MatrixAlgebra(ff,8);
> e11 := MM8!0;
> e12 := MM8!0;
> e13 := MM8!0;
> e11[1]  := BB8[1];
> e11[4]  := BB8[4];
> e11[5]  := BB8[5];
> e11[8]  := BB8[8];
```

```
> e12[2]  := BB8[2];
> e12[7]  := BB8[7];
> e13[3]  := BB8[3];
> e13[6]  := BB8[6];
> a1 := MM8!0;
> b1 := MM8!0;
> c1 := MM8!0;
> d1 := MM8!0;
> e1 := MM8!0;
> f1 := MM8!0;
> a1[1] :=  BB8[2];
> a1[5] :=  BB8[7];
> b1[1] :=  BB8[3];
> b1[4] :=  BB8[6];
> c1[2] :=  BB8[4];
> c1[7] :=  BB8[8];
> e1[3] :=  BB8[5];
> e1[6] :=  BB8[8];
> f1[3] :=  BB8[6];
> A1 := sub< MM8 | [e11, e12, e13, a1, b1, c1, d1, e1, f1] >;
> T1 := [ <1,1>,<1,4>,<1,5>,<2,6>,<3,8>,<4,5>,<5,4>,<6,8>];
> VV6 := VectorSpace(ff,6);
> BB6 := Basis(VV6);
> MM6 := MatrixAlgebra(ff,6);
> e21 := MM6!0;
> e22 := MM6!0;
> e23 := MM6!0;
> e22[1]  := BB6[1];
> e22[5]  := BB6[5];
> e22[6]  := BB6[6];
> e21[2]  := BB6[2];
> e21[4]  := BB6[4];
> e23[3]  := BB6[3];
> a2 := MM6!0;
> b2 := MM6!0;
> c2 := MM6!0;
> d2 := MM6!0;
> e2 := MM6!0;
> f2 := MM6!0;
> a2[4] := BB6[6];
> b2[2] := BB6[3];
> c2[1] := BB6[2];
> d2[1] := BB6[5];
> d2[5] := BB6[6];
> e2[3] := BB6[4];
> A2 := sub< MM6 | [e21, e22, e23, a2, b2, c2, d2, e2, f2]>;
> T2 := [ <1,2>, <1,6>, <2,5>, <3,8>, <1,7>, <5,7> ];
> VV8 := VectorSpace(ff,8);
```

```
> BB8 := Basis(VV8);
> MM8 := MatrixAlgebra(ff,8);
> e31 := MM8!0;
> e32 := MM8!0;
> e33 := MM8!0;
> e31[2]  := BB8[2];
> e31[6]  := BB8[6];
> e32[4]  := BB8[4];
> e33[1]  := BB8[1];
> e33[3]  := BB8[3];
> e33[5]  := BB8[5];
> e33[7]  := BB8[7];
> e33[8]  := BB8[8];
> a3 := MM8!0;
> b3 := MM8!0;
> c3 := MM8!0;
> d3 := MM8!0;
> e3 := MM8!0;
> f3 := MM8!0;a3[2] :=  BB8[4];
> b3[6] :=  BB8[8];
> b3[2] :=  BB8[7];
> c3[4] :=  BB8[6];
> e3[1] :=  BB8[2];
> e3[3] :=  BB8[6];
> f3[1] :=  BB8[3];
> f3[3] :=  BB8[5];
> f3[5] :=  BB8[7];
> f3[7] :=  BB8[8];
> A3 := sub< MM8 | [e31, e32, e33, a3, b3, c3, d3, e3, f3] >;
> T3 := [ <1,3>,<1,8>,<1,9>,<2,4>,<3,9>,<4,6>,<5,9>,<6,5>];
>
> m11 := BasicAlgebra( [<A1, T1>, <A2, T2>, <A3, T3>] );
> m11;
Basic algebra of dimension 22 over GF(2)
Number of projective modules: 3
Number of generators: 9
> s1 := SimpleModule(m11,1);
> s2 := SimpleModule(m11,2);
```

Now we compute the projective resolutions of the first and second simple modules. Then we find the degrees of their cohomology ring generators.

```
> prj1 := CompactProjectiveResolution(s1,20);
> prj2 := CompactProjectiveResolution(s2,20);
> CR1 := CohomologyRingGenerators(prj1);
> CR2 := CohomologyRingGenerators(prj2);
> DegreesOfCohomologyGenerators(CR1);
[ 3, 4, 5 ]
> DegreesOfCohomologyGenerators(CR2);
```

```
[ 1, 2 ]
```

Finally we look at the cohomology Ext(`s2`, `s1`) as a left module over the cohomology ring of `s1` and as a right module over the cohomology ring of `s2`.

```
> CR12 := CohomologyLeftModuleGenerators(prj1,CR1,prj2);
> DegreesOfCohomologyGenerators(CR12);
[ 1, 2, 3, 4 ]
> CR12 := CohomologyRightModuleGenerators(prj1,prj2,CR2);
> DegreesOfCohomologyGenerators(CR12);
[ 1 ]
```

So as a module over the cohomology ring of `s1` it is generated by 4 elements. But as a module over the cohomology ring of `s2` it is generated by a single element.
Next we get the chain complex for the projective resolution of the first simple module and the chain map for the third generator of the cohomology ring of the first simple module.

```
> pj1 := ProjectiveResolution(s1,20);
> pj1;
Basic algebra complex with terms of degree 20 down to 0
Dimensions of terms: 74 66 68 68 60 54 54 54 48 40 40 42 34 26 28 28 20 14 14
   14 8
> gen113 := CohomologyGeneratorToChainMap(pj1,CR1,3);
> gen113;
Basic algebra chain map of degree -5
```

We can compose this with itself.

```
> gen113*gen113;
Basic algebra chain map of degree -10
```

Now compute the kernel and the dimensions of the homology of the kernel.

```
> Ker, phi := Kernel(gen113);
> Ker, phi;
Basic algebra complex with terms of degree 20 down to 0
Dimensions of terms: 20 15 19 20 20 17 17 20 22 15 17 22 20 15 19 20 20 14 14
14 8
Basic algebra chain map of degree 0
> DimensionsOfHomology(Ker);
[ 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 3, 0, 0, 0 ]
```

We apply the same procedure in the case of the cokernel.

```
> Cok, mu := Cokernel(gen113);
> Cok, mu;
Basic algebra complex with terms of degree 20 down to 0
Dimensions of terms: 74 66 68 68 60 0 3 5 0 0 3 5 0 0 3 5 0 0 3 5 0
Basic algebra chain map of degree 0
> DimensionsOfHomology(Cok);
```

```
[ 0, 0, 0, 27, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2 ]
```

We can also check the image.

```
> Imm, theta, gamma := Image(gen113);
> Imm;
Basic algebra complex with terms of degree 20 down to 0
Dimensions of terms: 0 0 0 0 0 54 51 49 48 40 37 37 34 26 25 23 20 14 11 9 8
> DimensionsOfHomology(Imm);
[ 0, 0, 0, 0, 27, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0 ]
```

We can check that certain things make sense.

```
> IsChainMap(theta);
true
> IsChainMap(gamma);
true
```

85.10.1 Ext-Algebras

The ext-algebra of an algebra B is the algebra $Ext^*_B(S,S)$ for $S = S_1 \oplus \ldots \oplus S_n$ where $S_1, \ldots, S_n$ are all of the simple B-modules. In the event that the algebra B had finite global dimension, this is a finite dimensional algebra and we can form its basic algebra.

`ExtAlgebra(A, n)`

The function computes the information on the ext-algebra B of the basic algebra A where the projecive resolutions and cohomology have been computed to degree n. The function returns a record carrying the data:

(i) A free algebra F,

(ii) A list of relations in the elements of F, such that the ext-algebra is the quotient F/I where I is the ideal generated by the relations,

(iii) The sequence of chain maps of the generators. Each chain map goes from the projective resolution of one simple module to that of another.

(iv) The seqence of degrees of the generators.

(v) A sequence of sequences of sequences of basis element such the j^{th} element of the i^{th} sequence is a basis of in $Ext^*_A(S_i, S_j)$ where S_i is the i^{th} simple module.

(vi) A basis of the entire ext-algebra, the concatenation of the previous sequences.

(vii) The number of steps of the cohomology that were computed.

(viii) The global dimension that has been computed. This number is smaller than the number of steps only in the case that the global dimension of A is less than n, meaning that the n^{th} step in the projective resolution of any simple module is the zero module.

`BasicAlgebraOfExtAlgebra(ext)`

The function creates the basic algebra from a computed ext-algebra. The input is the output of the `ExtAlgebra` function. If the ext-algebra is not verified to be finite dimensional by the computation, then an error is returned.

`BasicAlgebraOfExtAlgebra(A)`

The function forms the basic algebra from a computed ext-algebra of the basic algebra A. If no ext-algebra for A has been computed or if the ext-algebra is not verified to be finite dimensional then an error is returned.

`BasicAlgebraOfExtAlgebra(A, n)`

The function creates the basic algebra for the ext-algebra of A computed to n steps. If no ext-algebra for A to n steps has been computed then it computes one. If the ext-algebra is not verified to be finite dimensional by the computation, then an error is returned.

`SumOfBettiNumbersOfSimpleModules(A, n)`

This function computes the Betti numbers of all of simple A-modules out to degree n. This is the dimension of the ext-algebra of A computed to degree n.

Example H85E20

We construct the basic algebra of the algebra B of lower triangular matrices over the field with 5 elements. Note that because the identity element of B is the sum of primitive idempotents of rank 1, B is actually isomorphic to its basic algebra.

```
> A := MatrixAlgebra(GF(5),10);
> U := A!0;
> ElementaryMatrix := function(i,j);
>     W := U;
>     W[i,j] := 1;
>     return W;
> end function;
> S := &cat[[ElementaryMatrix(i,j): i in [j .. 10]]:j in [1 .. 10]];
> B := sub<A|S>;
> B;
Matrix Algebra of degree 10 with 55 generators over GF(5)
> C := BasicAlgebra(B);
> C;
Basic algebra of dimension 55 over GF(5)
Number of projective modules: 10
Number of generators: 19
```

Note that C has the same dimension as B. The two are isomorphic, though they have different types.

```
> SumOfBettiNumbersOfSimpleModules(C,9);
19
```

```
> SumOfBettiNumbersOfSimpleModules(C,10);
19
```

From the above we see that C has global dimension at most 9. Consequently, we can compute the basic algebra of its ext-algebra.

```
> D:= ExtAlgebra(C,10);
> E := BasicAlgebraOfExtAlgebra(D);
> E;
Basic algebra of dimension 19 over GF(5)
Number of projective modules: 10
Number of generators: 19
> SumOfBettiNumbersOfSimpleModules(E,8);
54
> SumOfBettiNumbersOfSimpleModules(E,9);
55
> SumOfBettiNumbersOfSimpleModules(E,10);
55
```

Here we see that E has global dimension 9. So we compute the basic algebra of its ext-algebra.

```
> F := BasicAlgebraOfExtAlgebra(E,10);
> F;
Basic algebra of dimension 55 over GF(5)
Number of projective modules: 10
Number of generators: 19
> G := BasicAlgebraOfExtAlgebra(F,10);
> G;
Basic algebra of dimension 19 over GF(5)
Number of projective modules: 10
Number of generators: 19
```

So it would appear that C and F are isomorphic as well as E and G.

85.11 Group Algebras of p-groups

There is a special type for the basic algebras which are the modular group algebras of p-groups for p a prime. If G is a finite p and k is a field of characteristic p, then the commands `BasicAlgebra(`G, k`)` and `BasicAlgebra(`G`)` automatically create a basic algebra of type `AlgBasGrpP`. The type is optimized for the computation of cohomology rings. Included for this type are restriction and inflation maps. Most of the functions for modules and complexes are the same as for general basic algebras.

85.11.1 Access Functions

`Group(A)`

The group which defines the algebra A.

`PCGroup(A)`

The internal PC group of the algebra A.

`PCMap(A)`

The map from `Group(A)` to `PCGroup(A)` for an algebra A.

`AModule(M)`

Converts a `GModule` M over a p-group to a module over the basic algebra of that group.

`GModule(M)`

Returns the standard module of the algebra A as a module over `Group(A)` and as a module over `PCGroup(A)`.

`GModule(M)`

Converts a module M for the basic algebra of a p-group into a module over the p-group.

85.11.2 Projective Resolutions

`ResolutionData(A)`

Returns the data needed to compute the projective resolution of an A-module for an algebra A. The data is given as a record with the fields:

(a) The matrices of the `PCGenerators` of the p-group on the standard indecomposable projective module for the algebra (field name `PCgenMats`).

(b) The matrices of the minimal generators of the p-group on the standard indecomposable projective module for the algebra (field name `MingenMats`).

(c) The algebra A (field name `Algebra`).

`CompactProjectiveResolutionPGroup(M, n)`

`CompactProjectiveResolution(M, n)`

Computes the projective resolution of the module M out to n steps. The function returns a record with the fields:

(a) The list of the ranks of the projective modules in the resolution (field name `BettiNumbers`).

(b) The record of the boundary maps (field name `ResolutionRecord`).

(c) The module M (field name `Module`).

(d) The augmentation map (field name `AugmentationMap`).

(e) The type of the resolution, whether projective or injective (field name `Typ`).

`ProjectiveResolutionPGroup(PR)`

The projective resolution as a complex of modules over the basic algebra of the group algebra, computed from the compact projective resolution PR.

`ProjectiveResolution(M, n)`

The projective resolution of the module M computed as a complex out to n steps. The function also returns the augmentation map.

`ProjectiveResolution(PR)`

The projective resolution computed from a compact projective resolution PR as a complex. The function also returns the augmentation map.

85.11.3 Cohomology Generators

`AllCompactChainMaps(PR)`

Creates the data on the chain maps for all generators of the cohomology of the simple module k in degrees within the limits of the compact projective resolution PR of the simple module. The function returns a record having the following information.

(a) The record of the chain maps of the generators of cohomology (field name `ChainMapRecord`).

(b) The sequence of sizes of the chain map record (field name `ChainSizes`).

(c) The degrees of the chain maps (field name `ChainDegrees`).

(d) The list of cocycles representing the generators (field name `Cocycles`).

(e) The record of the products of the generators (field name `ProductRecord`).

(f) The locations of the products of the generators (field name `ProductLocations`). Much of the information is for use in the computation of the cohomology ring.

`CohomologyElementToChainMap(P, d, n)`

Creates a chain map from the projective resolution P to itself for the element number n in degree d of cohomology.

`CohomologyElementToCompactChainMap(PR, d, n)`

Creates a chain map in compact form from the compact projective resolution PR to itself for the element number n in degree d of cohomology.

85.11.4 Cohomology Rings

`CohomologyRing(k, n)`

`CohomologyRing(PR, AC)`

The cohomology ring of the unique simple module k for the basic algebra of the group algebra of a p-group. The input can be given either as the module k and the number of steps n or as the compact projective resolution PR of k together with AC, the calculation of the chain map generators of the cohomology. In the former case the compact resolution and the chain map of the generators are computed in the process. The ring is returned as a record having the following fields:

(a) The polynomial ring or free graded-commutative k-algebra R generated by the cohomology generators (field name `PolRing`).

(b) The ideal of relations in R satisfied by the cohomology generators (field name `RelationsIdeal`).

(c) The list of relations that have been computed (field name `ComputedRelations`).

(d) The chain maps giving the tops of the monomial in the cohomology generators (field name `MonomialData`).

(e) The number of computed steps in the resolution (field name `NumberOfSteps`).

`MinimalRelations(R)`

A minimal set of relations generating the relations ideal of a cohomology ring R.

85.11.5 Restrictions and Inflations

`RestrictionData(A,B)`

Assuming that A is the basic algebra of a p-group G and that B is the basic algebra of a subgroup of G, the function returns the change of basis matrix that make the standard free module for A into a direct sum of standard free modules for B. It also returns the inverse of the matrix and a set of coset representatives of the `PCGroup(B)` in `PCGroup(A)`.

`RestrictResolution(PR, RD)`

Takes the compact projective resolution PR for the trivial module of G and the resolution data RD for the basic algebra of a subgroup H and returns the restriction of the resolution to a complex of modules over the basic algebra for H.

`RestrictionChainMap(P1,P2)`

Computes the chain map from the resolution $P2$ of the simple module for the basic algebra of a subgroup H of a group G to the restriction to H of the resolution $P1$ of the simple module for the basic algebra of G. The inputs $P1$ and $P2$ must be in compact form.

`RestrictionOfGenerators(PR1, PR2, AC1, AC2, REL2)`

Computes the sequence of images of the generators of the cohomology ring of G restricted to a subgroup H. The input is the projective resolutions and cohomology generators for the basic algebra of G ($PR1$ and $AC1$) and for the basic algebra of the subgroup ($PR2$ and $AC2$), as well as the cohomology relations for the subgroup, $REL2$.

`InflationMap(PR2, PR1, AC2, AC1, REL1, theta)`

Returns the images of the generators of the cohomology ring of a quotient group Q in the cohomology ring of a group G. The input θ is the quotient map $G \to Q$. Other input is the projective resolutions and cohomology generators for the basic algebra of G ($PR1$ and $AC1$) and for the quotient group Q ($PR2$ and $AC2$) as well as the cohomology relations for G, $REL1$.

Example H85E21

We create the cohomology ring of a group G of order 64 and find a cyclic subgroup Z of the center of G. We compute the restriction of the cohomology of G to the cohomology of Z and also the inflation of the cohomology of G/Z to the cohomology ring of G.

```
> SetSeed(1);
> G := SmallGroup(64,7);
> Z := sub<G| Random(Center(G))>;
> G;
GrpPC : G of order 64 = 2^6
PC-Relations:
    G.1^2 = G.4,
    G.2^2 = G.5,
    G.3^2 = G.5,
    G.4^2 = G.6,
    G.2^G.1 = G.2 * G.3,
    G.3^G.1 = G.3 * G.5,
    G.3^G.2 = G.3 * G.5
> #Z, [G!Z.i: i in [1 .. Ngens(Z)]];
4 [ G.4 ]
```

So we see that Z has order 4 and is generated by the element $G.4$. Now construct the quotient and the basic Algebras.

```
> Q, mu := quo<G|Z>;
> A := BasicAlgebra(G);
> B := BasicAlgebra(Q);
> C := BasicAlgebra(Z);
```

Next we want the simple modules and the cohomology rings. We compute the cohomology out to 17 steps which should be more than enough to get the generators and relations.

```
> k := SimpleModule(A,1);
> kk := SimpleModule(B,1);
```

```
> kkk := SimpleModule(C,1);
> time R := CohomologyRing(k,17);
Time: 2.060
> time S := CohomologyRing(kk,17);
Time: 0.140
> time T := CohomologyRing(kkk,17);
Time: 0.060
```

The structure of the cohomology rings can be read from the following outputs.

```
> R`RelationsIdeal,S`RelationsIdeal,T`RelationsIdeal;
First the cohomology ring for $G$.
Ideal of Graded Polynomial ring of rank 6 over GF(2)
Lexicographical Order
Variables: $.1, $.2, $.3, $.4, $.5, $.6
Variable weights: 1 1 2 2 3 4
Basis:
[
    $.1^2,
    $.1*$.2,
    $.2^3,
    $.1*$.3,
    $.2*$.5,
    $.3^2,
    $.1*$.5 + $.2^2*$.3,
    $.3*$.5,
    $.5^2
]
```

Now the cohomology ring for Q.

```
Ideal of Graded Polynomial ring of rank 4 over GF(2)
Lexicographical Order
Variables: $.1, $.2, $.3, $.4
Variable weights: 1 1 3 4
Basis:
[
    $.1*$.2,
    $.1^3,
    $.1*$.3,
    $.2^2*$.4 + $.3^2
]
```

And finally the cohomology ring for Z.

```
Ideal of Graded Polynomial ring of rank 2 over GF(2)
Lexicographical Order
Variables: $.1, $.2
Variable weights: 1 2
Basis:
[
```

```
    $.1^2
]
```

Next we require the inputs for the restriction and inflation maps.

```
> Pr1 := k‘CompactProjectiveResolution;
> Pr2 := kk‘CompactProjectiveResolution;
> Pr3 := kkk‘CompactProjectiveResolution;
> Ac1 := k‘AllCompactChainMaps;
> Ac2 := kk‘AllCompactChainMaps;
> Ac3 := kkk‘AllCompactChainMaps;
```

Now the inflation map from Q to G sends the generators of the cohomology of Q to the given list of elements in the cohomology ring of G.

```
> inf := InflationMap(Pr2,Pr1,Ac2,Ac1,R,mu);
> inf;
[
    $.2,
    $.1,
    $.5,
    $.6
]
```

The restriction map from the cohomology ring of G to the cohomology ring of Z sends the generators of R to the corresponding elements in the computed sequence.

```
> res := RestrictionOfGenerators(Pr1,Pr3,Ac1,Ac3,T);
> res;
[
    0,
    0,
    0,
    $.2,
    0,
    0
]
```

Finally, a set of minimal relations is determined for the cohomology ring R.

```
> MinimalRelations(R);
[
    $.1^2,
    $.1*$.2,
    $.2^3,
    $.1*$.3,
    $.2*$.5,
    $.3^2,
    $.1*$.5 + $.2^2*$.3,
    $.3*$.5,
    $.5^2
```

]

85.12 A-infinity Algebra Structures on Group Cohomology

As described in [Kel01, **????**], an $A\infty$-algebra structure can be induced on H^*A for any differential graded algebra A. Consider $\mathrm{Ext}^*_R(S,S)$, for R a quiver algebra quotient and S the direct sum of all simple R-modules. Regarded as the homology of the endomorphism algebra of a projective resolution of S, Keller further demonstrates how this additional algebraic structure allows recovery of R from $\mathrm{Ext}^*_R(S,S)$.

An $A\infty$-algebra structure on a vector space V consists of higher structural operations $m_1,\ldots$ defined as $m_i : V^{\otimes i} \to V$ fulfilling the Stasheff axioms for all n:

$$\sum_{i+j-1=n, 0\le k\le n-j} \pm m_i(a_1,\ldots,a_{k-1},m_j(a_k,\ldots,a_{k+j}),a_{k+j+1},\ldots,a_n) = 0$$

An $A\infty$-algebra homomorphism from an $A\infty$-algebra A to an $A\infty$-algebra B is a family f_i of maps $A^{\otimes i} \to B$ such that the homomorphism axioms hold for all n:

$$\sum_{i+j-1=n, 0\le k\le n-j} \pm f_i(a_1,\ldots,a_{k-1},m_j(a_k,\ldots,a_{k+j}),a_{k+j+1},\ldots,a_n) =$$

$$\sum_{i_1+\ldots+i_r=n} \pm m_r(f_{i_1}(a_1,\ldots,a_{i_1}),\ldots,f_{i_r}(a_{n-i_r+1},\ldots,a_n))$$

According to a theorem fundamental to the algebraic uses of $A\infty$-techniques, for a differential graded algebra A, there is an $A\infty$-structure on H^*A and an $A\infty$-algebra homomorphism $f : H^*A \to A$ such that f_1 is a quasiisomorphism of differential graded algebras, and induced by the identity map on H^*A.

A blackbox method of calculation can be based on Kadeishvilis' proof of this statement, using the homomorphism axioms to recursively calculate any specific values that are needed, and choosing the m_i and f_i in such a way as not to violate the axioms. The following package implements this method for the special case of $\mathrm{Ext}^*_{kG}(k,k)$ for G a p-group, k a prime field of characteristic p and also the one-dimensional unique simple kG-module.

`AInfinityRecord(G,n)`

Constructs a record carrying all relevant information to calculate $A\infty$-operations on a group cohomology ring. Among the data carried can be found the cohomology ring in `R`, the cohomology ring quotient in `S`, the projective resolution used in `P`, the simple module resolved in `k` and the basic algebra in `A`.

`MasseyProduct(Aoo,terms)`

`HighProduct(Aoo,terms)`

Given an $A\infty$ object *Aoo* corresponding to a group cohomology ring, this intrinsic calculates the structure map $m_i(t_1 \otimes \ldots \otimes t_i)$, where the i give the length of terms, and $t_1, \ldots, t_i$ are the elements of terms.

`HighMap(Aoo,terms)`

Given an $A\infty$ object *Aoo* corresponding to a group cohomology ring, this intrinsic calculates the value of an $A\infty$-quasiisomorphism f at the point $t_1 \otimes \ldots \otimes t_i$, where the i gives the length of terms, and $t_1, \ldots, t_i$ are the elements of terms.

Example H85E22

The $A\infty$-structures on the cohomology rings of cyclic p-groups are well known examples in the literature: the $A\infty$-structure on $H^*(C_n, F_2)$ has one single higher structure nontrivial operation, namely m_n, which takes any n-tuple of odd coclasses to the even coclass of appropriate degree. In order to verify this for a specific example, we start by constructing an $A\infty$ record that contains all relevant information for the cohomology ring.

```
> Aoo := AInfinityRecord(CyclicGroup(4),10);
> S<x,y> := Aoo`S;
> HighProduct(Aoo,[x,x,x,x]);
y
> HighMap(Aoo,[x,x,x,x]);
Basic algebra chain map of degree -1
```

Example H85E23

The code as written handles odd characteristics well, with the sign choices featured in Kadeishvilis article [Kad80] embedded in the code.

```
> Aoo := AInfinityRecord(CyclicGroup(3),10);
> S<x,y> := Aoo`S;
> HighProduct(Aoo,[x,x,x]);
y
> HighMap(Aoo,[x,x,x]);
Basic algebra chain map of degree -1
```

85.12.1 Homological Algebra Toolkit

For the computation of $A\infty$-structures, several methods are used that would invite a wider use in a generic homological algebra toolkit.

`ActionMatrix(A,x)`

Produces a matrix of the right action of the Basic algebra element described by x in the Basic algebra A.

`CohomologyRingQuotient(CR)`

Computes the actual cohomology ring as a quotient ring of a multivariate polynomial ring from a cohomology ring record.

`LiftToChainmap(P,f,d)`

Lifts the function described by f to a chain map from P to P of degree d.

`NullHomotopy(f)`

Constructs a null homotopy of the null homotopic chain map f. If f is not null homotopic, the function will throw an error message, since in that case some of the equations encountered on the way are not solvable.

`IsNullHomotopy(f,H)`

Confirms that H is a null homotopy of f, in other words that $f = dH - Hd$, with d the differential of the corresponding chain complexes.

`ChainmapToCohomology(f,CR)`

Takes a chain map f and returns the element in the cohomology quotient ring to which the chain map corresponds.

`CohomologyToChainmap(xi,CR,P)`

Takes an element xi of a cohomology quotient ring of the cohomology ring record CR and a projective resolution corresponding to that cohomology ring, and returns a chain map in the coclass represented by xi.

Example H85E24

To illustrate the code that generates null homotopies, we consider the cohomology ring of a cyclic group, and pick out chain map representatives for the degree 1 coclass. Although this squares to zero, the corresponding chainmaps do not compose to the zero chainmap.

```
> A := BasicAlgebra(CyclicGroup(4));
> k := SimpleModule(A,1);
> P := ProjectiveResolution(k,5);
> R := CohomologyRing(k,5);
> S<x,y> := CohomologyRingQuotient(R);
> xi := CohomologyToChainmap(x,R,P);
> x*x;
```

```
0
> IsZero(xi*xi);
false
> ModuleMaps(xi*xi);
[*
    [0 0 1 0]
    [0 0 0 1]
    [0 0 0 0]
    [0 0 0 0],

    [0 0 1 0]
    [0 0 0 1]
    [0 0 0 0]
    [0 0 0 0],

    [0 0 1 0]
    [0 0 0 1]
    [0 0 0 0]
    [0 0 0 0],

    [0 0 1 0]
    [0 0 0 1]
    [0 0 0 0]
    [0 0 0 0]
*]
> H := NullHomotopy(xi*xi);
> ModuleMaps(H);
[*
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0],

    [0 1 0 0]
    [0 0 1 0]
    [0 0 0 1]
    [0 0 0 0],

    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0],

    [0 1 0 0]
    [0 0 1 0]
    [0 0 0 1]
    [0 0 0 0],
```

```
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 0]
*]
> IsNullHomotopy(xi*xi,H);
true
```

85.13 Bibliography

[**Kad80**] Tornike V. Kadeishvili. On the homology theory of fiber spaces. *Russian Math. Surveys*, (35:3):231–238, 1980. arXiv:math/0504437v1.

[**Kel01**] Bernhard Keller. Introduction to A-infinity algebras and modules. *Homology, Homotopy and Applications*, 3(1):1–35 (electronic), 2001.

86 QUATERNION ALGEBRAS

Chapter 86
QUATERNION ALGEBRAS

86.1 Introduction

A quaternion algebra A over a field K is a central simple algebra of dimension four over K, or equivalently when K does not have characteristic 2, an algebra generated by elements i, j which satisfy

$$i^2 = a,\ j^2 = b,\ ji = -ij$$

with $a, b \in K^*$. A quaternion algebra in MAGMA is a specialized type `AlgQuat`, which is a subtype of associative algebra `AlgAss` defined over a field. MAGMA can recognize if an associative algebra A is a quaternion algebra and will return a standard representation for A with a, b as above.

Examples of quaternion algebras include the ring $M_2(K)$ of 2×2-matrices over K with $a = b = 1$, as well as the division ring of Hamiltonians over $K = \mathbf{R}$ with $a = b = -1$. Every quaternion algebra over a finite field or an algebraically closed field is isomorphic to $M_2(K)$. Over a local field, there is a unique quaternion algebra (up to isomorphism) which is a division ring.

Every quaternion algebra A over K not isomorphic to $M_2(K)$ is a division algebra. Finding a zerodivisor in A is computationally equivalent to the problem of finding a K-rational point on a conic. Given a zerodivisor in A, MAGMA will compute an explicit isomorphism $A \to M_2(K)$. If L is an extension field of K, then $A_L = A \otimes_K L$ is a quaternion algebra over L, and we say L is a *splitting field* if $A_L \cong M_2(L)$. If $[L : K] = 2$, then L is a splitting field if and only if there exists a K-embedding $L \hookrightarrow A$, and such an embedding can be computed in MAGMA.

Further functionality is available for quaternion algebras A defined over number fields K. Such an algebra is said to be *unramified* at a noncomplex place v of K if K_v is a splitting field for A, otherwise A is *ramified* at v. Testing if an algebra is ramified at a place is encoded in the *Hilbert symbol*, which can be computed for any such algebra. The set S of ramified places of an algebra is finite and of even cardinality, and conversely, given such a set S there exists a quaternion algebra which is ramified only at S. One may compute the set of ramified places of A as well as constructing an algebra given its ramification set.

In MAGMA, there are algorithms for quaternion algebras which are analogous to those for number fields—computation of the ring of integers, class group, and unit group. One may compute a maximal order $O \subset A$, a p-maximal order for a prime p of K, and orders with given index in a maximal order. Secondly, a representative set of (right) ideal classes in O can be enumerated, and one can test if two ideals are isomorphic (hence if a given ideal is principal). Finally, for a definite quaternion algebra (defined over a totally real field), one may compute the group of units of norm 1 in O. Over $\mathbf{Q}$, these functions tie into machinery for constructing the Brandt module, with applications to modular forms.

Orders for quaternion algebras over the rational numbers and over rational function fields in MAGMA have type `AlgQuatOrd` and ideals have type `AlgQuatOrdIdl`. Over other number rings, orders have the general type `AlgAssVOrd` and ideals `AlgAssVOrdIdl` which is also the type for used for associative orders. The types `AlgQuatOrd`, `AlgQuatOrdElt` and `AlgQuatOrdIdl` inherit from the types `AlgAssVOrd`, `AlgAssVOrdElt` and `AlgAssVOrdIdl` respectively.

The main reference for material in this chapter is the book of Vignéras [Vig80]. Most nontrivial algorithms for quaternion algebras over the rationals and over number fields are described in [KV10].

IMPORTANT WARNING.

In MAGMA, the rationals are *not* considered to be a number field (the type `FldRat` is not a subtype of `FldNum`). Much of the functionality for quaternion algebras, and particularly for orders in quaternion algebras, is implemented only for algebras whose base field is a `FldNum` (while some, but not all, also works for algebras over the `FldRat`). To compute with algebras over $\mathbf{Q}$, in many cases the best solution is to create $\mathbf{Q}$ as a number field at the outset, using `RationalsAsNumberField()`, and create the algebra over this field instead of `Rationals()`.

86.2 Creation of Quaternion Algebras

A general constructor for a quaternion algebra over any field K creates a model in terms of two generators x and y and three relations

$$x^2 = a, \quad y^2 = b, \quad yx = -xy$$

with $a, b \in K^*$ if K has characteristic different from 2, and

$$x^2 + x = a, \quad y^2 = b, \quad yx = (x+1)y$$

if K has characteristic 2. The printing names i, j, and k are assigned to the generators x, y, and xy by default, unless the user assigns alternatives (see the function `AssignNames` below, or the example which follows).

Special constructors are provided for quaternion algebras over $\mathbf{Q}$, which return an algebra with a more general set of three defining quadratic relations. In general, the third generator need not be the product of the first two. This allows the creation of a quaternion algebra A such that the default generators $\{1, i, j, k\}$ form a basis for a maximal order.

`QuaternionAlgebra< K | a, b >`

For $a, b \in K^*$, this function creates the quaternion algebra A over the field K on generators i and j with relations $i^2 = a$, $j^2 = b$, and $ji = -ij$ or $ji = (i+1)j$, as $\mathrm{char}(K) \neq 2$ or $\mathrm{char}(K) = 2$, respectively. A third generator is set to $k = ij$. The field K may be of any MAGMA field type; for inexact fields, such as local fields, one should expect unstable results since one cannot test deterministically for element equality.

`AssignNames(~A, S)`

Given a quaternion algebra A and sequence S of strings of length 3, this function assigns the strings to the generators of A, i.e. the basis elements not equal to 1. This function is called by the bracket operators `< >` at the time of creation of a quaternion algebra.

Example H86E1

A general quaternion algebra can be created as follows. Note that the brackets `< >` can be used to give any convenient names to the generators of the ring.

```
> A<x,y,z> := QuaternionAlgebra< RationalField() | -1, -7 >;
> x^2;
-1
> y^2;
-7
> x*y;
z
```

In the case of this constructor the algebra generators are of trace zero and are pairwise anticommuting. This contrasts with some of the special constructors for quaternion algebras over the rationals described below.

Example H86E2

Similarly, we have the following for characteristic 2.

```
> A<i,j> := QuaternionAlgebra< GF(2) | 1, 1 >;
> i^2;
1 + i
> j^2;
1
```

`QuaternionAlgebra(N)`

`Al`	MonStgElt	*Default :* "*TraceValues*"
`Optimized`	BoolElt	*Default :* `true`

Given a positive squarefree integer N, this function returns a rational quaternion algebra of discriminant N. If the optional parameter `Al` is set to `"Random"`, or N is the product of an even number of primes (i.e. the algebra is indefinite), then a faster, probabilistic algorithm is used. If `Al` is set to `"TraceValues"` and N is a product of an odd number of primes, then an algebra basis is computed which also gives a basis for a maximal order (of discriminant N). If `Optimized` is `true` then an optimized representation of the algebra is returned.

`QuaternionAlgebra(N)`

Given a squarefree polynomial $N \in \mathbf{F}_q[x]$ for some odd q, this function returns a quaternion algebra over $\mathbf{F}_q(x)$ of discriminant N.

`QuaternionAlgebra(I)`

`Optimized` BoolElt *Default :* `true`

Given an ideal I of a number field F with an even number of prime ideal factors, the function returns the quaternion algebra A over F which is ramified exactly at the primes dividing I. The ideal I is not required to be squarefree, so A will be ramified at the radical of I. If `Optimized` is `true` then an optimized representation of the algebra is returned.

`QuaternionAlgebra(I, S)`

`Optimized` BoolElt *Default :* `true`

Given an ideal I and a subset S of real places of a number field F such that the number of prime ideal divisors of I has the same parity as S, the function returns the quaternion algebra which is ramified exactly at the primes dividing I and at the real places specified by the set S. If `Optimized` is `true` then an optimized representation of the algebra is returned.

`QuaternionAlgebra(S)`

`Optimized` BoolElt *Default :* `true`

Given an even set S of noncomplex places of a number field F, this function returns the quaternion algebra which is ramified exactly at S. If `Optimized` is `true` then an optimized representation of the algebra is returned.

Example H86E3

We illustrate the above constructors in the case of a general number field.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Foo := InfinitePlaces(F);
> Z_F := MaximalOrder(F);
> I := ideal<Z_F | 6>;
> A := QuaternionAlgebra(I);
> FactoredDiscriminant(A);
[
    Prime Ideal of Z_F
    Two element generators:
        [3, 0, 0]
        [2, 1, 0],
    Principal Prime Ideal of Z_F
    Generator:
        [2, 0, 0]
```

```
]
[]
> A := QuaternionAlgebra(ideal<Z_F | 1>, Foo[1..2]);
> FactoredDiscriminant(A);
[]
[ 1st place at infinity, 2nd place at infinity ]
```

`QuaternionAlgebra(D1, D2, T)`

This intrinsic creates the rational quaternion algebra $\mathbf{Q}\langle i, j\rangle$, where $\mathbf{Z}[i]$ and $\mathbf{Z}[j]$ are quadratic suborders of discriminant D_1 and D_2, respectively, and $\mathbf{Z}[ij - ji]$ is a quadratic suborder of discriminant $D_3 = D_1D_2 - T^2$. The values D_1D_2 and T must have the same parity and D_1, D_2 and D_3 must each be the discriminant of some quadratic order, i.e. nonsquare integers congruent to 0 or 1 modulo 4.

Example H86E4

The above constructor is quite powerful for constructing quaternion algebras with given ramification. For any i and j, a commutator element such as $ij - ji$ has trace zero, so in the above constructor, the minimal polynomial of this element is $x^2 + n$, where $n = (D_1D_2 - T^2)/4$.
In the following example we construct such a ring, and demonstrate some of the relations satisfied in this algebra. Note that the minimal polynomial is an element of the commutative polynomial ring over the base field of the algebra.
In particular, we note that the algebra is not in standard form.

```
> A<i,j,k> := QuaternionAlgebra(-7,-47,1);
> PQ<x> := PolynomialRing(RationalField());
> MinimalPolynomial(i);
x^2 - x + 2
> MinimalPolynomial(j);
x^2 - x + 12
> MinimalPolynomial(k);
x^2 - x + 24
> i*j;
k
> i*j eq -j*i;
false
```

From their minimal polynomials, we see that the algebra generators i, j, and k generate commutative subfields of discriminants -7, -47, and -95. The value $82 = (D_1D_2 - T^2)/4$, however, is a more important invariant of the ring. We give a preview of the later material by demonstrating the functionality for computing the determinant and ramified primes of an algebra over $\mathbf{Q}$.

```
> MinimalPolynomial(i*j-j*i);
x^2 + 82
> Discriminant(A);
41
> RamifiedPrimes(A);
```

```
[ 41 ]
```

A ramified prime must be inert or ramified in every subfield and must divide the norm of any commutator element $xy - yx$.

```
> x := i;
> y := j;
> Norm(x*y-y*x);
82
> Factorization(82);
[ <2, 1>, <41, 1> ]
> x := i-k;
> y := i+j+k;
> Norm(x*y-y*x);
1640
> Factorization(1640);
[ <2, 3>, <5, 1>, <41, 1> ]
> KroneckerSymbol(-7,2), KroneckerSymbol(-47,2), KroneckerSymbol(-95,2);
1 1 1
> KroneckerSymbol(-7,41), KroneckerSymbol(-47,41), KroneckerSymbol(-95,41);
-1 -1 -1
```

The fact that the latter Kronecker symbols are -1, indicating that 41 is inert in the quadratic fields of discriminants -7, -47, and -95, proves that 41 is a ramified prime, and 2 is not.

86.3 Creation of Quaternion Orders

Let R be a ring with field of fractions K, and let A be a quaternion algebra over K. An *R-order* in A is a subring $O \subset A$ which is a R-submodule of A with $O \cdot K = A$. An order is *maximal* if it is not properly contained in any other order.

One can create orders for number rings R, for $R = \mathbf{Z}$ or for $R = k[x]$ with k a field. Unlike commutative orders, it is important to note that maximal orders O of quaternion algebras are no longer unique: for any $x \in A$ not in the normalizer of O, we have another maximal order given by $O' = x^{-1}Ox \neq O$.

When $R = \mathbf{Z}$ or $R = k[x]$, the order O has type `AlgQuatOrd`. When R inherits from type `RngOrd` (a number ring), the order O has type `AlgAssVOrd`; see Section 81.4 for more information on constructors and general procedures for these orders.

See above (86.1) for an important warning regarding quaternion algebras over the rationals.

86.3.1 Creation of Orders from Elements

The creation of orders from elements of number rings is covered in Section 81.4. The creation of quaternion orders over the integers and univariate polynomial rings is covered in this section.

`QuaternionOrder(S)`

`IsBasis` BoolElt *Default :* `false`

Given S a sequence of elements in a quaternion algebra defined over $\mathbf{Q}$ or $\mathbf{F}_q(X)$, this function returns the order generated by S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$. If the set S does not generate an order, an error will be returned. If the parameter `IsBasis` is set to `true` then S will be used as the basis of the order returned.

`QuaternionOrder(R, S)`

`Order(R, S)`

`Check` BoolElt *Default :* `true`

Given a ring R and a sequence S of elements of a quaternion algebra over $\mathbf{Q}$ or $\mathbf{F}_q(X)$, this function returns the R-order with basis S. The sequence must have length four.

Example H86E5

First we construct an order over a polynomial ring.

```
> K<t> := FunctionField(FiniteField(7));
> A<i,j,k> := QuaternionAlgebra< K | t, t^2+t+1 >;
> O := QuaternionOrder( [i,j] );
> Basis(O);
[1, i, j, k ]
```

Next we demonstrate how to construct orders in quaternion algebras generated by a given sequence of elements. When provided with a sequence of elements of a quaternion algebra over $\mathbf{Q}$, Magma reduces the sequence so as to form a basis. When provided with the ring over which these elements are to be interpreted, the sequence must be a basis with initial element 1, and the order having this basis is constructed.

```
> A<i,j,k> := QuaternionAlgebra< RationalField() | -1, -3 >;
> B := [ 1, 1/2 + 1/2*j, i, 1/2*i + 1/2*k ];
> O := QuaternionOrder(B);
> Basis(O);
[ 1, 1/2*i + 1/2*k, 1/2 - 1/2*j, -1/2*i + 1/2*k ]
> S := QuaternionOrder(Integers(),B);
> Basis(S);
[ 1, 1/2 + 1/2*j, i, 1/2*i + 1/2*k ]
```

86.3.2 Creation of Maximal Orders

`MaximalOrder(A)`

A maximal order is constructed in the quaternion algebra A. The algebra A must be defined over a field K where K is either a number field, $\mathbf{Q}$, $\mathbf{F}_q(X)$ with q odd, or the field of fractions of a number ring. Over $\mathbf{F}_q(X)$ we use the standard algorithm [Fri97, IR93]. Over the rationals or over a number field, we use a variation of this algorithm optimized for the case of quaternion algebras. First, a factorization of the discriminant of a tame order (see below) is computed. Then, for each prime p dividing the discriminant, a p-maximal order compatible with the existing order is computed. The method used corresponds to Algorithm 4.3.8 in [Voi05]. See also [Voi11].

Example H86E6

The following is an example of a quaternion algebra which is unramified at all finite primes.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> A<alpha,beta,alphabeta> := QuaternionAlgebra<F | -3,b>;
> O := MaximalOrder(A);
> Factorization(Discriminant(O));
[]
```

Hence the algebra A has a maximal order of discriminant 1, or equivalently, A is unramified at all finite places of F.
Since we are working over a general order of a number field, we can no longer guarantee that an order will have a free basis, so it must be represented by a pseudomatrix. For more on pseudomatrices, see Section 55.10.

```
> Z_F := BaseRing(O);
> PseudoBasis(O);
[
    <Principal Ideal of Z_F
    Generator:
        Z_F.1, Z_F.1>,
    <Fractional Ideal of Z_F
    Two element generators:
        Z_F.1
        2/3*Z_F.1 + 1/6*Z_F.2 + 1/6*Z_F.3, 3/1*Z_F.1 + i>,
    <Principal Ideal of Z_F
    Generator:
        Z_F.1, j>,
    <Fractional Ideal of Z_F
    Two element generators:
        Z_F.1
        11/2*Z_F.1 + 1/6*Z_F.2 + 35/6*Z_F.3, 3/1*Z_F.1 - i + 3/1*Z_F.1*j + k>
```

```
]
```

The wide applicability of the above algorithm, is demonstrated by examining a "random" quaternion algebra over a "random" quadratic number field.

```
> for c := 1 to 10 do
>   D := Random([d : d in [-100..100] | not IsSquare(d)]);
>   K<w> := NumberField(x^2-D);
>   Z_K := MaximalOrder(K);
>   K<K1,w> := FieldOfFractions(Z_K);
>   a := Random([i : i in [-50..50] | i ne 0]) + Random([-50..50])*w;
>   b := Random([i : i in [-50..50] | i ne 0]) + Random([-50..50])*w;
>   printf "D = %o, a = %o, b = %o\n", D, a, b;
>   A := QuaternionAlgebra<K | a,b>;
>   O := MaximalOrder(A);
>   ds := [<pp[1],pp[2],HilbertSymbol(A,pp[1])> :
>           pp in Factorization(Discriminant(O))];
>   print ds;
>   for d in ds do
>     if d[3] eq 1 then
>       break c;
>     end if;
>   end for;
> end for;
D = 5, a = -46/1*K1 + 25/1*w, b = -10/1*K1 - 7/1*w
[
    <Prime Ideal of Z_K
    Two element generators:
        [31, 0]
        [5, 2], 2, -1>,
    <Prime Ideal of Z_K
    Two element generators:
        [11, 0]
        [6, 2], 2, -1>
]
...
```

For each such "random" quaternion algebra, we verify that the Hilbert symbol evaluated at each prime dividing the discriminant of the maximal order is -1, indicating that the algebra is indeed ramified at the prime.

`MaximalOrder(O)`

For O a quaternion order defined over $\mathbf{Z}$, $\mathbf{F}_q[X]$ with q odd or a number ring, this function returns a maximal order containing O.

`pMaximalOrder(O, p)`

For O a quaternion order defined over $\mathbf{Z}$, $\mathbf{F}_q[X]$ with q odd or a number ring and a prime (ideal) p, this function returns a p-maximal order O' containing the order O. The p-adic valuation of the discriminant of O' (which is either 0 or 1) is returned as a second return value.

`TameOrder(A)`

Given a quaternion algebra A, this function returns an order O having the property that the odd reduced discriminant of O is squarefree. The algebra A must be defined over a number field or field of fractions of a number ring. The algorithm ignores even primes and does not test the remaining odd primes for maximality.

86.3.3 Creation of Orders with given Discriminant

The following two functions together with the maximal order algorithms of the previous subsection allow the construction of arbitrary Eichler orders.

`Order(O, N)`

Given an order O in a quaternion algebra A over the rationals or $\mathbf{F}_q(x)$ with q odd, and some element N in the base ring of O, this function returns a suborder O' of O having index N. Currently, N and the level of O must be coprime and N must have valuation at most 1 at each ramified prime of A. The order O' is locally Eichler at all prime divisors of N that are not ramified in A. In particular, if O is Eichler and N is coprime to the discriminant of A, so is O'.

`Order(O, N)`

Given a maximal quaternion order O over a number ring, this function returns an Eichler order of level N inside O.

`GorensteinClosure(O)`

Given a quaternion order O over the integers, $\mathbf{F}_q[x]$ with q odd or a number ring, this function returns the smallest Gorenstein order containing O.

Example H86E7

First we construct a quaternion algebra A over $\mathbf{F}_5(x)$ ramified at x^2+x+1, then a maximal order M in A and finally an Eichler O order of discriminant $(x^2+x+1)(x^3+x+1)^5$.

```
> P<x> := PolynomialRing(GF(5));
> A := QuaternionAlgebra(x^2+x+1);
> M := MaximalOrder(A);
> O := Order(M, (x^3+x+1)^5);
> FactoredDiscriminant(O);
[
    <x^2 + x + 1, 1>,
    <x^3 + x + 1, 5>
]
```

86.3.4 Creation of Orders with given Discriminant over the Integers

When constructing quaternion orders over the integers, several shortcuts are available.

`QuaternionOrder(A, M)`

Given a quaternion algebra A and a positive integer M, this function returns an order of index M in a maximal order of the quaternion algebra A defined over $\mathbf{Q}$. The second argument M can have at most valuation 1 at any ramified prime of A.

`QuaternionOrder(N)`

`QuaternionOrder(N, M)`

Given positive integers N and M, this function returns an order of index M in a maximal order of the rational quaternion algebra A of discriminant N. The discriminant N must be a product of an odd number of distinct primes, and the argument M can be at most of valuation 1 at any prime dividing N. If M is omitted, the integer M defaults to 1, i.e., the function will return a maximal order.

`QuaternionOrder(D1, D2, T)`

This intrinsic constructs the quaternion order $\mathbf{Z}\langle x, y\rangle$, where $\mathbf{Z}[x]$ and $\mathbf{Z}[y]$ are quadratic subrings of discriminant D_1 and D_2, respectively, and $\mathbf{Z}[xy - yx]$ is a quadratic subring of discriminant $D_1 D_2 - T^2$.

Note that the container algebra of such a quaternion order is **not** usually in standard form (see the example below).

Example H86E8

The above constructors permit the construction of Eichler orders over $\mathbf{Z}$, if the discriminant N and the index M are coprime. More generally they allow the construction of an order whose index in an Eichler order divides the discriminant.

```
> A := QuaternionOrder(103,2);
> Discriminant(A);
206
> Factorization($1);
[ <2, 1>, <103, 1> ]
> _<x> := PolynomialRing(Rationals());
> [MinimalPolynomial(A.i) : i in [1..4]];
[
    x - 1,
    x^2 + 1,
    x^2 - x + 52,
    x^2 + 104
]
```

The constructor `QuaternionOrder(D1, D2, T)` may return an order whose container algebra is not in standard form.

```
> A := QuaternionOrder(-4, 5, 2);
```

```
> B := Algebra(A);
> B.1 * B.2 eq - B.2 * B.1;
false
```

86.4 Elements of Quaternion Algebras

For more information about elements of orders of associative algebras, see Section 81.4.

86.4.1 Creation of Elements

`A ! 0`
`Zero(A)`

The zero element of the quaternion algebra A.

`A ! 1`
`One(A)`

The identity element of the quaternion algebra A.

`A . i`
`Name(A, i)`

Given a quaternion algebra A and an integer $1 \leq i \leq 3$, returns the ith generator of A as an algebra over the base ring. Note that the element 1 is always the first element of a basis, and is never returned as a generating element.

`A ! x`

Return an element of the quaternion algebra A described by x, where x may be an algebra element, a module element, a sequence, an element of an order of an associative algebra or be coercible into the coefficient ring of A.

86.4.2 Arithmetic of Elements

`x + y`

The sum of x and y.

`x - y`

The difference of x and y.

`x * y`

The product of x and y.

`x / y`

The quotient of x by the unit y in the quaternion algebra.

`x eq y`

Returns `true` if the elements x and y are equal; otherwise `false`.

`x ne y`

Returns `true` if and only if the elements x and y are not equal.

`x in A`

Returns `true` if and only if x is in the algebra A.

`x notin A`

Returns `true` if and only if x is not in the algebra A.

`Conjugate(x)`

The conjugate $\bar{x}$ of the element x of a quaternion algebra, defined so that the reduced trace and reduced norm are $\bar{x} + x$ and $\bar{x}x$, respectively.

`ElementToSequence(x)`

`Eltseq(x)`

`Coordinates(x)`

Given an element x of a quaternion algebra or order, this function returns the sequence of coordinates of x in terms of the basis of its parent.

`Norm(x)`

The reduced norm $\mathrm{N}(x)$ of the element x of a quaternion algebra, defined so that the characteristic polynomial for x is $x^2 - \mathrm{Tr}(x)x + \mathrm{N}(x) = 0$, where $\mathrm{Tr}(x)$ is the reduced trace.

`Trace(x)`

The reduced trace $\mathrm{Tr}(x)$ of the element x of a quaternion algebra, defined so that the characteristic polynomial for x is $x^2 - \mathrm{Tr}(x)x + \mathrm{N}(x) = 0$, where $\mathrm{N}(x)$ is the reduced norm.

`CharacteristicPolynomial(x)`

The characteristic polynomial of degree 2 for the element x of a quaternion algebra over the base ring of its parent.

`MinimalPolynomial(x)`

The minimal polynomial of degree 1 or 2 for the element x of a quaternion algebra over the base ring of its parent.

Example H86E9

We demonstrate the relation between characteristic polynomial, and reduced trace and norm in the following example.

```
> A := QuaternionAlgebra< RationalField() | -17, -271 >;
> x := A![1,-2,3,0];
> Trace(x);
2
> Norm(x);
2508
> x^2 - Trace(x)*x + Norm(x);
0
```

Note that trace and norm of an element x of any algebra can be defined as the trace and norm of the linear operator corresponding to right-multiplication by x. The reduced trace and norm in a quaternion algebra A are taken instead to be the corresponding trace and determinant in any two-dimensional matrix representation of A, or equivalently, the sum and product of an element with its conjugate. The definition of norm and trace used for a general algebra can be realised in a quaternion algebra by the following code.

```
> P<X> := PolynomialRing(RationalField());
> M := RepresentationMatrix(x, A);
> M;
[   1   -2    3    0]
[  34    1    0    3]
[-813    0    1    2]
[   0 -813  -34    1]
> Trace(M);
4
> Factorization(CharacteristicPolynomial(M));
[
    <X^2 - 2*X + 2508, 2>
]
```

The general definition of trace (for the algebra) is twice the reduced trace, and the general definition of norm is the square of the reduced norm.

86.5 Attributes of Quaternion Algebras

`BaseField(A)`

`BaseRing(A)`

The base field of the quaternion algebra A.

`Basis(A)`

The basis of the algebra A.

`RamifiedPrimes(A)`

Given a quaternion algebra A over $\mathbf{Q}$ or $\mathbf{F}_q(X)$ with q odd, this function returns a list of primes or normalized irreducible polynomials corresponding to the finite ramified places of A.

Example H86E10

The sequence of ramified primes of a quaternion algebra A over $\mathbf{Q}$ determines the isomorphism class of the algebra.

```
> A := QuaternionAlgebra(-436,-503,22);
> RamifiedPrimes(A);
[ 17 ]
```

Provided the discriminant is of a size which can be factored, the ramified primes are determined efficiently using Hilbert symbols.

`RamifiedPlaces(A)`

`FactoredDiscriminant(A)`

Given a quaternion algebra A over $\mathbf{Q}$ or $\mathbf{F}_q(X)$ with q odd or a number field, this function returns the finite as well as infinite places where A is ramified.

Note: The first return value of these functions is always a list of ideals, even if the algebra is given over $\mathbf{Q}$ or $\mathbf{F}_q(X)$.

Example H86E11

This example shows the (minor) difference between `RamifiedPrimes` and `RamifiedPlaces`.

```
> F<x> := RationalFunctionField( GF(5) );
> A := QuaternionAlgebra< F | 2, x >;
> R<x>:= Integers(F);
> RamifiedPrimes(A);
[ x ]
> RamifiedPlaces(A);
[
    Ideal of Univariate Polynomial Ring in x over GF(5) generated by x
]
[ Infinity ]
```

`Discriminant(A)`

The reduced discriminant of a quaternion algebra A over $\mathbf{Q}$, $\mathbf{F}_q(X)$ with q odd or a number field. In the first two cases, the functions return the product of the ramified primes. Over number fields, they return the product of the ramified prime ideals as well as the sequence of ramified infinite places.

`StandardForm(A)`

Returns integers a and b in the base field F of the given quaternion algebra A such that there exists elements $i, j \in A$ where $i^2 = a$, $j^2 = b$, and $ji = -ij$. The third object returned is the standard quaternion algebra $B =$ `QuaternionAlgebra<F|a,b>`, and the fourth object is the homomorphism from A to B.

86.6 Hilbert Symbols and Embeddings

Let A be a quaternion algebra over Q, $\mathbf{F}_q(X)$ (with q odd) or a number field F with defining elements a, b, and let v be a place of F. If v is unramified in A (i.e. $A \otimes_F F_v \cong M_2(F_v)$, we define the *Hilbert symbol* $(a, b)_v$ to be 1, and otherwise we define $(a, b)_v = -1$.

`HilbertSymbol(a, b, p)`

`HilbertSymbol(A, p)`

`Al` MonStgElt *Default :* "*NormResidueSymbol*"

Computes the Hilbert symbol for the quaternion algebra A over F, namely $(a, b)_p$, where $a, b \in F$ and p is either a prime (if $a, b \in \mathbf{Q}$ or $\mathbf{F}_q(X)$) or a prime ideal. If $a, b \in \mathbf{Q}$, by default table-lookup is used to compute the Hilbert symbol; one can optionally insist on using the full algorithm by setting the parameter `Al` to the value `"Evaluate"`.

`IsRamified(p, A)`

`IsUnramified(p, A)`

Returns `true` if and only if the prime or prime ideal p is ramified (unramified) in the quaternion algebra A.

Example H86E12

We first verify the correctness of all Hilbert symbols over the rationals.

```
> QQ := Rationals();
> for a,b in [1..8] do
>   bl := HilbertSymbol(QQ ! a, QQ ! b,2 : Al := "Evaluate")
>                 eq NormResidueSymbol(a,b,2);
>   print <a,b,bl>;
>   if not bl then
>     break a;
>   end if;
> end for;
<1, 1, true>
<1, 2, true>
<1, 3, true>
...
```

For a second test, we input a quaternion algebra which is unramified at all finite places.

```
> P<x> := PolynomialRing(Rationals());
```

```
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> A := QuaternionAlgebra<F | -3,b>;
> symbols := [];
> for p in [p : p in [2..100] | IsPrime(p)] do
>   pps := Decomposition(Z_F,p);
>   for pp in pps do
>     Append(~symbols,HilbertSymbol(A,pp[1]));
>   end for;
> end for;
> symbols;
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

Finally, we test "random" quaternion algebras over quadratic extensions at even primes, the hardest case. We use the fact that the quaternion algebra (a, b) is ramified at a prime ideal p if and only if b is a norm from the extension $F(\sqrt{a})$, so we can test this condition using `IsLocalNorm`. Note that this takes substantially more time.

```
> for c in [2,-2,6,-6,-1,3,-3] do
>   K<s> := NumberField(x^2-c);
>   Z_K := MaximalOrder(K);
>   Z_Kmod8, f8 := quo<Z_K | 8>;
>   PPK<xK> := PolynomialRing(K);
>   for i := 1 to 10 do
>     S := [x+y*Z_K.2 : x,y in [0..7] | x*y ne 0];
>     a := Random(S);
>     b := Random(S);
>     A := QuaternionAlgebra<K | a,b>;
>     for pp in Decomposition(Z_K,2) do
>       hsym := HilbertSymbol(A,pp[1]);
>       if not IsIrreducible(xK^2-a) then
>         print <c, a, b, hsym eq 1>;
>         if hsym ne 1 then
>           break c;
>         end if;
>       else
>         lclsym := IsLocalNorm(AbelianExtension(ext<K | xK^2-a>),Z_K ! b,pp[1]);
>         bl := (hsym eq 1) eq lclsym;
>         print <c, a, b, bl>;
>         if not bl then
>           break c;
>         end if;
>       end if;
>     end for;
>   end for;
> end for;
<2, 5/1*Z_K.1 + 3/1*Z_K.2, Z_K.1 + 7/1*Z_K.2, true>
<2, 6/1*Z_K.1 + 4/1*Z_K.2, 4/1*Z_K.1 + Z_K.2, true>
```

```
<2, 7/1*Z_K.1 + Z_K.2, 2/1*Z_K.1 + 2/1*Z_K.2, true>
...
```

`pMatrixRing(A, p)`

`pMatrixRing(O, p)`

`Precision` RngIntElt *Default :*

Let A be a quaternion algebra A over a field F where F is the rationals, a number field of $\mathbf{F}_q(x)$ with q odd. Given A and a prime (ideal) p of the ring of integers R of F such that p is unramified in A, this function returns the matrix ring over the completion F_p of F at p, a map from $A \to M_2(F_p)$ and the embedding $F \to F_p$.

Given a p-maximal order O in A, the map from $A \to M_2(F_p)$ induces a map from $O \to \mathrm{M}_2(R_p)$.

`IsSplittingField(K, A)`

`HasEmbedding(K, A)`

`ComputeEmbedding` BoolElt *Default :* `false`

Given a quaternion algebra A defined over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field F and K a quadratic extension of F, the function returns `true` if and only if there exists an embedding $K \to A$ over F. This is done by comparison of ramified places in K and A (see [Vig80, Cor. III.3.5]). If no embedding exists, the second return value will be a witness place. If an embedding exists and the optional argument `ComputeEmbedding` is set to `true`, the second and third return values contain the result of a call to `Embed` as described below.

`Embed(K, A)`

`Al` MonStgElt *Default :* "*NormEquation*"

Given a quaternion algebra A defined over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field F and K a quadratic extension of F, returns an embedding $K \to A$ over F, given as an element of A, the image of the primitive generator of K, and the map $K \to A$.

The algorithm by default involves solving a relative norm equation. Alternatively, a naive search algorithm may be selected by setting the optional parameter `Al:="Search"`.

If there is no embedding, a runtime error occurs (or the `"Search"` runs forever). To check whether an embedding exists, use `HasEmbedding` (see immediately above).

`Embed(Oc, O)`

`Al` MonStgElt *Default :* "*NormEquation*"

Given a quadratic order O_c with base number ring R and a quaternion order O with base ring R, the function computes an embedding $O_c \hookrightarrow O$ over R. It returns the image of the second generator `Oc.2` of `Oc`; secondly it returns the embedding map $O_c \to O$.

The algorithm by default involves solving a relative norm equation. Alternatively, a naive search algorithm may be selected by setting the optional parameter `Al:="Search"`.

Notes. Let K be the number field containing Oc.

(i) `Oc.1, Oc.2` are the generators of Oc as a module, and `Oc.2` is unrelated to `K.1`, where K is the number field containing Oc.

(ii) To check whether an embedding of K into the algebra exists, one can use `HasEmbedding(K, Algebra(O) : ComputeEmbedding:=false)`.

Example H86E13

```
> F<b> := NumberField(Polynomial([1,-3,0,1]));
> A := QuaternionAlgebra<F | -3, b>;
> K := ext<F | Polynomial([2,-1,1])>;
> mu, iota := Embed(K, A);
> mu;
1/2 + 1/6*(-2*b^2 + 2*b + 7)*i + 1/2*(2*b^2 + b - 6)*j + 1/6*(-2*b^2 - b + 4)*k
> MinimalPolynomial(mu);
$.1^2 - $.1 + 2
> iota(K.1) eq mu;
true
```

86.7 Predicates on Algebras

A quaternion algebra A over a number field F with $[F : \mathbf{Q}] = h$ is *definite* (or *totally definite*) if F is totally real and $A \otimes_{\mathbf{Q}} \mathbf{R} \cong H^h$, where H is the division ring of real Hamiltonians, otherwise A is *indefinite*.

A quaternion algebra A over $\mathbf{F}_q(X)$ is called *definite* if the place corresponding to the degree valuation is ramified.

`IsDefinite(A)`

`IsIndefinite(A)`

Given a quaternion algebra A over a number field, $\mathbf{Q}$ or $\mathbf{F}_q(X)$ with q odd, returns `true` if and only if A is a (totally) definite or indefinite quaternion algebra, respectively.

86.8 Recognition Functions

A quaternion algebra A over a field K is isomorphic to the matrix ring $M_2(K)$ if and only if there exists a zerodivisor ϵ in A. Given such an ϵ, we can exhibit an explicit isomorphism; otherwise a zerodivisor will be computed first by finding a point on a conic (see [Vig80, Cor. I.2.4]).

Given an associative algebra, we also have an algorithm to recognize if the algebra is a quaternion algebra, and, if so, return an isomorphism to a quaternion algebra in standard form.

`IsMatrixRing(A)`

`Isomorphism` BoolElt *Default :* `false`

Returns `true` if and only if the quaternion algebra A with base field F is isomorphic to $M_2(F)$, or equivalently if A has no ramified places. The field F has to be $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field.

If A is isomorphic to $M_2(F)$ and `Isomorphism` is set to `true`, then $M_2(F)$ and an isomorphism $A \to M_2(F)$ are also returned.

`MatrixRing(A, eps)`

`MatrixAlgebra(A, eps)`

Given a quaternion algebra A and a zerodivisor $\epsilon \in A$, the function returns the matrix algebra $M_2(F)$ and an isomorphism $A \to M_2(F)$.

Example H86E14

```
> A := QuaternionAlgebra<Rationals() | -1, 1>;
> eps := A.3-1;
> MinimalPolynomial(eps), Norm(eps);
x^2 + 2*x
0
```

Thus, since ϵ has reduced norm 0, it is a zerodivisor: indeed, $\epsilon(\epsilon + 2) = 0$.

```
> M2F, phi := MatrixRing(A,eps);
> [<MinimalPolynomial(A.i), MinimalPolynomial(phi(A.i))> : i in [1..3]];
[
    <x^2 + 1, x^2 + 1>,
    <x^2 - 1, x^2 - 1>,
    <x^2 - 1, x^2 - 1>
]
```

`IsQuaternionAlgebra(B)`

Returns `true` if and only if the associative algebra B is a quaternion algebra; if true, it returns the associated quaternion algebra A in standard form and an algebra homomorphism from B to A. The algorithm used is [Voi05, Algorithm 4.2.9].

Example H86E15

We create an associative algebra which is known to be a quaternion algebra A and then recover A (or an isomorphic algebra).

```
> A := AssociativeAlgebra(QuaternionAlgebra<Rationals() | -1,1>);
> vecs := [&+[Random(10)*A.i : i in [1..4]] : j in [1..4]];
> Mchange := Matrix(Rationals(),4,4,&cat[Eltseq(vecs[i]) : i in [1..4]]);
> Mchange := Mchange^(-1);
> seq := [<i,j,k,((vecs[i]*vecs[j])*Mchange)[k]> : i,j,k in [1..4]];
> A := AssociativeAlgebra<Rationals(),4 | seq>;
> bl, Aquat, phi := IsQuaternionAlgebra(A);
> bl;
true
> Aquat;
Quaternion Algebra with base ring Rational Field
> Aquat.1^2, Aquat.2^2;
25 -3924/25
> phi;
Mapping from: AlgAss: A to AlgQuat: Aquat given by a rule
```

We now verify the functionality when a zerodivisor is encountered.

```
> A := Algebra(MatrixAlgebra(Rationals(),2));
> IsQuaternionAlgebra(A);
true Quaternion Algebra with base ring Rational Field
Mapping from: AlgAss: A to Quaternion Algebra with base ring Rational Field
given by a rule
```

The algebra $k < x, y >$ with $x^2 = y^2 = xy + yx = 0$ is not semisimple; the ideal generated by x, y is a nontrivial two-sided ideal. Similarly, a commutative algebra is not a quaternion algebra.

```
> A := Algebra(FPAlgebra<Rationals(), x,y | x^2, y^2, x*y+y*x>);
> IsQuaternionAlgebra(A);
false
> A := Algebra(FPAlgebra<Rationals(), x | x^4+x^2+1>);
> IsQuaternionAlgebra(A);
false
```

In characteristic 2, the algorithm also performs correctly, both for an associative but non-quaternion algebra and for the "universal" example of a quaternion algebra.

```
> A := Algebra(FPAlgebra<GF(2), x,y | x^2, y^2, x*y+y*x+1>);
> IsQuaternionAlgebra(A);
false
> F<a,b,x,y,z,w> := FieldOfFractions(PolynomialRing(GF(2),6));
> M := [[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1],
>       [0,1,0,0],[a,1,0,0],[0,0,0,1],[0,0,a,1],
>       [0,0,1,0],[0,0,1,1],[b,0,0,0],[b,b,0,0],
>       [0,0,0,1],[0,0,a,0],[0,b,0,0],[a*b,0,0,0]];
> A<alpha,beta> := AssociativeAlgebra<F,4 | M>;
```

```
> alpha^2+alpha+a;
(0 0 0 0)
> beta^2+b;
(0 0 0 0)
>
> bl, Aquat, phi := IsQuaternionAlgebra(A);
> bl;
true
> Aquat;
Quaternion Algebra with base ring Multivariate rational function field of
rank 6 over GF(2)
> theta := phi(x+y*alpha+z*beta+w*alpha*beta);
> Trace(theta);
y
> Norm(theta);
a*b*w^2 + a*y^2 + b*z^2 + b*z*w + x^2 + x*y
```

`MatrixRepresentation(A)`

`MatrixRepresentation(R)`

Given a quaternion algebra A over $\mathbf{Q}$ or a quaternion order R over $\mathbf{Z}$, this function returns a 2×2-matrix representation of A, defined over a quadratic extension.

86.9 Attributes of Orders

For further information about orders of associative algebras, see Section 81.4.

For a quaternion order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, MAGMA additionally defines the following functions.

`Algebra(S)`

`QuaternionAlgebra(S)`

The quaternion algebra for which S is an order.

`BasisMatrix(S)`

`EmbeddingMatrix(S)`

Returns the basis matrix of the quaternion order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$. The rows of the matrix give the basis elements of S with respect to the basis of the container algebra.

`Discriminant(S)`

Given an order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, this function returns the reduced discriminant of S as a positive integer or a normalized polynomial.

`FactoredDiscriminant(S)`

Given a quaternion order S, this function returns the factorisation of the reduced discriminant of S (that is, `Factorization(Discriminant(S))`).

`Conductor(S)`

`Level(S)`

Given an order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ in a quaternion algebra A, this function returns the reduced index of S in a maximal order of A containing it. Together with the reduced discriminant of the order, this serves to classify the local isomorphism class of an Eichler order.

`Normalizer(S)`

Let S be an order in a definite quaternion algebra A over a field F where F is the rationals, $\mathbf{F}_q(t)$ or a number field. This function returns a matrix group G isomorphic to the normalizer of S in A^* modulo F^*. A homomorphism from G to A^* is also returned.

86.10 Predicates of Orders

Let O be a quaternion order with base ring $\mathbf{Z}$, $\mathbf{F}_q[X]$ with q odd, or a number ring. Then MAGMA can test the following predicates.

`IsMaximal(O)`

Returns `true` if and only if the order O is maximal.

`IspMaximal(O, p)`

Returns `true` if and only if the order O is maximal at the prime or prime ideal p.

`IsEichler(O)`

`MaximalOrders` BOOLELT *Default :* `false`

Returns `true` if and only if the order O is *Eichler*, that is an intersection of two (not necessarily distinct) maximal orders. The function calls the `EichlerInvariant` intrinsic explained below.

If the optional argument `MaximalOrders` is set to `true`, the algorithm also returns two maximal orders such that O is their intersection.

`IsEichler(O, p)`

`MaximalOrders` BOOLELT *Default :* `false`

Returns `true` if and only if the completion of the order O at the prime (ideal) p is *Eichler*.

If the optional argument `MaximalOrders` is set to `true`, the algorithm also returns two p-maximal orders such that O is their intersection.

`EichlerInvariant(O, p)`

Returns the local Eichler invariant of O at some prime (ideal) p which divides the discriminant of O. Let R be the base ring of O and let J be the Jacobson radical of the R/p-algebra O/pO. If J has dimension 3 then the Eichler invariant is defined to be 0. Otherwise the quotient of O/pO by J is either isomorphic to a direct sum of two copies of R/p or a quadratic field extension of R/p. In the first case the Eichler invariant is 1, in the latter it is -1.

`IsHereditary(O)`

Returns `true` if and only if the order O is a hereditary order in a quaternion algebra A. That is, every lattice in A of full rank such that O is contained in its left order is a projective left O-module. The hereditary orders are precisely those with squarefree discriminant.

`IsHereditary(O, p)`

Returns `true` if and only if the completion of the order O at the prime (ideal) p is *hereditary*.

`IsGorenstein(O)`

Returns `true` if and only if the order O is a Gorenstein order. That is, the dual of O with respect to the trace bilinear form is a projective O-module.

`IsGorenstein(O, p)`

Returns `true` if and only if the completion of the order O at the prime (ideal) p is *Gorenstein*.

86.11 Operations with Orders

`O1 meet O2`

This returns the order obtained by intersecting the quaternion orders O_1 and O_2.

`O ^ x`

This returns the conjugate of the order O by an element x in the associative algebra A for which O is an order (in other words the order $x^{-1}Ox$).

86.12 Ideal Theory of Orders

The right (or left) ideals of an R-order O in a quaternion algebra A defined over a number field fall into finitely many isomorphism classes. Already, for the case of number rings, it is a computationally difficult problem to compute the class group, and due to the noncommutativity of quaternion algebras, the problem of enumerating ideal classes of quaternion orders is even more difficult because this set does not have the structure of a group.

All orders in this section are required to be *Eichler*. The reader should recall that this includes the maximal orders.

The algorithms underpinning the intrinsics described in this section, in particular, those for determining ideal classes, are described in [KV10]. The methods depend on whether A is definite or indefinite. For a definite algebra, we use a variation of Kneser's neighbouring method together with Eichler's mass formula (see [Vig80, Chapter 5] and [DG88]) to construct sufficient ideals to represent all ideal classes. We then test if two right (or left) ideals I, J are isomorphic (Section 86.14.3).

For an indefinite quaternion algebra, we use a theorem of Eichler [Rei03, Th. 35.14] which states that the reduced norm gives rise to an bijection of sets between the class groups of the order and the number ring. (Over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, therefore, every ideal of an indefinite quaternion order is principal.) We may compute representative ideals I of O whose reduced norm is in a specified ideal class. We then reduce the problem of testing for an isomorphism $I \cong J$ between two ideals to the similar problem over R. Here, the problem of explicitly computing such an isomorphism is much less difficult, as an order (or ideal) will have infinitely many elements of bounded norm, and in most cases a search amongst reduced bases will find a desired element.

Ideals of quaternion orders belong to the types `AlgQuatOrdIdl` and `AlgAssVOrdIdl` according as to whether the order O is defined over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ (both of type `AlgQuatOrd`) or over a number ring (of type `AlgAssVOrd`). For more on the constructions and functions for ideals of the latter type, see Section 81.4.

86.12.1 Creation and Access Functions

`LeftIdeal(S, X)`
`lideal< S | X >`
`RightIdeal(S, X)`
`rideal< S | X >`
`ideal< S | X >`

These intrinsics construct a left, right or two-sided ideal of the order S generated by the sequence X, which should contain elements that are coercible into the algebra of S.

In the case of the constructors `lideal< | >` and `rideal< | >`, the right hand side may also be a matrix or pseudo-matrix giving the basis of the ideal with respect to the basis of S.

`PrimeIdeal(S, p)`

Given a quaternion order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, the function returns the unique two-sided (integral) prime ideal P of S over the prime p of $\mathbf{Z}$ or $\mathbf{F}_q[X]$. If p exactly divides the discriminant of S, then P properly contains pS and need not be principal, and otherwise P is equal to pS.

`CommutatorIdeal(S)`

The two-sided ideal of the quaternion order S generated by elements of the form $xy - yx$.

`MaximalLeftIdeals(O, p)`

`MaximalRightIdeals(O, p)`

The integral ideals of norm p with left or right order O.

Example H86E16

We demonstrate the construction of the 2-sided prime ideals and their relationship with the commutator ideal.

```
> S := QuaternionOrder(2*5*11);
> P := PrimeIdeal(S, 2);
> I := ideal< S | [2] cat [ x*y-y*x : x, y in Basis(S) ] >;
> P eq I;
true
> Q := PrimeIdeal(S, 5);
> R := PrimeIdeal(S, 11);
> P*Q*R eq CommutatorIdeal(S);
true
```

By way of explanation, we note that the composition of ideals is well-defined, since each of these ideals is a 2-sided ideal for S, that is, a left ideal whose right order is also S. The collection of all prime ideals over the ramified primes of a maximal order forms an elementary 2-abelian class group, and the commutator ideal is the product of these prime ideals.

Example H86E17

First we create an integral ideal I of norm 2.

```
> QQ:= Rationals();
> A<i,j> := QuaternionAlgebra< QQ | -1, -11 >;
> S := MaximalOrder(A);
> P<x> := PolynomialRing(QQ);
> P ! MinimalPolynomial(i);
x^2 + 1
> I := lideal< S | 2, 1+i >;
> Norm(I);
2
> I in MaximalLeftIdeals(S, 2);
```

```
true
```

We now examine the basis of the ideal I.

```
> Basis(I);
[ 2, 1 + i, i + k, 3/2 + 1/2*i + 1/2*j + 1/2*k ]
> [ Eltseq(x) : x in Basis(I) ];
[
    [ 2, 0, 0, 0 ],
    [ 1, 1, 0, 0 ],
    [ 0, 1, 0, 1 ],
    [ 3/2, 1/2, 1/2, 1/2 ]
]
> BasisMatrix(I, A);
[  2   0   0   0]
[  1   1   0   0]
[  0   1   0   1]
[3/2 1/2 1/2 1/2]
```

If the ideal I is printed, the rational coordinates with respect to the basis of the quaternion order S will be shown, We can get this base change matrix if we call `BasisMatrix` with no additional parameters.

```
> I;
Left Ideal with basis Pseudo-matrix over Integer Ring
[2 0 0 0]
[1 1 0 0]
[0 0 2 0]
[1 0 1 1]
> BasisMatrix(I);
[2 0 0 0]
[1 1 0 0]
[0 0 2 0]
[1 0 1 1]
```

`LeftOrder(I)`

Given an ideal I of a quaternion order defined over $\mathbf{Z}$, $\mathbf{F}_q[X]$ or a number ring, this function returns the left order of I, defined as the ring of all elements of the quaternion algebra of I mapping I to itself under left multiplication.

`RightOrder(I)`

Given an ideal I of a quaternion order defined over $\mathbf{Z}$, $\mathbf{F}_q[X]$ or a number ring, this function returns the right order of I, defined as the ring of all elements of the quaternion algebra of I mapping I to itself under right multiplication.

Example H86E18

```
> K := NumberField(Polynomial([5,0,1]));
> K;
Number Field with defining polynomial $.1^2 + 5 over the Rational Field
> A := QuaternionAlgebra<K | 3, K.1>;
> O := MaximalOrder(A);
> I := lideal<O | O.2, Norm(O.2)>;
> Norm(I);
Principal Ideal
Generator:
    2/1*$.1
> LeftOrder(I) eq O;
true
> RightOrder(I) eq O;
true
```

86.12.2 Enumeration of Ideal Classes

The tools provided for calculating ideal classes in the case of a maximal or, more generally, an Eichler order S in a quaternion algebra over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field are described in this section.

`Mass(S)`

Given a definite order S of level N over R, this function returns the mass

$$\sum_i h_i/[S_i^* : R^*]$$

where $S_1, \ldots, S_t$ represent the conjugacy classes of Eichler orders of level N and h_i denotes the number of two-sided ideal classes of S_i.

`LeftIdealClasses(S)`

`RightIdealClasses(S)`

`Support` [RngOrdIdl] *Default :*

These intrinsics find representatives for the left or right locally free ideal classes of S, where S is an order in a quaternion algebra A defined over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field. The algorithms are guaranteed to work correctly only when S is a maximal order or an Eichler order.

For definite algebras, the support of the returned ideals may be specified using parameter `Support`; this should be a sequence of primes or prime ideals in the base ring which generate the narrow class group and which do not ramify in A. In this case, the routine will find ideal class representatives whose norms are divisible only by primes in the specified support. The algorithm usually runs faster if the support

is either not specified or if the support includes the prime divisors of the discriminant of S.

For indefinite algebras, the representatives are obtained from a ray class group of the base field.

`TwoSidedIdealClasses(S)`

`Support` [RngOrdIdl] *Default :*

Given an order S in a quaternion algebra, this function returns a sequence containing representatives for the two-sided locally free ideal classes of S. The algorithm is only guaranteed to work when S is a maximal order or an Eichler order.

If the optional argument `Support` is specified, Magma tries to construct representatives that have norm divisible by primes or prime ideals in the specified support. If this is not possible, the set specified using parameter `Support` will be enlarged by the prime divisors of the discriminant of S. If this enlarged set is still not large enough (which can only happen if the set does not generate the class of group of the base ring of S) an error will be raised.

`TwoSidedIdealClassGroup(S : Support)`

`Support` [RngOrdIdl] *Default :*

Given an order S in a quaternion algebra, this returns the two-sided ideal class group of S as an abstract abelian group, together with a map from the group to the set of two-sided ideal classes of S. Inverses with respect to this map can be calculated in a very efficient way and thus may be used to compute discrete logarithms. The algorithm is only guaranteed to work when S is a maximal order or an Eichler order.

The optional parameter `Support` may be used to specify a support: this should be a sequence of primes or prime ideals in the base ring which generate the narrow class group and which do not ramify in the parent algebra.

`ConjugacyClasses(S)`

Given a maximal order (or an Eichler order S of level N) in a quaternion algebra A, this function returns representatives for the conjugacy classes of maximal orders (or orders of level N) in A.

For definite algebras, the algorithm involves computing the right ideal classes (in fact, the two problems are computationally equivalent in practice).

Example H86E19

In the following example we construct a maximal order in the quaternion algebra ramified at 37, and enumerate its left ideal classes.

```
> S := QuaternionOrder(37);
> ideals := LeftIdealClasses(S);
> [ Basis(I) : I in ideals ];
[
  [ 1, i, j, k ],
```

```
  [ 2, 2*i, 1 + i + j, i + k ],
  [ 2, 2*i, i + j, k ]
]
```

We now compute the right orders of the two nontrivial left ideal classes.

```
> _, I, J := Explode(ideals);
> R1 := RightOrder(I);
> R2 := RightOrder(J);
> IsIsomorphic(R1,R2);
true
```

Although the ideals I and J are non-isomorphic left ideals over S, they have isomorphic right orders. In the example following the next section we explore this phenomenon further.

Example H86E20

In this example, we enumerate ideal classes for a definite quaternion algebra (over a totally real field).

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> Foo := InfinitePlaces(F);
> pp := Decomposition(Z_F, 17)[1][1];
> A := QuaternionAlgebra(pp, Foo);
> O := MaximalOrder(A);
> time Rideals := RightIdealClasses(O);
Time: 0.870
> #Rideals;
2
```

Example H86E21

In this example the classgroup of the base field is not trivial and there are some ramified prime ideals in the algebra. Hence we expect nontrival two-sided ideal classes.

```
> K:= QuadraticField(401);
> A:= QuaternionAlgebra< K | -1, -1>;
> RamifiedPlaces(A);
[
    Prime Ideal
    Two element generators:
        2
        $.2 + 1,
    Prime Ideal
    Two element generators:
        2
        $.2 + 2
]
```

```
[ 1st place at infinity, 2nd place at infinity ]
> M:= MaximalOrder(A);
> #TwoSidedIdealClasses(M);
10
> time #RightIdealClasses(M);
140
Time: 6.470
```

86.12.3 Operations on Ideals

In addition to operations on ideals of orders over more general rings (see Section 81.4), the following operations are defined for ideals of quaternion orders over $\mathbf{Z}$, $\mathbf{F}_q[X]$ and number rings.

`I * J`

The composite of I and J, where the right order of I equals the left order of J.

`I meet J`

Given ideals or orders I and J, this function returns the intersection $I \cap J$.

`Conjugate(I)`

Given an ideal I (of a quaternion algebra), this function returns the conjugate ideal.

`Norm(I)`

Given an ideal I over $\mathbf{Z}$, this function returns the reduced norm of the ideal, defined as the positive generator of the image of the reduced norm map in $\mathbf{Q}$.

Given an ideal I over $\mathbf{F}_q[X]$, this function returns the reduced norm of the ideal, defined as the normalized generator of the image of the reduced norm map in $\mathbf{F}_q(X)$.

Given an ideal I over a commutative order, this function returns the reduced norm of I as a fractional ideal of the order.

`Factorization(I)`

Given a two-sided ideal I of an hereditary order O, this function returns the unique factorization of I into two-sided O-ideals. The result is a sequence of tuples. The first entry of each tuple is a two-sided ideal of O, the second is its exponent. (If the base ring of I is $\mathbf{F}_q[X]$, then q is currently required to be odd.)

86.13 Norm Spaces and Basis Reduction

For definite quaternion orders or ideals one can compute reduced bases and Gram matrices.

If the base ring of the order or ideal is $\mathbf{Z}$ or $\mathbf{F}_q[X]$ with q odd, the Gram matrices can be made unique up to isomorphism. In fact, in these cases, the isomorphism testing of ideals and orders is based on this reduction.

`NormSpace(A)`

Given a quaternion algebra A over any field F not of characteristic 2, this function returns the underlying F-space with inner product the norm form. A map from A into the structure is returned as second value.

`NormSpace(S)`

Given a quaternion order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ (with q odd), this function returns the underlying module over its base ring, with inner product respect to the norm. A map from O into the structure is returned as second value.

`GramMatrix(S)`
`GramMatrix(I)`

The Gram matrix of the quaternion order S or ideal I over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ with respect to the norm on the basis for S.

`ReducedGramMatrix(S)`

Given an order or ideal S over $\mathbf{Z}$ in a definite quaternion algebra, this function returns the Gram matrix G of the corresponding lattice.

The quaternion ideal machinery makes use of a Minkowski basis reduction algorithm which returns a unique normalized reduced Gram matrix G for any definite quaternion ideal. This forms the core of the isomorphism testing for quaternion ideals.

`ReducedBasis(S)`

Given an order or ideal S over $\mathbf{Z}$ in a definite quaternion algebra, this function returns some basis B of S the Gram matrix G of the corresponding lattice associated with S. Note that while there exists a unique Minkowski-reduced Gram matrix G, the basis B is not unique.

Example H86E22

Recall that Minkowski basis reduction is used which returns a unique normalized reduced Gram matrix G for any definite quaternion ideal. This forms the core of the isomorphism testing for quaternion ideals. We illustrate this by applying it to representatives of the set of left ideal classes of an order.

```
> A := QuaternionOrder(19,2);
> ideals := LeftIdealClasses(A);
> #ideals;
5
```

```
> [ (1/Norm(I))*ReducedGramMatrix(I) : I in ideals ];
[
    [ 2  0  1  1]
    [ 0  2  1  1]
    [ 1  1 20  1]
    [ 1  1  1 20],

    [6 0 1 3]
    [0 6 3 1]
    [1 3 8 1]
    [3 1 1 8],

    [6 0 1 3]
    [0 6 3 1]
    [1 3 8 1]
    [3 1 1 8],

    [ 4  0  1 -1]
    [ 0  4  1  1]
    [ 1  1 10  0]
    [-1  1  0 10],

    [ 4  0  1 -1]
    [ 0  4  1  1]
    [ 1  1 10  0]
    [-1  1  0 10]
]
```

`ReducedGramMatrix(S)`

`ReducedBasis(S)`

`Canonical` BoolElt *Default :* `false`

Given an order or ideal S over $\mathbf{F}_q[X]$ in a quaternion algebra A, this function returns a Gram matrix and/or a basis of S whose Gram matrix is in dominant diagonal form (see the function `DominantDiagonalForm` in Section 31.2.1). The Gram matrix will not be unique unless A is definite and `Canonical` is set to `true`.

`ReducedBasis(O)`

`ReducedBasis(I)`

Returns a "reduced" basis for the order O or the ideal I over some number ring. If O or I arise from a definite quaternion algebra, then this basis is LLL-reduced with respect to the norm form; otherwise, the basis is reduced with respect to a Minkowski-like embedding (see [KV10, Section 4]).

`OptimizedRepresentation(O)`

`OptimisedRepresentation(O)`

Given an order O contained in a quaternion algebra A over $\mathbf{Q}$ or a number field F, this function returns a new quaternion algebra A' such that $A' = ((a, b)/F)$ where a and b are small (with respect to O), and, as second return value, an isomorphism $A \to A'$.

`OptimizedRepresentation(A)`

`OptimisedRepresentation(A)`

Given a quaternion algebra A over $\mathbf{Q}$ or a number field F, this function returns a new quaternion algebra A' such that $A' = ((a, b)/F)$ where a and b are small. An isomorphism $A \to A'$ is returned as second value.

`Enumerate(O, A, B)`

The sequence of all elements x (up to sign) in the definite quaternion order O or ideal I over $\mathbf{Z}$ such that the reduced norm of x lies in the interval $[A, \ldots B]$ or $[0, \ldots B]$, respectively.

`Enumerate(O, A, B)`

`Enumerate(O, B)`

The sequence of elements x (up to sign) in the definite quaternion order O or ideal I over a number ring such that the absolute trace of the norm of x lies in the interval $[A, \ldots B]$ or $[0, \ldots B]$, respectively.

86.14 Isomorphisms

86.14.1 Isomorphisms of Algebras

Two quaternion algebras A, B over a common field F are isomorphic algebras if and only if they share the same ramified places. Finding an explicit isomorphism is much harder. Currently MAGMA embeds the first standard generator of A into B and then finds another element perpendicular to that image having the correct minimal polynomial. In particular, this requires the construction of two points on a conic (or equivalently, the solution of two norm equations) over quadratic extensions of F.

`IsIsomorphic(A, B)`

`Isomorphism` BOOLELT *Default :* `false`

Given two quaternion algebras A, B over $\mathbf{Q}$, $\mathbf{F}_q(X)$ with q odd or a number field, this function returns `true` if and only if they are isomorphic.

If the algebras are isomorphic and `Isomorphism` is set to true, an isomorphism $A \to B$ is also returned.

86.14.2 Isomorphisms of Orders

Two orders S, T in a quaternion algebra A are isomorphic if and only if they are conjugate in A.

In a definite algebra, we use the fact that this conjugation induces an isometry of the quadratic $\mathbf{Z}$- or $\mathbf{F}_q[X]$-modules S and T equipped with the (absolute) norm form. Over an indefinite algebra, we use the fact that any connecting ideal I having left order S and right order T must be isomorphic to a right ideal of T. See [KV10] for details.

`IsIsomorphic(S, T)`

`IsConjugate(S, T)`

`FindElement`	BoolElt	*Default :* `false`
`ConnectingIdeal`	AlgAssVOrdIdl	*Default :*

Given orders S and T in a quaternion algebra A over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field, this function returns `true` if and only if S and T are isomorphic.

For indefinite algebras, the orders are currently required to be maximal.

If `FindElement` is set, the second return value is an isomorphism from S to T and the third return value is an element a with $T = a^{-1}Sa$, inducing the isomorphism. For indefinite algebras, this search is expensive and may sometimes fail (as in `IsIsomorphic` for ideals, see below).

For indefinite algebras defined over number fields, part of the computation may be faster when a connecting ideal is specified using the parameter `ConnectingIdeal`; this should be an ideal with left order S and right order T.

`Isomorphism(S, T)`

Given two isomorphic definite quaternion orders S, T over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ (with q odd), this function returns an algebra isomorphism. For orders over number rings the intrinsic `IsIsomorphic` should be used with the optional argument `FindElement` set.

86.14.3 Isomorphisms of Ideals

Two right (left) ideals I, J of an order O in a quaternion algebra A are isomorphic O-modules if and only if $xI = J$ ($Ix = J$) for some $x \in A^*$.

To decide whether I and J are isomorphic, we test whether the *colon ideal* $(J : I) = \{x \in A : xI \subset J\}$ (similarly defined if I, J are left ideals) is principal, or not.

Over $\mathbf{Z}$, to accomplish this task we compute a Minkowski-reduced Gram matrix of $(J : I)$ which produces an element of smallest norm. Over $\mathbf{F}_q[X]$ we compute a Gram matrix which has dominant diagonal form (see [Ger03]) which also produces a suitable element. Over other number rings R, we search $(J : I)$ for an element of the required norm by computing a reduced basis. This method runs very quickly for "reasonably small" input. (See [KV10, DD08] for further details.)

`IsIsomorphic(I, J)`

Given ideals I and J of the same order O in a quaternion algebra A over $\mathbf{Q}$, $\mathbf{F}_q(X)$ (with q odd) or a number field, this function returns `true` if and only if the quaternion ideals I and J are isomorphic (left or right) O-ideals. If I and J are isomorphic, there exists some $x \in A$ such that $xI = J$ (in the case of right O-ideals) or $Ix = J$ (for left ideals). For definite algebras, such an element x is always returned. For indefinite algebras over $\mathbf{Z}$ or a number ring, a search for such an element is made, and if one is found then it is returned.

`IsPrincipal(I)`

Given a left (or right) ideal I over $\mathbf{Z}, \mathbf{F}_q[X]$ or a number ring, this function returns `true` if and only if I is a principal ideal; if so, a generator is returned as the second value.

`IsLeftIsomorphic(I, J)`

`IsRightIsomorphic(I, J)`

Given two definite ideals over $\mathbf{Z}$ or $\mathbf{F}_q[X]$ (with q odd) with the same left (or right) order S, this function returns `true` if and only if they are isomorphic as S-modules. The isomorphism and the transforming scalar in the quaternion algebra are returned as second and third values if `true`.

`IsLeftIsomorphic(I, J)`

`IsRightIsomorphic(I, J)`

Given two left (or right) ideals I and J over a number ring, with the same left order O, this function returns `true` if and only if they are isomorphic as O-modules. The isomorphism is given by multiplication as a second return value.

`LeftIsomorphism(I, J)`

Given two isomorphic left ideals over a definite order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, this function returns the S-module isomorphism between them, followed by the quaternion algebra element which defines the isomorphism by right-multiplication.

`RightIsomorphism(I, J)`

Given two isomorphic right ideals over a definite order S over $\mathbf{Z}$ or $\mathbf{F}_q[X]$, this function returns the S-module isomorphism between them, followed by the quaternion algebra element which defines the isomorphism by left-multiplication.

86.14.4 Examples

Example H86E23

In this example, we create two quaternion algebras over $\mathbf{F}_7$, show that they are isomorphic and find an isomorphism between them.

```
> F<x> := RationalFunctionField( GF(7) );
> Q1 := QuaternionAlgebra< F | (x^2+x-1)*(x+1), x >;
> a := x^3 + x^2 + 3;
> b := x^13 + 4*x^11 + 2*x^10 + x^9 + 6*x^8 + 4*x^5 + 3*x^4 + x;
> Q2:= QuaternionAlgebra< F | a, b >;
> ok, phi:= IsIsomorphic(Q1, Q2 : Isomorphism);
> ok;
true
> forall{ <x,y> : x,y in Basis(Q1) |  phi(x*y) eq phi(x)*phi(y) };
true
```

Example H86E24

In this example, we create two ideals, show that they have isomorphic right orders, and then explicitly exhibit the isomorphism.

```
> A := QuaternionAlgebra(37);
> S := MaximalOrder(A);
> ideals := LeftIdealClasses(S);
> _, I, J := Explode(ideals);
> R := RightOrder(I);
> Q := RightOrder(J);
> IsIsomorphic(R,Q);
true
> // Get the x which conjugates R to Q:
> _, pi := Isomorphism(R,Q);
> Norm(pi);
37
> J := lideal< S | [ x*pi : x in Basis(J) ] >;
> RightOrder(J) eq R;
true
```

Example H86E25

We construct two non-isomorphic left ideals with the same left and right orders, then investigate their isomorphisms as right ideals.

```
> S := QuaternionOrder(37);
> ideals := LeftIdealClasses(S);
> _, I, J := Explode(ideals);
> R := RightOrder(I);
> _, pi := Isomorphism(R,RightOrder(J));
```

```
> J := lideal< S | [ x*pi : x in Basis(J) ] >;
> IsLeftIsomorphic(I,J);
false
> IsRightIsomorphic(I,J);
true Mapping from: AlgQuatOrd: I to AlgQuatOrd: J given by a rule [no inverse]
1 + i - 2*k
> h, x := RightIsomorphism(I,J);
> y := [1,2,-1,3];
> y := &+[y[i]*b[i] : i in [1 .. 4]] where b is Basis(I);
> h(y);
[-73  15  31   4]
> x*y;
-73 + 15*i + 31*j + 4*k
```

The existence of an isomorphism as a right ideal is due to the fact that the two-sided ideals of R do not have non-isomorphic counterparts in S.

```
> TwoSidedIdealClasses(R);
[ Ideal with basis Pseudo-matrix over Integer Ring
1 * [1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
, Ideal with basis Pseudo-matrix over Integer Ring
1 * [37  0 32 18]
[ 0 37 10  2]
[ 0  0  1  0]
[ 0  0  0  1]
 ]
> TwoSidedIdealClasses(S);
[ Ideal with basis Pseudo-matrix over Integer Ring
1 * [1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
 ]
```

Thus while `Conjugate(I)*J` is in the non-principal R-ideal class, the ideal `I*Conjugate(J)` represents the unique principal ideal class of S.

Example H86E26

We exhibit isomorphism testing for ideals of orders over number rings.

```
> P<x> := PolynomialRing(Rationals());
> F<b> := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> F := FieldOfFractions(Z_F);
> A<alpha,beta,alphabeta> := QuaternionAlgebra<F | -3, b>;
> O := Order([alpha,beta,alphabeta]);
```

```
> O;
Order of Quaternion Algebra with base ring F
with coefficient ring Maximal Equation Order with defining polynomial x^3 - 3*x
    - 1 over its ground order
> I := ideal<O | 2>;
> I eq (I + ideal<O | 2>);
true
> I eq (I + ideal<O | 3>);
false
>
> Foo := InfinitePlaces(F);
> A := QuaternionAlgebra(ideal<Z_F | 2*3*5>, Foo);
> IsDefinite(A);
true
> O := MaximalOrder(A);
> I := rideal<O | Norm(O.2), O.2>;
> J := rideal<O | Norm(O.3), O.3>;
> IsIsomorphic(I, J);
true (F.2 + F.3) + (27/9190*F.1 - 143/9190*F.2 - 73/9190*F.3)*i +
(-251/27570*F.1 + 7/2757*F.2 + 10/2757*F.3)*k
```

86.15 Units and Unit Groups

Let S be a definite quaternion order over $\mathbf{Z}$, $\mathbf{F}_q[X]$ with q odd, or a number ring. In the first two cases, the unit group of S is finite and can be read off any reduced Gram matrix of S. If the base ring of S is some number ring R, then an explicit description of the finite quotient S^*/R^* is given in [Vig76].

`NormOneGroup(S)`

`ModScalars` BoolElt *Default :* `false`

Returns a group G isomorphic to the group S_1 of elements in S with reduced norm 1 (unless `ModScalars` is set, in which case G is isomorphic to S_1 modulo $\{\pm 1\}$).

The second object returned is a map from G to S expressing the isomorphism.

`Units(S)`

This intrinsic computes the set of units S^* for the definite order S. When the base ring of S is Z or $\mathbf{F}_q[X]$, the returned sequence contains all units of S; when the base field is a number field, the returned sequence contains representatives modulo the unit group of the base ring (since the unit group is infinite in general).

`MultiplicativeGroup(S)`

`UnitGroup(S)`

This intrinsic computes the unit group S^* of the definite quaternion order S. The function returns an abstract group G, and a map from G to S. When the base ring of S is Z or $\mathbf{F}_q[X]$, G represents the full group of units of S; when the base field is a number field, G represents the unit group of S modulo the unit group of the base ring (since the unit group is infinite in general).

Example H86E27

The following example illustrates the unit group computation for an order in a definite quaternion algebra over $\mathbf{Q}$.

```
> A := QuaternionAlgebra< RationalField() | -1, -1 >;
> S1 := MaximalOrder(A);
> S2 := QuaternionOrder(A,2);
> G1, h1 := UnitGroup(S1);
> #G1;
24
> [ A | h1(g) : g in G1 ];
[ 1, -1, -j, -k, i, -1/2 + 1/2*i - 1/2*j - 1/2*k, 1/2 + 1/2*i - 1/2*j + 1/2*k,
-1/2 - 1/2*i - 1/2*j + 1/2*k, -1/2 + 1/2*i + 1/2*j + 1/2*k, 1/2 - 1/2*i - 1/2*j
- 1/2*k, 1/2 + 1/2*i + 1/2*j - 1/2*k, 1/2 - 1/2*i + 1/2*j + 1/2*k, -1/2 - 1/2*i
+ 1/2*j - 1/2*k, 1/2 - 1/2*i + 1/2*j - 1/2*k, -1/2 + 1/2*i + 1/2*j - 1/2*k, 1/2
+ 1/2*i - 1/2*j - 1/2*k, -1/2 - 1/2*i - 1/2*j - 1/2*k, 1/2 + 1/2*i + 1/2*j +
1/2*k, -1/2 - 1/2*i + 1/2*j + 1/2*k, -1/2 + 1/2*i - 1/2*j + 1/2*k, 1/2 - 1/2*i -
1/2*j + 1/2*k, k, -i, j ]
> G2, h2 := UnitGroup(S2);
> #G2;
8
> [ A | h2(g) : g in G2 ];
[ 1, -1, -j, j, k, -k, -i, i ]
```

The unit groups of orders in indefinite quaternion algebras A are infinite arithmetic groups, which are twisted analogues of the groups $\mathrm{SL}_2(\mathbf{Z})$ and their families of subgroups. These are studied in relation to their actions on the upper half complex plane, via an embedding in $\mathrm{GL}_2(\mathbf{R})$ provided by some isomorphism $A \otimes \mathbf{R} \cong M_2(\mathbf{R})$.

Example H86E28

Now we exhibit unit group computations over a number ring.

```
> P<x> := PolynomialRing(Rationals());
> F := NumberField(x^3-3*x-1);
> Z_F := MaximalOrder(F);
> Foo := InfinitePlaces(F);
```

We use `SetSeed` since the following line makes random choices.

```
> SetSeed(1);
```

```
> A := QuaternionAlgebra(ideal<Z_F | 2>, Foo);
> IsDefinite(A);
true
> O := MaximalOrder(A);
> U, h := UnitGroup(O);
> U;
Permutation group U acting on a set of cardinality 12
Order = 12 = 2^2 * 3
    Id(U)
    (1, 2, 4)(3, 6, 7)(5, 9, 10)(8, 12, 11)
    (1, 3)(2, 5)(4, 8)(6, 11)(7, 9)(10, 12)
> #Units(O);
12
```

86.16 Bibliography

[**DD08**] L. Dembele and S. Donnelly. Computing Hilbert Modular Forms Over Fields With Nontrivial Class Group. In S. Pauli F. Hess and M.Pohst, editors, *ANTS VIII*, volume 5011 of *LNCS*. Springer-Verlag, 2008.

[**DG88**] M. Denert and J. Van Geel. The Class Number of Hereditary Orders in Non-Eichler Algebras over Global Fields. *Math. Ann.*, 282:379–393, 1988.

[**Fri97**] Carsten Friedrichs. Berechnung relativer Ganzheitsbasen mit dem Round-2-Algorithmus. Diplomarbeit, Technische Universität Berlin, 1997. URL:http://www.math.tu-berlin.de/∼kant/publications/diplom/friedrichs.ps.gz.

[**Ger03**] Larry J. Gerstein. Definite quadratic forms over $\mathbf{F}_q[X]$. *J. Algebra*, 268(1):252–263, 2003.

[**IR93**] Gábor Ivanyos and Lajos Rónyai. Finding maximal orders in semisimple algebras over $\mathbf{Q}$. *Comput. Complexity*, 3(3):245–261, 1993.

[**KV10**] M. Kirschmer and J. Voight. Algorithmic enumeration of ideal classes for quaternion orders. *SIAM J. Comput. (SICOMP)*, 39(5):1714–1747, 2010.

[**Rei03**] Irving Reiner. *Maximal Orders*, volume 28 of *LMS Monographs*. Oxford University Press, 2003.

[**Vig76**] M.-F. Vignéras. Simplification pour les ordres des corps de quaternions totalement définits. *J. Reine Angew. Math.*, 286-287:257–277, 1976.

[**Vig80**] M.-F. Vignéras. *Arithmétique des Algèbres de Quaternions*, volume 800 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1980.

[**Voi05**] John Voight. *Quadratic forms and quaternion algebras: Algorithms and arithmetic*. Dissertation, University of California, Berkeley, 2005.

[**Voi11**] John Voight. Identifying the matrix ring: algorithms for quaternion algebras and quadratic forms. 2011.

87 ALGEBRAS WITH INVOLUTION

Chapter 87
ALGEBRAS WITH INVOLUTION

87.1 Introduction

In this chapter we describe techniques for computing with $*$-*algebras*, namely algebras equipped with an anti-automorphism $x \mapsto x^*$ of order at most 2 (an *involution* or *star*). For further information on involutions and the structure of $*$-algebras see [Alb61] and [KMRT98].

The principal application of these techniques is currently to isometry groups of systems of reflexive forms (and the intimately related study of intersections of classical groups). However, it is also possible to use the techniques to compute with group algebras of moderate dimension.

To any set of reflexive forms defined on a common vector space (a *system of forms*) one may associate a matrix $*$-algebra called the *adjoint algebra* of the system. The group of units of this adjoint algebra contains a natural subgroup of *unitary elements*, namely those elements x satisfying the condition $x^* = x^{-1}$. The group of unitary elements coincides with the group of isometries of the system of forms, which is also the intersection of the general classical groups associated with these forms.

The StarAlgebras package provides functions that enable the user to investigate the structure of $*$-algebras. It also provides functions to compute and determine the structure of the group of isometries of a system of reflexive forms, and to compute intersections of arbitrary collections of classical groups defined on a common vector space.

The algorithms are mainly due to Peter Brooksbank and James Wilson [BW11a, BW11b].

87.2 Algebras with Involution

This section introduces two general constructions for $*$-algebras:

(i) The algebra of adjoints of a system of reflexive (alternating, reflexive, or Hermitian) forms $[\phi_1, \ldots, \phi_e]$ defined on a common vector space V.

(ii) The group algebra $K[G]$, where K is any ring and G is a finite group.

We also provide a constructor function for simple $*$-algebras.

87.2.1 Reflexive Forms

A *reflexive form* on a K-vector space V is a bilinear function $\phi : V \times V \to K$ such that, whenever $\phi(u, v) = 0$ for $u, v \in V$, we also have $\phi(v, u) = 0$. Reflexive forms ϕ and ψ on V are *isometric* if $\phi(u, v) = \psi(u, v)$ for all $u, v \in V$; they are *similar* if there exists $a \in K$ such that ϕ and $a\psi$ are isometric. The *radical* of a reflexive form ϕ is the subspace $\mathrm{rad}\phi = \{u \in V : \phi(u, V) = 0\}$, and ϕ is *nondegenerate* if $\mathrm{rad}\phi = 0$.

A fundamental result of Birkhoff and von Neumann states that there are three similarity classes of reflexive forms: *alternating*, *symmetric*, and *Hermitian*. Each such form ϕ is represented by a matrix and an automorphism of K. Specifically, regarding V as the space of row vectors, we specify F and α such that $\phi(u, v) = u^\alpha F v^{\mathrm{tr}}$, where α is the identity if ϕ is bilinear, or the automorphism $x \mapsto \overline{x}$ of order 2 if ϕ is Hermitian. Thus a matrix $g \in \mathrm{GL}(d, K)$ is an *isometry* (respectively *similarity*) of the reflexive form if $g^\alpha F g^{\mathrm{tr}} = F$ (respectively $g^\alpha F g^{\mathrm{tr}} = aF$).

`IsometryGroup(F : parameters)`

`Auto` RngIntElt *Default :* 0

The group of isometries of the (possibly degenerate) reflexive form represented by the matrix F with entries in a finite field $\mathbf{F}_{p^e}$. The field automorphism associated with F is specified by the parameter `Auto`, which represents the exponent f in the map $x \mapsto x^{p^f}$. The default is that F is bilinear on its base ring.

`SimilarityGroup(F : parameters)`

`Auto` RngIntElt *Default :* 0

The group of similarities of the (possibly degenerate) reflexive form represented by the matrix F with entries in a finite field $\mathbf{F}_{p^e}$. The field automorphism associated with F is specified by the parameter `Auto`, which represents the exponent f in the map $x \mapsto x^{p^f}$. The default is that F is bilinear on its base ring.

87.2.2 Systems of Reflexive Forms

A *system of forms* is a sequence $[\phi_1, \ldots, \phi_e]$, where each ϕ_i is a reflexive form on a common K-vector space V. The radical of a system of forms is the intersection of the radicals of the individual forms in the system; A system is *nondegenerate* if its radical is zero.

A classical group on a K-vector space V preserves a reflexive form on V that is unique up to similarity. Hence systems of forms arise naturally from sets of classical groups having V as their common defining module.

Systems of forms also arise naturally from the study of p-groups. Let G be a finite p-group. Let $G = \gamma_1(G) > \gamma_2(G) > \ldots > \gamma_m(G) = 1$ denote the lower central series of G, and let $1 = \zeta_1(G) < \zeta_2(G) < \ldots < \zeta_n(G) = G$ denote its upper central series. Let $\Phi(G)$ be the Frattini subgroup of G, and put $N := \langle \Phi(G), \zeta_{n-1}(G) \rangle$. Then $V = G/N$ and $W = \gamma_2(G)/\gamma_3(G)$ are $\mathrm{GF}(p)$-vector spaces and commutation in G induces a bilinear map $V \times V \to W$. One now obtains a system of forms associated to G by choosing bases for V and W.

Just as matrices are useful representations of bilinear forms, so are systems of forms convenient computational models for *bilinear maps*.

PGroupToForms(G)

PGroupToForms(G)

Return a system of forms associated to the p-group G. For matrix groups, the input must be a class 2 p-group.

Example H87E1

We construct a system of forms associated to a Sylow 7-subgroup of $\mathrm{GL}(3,7)$.

```
> S := ClassicalSylow(GL (3, 7), 7);
> G := PCGroup(S);
> Forms := PGroupToForms(G);
> Forms;
[
    [0 1]
    [6 0]
]
```

Now we apply the same construction working directly with the matrix group as a p-group of class 2.

```
> Forms := PGroupToForms(S);
> Forms;
[
    [0 1]
    [6 0]
]
```

It is preferable to input a PC-group if such a representation can readily be obtained. The option to use a matrix group for p-groups of class 2 enables the user to construct an associated system of forms in situations where it requires considerably more time to first construct a PC-representation.

87.2.3 Basic Attributes of $*$-Algebras

Using the following functions one can ascertain whether or not a given algebra has an assigned involution, and also access the map if it has.

IsStarAlgebra(A)

IsStarAlgebra(A)

Return `true` if and only if A has an assigned involution.

Star(A)

Star(A)

Return the involution associated to the $*$-algebra A.

87.2.4 Adjoint Algebras

Let $S = [\phi_1, \ldots, \phi_e]$ be a system of reflexive forms on a K-vector space V. Let R denote the algebra $\mathrm{End}_K(V)$ and R^{op} denote its opposite ring. Then the *algebra of adjoints* of S is defined as follows:

$$\mathrm{Adj}(S) = \{(x, y) \in R \times R^{\mathrm{op}} : \phi_i(ux, v) = \phi_i(u, yv)\ \forall u, v \in V,\ \forall i \in \{1, \ldots, e\}\}.$$

As the ϕ_i are reflexive forms, $(x, y) \in \mathrm{Adj}(S)$ if and only if $(y, x) \in \mathrm{Adj}(S)$. If S is nondegenerate, then y is uniquely determined by x; thus we identify $\mathrm{Adj}(S)$ with its projection onto R and the assignment $x^* := y$ equips $\mathrm{Adj}(S)$ with an involution.

The function to compute $\mathrm{Adj}(S)$ is an implementation of the algorithms in [BW11a, Proposition 5.1] and [BW11b, Section 5].

`AdjointAlgebra(S : parameters)`

`Autos` SeqEnum *Default :* $[0, .., 0]$

Given a sequence S containing a nondegenerate system of reflexive forms, this function returns the $*$-algebra of adjoints of the forms in S. The parameter `Autos` is used to specify a list of Frobenius exponents associated with the given forms. By default all forms are considered to be bilinear over their common base ring, which must be a finite field. Note that the individual forms in the system are allowed to be degenerate.

Example H87E2

We construct the algebra of adjoints of a particular pair of forms on a vector space of dimension 3 over GF(5^2). The matrices have entries in GF(5) but we regard them as forms over the larger field, and define the first (symmetric) form to be Hermitian over the larger field.

```
> MA := MatrixAlgebra(GF (25), 3);
> F := MA![1,2,0,2,3,4,0,4,1];
> G := MA![0,1,0,4,0,0,0,0,0];
> A := AdjointAlgebra([F, G] : Autos := [1, 0]);
> IsStarAlgebra(A);
true
> Degree(A);
6
> BaseRing(A);
Finite field of size 5
```

Observe that the function has converted the given system of forms to a new system over GF(5) and returned an algebra of degree 6 over the subfield. Now we access the involution on A and apply it to a generator of A.

```
> star := Star(A);
> A.3;
[1 0 2 0 2 0]
[0 1 0 2 0 2]
```

```
[1 0 4 0 4 0]
[0 1 0 4 0 4]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
> A.3@star;
[4 0 3 0 3 0]
[0 4 0 3 0 3]
[4 0 1 0 1 0]
[0 4 0 1 0 1]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
```

87.2.5 Group Algebras

If G is a finite group and R is any ring, then the group algebra $A = R[G]$ possesses a natural involution. Thus each group algebra may be regarded as a $*$-algebra.

`StarOnGroupAlgebra(A)`

The natural involution on the group algebra A induced by inversion on the underlying group. Specifically, if $a = \sum_{g \in G} \alpha_g g \in A$, then $a^* = \sum_{g \in G} \alpha_g g^{-1}$.

`GroupAlgebraAsStarAlgebra(R, G)`

Construct the group algebra $R[G]$ equipped with the natural involution afforded by inversion in G.

Example H87E3

We construct the group algebra $Z[S_3]$ as a $*$-algebra.

```
> G := SymmetricGroup(3);
> K := Integers();
> A := GroupAlgebraAsStarAlgebra(K, G);
> IsStarAlgebra(A);
true;
```

Now access the involution on A and apply it to an element of A.

```
> star := Star(A);
> a := A![0,0,1,0,3,0];
> a;
(1, 3, 2) + 3*(1, 2)
> a@star;
(1, 2, 3) + 3*(1, 2)
```

Alternatively, $Z[S_3]$ can be constructed using the standard constructor and the involution can be attached later.

```
> A := GroupAlgebra(K, G);
```

```
> IsStarAlgebra(A);
false
> StarOnGroupAlgebra(A);
Mapping from: AlgGrp: A to AlgGrp: A given by a rule [no inverse]
> IsStarAlgebra(A);
true
```

87.2.6 Simple $*$-Algebras

The Artinian *simple $*$-algebras* – those having no proper $*$-invariant ideals – were classified by Albert [Alb61]. They come in two basic flavours: *classical* and *exchange*. The classical types are simple as algebras and arise as adjoints of nondegenerate reflexive forms. A simple $*$-algebra of exchange type is a direct sum of two isomorphic simple algebras where the involution interchanges the two factors. Naturally extending the MAGMA classification of reflexive forms (see the manual entry for the function `ClassicalForms`) a simple $*$-algebra S is assigned a `name` as follows:

`"symplectic"` if S is defined by an alternating form;

`"orthogonalcircle"` if S is defined by a symmetric form in odd dimension;

`"orthogonalplus"` if S is defined by a symmetric form of maximal Witt index;

`"orthogonalminus"` if S is defined by a symmetric form of non-maximal Witt index;

`"unitary"` if S is defined by an Hermitian form; and

`"exchange"` if S has exchange type.

Every simple $*$-algebra is isomorphic to a standard simple $*$-algebra having one of these six names defined naturally on a suitable vector space.

We note that involutions on simple $*$-algebras are often classified as being "of the first kind" or "of the second kind" according to whether or not they induce the identity on the center of the algebra. Thus, involutions of the second kind are unitary and exchange, and the others are all involutions of the first kind [KMRT98].

`SimpleStarAlgebra(name, d, K)`

Given a `name`, a positive integer d, and a field K, this function constructs the standard copy of the simple $*$-algebra of type `name` defined naturally on a K-vector space of dimension d.

Example H87E4

We construct the standard copy of the simple $*$-algebra of exchange type on the vector space of dimension 4 over GF(16).

```
> K := GF(16);
> S := SimpleStarAlgebra("exchange", 4, K);
> Dimension(S);
8
> IsStarAlgebra(S);
```

```
true;
> w := K.1;
> s := S.1 * S.2 + w * S.1;
> s;
[   K.1      1      0      0]
[     0      0      0      0]
[     0      0      0      0]
[     0      0      0      0]
> star := Star(S);
> s@star;
[     0      0      0      0]
[     0      0      0      0]
[     0      0    K.1      0]
[     0      0      1      0]
```

87.3 Decompositions of $*$-Algebras

Every finite-dimensional K-algebra A has a decomposition $A = J \oplus W$, where J is the Jacobson radical of A and W is a semisimple subring of A. We refer to such decompositions as *Wedderburn decompositions*. The procedure that computes Wedderburn decompositions is adapted from an analogous MAGMA function written by W. de Graaf for algebras defined by structure constants.

If A is a $*$-algebra and the characteristic of K is not 2, then it follows from a result of Taft [Taf57] that A has a Wedderburn decomposition of the form $A = J \oplus T$ in which T is invariant under the involution of A. We refer to such decompositions as *Taft decompositions*. The procedure that computes Taft decompositions is based on Taft's original proof, and is described in [BW11a, Proposition 4.3].

`WedderburnDecomposition(A)`

`WedderburnDecomposition(A)`

A Wedderburn decomposition is constructed for the $*$-algebra A. Specifically, the Jacobson radical, J, of A, and a semisimple complement, W, to J in A are computed. Here A may be either a matrix algebra or a group algebra over any field.

`TaftDecomposition(A)`

`TaftDecomposition(A)`

A Taft decomposition is constructed for the $*$-algebra A. Specifically, the Jacobson radical, J, of A and a $*$-invariant Wedderburn complement to J in A are computed. This function requires that the base ring of A has characteristic different from 2. Here A may be either a matrix $*$-algebra or a group algebra.

Example H87E5

We compute a Wedderburn decomposition of the group algebra GF(5)[A_5] equipped with its natural involution.

```
> K := GF(5);
> G := AlternatingGroup(5);
> A := GroupAlgebraAsStarAlgebra(K, G);
> J, W := WedderburnDecomposition(A);
```

We check dimensions and the $*$-invariance of T.

```
> Dimension(J); Dimension(W);
25
35
> forall { i : i in [1..Ngens (W)] | W.i@Star(A) in W };
false
```

Now find a $*$-invariant decomposition.

```
> J, T := TaftDecomposition(A);
> Dimension(J); Dimension(W);
25
35
> forall { i : i in [1..Ngens(T)] | T.i@Star(A) in T };
true
```

87.4 Recognition of $*$-Algebras

In this section we describe methods that facilitate structural examinations of $*$-algebras. *All of the functions in this section require that the base ring of the given algebra is a finite field of odd order.* The functions are implementations of the methods described in [BW11a, Sections 4.2 and 4.3].

87.4.1 Recognition of Simple $*$-Algebras

If A is a simple $*$-algebra, then we *constructively recognise* A by finding an explicit inverse isomorphisms between A and the standard copy of the simple $*$-algebra which is isomorphic to A. The latter is the output of the function `SimpleStarAlgebra` with the appropriate input parameters.

`RecogniseClassicalSSA(A)`

Given a matrix $*$-algebra A, this function first decides whether or not A is a simple $*$-algebra of classical type. If it is, the standard $*$-algebra, T, corresponding to A, a $*$-isomorphism from A to T, and its inverse from T to A are returned.

`RecogniseExchangeSSA(A)`

Given a matrix $*$-algebra A, this function first decides whether or not A is a simple $*$-algebra of exchange type. If it is, the standard $*$-algebra, T, corresponding to A, a $*$-isomorphism from A to T, and its inverse from T to A are returned.

Example H87E6

We build a particular simple $*$-algebra of symplectic type and recognise it constructively.

```
> MA := MatrixAlgebra(GF(7), 4);
> F := MA![0,1,3,4,6,0,0,1,4,0,0,2,3,6,5,0];
> F;
[0 1 3 4]
[6 0 0 1]
[4 0 0 2]
[3 6 5 0]
> A := AdjointAlgebra([F]);
> isit, T, f, g := RecogniseClassicalSSA(A);
> isit;
true;
```

A quick check that f is, as claimed, a $*$-isomorphism.

```
> (A.1 + A.2)@f eq (A.1@f) + (A.2@f);
true
> (A.1 * A.2)@f eq (A.1@f) * (A.2@f);
true
> (A.2@Star(A))@f eq (A.2@f)@Star(T);
true
```

87.4.2 Recognition of Arbitrary $*$-Algebras

If A is an arbitrary $*$-algebra, then we constructively recognise A as follows:

(i) Find a decomposition $A = J \oplus T$, where J is the Jacobson radical of A and T is a $*$-invariant semisimple complement to J in A;

(ii) Find a decomposition $T = I_1 \oplus \ldots \oplus I_t$ of T into minimal $*$-ideals; and

(iii) For each $j \in \{1, \ldots, t\}$ constructively recognise the simple $*$-algebra I_j.

`RecogniseStarAlgebra(A)`

`RecogniseStarAlgebra(A)`

Constructively recognise the $*$-algebra A given as a matrix $*$-algebra or a group algebra.

There are several functions available that permit easy access to structural information about a $*$-algebra that has been constructively recognised. (In fact all of these functions also initiate a constructive recognition of the input $*$-algebra if the recognition has not already been carried out.) For all of the access functions A can be either a matrix $*$-algebra or a group algebra.

`IsSimpleStarAlgebra(A)`

`IsSimpleStarAlgebra(A)`

Return `true` if and only if A is a simple $*$-algebra.

`SimpleParameters(A)`

`SimpleParameters(A)`

Given a $*$-algebra A, this function returns the parameters that determine (up to $*$-isomorphism) the minimal $*$-ideals of the semisimple quotient A/J, where J is the Jacobson radical of A. The parameters are returned in the form of a sequence.

`NormGroup(A)`

Given a $*$-algebra A, this function returns the group of unitary elements of A, namely the group consisting of all units in A satisfying the condition $x^* = x^{-1}$. The function is based on methods described in [BW11a, Section 5].

Example H87E7

Our first example illustrates how the $*$-algebra machinery may be used to distinguish between group algebras over GF(5) for the dihedral and quaternion groups of order 8. Those group algebras are isomorphic as algebras, but the example shows that they are nonisomorphic as $*$-algebras.

```
> K := GF(5);
> G1 := SmallGroup(8, 3);
> G2 := SmallGroup(8, 4);
> A1 := GroupAlgebraAsStarAlgebra(K, G1);
> A2 := GroupAlgebraAsStarAlgebra(K, G2);
> J1, T1 := TaftDecomposition(A1);
> J2, T2 := TaftDecomposition(A2);
> Dimension(J1); Dimension(J2);
0
0
```

Thus (as we know from Maschke's theorem) both GF(5)$[D_8]$ and GF(5)$[Q_8]$ are semisimple. We now recognise them as $*$-algebras and examine their minimal $*$-ideals.

```
> RecogniseStarAlgebra(A1);
true
> RecogniseStarAlgebra(A2);
true
> SimpleParameters(A1);
[ <"orthogonalcircle", 1, 5>, <"orthogonalcircle", 1, 5>,
<"orthogonalcircle", 1, 5>, <"orthogonalcircle", 1, 5>,
<"orthogonalplus", 2, 5> ]
> SimpleParameters(A2);
[ <"orthogonalcircle", 1, 5>, <"orthogonalcircle", 1, 5>,
<"orthogonalcircle", 1, 5>, <"orthogonalcircle", 1, 5>,
<"symplectic", 2, 5>
]
```

Both group algebras decompose into four 1-dimensional $*$-ideals, and one 4-dimensional $*$-ideal. However, the latter has type `"orthogonalplus"` for GF(5)$[D_8]$, but type `"symplectic"` for GF(5)$[Q_8]$.

Example H87E8

Our second example shows how to use $*$-algebra functions to distinguish between two p-groups of class 2 and order 43^6. The first group is a Sylow 43-subgroup of $\mathrm{GL}(3, 43^2)$.

```
> P1 := ClassicalSylow(GL(3, 43^2), 43);
> Forms1 := PGroupToForms(P1);
> A1 := AdjointAlgebra(Forms1);
> RecogniseStarAlgebra(A1);
true
> SimpleParameters(A1);
[ <"symplectic", 2, 1849> ]
```

The second group is constructed as a subgroup of $\mathrm{GL}(3, \mathrm{GF}(43)[x]/(x^2))$.

```
> R<x> := PolynomialRing(GF(43));
> S, f := quo< R | x^2 >;
> G := GL(3, S);
> Ua := G![1,1,0,0,1,0,0,0,1];
> Wa := G![1,0,0,0,1,1,0,0,1];
> Ub := G![1,x@f,0,0,1,0,0,0,1];
> Wb := G![1,0,0,0,1,x@f,0,0,1];
> P2 := sub< G | [ Ua, Wa, Ub, Wb ] >;
> Forms2 := PGroupToForms(P2);
> A2 := AdjointAlgebra(Forms2);
> RecogniseStarAlgebra(A2);
true
> SimpleParameters(A2);
[ <"symplectic", 2, 43> ]
```

Since A_1 and A_2 are non-isomorphic $*$-algebras, it follows that P_1 and P_2 are non-isomorphic groups.

87.5 Intersections of Classical Groups

The main application of the $*$-algebra machinery is to the study of the group preserving each form in a system of forms; the so-called *isometry group* of the system. An essentially equivalent (but perhaps more familiar) problem is that of computing the intersection of a set of classical groups defined on a common vector space. The main functions are implementations of the algorithms presented in [BW11a, Theorem 1.2] and [BW11b, Theorem 1.1].

`IsometryGroup(S : parameters)`

`Autos`	SeqEnum	*Default :* $[0, .., 0]$
`DisplayStructure`	BoolElt	*Default :* `false`

Given a sequence S containing a system of reflexive forms, this function returns the group of isometries of the system. In addition to allowing the individual forms to be degenerate, the function handles degenerate systems.

The field automorphisms associated to the individual forms are specified using the parameter `Autos`; the default is that all forms in the system are bilinear over their common base ring. As well as finding generators for the isometry group, the procedure determines the structure of this group. The parameter `DisplayStructure` may be used to display this structure.

Example H87E9

We compute the isometry group of the system of forms associated to a particular p-group.

```
> G := ClassicalSylow(Sp (4, 5^2), 5);
> S := PGroupToForms(G);
> Parent(S[1]);
Full Matrix Algebra of degree 6 over GF(5)
> I := IsometryGroup(S : DisplayStructure := true);
   G
   |   GL ( 1 , 5 ^ 1 )
   *
   |   5 ^ 4    (unipotent radical)
   1
> #I;
2500
```

`ClassicalIntersection(S)`

Given a sequence S containing a number of classical groups, each one of which preserves (up to similarity) a unique nondegenerate reflexive form on a common finite vector space V, this function returns the intersection of the groups. It is not required that a classical group G in S be the full group of isometries.

Example H87E10

In our final example we intersect two quasisimple classical groups. First we construct a symplectic group $\mathrm{Sp}(F_1)$ for a particular skew-symmetric matrix F_1.

```
> K := GF(3);
> M := UpperTriangularMatrix
>        (K,[0,2,1,0,1,2,1,1,1,2,0,0,1,2,1,0,1,0,1,2,2]);
> F1 := M - Transpose(M);
```

```
> G1 := IsometryGroup(F1);
```

First check that G1 is a group of isometries.

```
> forall{ g : g in Generators(G1) | g*F1*Transpose(g) eq F1 };
true
```

Next we construct a quasisimple orthogonal group $\Omega^-(F_2)$ for a particular symmetric matrix F_2.

```
> F2 := SymmetricMatrix
>         (K, [1,2,0,1,2,2,1,0,2,2,1,0,0,0,1,2,1,1,0,1,0]);
> C := TransformForm(F2, "orthogonalminus");
> G := OmegaMinus(6, 3);
> G2 := G^(C^-1);
```

First check that G2 is a group of isometries.

```
> forall { g : g in Generators(G2) | g*F2*Transpose(g) eq F2 };
true
```

Finally compute the intersection of G_1 and G_2 and ask for the order of this intersection group.

```
> I := ClassicalIntersection([G1, G2]);
> #I;
14
```

87.6 Bibliography

[**Alb61**] A. Adrian Albert. *Structure of Algebras*. American Mathematical Society, Providence, RI, 1961. Revised printing.

[**BW11a**] Peter A. Brooksbank and James B. Wilson. Computing isometry groups of Hermitian maps. *Transactions of the American Mathematical Society*, 2011. to appear.

[**BW11b**] Peter A. Brooksbank and James B. Wilson. Intersecting two classical groups. Preprint, 2011.

[**KMRT98**] Max-Albert Knus, Alexander Merkurjev, Markus Rost, and Jean-Pierre Tignol. *The Book of Involutions*. American Mathematical Society, Providence, RI, 1998. Preface by Jacques Tits.

[**Taf57**] E.J. Taft. Invariant Wedderburn factors. *Illinois Journal of Mathematics*, 1:565–573, 1957.

88 CLIFFORD ALGEBRAS

Chapter 88
CLIFFORD ALGEBRAS

88.1 Introduction

Given a quadratic form Q defined on a vector space V over a field F, the Clifford algebra of Q is an associative F-algebra C with a vector space homomorphism $f : V \to C$ such that $f(v)^2 = Q(v)$ for all $v \in V$. Furthermore, the triple (C, V, f) has the universal property that if A is any associative algebra with a homomorphism $g : V \to A$ such that $g(v)^2 = Q(v)$ for all $v \in V$, then there is a unique algebra homomorphism $h : C \to A$ such that $hf = g$. It can be shown that f is injective and therefore we may identify V with its image in C. If the dimension of V is n, then the dimension of C is 2^n.

The primary references for quadratic forms and Clifford algebras are [Che97] and [Art57].

88.2 Clifford Algebras and their Elements

Clifford algebras are represented in MAGMA as structure constant algebras and so many of the functions described in Chapter 79 apply to Clifford algebras. The MAGMA type of a Clifford algebra is `AlgClff` and all Clifford algebras have the attributes

space: the quadratic space from which the Clifford is derived;

embedding: the standard embedding of the quadratic space into the Clifford algebra;

mainInvolutionMatrix: the matrix of the antiautomorphism of the Clifford algebra that reverses the multiplication.

Let C be the Clifford algebra of the quadratic form Q defined on the vector space V. If e_1, e_2, ..., e_n is a basis for V, a basis for C is the set of all products $e_1^{i_1} e_2^{i_2} \cdots v_n^{i_n}$, where i_k is 0 or 1 for all k. The function $k \mapsto i_k$ is the characteristic function of a subset of $\{1, 2, \ldots, n\}$, namely $S = \{k \mid i_k eq 1\}$. The map $S \mapsto 1 + \sum_{k \in S} 2^{k-1}$ is a bijection between the subsets of $\{1, 2, \ldots, n\}$ and the integers in the interval $[1 \ldots 2^n]$.

Thus the elements of C can be represented by a sequence of pairs $\langle S, a \rangle$ where S is a subset of $\{1, 2, \ldots, n\}$ and a is a field element. Multiplication is determined by the fact that for all $u, v \in V$ we have

$$v^2 = Q(v) \cdot 1 \quad \text{and} \quad uv + vu = \beta(u, v) \cdot 1,$$

where β is the polar form of Q.

`CliffordAlgebra(Q)`

This function returns a triple C, V, f, where C is the Clifford algebra of the quadratic form Q, V is the quadratic space of Q, and f is the standard embedding of V into C.

`CliffordAlgebra(V)`

If V is a quadratic space with quadratic form Q, this function returns the pair C, f, where C is the Clifford algebra of Q and f is the standard embedding of V into C.

88.2.1 Elements of a Clifford Algebra

`elt< C | `$r_1, r_2, \ldots, r_m$` >`

Given a Clifford algebra C of dimension $m = 2^n$ over a field F, and field elements r_1, r_2, ..., $r_m \in F$ construct the element $r_1 * C.1 + r_2 * C.2 + \cdots + r_m * C.m$ of C.

`C ! L`

Given a Clifford algebra C of dimension $m = 2^n$ and a sequence $L = [r_1, r_2, \ldots, r_m]$ of elements of the base ring R of C, construct the element $r_1 * C.1 + r_2 * C.2 + \cdots + r_m * C.m$ of C.

`BasisProduct(A, i, j)`

Return the product of the i-th and j-th basis element of the Clifford algebra C.

`BasisElement(C, L)`

The basis element $C.j$ of the Clifford algebra C corresponding to the subset L of $\{1, 2, \ldots, n\}$ where $j = 1 + \sum_{k \in L} 2^{k-1}$. If e_1, e_2, ..., e_n is the standard basis for the vector space on which C is based, this corresponds to the product $e_{i_1} * e_{i_2} * \cdots * e_{i_h}$, where $L = \{i_1, i_2, \ldots, i_h\}$ and $i_1 < i_2 < \cdots < i_h$.

88.3 Bibliography

[**Art57**] E. Artin. *Geometric Algebra*. Interscience Publishers, New York, 1957.

[**Che97**] Claude Chevalley. *The algebraic theory of spinors and Clifford algebras*. Springer-Verlag, Berlin, 1997. Collected works. Vol. 2, Edited and with a foreword by Pierre Cartier and Catherine Chevalley, With a postface by J.-P. Bourguignon.

PART XII
REPRESENTATION THEORY

89 MODULES OVER AN ALGEBRA

Chapter 89
MODULES OVER AN ALGEBRA

89.1 Introduction

Let A be an algebra over a field K and let M be a vector space over K. We say that M is a *(right) A-module* if for each $a \in A$ and $m \in M$, a product $ma \in M$ is defined such that

$$(m+n)a = ma + na, m(a+b) = ma + mb,$$
$$m(ab) = (ma)b, m1 = m,$$
$$m(ka) = (ma)k = (mk)a,$$

for all $a, b \in A, m, n \in M, k \in K$.

Recall that a *representation* of an algebra A over a field K is an algebra homomorphism of A into $\mathrm{Hom}_K(M, M)$, for some K-module M. Taking M to be an A-module, and defining a mapping $\rho : M \to M$ by

$$\rho(m) := ma \quad (a \in A, m \in M)$$

then it is an easy exercise to show that ρ is a representation of A. A *matrix representation of degree n* of the algebra A is an algebra homomorphism of A into $M_n(K)$, the complete matrix algebra of degree n over K. Suppose M has finite K-dimension n and choose a basis for M. If, for each $a \in A$, we associate the matrix corresponding to the action of a on the basis elements, we obtain a matrix representation of A. Thus, each A-module of finite K-dimension *affords* a matrix representation of the algebra A. An important special case occurs when $A = K[G]$, the group algebra of a group G. In this case the theory of A-modules coincides with the theory of group representations.

Throughout this chapter we shall use the term *A-module* when referring to modules as defined above. For Magma V2.19, A-modules are restricted to one of the following cases:

(i) A matrix algebra A defined over a field.

(ii) A matrix algebra A defined over a field corresponding to a representation of a group G of finite degree. In this case the user is (implicitly) computing with the group algebra $K[G]$.

(iii) A matrix algebra A defined over an Euclidean Domain R. However, as currently the action of A may be used only in the construction of submodules, discussion will be limited to the case in which the coefficient ring is a field.

Note that in all of the above cases A has finite K-dimension.

Magma provides a range of facilities for defining and computing with A-modules. In particular, extensive machinery is provided for creating $K[G]$-modules which is described in the following chapter. It should be noted however, that many advanced functions only apply when A is an algebra over a finite field. Since the R-module of n-tuples, $R^{(n)}$, underlies an A-module, the operations for $R^{(n)}$ are also applicable.

89.2 Modules over a Matrix Algebra

This section describes function dealing with modules over a matrix algebra, for which there are the most number of operations available.

89.2.1 Construction of an A-Module

89.2.1.1 General Constructions

`RModule(A)`

Given a subalgebra A of $M_n(K)$, create the right A-module M with underlying vector space $K^{(n)}$, where the action of $a \in A$ is given by $m * a$, $m \in M$.

`RModule(Q)`

Given the subalgebra A of $M_n(K)$ generated by the terms of the sequence Q, create the right A-module M with underlying vector space $K^{(n)}$, where the action of $a \in A$ is given by $m * a$, $m \in M$.

Example H89E1

We construct the 6-dimensional module over $\mathbf{F}_2$ with an action given by the matrices

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```
> A := MatrixAlgebra<GF(2), 6 |
>   [ 1,0,0,1,0,1,
>     0,1,0,0,1,1,
>     0,1,1,1,1,0,
>     0,0,0,1,1,0,
>     0,0,0,1,0,1,
>     0,1,0,1,0,0 ],
>   [ 0,1,1,0,1,0,
>     0,0,1,1,1,1,
>     1,0,0,1,0,1,
>     0,0,0,1,0,0,
>     0,0,0,0,1,0,
>     0,0,0,0,0,1 ] >;
> M := RModule(A);
> M;
RModule M of dimension 6 over GF(2)
```

89.2.1.2 Constructions for $K[G]$-Modules

Although $K[G]$-modules are discussed in the next chapter, it is convenient to use them as examples in this chapter and so we give two basic constructions here. The reader is referred to the $K[G]$-module chapter for many other techniques for constructing these modules.

`GModule(G, Q)`

`Check` BoolElt *Default :* `true`

Let G be a group defined on r generators and let Q be a sequence of r invertible elements of $M_n(K)$ or $GL(n, K)$. It is assumed that the mapping from G to Q defined by $\phi(G.i) \mapsto Q[i]$, for $i = 1, \ldots, r$, is a group homomorphism from G into $GL(n, K)$. The function constructs a $K[G]$-module M of dimension n, where the action of the generators of G is given by the terms of Q.

`PermutationModule(G, K)`

Given a permutation group G and a field K, create the natural permutation module for G over K.

89.2.2 Accessing Module Information

This section deals with the underlying vector space of a module M, which is a module over the algebra A.

89.2.2.1 The Underlying Vector Space

`M . i`

Given an A-module M and a positive integer i, return the i-th generator of M.

`CoefficientRing(M)`

`BaseRing(M)`

Given an A-module M, where A is an algebra over the field K, return K.

`Generators(M)`

The generators for the A-module M, returned as a set.

`Parent(u)`

Given an element u belonging to the A-module M, return M.

89.2.2.2 The Algebra

`Action(M)`

`RightAction(M)`

Given an A-module M, return the matrix algebra A giving the action of A on M.

`MatrixGroup(M)`

`Check` BoolElt *Default :* `true`

Given an $R[G]$-module M, return the matrix group whose generators are the (invertible) generators of the acting algebra of M.

`ActionGenerator(M, i)`

The i-th generator of the (right) acting matrix algebra for the module M.

`NumberOfActionGenerators(M)`

`Ngens(M)`

The number of action generators (the number of generators of the algebra) for the A-module M.

`Group(M)`

Given an $R[G]$-module M, return the group G.

Example H89E2

We illustrate the use of several of these access functions by applying them to the 6-dimensional representation of a matrix algebra defined over $\mathbf{F}_2$.

```
> F2 := GF(2);
> F := MatrixAlgebra(F2, 6);
> A := sub< F |
>   [ 1,0,0,1,0,1,
>     0,1,0,0,1,1,
>     0,1,1,1,1,0,
>     0,0,0,1,1,0,
>     0,0,0,1,0,1,
>     0,1,0,1,0,0 ],
>   [ 0,1,1,0,1,0,
>     0,0,1,1,1,1,
>     1,0,0,1,0,1,
>     0,0,0,1,0,0,
>     0,0,0,0,1,0,
>     0,0,0,0,0,1 ] >;
> T := RModule(F2, 6);
> M := RModule(T, A);
> Dimension(M);
6
```

```
> BaseRing(M);
Finite field of size 2
```

We set R to be the name of the matrix ring associated with M. Using the generator subscript notation, we can access the matrices giving the (right) action of A.

```
> R := RightAction(M);
> R.1;
[1 0 0 1 0 1]
[0 1 0 0 1 1]
[0 1 1 1 1 0]
[0 0 0 1 1 0]
[0 0 0 1 0 1]
[0 1 0 1 0 0]
> R.2;
[0 1 1 0 1 0]
[0 0 1 1 1 1]
[1 0 0 1 0 1]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
```

We display full details of the module.

```
> M: Maximal;
Module M of dimension 6 with base ring GF(2)
Generators of acting algebra:

[1 0 0 1 0 1]
[0 1 0 0 1 1]
[0 1 1 1 1 0]
[0 0 0 1 1 0]
[0 0 0 1 0 1]
[0 1 0 1 0 0]

[0 1 1 0 1 0]
[0 0 1 1 1 1]
[1 0 0 1 0 1]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
```

89.2.3 Standard Constructions

Given one or more existing modules, various standard constructions are available to construct new modules.

89.2.3.1 Changing the Coefficient Ring

`ChangeRing(M, S)`

Given an A-module M with base ring R, together with a ring S, such that there is a natural homomorphism from R to S, construct the module N with base ring S where N is obtained from M by coercing the components of the vectors of M into N. The corresponding homomorphism from M to N is returned as a second value.

`ChangeRing(M, S, f)`

Given a module M with base ring R, together with a ring S, and a homomorphism $f : R \to S$, construct the module N with base ring S, where N is obtained from M by applying f to the components of the vectors of M. The corresponding homomorphism from M to N is returned as a second value.

89.2.3.2 Direct Sum

`DirectSum(M, N)`

Given R-modules M and N, construct the direct sum D of M and N as an R-module. The embedding maps from M into D and from N into D respectively and the projection maps from D onto M and from D onto N respectively are also returned.

`DirectSum(Q)`

Given a sequence Q of R-modules, construct the direct sum D of these modules. The embedding maps from each of the elements of Q into D and the projection maps from D onto each of the elements of Q are also returned.

89.2.3.3 Changing Basis

`M ^ T`

Given a $K[G]$-module M of dimension n over the field K, and a nonsingular $n \times n$ matrix T over K, construct the $K[G]$-module N which corresponds to taking the rows of T as a basis for M.

89.2.4 Element Construction and Operations

89.2.4.1 Construction of Module Elements

`elt< M | a`$_1$`, ..., a`$_n$ `>`

Given a module M with underlying vector space $K^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to K, construct the element $m = (a_1, \ldots, a_n)$ of M. Note that if m is not an element of M, an error will result.

`M ! Q`

Given the module M with underlying vector space K^n, and a sequence $Q = [a_1, \ldots, a_n]$ with universe K, construct the element $m = (a_1, \ldots, a_n)$ of M. Note that if m is not an element of M, an error will result.

`Zero(M)`

`M ! 0`

The zero element for the A-module M.

`Random(M)`

Given a module M defined over a finite ring or field, return a random vector.

89.2.4.2 Deconstruction of Module Elements

`ElementToSequence(u)`

`Eltseq(u)`

Given an element u belonging to the A-module M, return u in the form of a sequence Q of elements of K.

89.2.4.3 Action of the Algebra on the Module

`u * a`

Given a vector u belonging to an A-module M, and an element $a \in A$ return the image of u under the action of a.

`u * g`

Given a vector u belonging to an $K[G]$-module M, and an element g belonging to the group G, return the image of u under the action of $K[G]$ on the module M.

89.2.4.4 Arithmetic with Module Elements

`u + v`

Sum of the elements u and v, where u and v lie in the same A-module M.

`-u`

Additive inverse of the element u.

`u - v`

Difference of the elements u and v, where u and v lie in the same A-module M.

`k * u`

Given an element u in an A-module M, where A is a K-algebra and an element $k \in K$, return the scalar product $k * u$ as an element of M.

`u * k`

Given an element u in an A-module M, where A is a K-algebra and an element $k \in K$, return the scalar product $u * k$ as an element of M.

`u / k`

Given an element u in an A-module M, where A is a K-algebra and a non-zero element $k \in K$, return the scalar product $u * (1/k)$ as an element of M.

89.2.4.5 Indexing

`u[i]`

Given an element u belonging to a submodule M of the R-module $R^{(n)}$ and a positive integer i, $1 \leq i \leq n$, return the i-th component of u (as an element of the ring R).

`u[i] := x`

Given an element u belonging to a submodule M of the R-module $T = R^{(n)}$, a positive integer i, $1 \leq i \leq n$, and an element x of the ring R, redefine the i-th component of u to be x. The parent of u is changed to T (since the modified element u need not lie in M).

89.2.4.6 Properties of Module Elements

`IsZero(u)`

Returns `true` if the element u of the A-module M is the zero element.

`Support(u)`

A set of integers giving the positions of the non-zero components of the vector u.

89.2.5 Submodules

89.2.5.1 Construction

`sub< M | L >`

Given an A-module M, construct the submodule N generated by the elements of M specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of R defining an element of M;

(b) A set or sequence whose terms are elements of M;

(c) A submodule of M;

(d) A set or sequence whose terms are submodules of M.

The generators stored for N consist of the elements specified by terms L_i together with the stored generators for submodules specified by terms of L_i. Repetitions of an element and occurrences of the zero element are removed (unless N is trivial).

The constructor returns the submodule N as an A-module together with the inclusion homomorphism $f : N \to M$.

`ImageWithBasis(X, M)`

`Check` BoolElt *Default :* `true`

Given a basis matrix X for a A-submodule of the A-module M, return the submodule N of M such that the morphism of N into M is X.

`Morphism(M, N)`

If the A-module M was created as a submodule of the module N, return the inclusion homomorphism $\phi : M \to N$ as an element of $\mathrm{Hom}_A(M, N)$. Thus, ϕ gives the correspondence between elements of M (represented with respect to the standard basis of M) and elements for N.

Example H89E3

We construct a submodule of the permutation module for $L(3, 4)$ in its representation of degree 21.

```
> G := PSL(3, 4);
> M := PermutationModule(G, GF(2));
> x := M![0,0,0,1,0,1,0,0,0,1,1,0,0,0,1,0,1,1,0,0,1];
> N := sub< M | x >;
> N:Maximal;
GModule N of dimension 9 over GF(2)
Generators of acting algebra:

[1 0 0 0 1 0 1 0 1]
[0 1 0 1 1 1 0 0 0]
[0 0 1 1 1 1 1 0 1]
[0 0 0 0 0 1 1 0 0]
[0 0 0 1 0 0 1 0 0]
[0 0 0 0 1 0 1 0 0]
[0 0 0 1 1 1 0 0 0]
[0 0 0 0 1 1 0 0 1]
[0 0 0 1 0 1 0 1 1]

[0 0 0 0 0 1 0 1 1]
[1 0 0 0 0 0 0 0 1]
[0 1 1 0 0 1 0 0 1]
[0 0 0 0 0 1 0 0 0]
[0 0 1 0 0 1 0 0 0]
[0 0 1 0 1 1 0 0 1]
[0 0 1 1 0 0 0 0 1]
[0 0 1 0 0 0 0 0 1]
[0 0 0 0 0 0 1 0 0]
```

Note that as a $\mathbf{F}_2$-module V has dimension 1, while as a $K[G]$-module it has dimension 9. The submodule N is defined on a reduced basis so we use `Morphism` to see N embedded in M.

```
> phi := Morphism(N, M);
```

```
> [ phi(x) : x in Basis(N) ];
[
    M: (1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 1),
    M: (0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 1 1),
    M: (0 0 1 0 0 0 0 0 0 1 0 1 1 0 1 0 1 1 0 1 0),
    M: (0 0 0 1 0 0 0 0 0 0 1 0 1 1 0 1 0 1 1 0 1),
    M: (0 0 0 0 1 0 0 0 0 1 1 0 1 1 1 0 0 0 0 1 1),
    M: (0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 1 1 0 1 0 0),
    M: (0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 1 1 0 1 0),
    M: (0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 1 1 0 1),
    M: (0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 0 1 0 1 1)
]
```

89.2.5.2 Membership and Equality

The operators described below refer to the underlying vector space.

`u in M`

Returns `true` if the element u lies in the A-module M.

`N subset M`

Returns `true` if the A-module N is contained in the A-module M.

`N eq M`

Returns `true` if the A-modules N and M are equal, where N and M are contained in a common A-module.

89.2.5.3 Operations on Submodules

`M + N`

Sum of the submodules M and N, where M and N belong to a a common A-module.

`M meet N`

Intersection of the submodules M and N, where M and N belong to a common A-module.

89.2.6 Quotient Modules

`quo< M | L >`

Given an A-module M, construct the quotient module $P = M/N$ as an A-module, where N is the submodule generated by the elements of M specified by the list L. Each term L_i of the list L must be an expression defining an object of one of the following types:

(a) A sequence of n elements of R defining an element of M;

(b) A set or sequence whose terms are elements of M;

(c) A submodule of M;

(d) A set or sequence whose terms are submodules of M.

The generators constructed for N consist of the elements specified by terms L_i together with the stored generators for submodules specified by terms of L_i. The constructor returns the quotient module P as an A-module together with the natural homomorphism $f : M \to P$.

`Morphism(M, N)`

If the A-module N was created as a quotient module of the module M, return the natural homomorphism $\phi : M \to N$ as an element of $\mathrm{Hom}_R(M, N)$. Thus ϕ gives the correspondence between elements of M and elements of N (represented with respect to the standard basis for N).

Example H89E4

We construct a quotient module of the permutation module for $L(3, 4)$ considered above.

```
> G := PSL(3, 4);
> M := PermutationModule(G, GF(2));
> x := M![0,0,0,1,0,1,0,0,0,1,1,0,0,0,1,0,1,1,0,0,1];
> N := sub< M | x >;
> N;
GModule N of dimension 9 over GF(2)
> Q, phi := quo< M | x >;
> Q;
GModule Q of dimension 12 over GF(2)
```

We locate the kernel of the epimorphism ϕ and check that it is the same as N.

```
> K :=  Kernel(phi);
GModule Ker of dimension 9 over GF(2)
> K eq N;
true
```

Given an element x in the codomain Q of the epimorphism ϕ, the value returned as the preimage of x is a representative element of the coset of the kernel that is the actual preimage of x. Since

we are working in a module over a finite field, we can explicitly construct the full preimage `PreIm` of x.

```
> x := Q![0,0,0,1,1,0,0,0,0,1,0,0];
> PreIm := { x@@phi + k : k in K };
> #PreIm;
512
```

89.2.7 Structure of a Module

Most of the functions described in this section assume that the base ring is a **finite field**.

89.2.7.1 Reducibility

`Meataxe(M)`

Given an A-module M with base ring a finite field attempt to find a proper submodule N of M or else prove that M is irreducible. If a splitting of M is found, three values are returned:

(a) An A-module N corresponding to the induced action of A on S;

(b) An A-module P corresponding to the induced action of A on the quotient space M/N;

(c) Let ρ, ν and π denote the representations of A afforded by modules M, N and P, respectively. The third value returned is an invertible matrix T which conjugates the matrices of $\rho(A)$ into reduced form. Specifically, if $a \in A$, then

$$T * \rho(a) * T^{-1} = \begin{pmatrix} \nu(a) & 0 \\ * & \pi(a) \end{pmatrix}$$

If M is proved to be irreducible, the function simply returns M. The fact that M is irreducible is recorded as part of the data structure for M.

`IsIrreducible(M)`

Returns `true` if and only if the A-module M is irreducible. If M is reducible, a proper submodule S of M together with the corresponding quotient module $Q = M/S$, are also returned.

`IsAbsolutelyIrreducible(M)`

Returns `true` if and only if the A-module M is absolutely irreducible. Return also a matrix algebra generator for the endomorphism algebra E of M (a field), as well as the dimension of E.

`AbsolutelyIrreducibleModule(M)`

Let A be an algebra over a field K. Given an irreducible A-module M that is not absolutely irreducible over K, return an irreducible module N that is a constituent of the module M considered as a module over the splitting field for M. Note that the module N, while not unique, is absolutely irreducible.

Example H89E5

Consider the group $O_5(3)$ given as a permutation group of degree 45. We construct the permutation module, and we apply the Meataxe manually to find an irreducible constituent.

```
> SetSeed(3);
> O53 := PermutationGroup<45 |
>     (2,3)(4,6)(7,9)(8,11)(12,16)(13,14)(15,19)(18,22)(20,25)(21,26)(27,33)
>       (28,35) (29,34)(31,38)(36,43)(39,41),
>     (1,2,4,7,10,14,16,3,5)(6,8,12,17,21,27,34,41,44)(9,13,18,23,29,37,33,40,43)
>       (11,15,20)(19,24,30,25,31,22,28,36,38)(26,32,39)(35,42,45)>;
>
> P := PermutationModule(O53, GF(2));
> A, B := Meataxe(P); A; B;
GModule A of dimension 20 over GF(2)
GModule B of dimension 25 over GF(2)
> A, B := Meataxe(A); A; B;
GModule A of dimension 14 over GF(2)
GModule B of dimension 6 over GF(2)
> IsIrreducible(A);
true
```

`MinimalField(M)`

Let A be an algebra over a finite field K. Given an A-module M over K, return the smallest subfield of K, over which M can be realised.

`IsPermutationModule(M)`

Returns `true` if and only if the generators of the matrix algebra A are permutation matrices, for a given A-module M.

89.2.7.2 Composition Series

`CompositionSeries(M)`

Given an A-module M, construct a composition series by repeatedly applying the meataxe. The function returns three values:

(a) The composition series as a sequence of A-modules;

(b) The composition factors as a sequence of A-modules in the order determined by the composition series (a);

(c) A transformation matrix t such that for each $a \in A$, $t * a * t^{-1}$ is in reduced form.

`CompositionFactors(M)`

Given an A-module M, construct the composition factors by repeatedly applying the meataxe. The composition factors are returned in the form of a sequence of R-modules in the order determined by a composition series for M. If M is irreducible, the function returns a sequence containing M alone.

`Constituents(M)`

Given an A-module M, construct the constituents C of M, i.e., a sequence of representatives for the isomorphism classes of composition factors of M. A sequence I of indices is also returned, so that that i-th element of C is the $I[i]$-th composition factor of M.

`ConstituentsWithMultiplicities(M)`

Given an A-module M, return the constituents of M, together with their multiplicities. A sequence I of indices is also returned, so that that i-th element of C is the $I[i]$-th composition factor of M.

Example H89E6

We continue with the $O_5(3)$ example from the previous section. We notice that the constituent of dimension of 8 is not absolutely irreducible, so we lift it to over an extension field.

```
> O53 := PermutationGroup<45 |
>     (2,3)(4,6)(7,9)(8,11)(12,16)(13,14)(15,19)(18,22)(20,25)(21,26)(27,33)
>       (28,35) (29,34)(31,38)(36,43)(39,41),
>     (1,2,4,7,10,14,16,3,5)(6,8,12,17,21,27,34,41,44)(9,13,18,23,29,37,33,40,43)
>       (11,15,20)(19,24,30,25,31,22,28,36,38)(26,32,39)(35,42,45)>;
>
> P := PermutationModule(O53, GaloisField(2));
> Constituents(P);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 8 over GF(2),
    GModule of dimension 14 over GF(2)
]
> ConstituentsWithMultiplicities(P);
[
    <GModule of dimension 1 over GF(2), 3>,
    <GModule of dimension 6 over GF(2), 1>,
    <GModule of dimension 8 over GF(2), 1>,
    <GModule of dimension 14 over GF(2), 2>
]
> S, F := CompositionSeries(P);
> S, F;
[
    GModule of dimension 14 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 21 over GF(2),
    GModule of dimension 29 over GF(2),
    GModule of dimension 30 over GF(2),
    GModule of dimension 31 over GF(2),
    GModule P of dimension 45 over GF(2)
]
```

```
[
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 8 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2)
]
> IndecomposableSummands(P);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 44 over GF(2)
]
> C := Constituents(P);
> C;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 8 over GF(2),
    GModule of dimension 14 over GF(2)
]
> [IsAbsolutelyIrreducible(M): M in C];
[ true, true, false, true ]
> DimensionOfEndomorphismRing(C[3]);
2
> L := GF(2^2);
> E := ChangeRing(C[3], L);
> E;
GModule E of dimension 8 over GF(2^2)
> CE := CompositionFactors(E);
> CE;
[
    GModule of dimension 4 over GF(2^2),
    GModule of dimension 4 over GF(2^2)
]
> IsAbsolutelyIrreducible(CE[1]);
true
> IsIsomorphic(CE[1], CE[2]);
false
```

89.2.7.3 Socle Series

`IsSemisimple(M)`

Given an A-module M, which is a $K[G]$-module (`ModGrp`) for a field K, return whether M is semisimple.

If M is a semisimple module defined over a matrix algebra, the function returns as second return value a list of the ranks of the primitive idempotents of the algebra. This is also a list of the multiplicities of the simple modules of the algebra as composition factors in a composition series for the module.

`MaximalSubmodules(M)`

Given an A-module M, return a sequence containing the maximal submodules of M.

`Limit` RngIntElt *Default :* 0

If a limit L is provided, only up L submodules are calculated, and the second return value indicates whether all of the submodules are returned.

`JacobsonRadical(M)`

The Jacobson radical of the A-module M.

`MinimalSubmodules(M)`

Given an A-module M, return a sequence containing the minimal submodules of M.

`Limit` RngIntElt *Default :* 0

If a limit L is provided, only up L submodules are calculated, and the second return value indicates whether all of the submodules are returned.

`MinimalSubmodules(M, F)`

Given an A-module M and an irreducible module F, return a sequence containing those minimal submodules of M, each of which is isomorphic to F.

`Limit` RngIntElt *Default :* 0

If a limit L is provided, only up L submodules are calculated, and the second return value indicates whether all of the submodules are returned.

`MinimalSubmodule(M)`

Given an A-module M, return a single minimal (or irreducible) submodule of M; if M is itself irreducible, M is returned.

`Socle(M)`

Given a A-module M, return its socle, i.e. the sum of the minimal submodules of M.

`SocleSeries(M)`

A socle series S for the A-module M, together with the socle factors corresponding to the terms of S and a matrix T giving the transformation of M into (semi-simple) reduced form. The socle series, as returned, does not include the trivial module but does include M.

`SocleFactors(M)`

The factors corresponding to the terms of a socle series for the A-module M. The factors are returned in the form of a sequence of A-modules in the order determined by a socle series for M. If M is irreducible, the function returns a sequence containing M alone.

Example H89E7

We continue with the $O_5(3)$ example from the previous section.

```
> O53 := PermutationGroup<45 |
>   (2,3)(4,6)(7,9)(8,11)(12,16)(13,14)(15,19)(18,22)(20,25)(21,26)(27,33)
>     (28,35) (29,34)(31,38)(36,43)(39,41),
>   (1,2,4,7,10,14,16,3,5)(6,8,12,17,21,27,34,41,44)(9,13,18,23,29,37,33,40,43)
>     (11,15,20)(19,24,30,25,31,22,28,36,38)(26,32,39)(35,42,45)>;
>
> P := PermutationModule(O53, FiniteField(2));
> MaximalSubmodules(P);
[
    GModule of dimension 31 over GF(2),
    GModule of dimension 44 over GF(2)
]
> JacobsonRadical(P);
GModule of dimension 30 over GF(2)
> MinimalSubmodules(P);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2)
]
> Soc := Socle(P);
> Soc;
GModule Soc of dimension 15 over GF(2)
> SocleSeries(P);
[
    GModule of dimension 15 over GF(2),
    GModule of dimension 22 over GF(2),
    GModule of dimension 30 over GF(2),
    GModule of dimension 31 over GF(2),
    GModule P of dimension 45 over GF(2)
]
> SocleFactors(P);
[
```

```
    GModule of dimension 15 over GF(2),
    GModule of dimension 7 over GF(2),
    GModule of dimension 8 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2)
]
```

89.2.8 Decomposability and Complements

The functions in this section currently apply only in the case in which A is an algebra over a finite field.

`IsDecomposable(M)`

Given an A-module M, return `true` iff M is decomposable. If M is decomposable and defined over a finite field, the function also returns proper submodules S and T of M such that $M = S \oplus T$.

`DirectSumDecomposition(M)`

`IndecomposableSummands(M)`

`Decomposition(M)`

Given an A-module M, return a sequence Q of indecomposable summands of M. Each element of Q is an indecomposable submodule of M and M is equal to the (direct) sum of the terms of Q. If M is indecomposable, the sequence Q consists of M alone.

`HasComplement(M, S)`

`IsDirectSummand(M, S)`

Given an A-module M and a submodule S of M, determine whether S has a A-invariant complement in M. If this is the case, the value `true` is returned together with a submodule T of M such that $M = S \oplus T$; otherwise the value `false` is returned.

`Complements(M, S)`

Given an A-module M and a submodule S of M, return all A-invariant complements of S in M.

Example H89E8

```
> A := MatrixAlgebra<GF(2), 6 |
>   [ 1,0,0,1,0,1,
>     0,1,0,0,1,1,
>     0,1,1,1,1,0,
>     0,0,0,1,1,0,
>     0,0,0,1,0,1,
>     0,1,0,1,0,0 ],
>   [ 0,1,1,0,1,0,
>     0,0,1,1,1,1,
>     1,0,0,1,0,1,
>     0,0,0,1,0,0,
>     0,0,0,0,1,0,
>     0,0,0,0,0,1 ] >;
> M := RModule(RSpace(GF(2), 6), A);
> M;
RModule M of dimension 6 over GF(2)
> IsDecomposable(M);
false
> MM := DirectSum(M, M);
> MM;
RModule MM of dimension 12 over GF(2)
> l, S, T := IsDecomposable(MM);
> l;
true;
> S;
RModule S of dimension 6 over GF(2)
> HasComplement(MM, S);
true
> Complements(MM, S);
[
    RModule of dimension 6 over GF(2),
    RModule of dimension 6 over GF(2)
]
> IndecomposableSummands(MM);
[
    RModule of dimension 6 over GF(2),
    RModule of dimension 6 over GF(2)
]
> Q := IndecomposableSummands(MM);
> Q;
[
    RModule of dimension 6 over GF(2),
    RModule of dimension 6 over GF(2)
]
> Q[1] meet Q[2];
RModule of dimension 0 over GF(2)
```

```
> Q[1] + Q[2];
RModule MM of dimension 12 over GF(2)
```

89.2.9 Lattice of Submodules

Let M be an A-module. MAGMA can construct the lattice L of all submodules of M if this is not too large. Various properties of the lattice L may then be examined. The elements of L are called *submodule-lattice elements* and are numbered from 1 to n where n is the cardinality of L. Once the lattice has been constructed, the result of various lattice operations, such as meet and intersection, are available without the need for any module-theoretic calculation. Certain information about M and its submodules may then be obtained by analyzing L. Given an element of L, one can easily create the submodule N of M corresponding to it and one can also create the element of L corresponding to any submodule of M.

The functions is this section currently apply only in the case in which A is an algebra over a finite field.

89.2.9.1 Creating Lattices

`SubmoduleLattice(M)`

`Limit`	RNGINTELT	*Default :* 0
`CodimensionLimit`	RNGINTELT	*Default :* -1

Given an A-module M, construct the lattice L of submodules of M. If a limit n is provided, at most n submodules are calculated, and the second return value indicates whether the returned lattice L is the full lattice of submodules of M.

`SubmoduleLatticeAbort(M, n)`

Given an A-module M and a positive integer n, construct the lattice L of submodules of M, provided that the number of submodule does not exceed n. In this case the value `true` and the lattice L are returned. If M has more than n submodules, the function aborts and returns the value `false`.

`SetVerbose("SubmoduleLattice", i)`

Control verbose printing for the submodule lattice algorithm. The level i can be 2 for maximal printing or 1 for moderate printing. The algorithm works down a composition series of the module and a summary is printed for each level.

`Submodules(M)`

`CodimensionLimit`	RNGINTELT	*Default :* `Dimension(M)`

Given an A-module M, return a sequence containing all submodules of M sorted by dimension.

Example H89E9

We create the lattice of submodules for the A-module $\mathbf{F}_3[\mathbf{Z}_6]$ with level 1 verbose printing turned on.

```
> M := PermutationModule(CyclicGroup(6), GF(3));
> SetVerbose("SubmoduleLattice", 1);
> L := SubmoduleLattice(M);
Submodule Lattice; Dimension: 6, Composition length: 6
Starting level 4; Current number of modules: 2
Starting level 3; Current number of modules: 3
Starting level 2; Current number of modules: 6
Starting level 1; Current number of modules: 9
Starting level 0; Current number of modules: 12
Change basis time: 0.010
Jacobson radical time: 0.060
Complement time: 0.070
Total time: 0.250
> #L;
16
```

89.2.9.2 Operations on Lattices

In the following, L is the lattice of submodules for a module M.

`#L`

The cardinality of L, i.e. the number of submodules of M.

`L ! i`

Create the i-th element of the lattice L. The number i is insignificant (i.e. the elements of L are not numbered in any special way), but this allows one to uniquely identify each element of the lattice L.

`L ! S`

Create the element of the lattice L corresponding to the submodule S of M.

`Bottom(L)`

Create the bottom of the lattice L, i.e. the element of L corresponding to the zero-submodule of M. If the lattice was created with a limit on the number of submodules and the lattice is partial, the bottom of the lattice may not be the zero submodule.

`Random(L)`

Create a random element of L.

`Top(L)`

Create the top of the lattice L, i.e. the element of L corresponding to M.

89.2.9.3 Operations on Lattice Elements

In the following, L is the lattice of submodules for a module M. Elements of L are identified with the integers $[1..\#L]$ but not in any particular order.

`IntegerRing() ! e`

The integer corresponding to lattice element e.

`e + f`

The sum of lattice elements e and f, i.e. the lattice element corresponding to the sum of the modules corresponding to e and f.

`e meet f`

The intersection of lattice elements e and f.

`e eq f`

Returns `true` if and only if lattice elements e and f are equal.

`e subset f`

Returns `true` if and only if e is under f in the lattice L, i.e. the submodule corresponding to e is a submodule of the submodule corresponding to f.

`MaximalSubmodules(e)`

The maximal submodules of e, returned as a set of lattice elements.

`MinimalSupermodules(e)`

The minimal supermodules of e, returned as a set of lattice elements.

`Module(e)`

The submodule of M corresponding to the element e of the lattice L.

89.2.9.4 Properties of Lattice Elements

`Dimension(e)`

The dimension of the submodule of M corresponding to e.

`JacobsonRadical(e)`

The Jacobson radical of e, i.e. the lattice element corresponding to the Jacobson radical of the submodule corresponding to e.

`Morphism(e)`

The morphism from the module corresponding to e to M.

Example H89E10

We create the lattice of submodules for the A-module $\mathbf{F}_3[\mathbf{Z}_6]$.

```
> SetSeed(1);
> M := PermutationModule(CyclicGroup(6), GF(3));
> L := SubmoduleLattice(M);
> #L;
16
> T := Top(L);
> B := Bottom(L);
> T;
16
> B;
1
> // Check that element of L corresponding to M is T
> L ! M;
1
> (L ! M) eq T;
true
> // Check that module corresponding to B is zero-submodule of M
> Module(B);
GModule of dimension 0 with base ring GF(3)
```

We next find the minimal supermodules (immediate parents) of B in L and then determine the actual A-submodules to which they correspond.

```
> S := MinimalSupermodules(B);
> S;
{ 2, 3 }
> Module(L ! 2);
GModule of dimension 1 with base ring GF(3)
> Module(L ! 3);
GModule of dimension 1 with base ring GF(3)
> Dimension(L ! 2);
1
> Morphism(L ! 2);
[1 1 1 1 1 1]
> Morphism(L ! 3);
[1 2 1 2 1 2]
> // Set A to the sum of these elements
> A := L!2 + L!3;
> A;
5;
> // Note that A has dimension 2 and its morphism is the sum of the previous
> Dimension(A);
2
> Morphism(A);
[1 0 1 0 1 0]
[0 1 0 1 0 1]
```

```
> MaximalSubmodules(A);
{ 2, 3}
> S!2 subset A;
true
```

We now find the maximal submodules of L, and examine one, S, in detail.

```
> MaximalSubmodules(T);
{ 14, 15 }
> A := L ! 14;
> Dimension(A);
5
> Morphism(A);
[1 0 0 0 0 1]
[0 1 0 0 0 2]
[0 0 1 0 0 1]
[0 0 0 1 0 2]
[0 0 0 0 1 1]
> S := Module(A);
> S;
GModule S of dimension 5 with base ring GF(3)
```

Finally, we compute the Jacobson radical of S directly, and also obtain it from the lattice, checking that the two methods match.

```
> J := JacobsonRadical(S);
> J;
GModule J of dimension 3 with base ring GF(3)
> L ! J;
8
> JacobsonRadical(A);
8
```

89.2.10 Homomorphisms

Let M and N be A-modules where A is an algebra defined over a field K. Then $\mathrm{Hom}_A(M, N)$ consists of all K-homomorphisms from M to N which commute with the action of A. The type of such (matrix) homomorphisms, called *A-homs*, is `ModMatGrpElt`.

The functions is this section currently apply only in the case in which A is an algebra over a finite field.

89.2.10.1 Creating Homomorphisms

`hom< M -> N | X >`

Given A-modules M and N, create the (map) homomorphism from M to N given by matrix X.

`H ! f`

Given matrix space H, which is $\mathrm{Hom}_A(M, N)$ for A-modules M and N, together with a homomorphism f from M to N, create the matrix corresponding to the map f.

`IsModuleHomomorphism(X)`

Given a matrix X belonging to $\mathrm{Hom}_K(M, N)$, where M and N are A-modules, return `true` if X is an A-homomorphism.

89.2.10.2 $\mathrm{Hom}(M, N)$

`Hom(M, N)`

Given A-modules M and N, construct the vector space of homomorphisms, $\mathrm{Hom}_K(M, N)$, where K is the field over which A is defined.

`AHom(M, N)`

Given A-modules M and N, construct the vector space of homomorphisms, $\mathrm{Hom}_A(M, N)$, as a submodule of $\mathrm{Hom}_K(M, N)$.

`GHomOverCentralizingField(M, N)`

Given A-modules M and N, construct $\mathrm{Hom}_{L[G]}(M, N)$ as a subspace of $\mathrm{Hom}_K(M, N)$ where L is the centralizing field of M.

Example H89E11

We construct a 12-dimensional module M and a 9-dimensional submodule P of M for a soluble group of order 648 over $\mathbf{F}_3$. We then construct $H = \mathrm{Hom}_A(M, N)$ and perform map operations on elements of H.

```
> G := PermutationGroup< 12 |
>         (1,6,7)(2,5,8,3,4,9)(11,12),
>         (1,3)(4,9,12)(5,8,10,6,7,11) >;
> K := GF(3);
> P := PermutationModule(G, K);
> M := sub< P | [1,0,0,0,0,1,0,0,1,0,0,1] >;
> M;
GModule M of dimension 9 over GF(3)
> H := AHom(P, M);
> H: Maximal;
KMatrixSpace of 12 by 9 GHom matrices and dimension 2 over GF(3)
Echelonized basis:
```

```
[1 1 1 0 0 0 0 0 0]
[1 1 1 0 0 0 0 0 0]
[1 1 1 0 0 0 0 0 0]
[0 0 0 1 1 0 0 0 0]
[0 0 0 1 1 0 0 0 0]
[0 0 0 1 1 0 0 0 0]
[0 0 0 0 0 1 1 0 0]
[0 0 0 0 0 1 1 0 0]
[0 0 0 0 0 1 1 0 0]
[0 0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1 1]

[0 0 0 1 1 1 1 1 1]
[0 0 0 1 1 1 1 1 1]
[0 0 0 1 1 1 1 1 1]
[1 1 1 0 0 1 1 1 1]
[1 1 1 0 0 1 1 1 1]
[1 1 1 0 0 1 1 1 1]
[1 1 1 1 1 0 0 1 1]
[1 1 1 1 1 0 0 1 1]
[1 1 1 1 1 0 0 1 1]
[1 1 1 1 1 1 1 0 0]
[1 1 1 1 1 1 1 0 0]
[1 1 1 1 1 1 1 0 0]
> // We write down a random homomorphism from M to P.
> f := 2*H.1 + H.2;
> f;
[2 2 2 1 1 1 1 1 1]
[2 2 2 1 1 1 1 1 1]
[2 2 2 1 1 1 1 1 1]
[1 1 1 2 2 1 1 1 1]
[1 1 1 2 2 1 1 1 1]
[1 1 1 2 2 1 1 1 1]
[1 1 1 1 1 2 2 1 1]
[1 1 1 1 1 2 2 1 1]
[1 1 1 1 1 2 2 1 1]
[1 1 1 1 1 1 1 2 2]
[1 1 1 1 1 1 1 2 2]
[1 1 1 1 1 1 1 2 2]
> Ker := Kernel(f);
> Ker;
GModule Ker of dimension 8 with base ring GF(3)
```

If we print the morphism associated with Ker, we see generators for Ker as a submodule of P.

```
> Morphism(Ker, P);
[1 0 2 0 0 0 0 0 0 0 0 0]
[0 1 2 0 0 0 0 0 0 0 0 0]
```

```
[0 0 0 1 0 2 0 0 0 0 0 0]
[0 0 0 0 1 2 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 2 0 0 0]
[0 0 0 0 0 0 0 1 2 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 2]
[0 0 0 0 0 0 0 0 0 0 1 2]
> // Examine the image of f and its morphism to P.
> Im := Image(f);
> Im;
GModule Im of dimension 4 with base ring GF(3)
> Morphism(Im, P);
[1 1 1 0 0 0 0 0 0 0 0 0]
[0 0 0 1 1 1 0 0 0 0 0 0]
[0 0 0 0 0 0 1 1 1 0 0 0]
[0 0 0 0 0 0 0 0 0 1 1 1]
```

Example H89E12

We construct a G-homomorphism module H_1 for a G-module and then the homomorphism module $H = \mathrm{Hom}(H_1, H_1)$ with right matrix action which is equivalent to (the right representation of) the endomorphism module of H.

```
> P := GModule(CyclicGroup(11), GF(3));
> F := Constituents(P);
> F;
[
    GModule of dimension 1 over GF(3),
    GModule of dimension 5 over GF(3),
    GModule of dimension 5 over GF(3)
]
> H1 := GHom(P, F[2]);
> H1;
KMatrixSpace of 2 by 3 matrices and dimension 1 over Rational Field
> H := Hom(H1, H1, "right");
> H: Maximal;
KMatrixSpace of 5 by 5 matrices and dimension 5 over GF(3)
Echelonized basis:

[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]

[0 1 0 0 0]
[1 1 1 2 1]
[2 0 2 1 1]
[2 1 0 0 0]
```

```
[0 2 1 0 0]

[0 0 1 0 0]
[2 0 2 1 1]
[2 2 2 2 2]
[2 0 1 0 2]
[1 0 1 2 1]

[0 0 0 1 0]
[2 1 0 0 0]
[2 0 1 0 2]
[2 2 1 2 2]
[2 1 1 0 1]

[0 0 0 0 1]
[0 2 1 0 0]
[1 0 1 2 1]
[2 1 1 0 1]
[2 1 0 0 2]
```

89.2.10.3 Endo– and Automorphisms

`EndomorphismAlgebra(M)`

`EndomorphismRing(M)`

`Direct` BoolElt *Default :* `false`

Given a A-module M with base ring K, construct $E = \mathrm{End}_A(M)$ as a subring E of the complete matrix ring $K^{(n\times n)}$.

`CentreOfEndomorphismRing(M)`

Given a A-module M with base ring K, construct the centre of $\mathrm{End}_A(M)$ as a subring Z of the complete matrix ring $K^{(n\times n)}$. This is equivalent to `Centre(EndomorphismRing(M))` but will often be much faster.

`AutomorphismGroup(M)`

Given a A-module M with base ring K, construct $\mathrm{Aut}(M)$ as a subgroup G of the general linear group $GL(n, K)$. Thus, G is the group of units of $\mathrm{End}(M)$.

`IsIsomorphic(M, N)`

Returns `true` if the A-modules M and N are isomorphic, `false` otherwise. If M and N are isomorphic, the function also returns a matrix T such that $M^T = N$. Note that the action generators of M and N must match, so the function effectively determines whether there is an invertible matrix T such that `T^-1*ActionGenerator(M, i)*T` equals `ActionGenerator(N, i)` for each i.

Example H89E13

We construct the endomorphism ring for a permutation module over $\mathbf{F}_3$ for a soluble group of order 648.

```
> G := PermutationGroup< 12 |
>         (1,6,7)(2,5,8,3,4,9)(11,12),
>         (1,3)(4,9,12)(5,8,10,6,7,11) >;
> P := PermutationModule(G, GF(3));
> time End := EndomorphismAlgebra(P);
Time: 0.000
> End;
Matrix Algebra of degree 12 and dimension 3 over GF(3)
```

Thus, the permutation module P has 27 endomorphisms.

```
> time Aut := AutomorphismGroup(P);
Time: 0.010
> Aut;
MatrixGroup(12, GF(3))
Generators:
    [1 0 0 1 1 1 1 1 1 1 1 1]
    [0 1 0 1 1 1 1 1 1 1 1 1]
    [0 0 1 1 1 1 1 1 1 1 1 1]
    [1 1 1 1 0 0 1 1 1 1 1 1]
    [1 1 1 0 1 0 1 1 1 1 1 1]
    [1 1 1 0 0 1 1 1 1 1 1 1]
    [1 1 1 1 1 1 1 0 0 1 1 1]
    [1 1 1 1 1 1 0 1 0 1 1 1]
    [1 1 1 1 1 1 0 0 1 1 1 1]
    [1 1 1 1 1 1 1 1 1 1 0 0]
    [1 1 1 1 1 1 1 1 1 0 1 0]
    [1 1 1 1 1 1 1 1 1 0 0 1]

    [2 1 1 0 0 0 0 0 0 0 0 0]
    [1 2 1 0 0 0 0 0 0 0 0 0]
    [1 1 2 0 0 0 0 0 0 0 0 0]
    [0 0 0 2 1 1 0 0 0 0 0 0]
    [0 0 0 1 2 1 0 0 0 0 0 0]
    [0 0 0 1 1 2 0 0 0 0 0 0]
    [0 0 0 0 0 0 2 1 1 0 0 0]
    [0 0 0 0 0 0 1 2 1 0 0 0]
    [0 0 0 0 0 0 1 1 2 0 0 0]
    [0 0 0 0 0 0 0 0 0 2 1 1]
    [0 0 0 0 0 0 0 0 0 1 2 1]
    [0 0 0 0 0 0 0 0 0 1 1 2]

    [0 1 1 0 0 0 0 0 0 0 0 0]
    [1 0 1 0 0 0 0 0 0 0 0 0]
    [1 1 0 0 0 0 0 0 0 0 0 0]
```

```
    [0 0 0 0 1 1 0 0 0 0 0 0]
    [0 0 0 1 0 1 0 0 0 0 0 0]
    [0 0 0 1 1 0 0 0 0 0 0 0]
    [0 0 0 0 0 0 0 1 1 0 0 0]
    [0 0 0 0 0 0 1 0 1 0 0 0]
    [0 0 0 0 0 0 1 1 0 0 0 0]
    [0 0 0 0 0 0 0 0 0 0 1 1]
    [0 0 0 0 0 0 0 0 0 0 1 0 1]
    [0 0 0 0 0 0 0 0 0 0 1 1 0]
> #Aut;
18
> IsAbelian(Aut);
true
> AbelianInvariants(Aut);
[ 3, 6 ]
```

The module has 18 automorphisms. The automorphism group is isomorphic to the abelian group $\mathbf{Z}_2 \times \mathbf{Z}_3 \times \mathbf{Z}_3$.

89.3 Modules over a General Algebra

89.3.1 Introduction

This section describes the functionality for modules over general algebras in MAGMA. A left-module over an algebra A is a module M together with a bilinear map $A \times M \to M$. A right-module over A is a module M together with a bilinear map $M \times A \to M$. MAGMA provides functionality for both kinds of modules.

89.3.2 Construction of Algebra Modules

`Module(A, m)`

For an algebra A this function creates a module over A. If the module will be a left-module then m is a map from the Cartesian product $A \times M$ to M. If the module will be a right-module then m is a map from $M \times A$ to M. Here M has to be an R-module, where R is the coefficient field of A.

Example H89E14

We create the right-module over the full matrix algebra of 3×3 - matrices acting on its natural module.

```
> A:= MatrixAlgebra(Rationals(), 3);
> V:= RModule(Rationals(), 3);
> m:= map< CartesianProduct(V, A) -> V | t :-> t[1]*t[2] >;
> Module(A, m);
Right Module of Full Matrix Algebra of degree 3 over Rational Field
```

89.3.3 The Action of an Algebra Element

`a ^ v`

Given an element v of a left-module over an algebra A, and an element a of A computes the result of letting a act on v.

`v ^ a`

Given an element v of a right-module over an algebra A and an element a of A computes the result of letting a act on v.

`ActionMatrix(M, a)`

Given a module M over an algebra A and an element a of A returns the matrix of the action of a on M. If M is a left-module then the i-th column of this matrix contains the coordinates of the image of a acting on the i-th basis element of M. If A is a right-module then the rows contain these coordinates.

Example H89E15

```
> A:= MatrixAlgebra(Rationals(), 3);
> V:= RModule(Rationals(), 3);
> m:= map< CartesianProduct(V, A) -> V | t :-> t[1]*t[2] >;
> M:=Module(A, m);
> M.1^A.1;
M: (1 0 0)
> ActionMatrix(M, A.2);
[0 1 0]
[0 0 1]
[1 0 0]
```

89.3.4 Related Structures of an Algebra Module

`Algebra(M)`

This returns the algebra over which the algebra module M is defined.

`CoefficientRing(M)`

Returns the ground field of the algebra module M.

`Basis(M)`

Returns a sequence containing the basis vectors of the algebra module M.

89.3.5 Properties of an Algebra Module

`IsLeftModule(M)`

This returns `true` if the algebra module M is a left-module, and `false` if it is a right module.

`IsRightModule(M)`

This returns `true` if the algebra module M is a right-module, and `false` if it is a left module.

`Dimension(M)`

The dimension of the algebra module M.

89.3.6 Creation of Algebra Modules from other Algebra Modules

`DirectSum(Q)`

Given a sequence Q of algebra modules (all defined over the same algebra, and all left (respectively right) modules), returns the module M that is the direct sum of the modules in Q. Furthermore, two sequences of mappings are returned. The i-th element of the first sequence is the embedding of the i-th element of Q into M. The i-th element of the second sequence is the projection of M onto the i-th element of Q.

`SubalgebraModule(B, M)`

Given an algebra module M over the algebra A, and a subalgebra B of A, return M as a B-module.

`ModuleWithBasis(Q)`

Given a sequence Q containing the elements of a particular basis of an algebra module M, create an algebra module that is isomorphic to M, but with basis Q. (Or, more precisely, the basis vectors of the module V that is returned are in bijection with Q. The action of an algebra element on the i-th basis vector of V is computed by computing it on the i-th vector in Q and expressing the result as a linear combination of the elements of Q. The resulting coordinates are used to form the corresponding element of V.) This can be used to compute the action of algebra elements with respect to a given basis of M.

Example H89E16

```
> A:= MatrixAlgebra(Rationals(), 3);
> V:= RModule(Rationals(), 3);
> m:= map< CartesianProduct(V, A) -> V | t :-> t[1]*t[2] >;
> M:=Module(A, m);
> N:=DirectSum([ M, M ]);
> ActionMatrix(N, A.1);
[1 0 0 0 0 0]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
> W:= ModuleWithBasis([ M.1+M.2+M.3, M.2+M.3, M.3 ]);
> ActionMatrix(W, A.1);
[ 1 -1  0]
[ 0  0  0]
[ 0  0  0]
```

`sub< M | S >`

`sub< M | e1, ..., en >`

Return the submodule of M containing the elements in the sequence S or the elements $e1$, ..., en.

`quo< M | S >`

`quo< M | e1, ..., en >`

Construct the quotient module of M by the submodule S of M, the submodule containing the elements in the sequence S or the elements $e1, ..., en$.

90 $K[G]$-MODULES AND GROUP REPRESENTATIONS

Chapter 90
$K[G]$-MODULES AND GROUP REPRESENTATIONS

90.1 Introduction

A module over a group algebra, $K[G]$, where K is a field and G is a group, is an important special case of modules over an algebra. This case coincides with the theory of group representations. MAGMA provides extensive machinery for constructing $K[G]$-modules. It should be noted however, that some advanced functions apply only when K is a finite field.

In this chapter the machinery for constructions peculiar to $K[G]$-modules will be described. In addition, a number of operations that apply only to $K[G]$-modules are described. All of the operations for A-modules also apply to $K[G]$-modules and are not repeated in this chapter.

90.2 Construction of $K[G]$-Modules

The following functions provide for the construction of finite-dimensional $K[G]$-modules for a group G, where the action of G is given in terms of a matrix representation of G. Note that an Euclidean Domain may appear in place of the field K.

90.2.1 General $K[G]$-Modules

`GModule(G, A)`

Let G be a group defined on r generators and let A be a subalgebra of the matrix algebra $M_n(K)$, also defined by r non-singular matrices. It is assumed that the mapping from G to A defined by $\phi(G.i) \mapsto A.i$, for $i = 1, \ldots, r$, is a group homomorphism. Let M be an n-dimensional vector space over K. The function constructs a $K[G]$-module M of dimension n, where the action of the i-th generator of G on M is given by the i-th generator of A.

`GModule(G, Q)`

Let G be a group defined on r generators and let Q be a sequence of r invertible elements of $M_n(K)$ or $GL(n, K)$. It is assumed that the mapping from G to Q defined by $\phi(G.i) \mapsto Q[i]$, for $i = 1, \ldots, r$, is a group homomorphism from G into the matrix algebra A defined by the terms of Q. The function constructs a $K[G]$-module M of dimension n, where the action of G is defined by the matrix algebra A.

`TrivialModule(G, K)`

Create the trivial $K[G]$-module for the group G.

Example H90E1

We construct a 3-dimensional module for PSL(2, 7) over $\mathbf{F}_2$. The action of the group on M is described in terms of two elements, x and y, belonging to the ring of 3×3 matrices over $\mathbf{F}_2$.

```
> PSL27 := PermutationGroup< 8 | (2,3,5)(6,7,8), (1,2,4)(3,5,6) >;
> S := MatrixAlgebra< FiniteField(2), 3 |
>        [ 0,1,0,  1,1,1,  0,0,1 ], [ 1,1,1,  0,1,1,  0,1,0 ] >;
> M := GModule(PSL27, S);
> M: Maximal;
GModule M of dimension 3 with base ring GF(2)
Generators of acting algebra:

[0 1 0]
[1 1 1]
[0 0 1]

[1 1 1]
[0 1 1]
[0 1 0]
```

Example H90E2

We write a function which, given a matrix algebra A, together with a matrix group G acting on A by conjugation (so A is closed under action by G), computes the G-module M representing the action of G on A. We then construct a particular (nilpotent) upper-triangular matrix algebra A, a group $G = 1 + A$ which acts on A, and finally construct the appropriate G-module M.

```
> MakeMod := function(A, G)
>     // Make G-module M of G acting on A by conjugation
>     k := CoefficientRing(A);
>     d := Dimension(A);
>     S := RMatrixSpace(A, k);
>     return GModule(
>         G,
>         [
>             MatrixAlgebra(k, d) |
>             &cat[
>                 Coordinates(S, S.j^g): j in [1 .. d]
>             ] where g is G.i: i in [1 .. Ngens(G)]
>         ]
>     );
> end function;
>
> MakeGroup := function(A)
>     // Make group G from upper-triangular matrix algebra A
>     k := CoefficientRing(A);
>     n := Degree(A);
>     return MatrixGroup<n, k | [Eltseq(1 + A.i): i in [1 .. Ngens(A)]]>;
```

```
> end function;
>
> k := GF(3);
> n := 4;
> M := MatrixAlgebra(k, n);
> A := sub<M |
>      [0,2,1,1, 0,0,1,1, 0,0,0,1, 0,0,0,0],
>      [0,1,0,0, 0,0,2,2, 0,0,0,1, 0,0,0,0]>;
> G := MakeGroup(A);
> G;
MatrixGroup(4, GF(3)) of order 3^4
Generators:
    [1 2 1 1]
    [0 1 1 1]
    [0 0 1 1]
    [0 0 0 1]

    [1 1 0 0]
    [0 1 2 2]
    [0 0 1 1]
    [0 0 0 1]
> M := MakeMod(A, G);
> M: Maximal;
GModule M of dimension 5 over GF(3)
Generators of acting algebra:

[1 1 1 2 0]
[0 1 1 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]

[1 2 2 2 0]
[0 1 1 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

90.2.2 Natural $K[G]$-Modules

The following functions provide for the construction of $K[G]$-modules for a group G in one of its natural actions. Note that an Euclidean Domain may be used in place of the field K.

`GModule(G, K)`

Given a finite permutation group G and a ring K, create the natural permutation module for G over K.

GModule(G)

Given a matrix group G defined as a subgroup of the group of units of the ring $Mat_n(K)$, where K is a field, create the natural $K[G]$-module for G.

Example H90E3

Given the Mathieu group M_{11} presented as a group of 5×5 matrices over $\mathbf{F}_3$, we construct the natural $K[G]$-module associated with this representation.

```
> G := MatrixGroup<5, FiniteField(3) |
>      [ 2,1,2,1,2,  2,0,0,0,2,  0,2,0,0,0,  0,1,2,0,1,  1,0,2,2,1],
>      [ 2,1,0,2,1,  1,2,0,2,2,  1,1,2,1,1,  0,2,0,1,1,  1,1,2,2,2] >;
> Order(G);
7920
>
> M := GModule(G);
> M : Maximal;
GModule M of dimension 5 with base ring GF(3)
Generators of acting algebra:

[2 1 2 1 2]
[2 0 0 0 2]
[0 2 0 0 0]
[0 1 2 0 1]
[1 0 2 2 1]

[2 1 0 2 1]
[1 2 0 2 2]
[1 1 2 1 1]
[0 2 0 1 1]
[1 1 2 2 2]
```

90.2.3 Action on an Elementary Abelian Section

GModule(G, A, B)

GModule(G, A)

Given a group G, a normal subgroup A of G and a normal subgroup B of A such that the section A/B is elementary abelian of order p^n, create the $K[G]$-module M corresponding to the action of G on A/B, where K is the field $\mathbf{F}_p$. If B is trivial, it may be omitted. The function returns

(a) the module M; and

(b) the homomorphism $\phi : A/B \to M$.

Example H90E4

We construct a module M for the wreath product G of the alternating group of degree 4 with the cyclic group of degree 3. The module is given by the action of G on an elementary abelian normal subgroup H of order 64.

```
> G := WreathProduct(AlternatingGroup(4), CyclicGroup(3));
> G := PCGroup(G);
> A := pCore(G, 2);
> A;
GrpPC of order 64 = 2^6
Relations:
A.1^2 = Id(A),
A.2^2 = Id(A),
A.3^2 = Id(A),
A.4^2 = Id(A),
A.5^2 = Id(A),
A.6^2 = Id(A)
> M := GModule(G, A, sub<G|>);
> M;
GModule of dimension 6 with base ring GF(2)
```

90.2.4 Permutation Modules

The following functions provide for the construction of permutation modules for a group G. Note that an Euclidean Domain may be used in place of the field K.

`PermutationModule(G, H, K)`

Given a group G, a subgroup H of finite index in G and a field K, create the $K[G]$-module for G corresponding to the permutation action of G on the cosets of H.

`PermutationModule(G, K)`

Given a permutation group G and a field K, create the natural permutation module for G over K.

`PermutationModule(G, V)`

Given a permutation group G of degree n and an n-dimensional vector space V, create the natural permutation module for G over K.

`PermutationModule(G, u)`

Given a permutation group G of degree n, and a vector u belonging to the vector space $V = K^{(n)}$, construct the $K[G]$-module corresponding to the action of G on the K-subspace of V generated by the set of vectors obtained by applying the permutations of G to the vector u.

Example H90E5

We construct the permutation module for the Mathieu group M_{12} over the field $\mathbf{F}_2$.

```
> M12 := PermutationGroup<12 |
>       (1,2,3,4,5,6,7,8,9,10,11),
>       (1,12,5,2,9,4,3,7)(6,10,11,8) >;
> M := PermutationModule(M12, FiniteField(2));
> M : Maximal;
GModule M of dimension 12 with base ring GF(2)
Generators of acting algebra:

[0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0]
[1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1]

[0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0]
```

Example H90E6

We construct the constituent of the permutation module for the alternating group of degree 7 that contains the vector $(1, 0, 1, 0, 1, 0, 1)$.

```
> A7 := AlternatingGroup(7);
> V  := VectorSpace(FiniteField(2), 7);
> x  := V![1,0,1,0,1,0,1];
> M  := PermutationModule(A7, x);
> M : Maximal;
GModule of dimension 6 with base ring GF(2)
```

```
Generators of acting algebra:

[1 0 1 0 0 0]
[0 0 1 0 1 0]
[0 1 1 0 0 0]
[0 0 1 0 0 0]
[0 0 1 1 0 0]
[0 0 1 0 0 1]

[0 0 0 0 0 1]
[0 1 0 0 0 0]
[1 0 0 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 1 0 0 0]
```

90.2.5 Action on a Polynomial Ring

`GModule(G, P, d)`

Let G be a permutation group of degree n or a or matrix group of degree n over a finite field, $P = K[x_1, \ldots, x_n]$ a polynomial ring over a field K in n variables, and d a non-negative integer. This function creates the $K[G]$-module M corresponding to the action of G on the space of homogeneous polynomials of degree d of the polynomial ring P. The function also returns the isomorphism f between the space of homogeneous polynomials of degree d of P and M, together with an indexed set of monomials of degree d of P which correspond to the columns of M.

`GModule(G, I, J)`

Let G be a permutation group of degree n or a or matrix group of degree n over a finite field, I an ideal of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ in n variables over a field K, and J a zero-dimensional subideal of I. This function creates the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient I/J. The function also returns the isomorphism f between the quotient space I/J and M, together with an indexed set of monomials of P, forming a (vector space) basis of I/J, and which correspond to the columns of M.

`GModule(G, Q)`

Let G be a permutation group of degree n or a or matrix group of degree n over a finite field and $Q = I/J$ a finite-dimensional quotient ring of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ in n variables over a field K. This function creates the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient Q. The function also returns the isomorphism f between the quotient ring Q and M, together with an indexed set of monomials of P, forming a (vector space) basis of Q, and which correspond to the columns of M.

Example H90E7

Let T be the polynomial ring in five indeterminates over $GF(5)$. We create the representation of the alternating group of degree 5 that corresponds to its action on the space H_4 of homogeneous polynomials of degree 4 of T.

```
> G := Alt(5);
> R<[x]> := PolynomialRing(GF(5), 5);
> M, f := GModule(G, R, 4);
> M;
GModule M of dimension 70 over GF(5)
```

Thus, the action of $Alt(5)$ on H_4 yields a 70-dimensional module. We find its irreducible constituents.

```
> Constituents(M);
[
    GModule of dimension 1 over GF(5),
    GModule of dimension 3 over GF(5),
    GModule of dimension 5 over GF(5)
]
> t := x[1]^4 + x[2]^4 + x[3]^4 + x[4]^4 + x[5]^4;
> v := f(t); v;
M: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1)
> v@@f;
x[1]^4 + x[2]^4 + x[3]^4 + x[4]^4 + x[5]^4
```

90.3 The Representation Afforded by a $K[G]$-module

```
GModuleAction(M)
```

Given a $K[G]$-module M, return the action of G on M as homomorphism f of G into the matrix group $GL_n(K)$.

```
Representation(M)
```

Given a $K[G]$-module M, return the action of G on M as homomorphism f of G into the matrix algebra $M_n(K)$.

Example H90E8

The function `Representation` allows the easy calculation of group characters. We illustrate this with the 6-dimension module for the group A_7 constructed above.

```
> A7 := AlternatingGroup(7);
> M  := PermutationModule(A7, Vector(GF(11), [1,0,1,0,1,0,1]));
> phi := Representation(M);
> [ Trace(phi(c[3])) : c in Classes(A7) ];
[ 7, 3, 4, 1, 1, 2, 0, 0, 0 ]
```

Example H90E9

We present a procedure which, given a $K[G]$-module M, constructs its dual D.

```
> DualModule := function(M)
>       G := Group(M);
>       f := Representation(M);
>       return GModule(G, [ Transpose(f(G.i))^-1 : i in [1 .. Ngens(G)] ]);
> end function;
```

`ActionGenerator(M, i)`

`RightActionGenerator(M, i)`

The i-th generator of the (right) acting matrix algebra for the module M. That is, the image of the i-th group generator in the corresponding representation.

`ActionGenerators(M)`

Return the matrices giving the action on the module M as a sequence. These are the images of the generators of the group in the corresponding representation.

`NumberOfActionGenerators(M)`

`Nagens(M)`

The number of action generators (the number of generators of the algebra) for the $R[G]$-module M.

`ActionGroup(M)`

The matrix group generated by the action generators of M.

`Sections (G)`

Given a matrix group G defined over a finite field K, return the action of G on each composition factor of the natural K[G]-module for G.

Example H90E10

We construct the tensor square T of the natural module M of the matrix group $G = SL(3, 5)$ and then determine the action of G on each composition factor of T.

```
> G := SL(3, 5);
> M := GModule(G);
> T := TensorProduct(M, M);
> A := ActionGroup(T);
> S := Sections(A);
> #S;
2
```

There are just two composition factors of T, the symmetric square and the exterior square of M.

```
> S[2];
MatrixGroup(3, GF(5))
Generators:

    [1 0 0]
    [0 2 0]
    [0 0 3]

    [0 1 0]
    [1 0 1]
    [1 0 0]
```

90.4 Standard Constructions

Given one or more existing modules, various standard constructions are available to construct new modules.

90.4.1 Changing the Coefficient Ring

In this collection of functions will be found utilities for changing the base ring of the module. Note that several of the functions for rewriting over a minimal field are restricted to rings $K[G]$ where K is a finite field.

`ChangeRing(M, S)`

Given an A-module M with base ring R, together with a ring S, such that there is a natural homomorphism from R to S, construct the module N with base ring S where N is obtained from M by coercing the components of the vectors of M into N. The corresponding homomorphism from M to N is returned as a second value.

`ChangeRing(M, S, f)`

Given a module M with base ring R, together with a ring S, and a homomorphism $f : R \to S$, construct the module N with base ring S, where N is obtained from M by applying f to the components of the vectors of M. The corresponding homomorphism from M to N is returned as a second value.

90.4.2 Writing a Module over a Smaller Field

The functions in this section currently only apply to $K[G]$-modules defined over a finite field K.

`IsRealisableOverSmallerField(M)`

Given a $K[G]$-module M, where K is a finite field, return true if M can be realised over a proper subfield F of K. The equivalent $F[G]$-module is also returned. The Glasby-Howlett algorithm is used to determine the smallest field over which M can be realised.

`IsRealisableOverSubfield(M, F)`

Let M be a $K[G]$-module, where K is a finite field of characteristic p, and let F be a finite field also of characteristic p. If it is possible to realise M over the subfield F of K, return true and the equivalent $F[G]$-module.

`WriteOverSmallerField(M, F)`

Given a module M of dimension d over a finite field E having degree e and a subfield F of E having degree f, write the action of M as $d * e/f$ by $d * e/f$ matrices over F and return the module and the isomorphism.

`AbsoluteModuleOverMinimalField(M, F)`

Let M be a $K[G]$-module, where K is a finite field of characteristic p, and let F be a finite field also of characteristic p. This function returns the module obtained by writing M over the smallest possible field containing F subject to the condition that the dimension of M does not increase. The Glasby-Howlett algorithm is used to determine the smallest field over which M can be realised.

`AbsoluteModuleOverMinimalField(M)`

Verbose	`Reduce`	*Maximum :* 2
Verbose	`Cohomology`	*Maximum :* 2
Verbose	`GrunwaldWang`	*Maximum :* 2

Let M be a $K[G]$-module, where K is a finite field of characteristic p or a number field. This function returns the module obtained by writing M over a field of smallest possible degree subject to the condition that the dimension of M does not increase. For modules over finite fields, a field of smallest degree is always a subfield of K, in this case, the Glasby-Howlet algorithm is used. For number fields, a different field might be necessary and a combination of Plesken's method and a constructive version of the Grunwald-Wang theorem is used.

`Minimize(R)`

`All`	BoolElt	*Default :* `false`
`Char`	AlgChtrElt	*Default :* `false`
`FindSmallest`	BoolElt	*Default :* `false`
Verbose	`Reduce`	*Maximum :* 2
Verbose	`Cohomology`	*Maximum :* 2
Verbose	`GrunwaldWang`	*Maximum :* 2

Let $R : G \to \mathrm{Gl}(n, K)$ be an absolutely irreducible representation over some number field K. This function tries to find minimal subfields k of K that afford R, ie. it tries to write the representation over a smaller field. In general however, there might be number field k not contained in K of smaller degree that afford R. If `All` is given, then instead of a single representation over a minimal degree subfield of K, a list of representations over all minimal subfields of K is returned instead. If `Char` is given, it should be set to the character of the representation. If `FindSmallest` is given, the field K will be extended by some auxiliary field A such that KA will contain a minimal degree field affording R. This involves a constructive version of the Grunwald-Wang theorem and can be computationally expensive if the degree of KA is too large.

`AbsoluteModulesOverMinimalField(Q, F)`

Let Q be a sequence of $K[G]$-modules, where K is a finite field of characteristic p, and let F be a finite field also of characteristic p. This function returns the sequence of modules obtained by writing each module M of Q over the smallest possible field containing F subject to the condition that the dimension of M does not increase. Thus, the effect of the function is to apply the function `AbsoluteModuleOverMinimalField` to each module of Q. The Glasby-Howlett algorithm is used to determine the smallest field over which the modules M of Q can be realised.

`ModuleOverSmallerField(M, F)`

Let M be a $K[G]$-module of dimension d, where K is a finite field of characteristic p, and let F be a subfield of K of index n. This function returns the $F[G]$-module N obtained by writing the action of M as $dn \times dn$ matrices over F.

`ModulesOverSmallerField(Q, F)`

Let Q be a sequence of $K[G]$-modules, where K is a finite field of characteristic p, and let F be a subfield of K of index n. This function returns the sequence R of $F[G]$-modules obtained by applying the function `ModuleOverSmallerField` to each term of Q. That is, each term N of R is formed by writing the action of the corresponding term of Q as $dn \times dn$ matrices over F.

`ModulesOverCommonField(M, N)`

Given $K[G]$-modules M and N, change their base fields to K, where K is the smallest field containing the base fields of M and N.

`WriteGModuleOver(M, K)`

`Char`	AlgChtrElt	*Default :* `false`
`Subfield`	BoolElt	*Default :* `false`
Verbose	`Reduce`	*Maximum :* 2
Verbose	`Cohomology`	*Maximum :* 2
Verbose	`GrunwaldWang`	*Maximum :* 2

Given a $L[G]$ module M and some number field K, try to write M over K. If `Char` is specified, it should be set to the character of this module. If `Subfield` is given, the module will be rewritten over a minimal degree subfield of K.

`WriteRepresentationOver(R, K)`

`Char`	AlgChtrElt	*Default :* `false`
`Subfield`	BoolElt	*Default :* `false`
Verbose	`Reduce`	*Maximum :* 2
Verbose	`Cohomology`	*Maximum :* 2
Verbose	`GrunwaldWang`	*Maximum :* 2

Given an absolutely irreducible representation $R : G \to \mathrm{Gl}(n, L)$ and some normal number field K, try to write R over K. If `Char` is specified, it should be set to the character of this representation. If `Subfield` is given, the representation will be rewritten over a minimal degree subfield of K.

Example H90E11

We will work with the G-module and character of the unique 2-dimensional character of Q_8. It is well known that, while the character is defined over Q, the corresponding representation can only be defined over fields where -1 is the sum of 2 squares.

```
> G := TransitiveGroup(8, 5);
> TransitiveGroupDescription(G);
Q_8(8);
> R := AbsolutelyIrreducibleModules(G, Rationals());
> R;
[
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 2 over Cyclotomic Field
    of order 4 and degree 2
]
```

```
> R := R[5];
> WriteGModuleOver(R, CyclotomicField(5));
GModule of dimension 2 over Cyclotomic Field
of order 5 and degree 4
```

So $Q(\zeta_5)$ is an example of a field affording the module but having no minimal degree subfield (of degree 2 here) affording R!.

```
> AbsoluteModuleOverMinimalField($1);
GModule of dimension 2 over Number Field with
defining polynomial Qx.1^2 - Qx.1 + 1 over the
Rational Field
```

Note that the base field returned here is $Q(\zeta_3)$ which is of degree 2 but different from $Q(\zeta_4)$ that was found initially. In general there are infinetely many minimal degree splitting fields.
If we try to realize R over a field where -1 cannot be written as a sums of two squares we get an error:

```
>  WriteGModuleOver(R, QuadraticField(3));
>> WriteGModuleOver(R, QuadraticField(3));
                    ^
Runtime error in 'WriteGModuleOver': The G-module
cannot be realised over K
```

We can try to find a minimal field containing $Q(\sqrt{3})$ by computing the local Schur-indices and then obtain a splitting field:

```
> k := QuadraticField(3);
> SchurIndices(Character(R), k);
[ <1st place at infinity, 2>, <2nd place at
infinity, 2> ]
> A := SplittingField($1);
> A;
FldAb, defined by (<3>, [1      2])
of structure: Z/2
```

So the splitting field is returned as an abelian extension. We can see that A is of degree 2 over k and will be ramified at most at 3 and both infinite places. In order to use it to rewrite the module, we need to convert to a number field over Q first:

```
> A := NumberField(A);
> A;
Number Field with defining polynomial $.1^2 + 1
over k
> A := AbsoluteField(A);
> A;
Number Field with defining polynomial Qx.1^4 -
    4*Qx.1^2 + 16 over the Rational Field
> WriteGModuleOver(R, A);
GModule of dimension 2 over A
> WriteGModuleOver(R, A:Subfield);
```

```
GModule of dimension 2 over Number Field with
defining polynomial Qx.1^2 - Qx.1 + 1 over the
Rational Field
```

90.4.3 Direct Sum

`DirectSum(M, N)`

Given $K[G]$-modules M and N, construct the direct sum D of M and N as an $K[G]$-module. The embedding maps from M into D and from N into D respectively and the projection maps from D onto M and from D onto N respectively are also returned.

`DirectSum(Q)`

Given a sequence Q of $K[G]$-modules, construct the direct sum D of these modules. The embedding maps from each of the elements of Q into D and the projection maps from D onto each of the elements of Q are also returned.

90.4.4 Tensor Products of $K[G]$-Modules

`TensorProduct(M, N)`

Let M and N be two $K[G]$ modules. This function constructs the tensor product, $M \otimes_A N$, with diagonal action.

`TensorPower(M, n)`

Given a $K[G]$-module M and an integer $n \geq 1$, construct the n-th tensor power of M.

`ExteriorSquare(M)`

Given a $K[G]$-module M, construct the A-submodule of $M \otimes_A M$ consisting of the skew tensors.

`SymmetricSquare(M)`

Given a $K[G]$-module M, construct the A-submodule of $M \otimes_A M$ consisting of the symmetric tensors.

90.4.5 Induction and Restriction

`Dual(M)`

Given an $K[G]$-module M, construct the $K[G]$-module which is the K-dual, $\mathrm{Hom}_K(M, K)$, of M.

`Induction(M, G)`

Given a $K[H]$-module M and a supergroup G of H, construct the $K[G]$-module obtained by inducing M up to G.

`Induction(R, G)`

Given a representation R of a subgroup of G, construct the representation of G obtained by inducing R up to G.

`Restriction(M, H)`

Given a $K[G]$-module M and a subgroup H of G, form the $K[H]$-module corresponding to the restriction of M to the subgroup H.

Example H90E12

Starting with the permutation module M over $\mathbf{F}_2$ for the Mathieu group M_{22}, we apply the induction and restriction functions to find new irreducible modules for M_{22}.

```
> SetSeed(1);
> G := PermutationGroup< 22 |
>         (1,2,4,8,16,9,18,13,3,6,12)(5,10,20,17,11,22,21,19,15,7,14),
>         (1,18,4,2,6)(5,21,20,10,7)(8,16,13,9,12)(11,19,22,14,17),
>         (1,18,2,4)(3,15)(5,9)(7,16,21,8)(10,12,20,13)(11,17,22,14) >;
> M := PermutationModule(G, GaloisField(2));
> M;
GModule M of dimension 22 with base ring GF(2)
> CM := Constituents(M);
> CM;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 10 over GF(2),
    GModule of dimension 10 over GF(2)
]
```

We restrict the module M to the stabilizer of a point in M_{22} and then induce back up, a constituent of the restriction.

```
> L34 := Stabilizer(G, 1);
> N := Restriction(M, L34);
> N;
GModule N of dimension 22 with base ring GF(2)
> CN := Constituents(N);
> CN;
[
```

```
    GModule of dimension 1 over GF(2),
    GModule of dimension 9 over GF(2),
    GModule of dimension 9 over GF(2)
]
> Ind1 := Induction(CN[1], G);
> Ind1;
GModule Ind1 of dimension 22 over GF(2)
> Constituents(Ind1);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 10 over GF(2),
    GModule of dimension 10 over GF(2)
]
> Ind2 := Induction(CN[2], G);
> Ind2;
GModule Ind2 of dimension 198 over GF(2)
> Constituents(Ind2);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 10 over GF(2),
    GModule of dimension 10 over GF(2),
    GModule of dimension 34 over GF(2),
    GModule of dimension 98 over GF(2)
]
```

Thus, inducing up the 1-dimensional constituent of N gives us irreducible modules for G having the same dimensions as those appearing as constituents of M. However, inducing up the 9-dimensional module gives us irreducible modules of new dimensions: 34 and 98. Hence starting out with only the permutation module for M_{22} over $\mathbf{F}_2$, we have found 5 irreducible modules for the group.

90.4.6 The Fixed-point Space of a Module

`Fix(M)`

Given an $K[G]$-module M, construct the largest submodule of M on which G acts trivially, i.e. the fixed-point space of M.

90.4.7 Changing Basis

`M ^ T`

Given a $K[G]$-module M of dimension n over the field K, and a nonsingular $n \times n$ matrix T over K, construct the $K[G]$-module N which corresponds to taking the rows of T as a basis for M.

90.5 The Construction of all Irreducible Modules

The construction of all irreducible K[G]-modules for a finite group G is of major interest. If G is soluble there is a very effective method that dates back to Schur. This proceeds by working up a composition series for G and constructing the irreducibles for each subgroup by inducing or extending representations from the previous subgroup. This works equally well over finite fields and over fields of characteristic zero. If G is non-soluble the situation is more difficult. Starting with a faithful representation, by a theorem of Burnside, it is possible to construct all representations by splitting tensor powers of the faithful representation. An algorithm based on this idea developed by Cannon and Holt uses the Meataxe to split representations and works well over small finite fields. A similar algorithm for rational representations using a different method for splitting representations is under development by Steel. Methods based on the theorem of Burnside, will be referred to as *Burnside algorithms*.

90.5.1 Generic Functions for Finding Irreducible Modules

The functions described in this section construct all irreducible representations of a finite group. The choice of algorithm depends upon the type of group and the kind of field given. The individual algorithms may be invoked directly by means of intrinsic functions described in subsequent sections.

`IrreducibleModules(G, K : parameters)`

`AbsolutelyIrreducibleModules(G, K : parameters)`

Let G be a finite group and let K be a field. If G is soluble then K may be either a finite field, the rational field or a cyclotomic field whose order divides the exponent of G. If G is non-soluble, then currently K is restricted to being a finite field or the rational field. These functions construct all of the irreducible or absolutely irreducible G-modules over K. Since V2.16, if K is the rational field, then by default a new algorithm is used. Otherwise, if G is soluble, Schur's algorithm is normally used while if G is non-soluble, the Burnside method is used.

`Alg` MonStgElt *Default :*

The parameter `Alg` may be used to override the usual choice of algorithm if desired. Note that Schur's algorithm may only be applied to soluble groups, while, for the time being, Burnside's method requires that the field K is finite.

The Schur algorithm is usually very fast and is often able to find the complex representations more quickly than it is possible to compute the character table. The speed of the Burnside algorithm is determined firstly by the maximal degree of the irreducible modules and secondly by the number of irreducible modules. It will start to become quite slow if G has modules of dimension in excess of 1000. In order to prevent the Burnside method from wasting huge amounts of time, the algorithm takes a parameter which controls the degree of the largest module that will be consider for splitting.

`DimLim` RngInt *Default :* 2000

The parameter `DimLim` only affects the Burnside algorithm where it is used to limit the dimension of the modules which will be considered for splitting. If this limit prevents all irreducibles being found, a warning message is output and those irreducibles that have been found will be returned. This possibility allows the user to determine a sample of low degree modules without using excessive time.

Example H90E13

We take a group of order $416 = 2^5 13$ and compute its irreducible modules over various fields.

```
> G := SmallGroup(416, 136);
> G;
GrpPC : G of order 416 = 2^5 * 13
PC-Relations:
    G.1^2 = Id(G),
    G.2^2 = G.5,
    G.3^2 = Id(G),
    G.4^2 = G.5,
    G.5^2 = Id(G),
    G.6^13 = Id(G),
    G.3^G.1 = G.3 * G.5,
    G.3^G.2 = G.3 * G.4,
    G.4^G.2 = G.4 * G.5,
    G.4^G.3 = G.4 * G.5,
    G.6^G.1 = G.6^12
```

We first compute the $K[G]$-modules for the finite fields $K = GF(p)$, where p runs through the primes dividing the order of G.

```
> IrreducibleModules(G, GF(2));
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 12 over GF(2)
]
> IrreducibleModules(G, GF(13));
[
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 1 over GF(13),
    GModule of dimension 2 over GF(13),
    GModule of dimension 2 over GF(13),
    GModule of dimension 4 over GF(13)
```

```
]
```

We now compute the $K[G]$-modules where K is the rational field.

```
> time L := IrreducibleModules(G, Rationals());
Time: 16.170
> L;
[
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 1 over Rational Field,
    GModule of dimension 2 over Rational Field,
    GModule of dimension 2 over Rational Field,
    GModule of dimension 8 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 24 over Rational Field,
    GModule of dimension 96 over Rational Field
]
```

Finally, we compute the $K[G]$-modules taking K to be the splitting field over the rationals and verify that the number of irreducibles is equal to the number of conjugacy classes of G.

```
> Exponent(G);
104
> mods := IrreducibleModules(G, CyclotomicField(104));
> #mods;
53;
> #Classes(G);
53
> [ Dimension(N) : N in mods ];
[ 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4,
  4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
> X := CharacterTable(G);
> [ Degree(x) : x in X ];
[ 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4,
  4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4 ]
```

90.5.2 The Burnside Algorithm

The Burnside algorithm finds all irreducible $K[G]$-modules with a faithful $K[G]$-module P and looks for the distinct irreducibles among the tensor powers of P. Both irreducible and absolutely irreducible modules may be found. At present the algorithm is restricted to finite fields. In more detail the algorithm starts with a some faithful permutation module over the given field, splits it into irreducibles using the meataxe, and constructing further modules to split as tensor products of those already found. A warning is printed if all irreducible modules are not found.

`AbsolutelyIrreducibleModulesBurnside(G, K : parameters)`

`DimLim` RngIntElt *Default :* 2000

Given a finite group G and a finite field K, this function constructs the absolutely irreducible $K[G]$-modules over extensions of K. Currently, the group G is restricted to a permutation group. The maximum dimension of a module considered for splitting is controlled by the parameter `DimLim`, which has default value 2000.

`IrreducibleModulesBurnside(G, K : parameters)`

`DimLim` RngIntElt *Default :* 2000

Given a finite group G and a finite field K, this function constructs the irreducible $K[G]$-modules over K. Currently, the group G is restricted to a permutation group. The maximum dimension of a module considered for splitting is controlled by the parameter `DimLim`, which has default value 2000.

`AbsolutelyIrreducibleConstituents(M)`

Given an irreducible module M, return M if it is already absolutely irreducible, else return the absolutely irreducible modules obtained by finding the constituents of M after extending the base field of M to a splitting field.

Example H90E14

We find all irreducible modules for M_{11} over $GF(2)$, and all absolutely irreducible modules of characteristic 2. The Burnside algorithm is used by default.

```
> load m11;
Loading "/home/magma/libs/pergps/m11"
M11 - Mathieu group on 11 letters - degree 11
Order 7 920 = 2^4 * 3^2 * 5 * 11;  Base 1,2,3,4
Group: G
> IrreducibleModules(G, GF(2));
[
  GModule of dimension 1 over GF(2),
  GModule of dimension 10 over GF(2),
  GModule of dimension 44 over GF(2),
  GModule of dimension 32 over GF(2)
]
> AbsolutelyIrreducibleModules(G, GF(2));
```

```
[
  GModule of dimension 1 over GF(2),
  GModule of dimension 10 over GF(2),
  GModule of dimension 44 over GF(2),
  GModule of dimension 16 over GF(2^2),
  GModule of dimension 16 over GF(2^2)
]
```

90.5.3 The Schur Algorithm for Soluble Groups

This collection of functions allows the user to find all irreducible modules of a finite soluble group. The group is first replaced by an isomorphic group defined by a power-conjugate presentation. The irreducibles are then found using Schur's method of working up the composition series for G defined by the pc-presentation.

`IrreducibleModules(G, K : parameters)`

`AbsolutelyIrreducibleModulesSchur(G, K: parameters)`

Let G be a finite soluble group and let K be one the following types of field: a finite field, the rational field or a cyclotomic field. The order of a cyclotomic field must divide the exponent of G. The function constructs all absolutely irreducible representations of G over appropriate extensions or subfields of the field K. The modules returned are non-isomorphic and consist of all distinct modules, subject to the conditions imposed. In the case when K is a finite field, the Glasby-Howlett algorithm is used to determine the minimal field over which an irreducible module may be realised. If K has characteristic 0, the field over which an irreducible module is given may not be minimal.

The irreducible modules are found using Schur's method of climbing the composition series for G defined by the pc-presentation.

`Process` BoolElt *Default :* `true`

If the parameter `Process` is set true then the list is a list of pairs comprising an integer and a representation. This list or any sublist of it is a suitable value for the argument L in the last versions of the function, and in this case only the representations in L will be extended up the series. This allows the user to inspect the representations produced along the way and cull any that are uninteresting.

`GaloisAction` MonStgElt *Default :* "Yes"

Possible values are "Yes", "No" and "Relative". The default is "Yes" for intermediate levels and "No" for the whole group. The value "Yes" means that it only lists one irreducible from each orbit of the action of the absolute Galois group $Gal(K/\text{primefield}(K))$. Setting this parameter to "No" turns this reduction off (thus listing all inequivalent representations), while setting it to "Relative" uses the group $Gal(K/k)$.

`MaxDimension` RngIntElt *Default :*

Restrict the irreducible to those of dimension $\leq$ MaxDimension. Default is no restriction.

`ExactDimension` SetEnum *Default :*

If `ExactDimension` is assigned a set S of positive integers, attention is restricted to irreducible having dimensions lying in the set S. The default is equivalent to taking the set of all positive integers.

If both `MaxDimension` and `ExactDimension` are assigned values, then irreducible having dimensions that are either bounded by `MaxDimension` or contained in `ExactDimension` are produced.

`IrreducibleModulesSchur(G, K: parameters)`

Compute irreducible modules for G over the given field K. All arguments and parameters are as for the absolutely irreducible case.

The computation proceeds by first computing the absolutely irreducible representations subject to the given conditions, then rewriting over the field K, with a consequent change of dimension of the representation.

Example H90E15

We compute representations of the dihedral group of order 20.

```
> G := DihedralGroup(GrpPC, 10);
> FactoredOrder(G);
[ <2, 2>, <5, 1> ]
```

First some modular representations with characteristic 2.

```
> r := IrreducibleModulesSchur(G, GF(2));
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2)
*]
> r := AbsolutelyIrreducibleModulesSchur(G, GF(2));
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(2^2),
    GModule of dimension 2 over GF(2^2)
*]
> r := AbsolutelyIrreducibleModulesSchur(G, GF(2) : GaloisAction:="Yes");
> r;
[*
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(2^2)
*]
```

The irreducible representation of dimension 4 is not absolutely irreducible, as over $GF(4)$ it splits into two Galois-equivalent representations.

Finding irreducible modules over the complex field is straightforward, despite not being able to use the complex field as the field argument. We could instead specify the cyclotomic field having order equal to the exponent of G, but it is preferable to ask for all absolutely irreducible modules over the rationals.

```
> r := AbsolutelyIrreducibleModulesSchur(G, Rationals());
> r;
[*
   GModule of dimension 1 over Rational Field,
   GModule of dimension 1 over Rational Field,
   GModule of dimension 1 over Rational Field,
   GModule of dimension 1 over Rational Field,
   GModule of dimension 2 over Cyclotomic Field of order 5 and degree 4,
   GModule of dimension 2 over Cyclotomic Field of order 5 and degree 4,
   GModule of dimension 2 over Cyclotomic Field of order 5 and degree 4,
   GModule of dimension 2 over Cyclotomic Field of order 5 and degree 4
*]
> Representation(r[6])(G.2);
[zeta_5^3        0]
[       0 zeta_5^2]
```

AbsolutelyIrreducibleRepresentationsInit(G, F : *parameters*)

AbsolutelyIrreducibleModulesInit(G, F : *parameters*)

IrreducibleRepresentationsInit(G, F : *parameters*)

IrreducibleModulesInit(G, F : *parameters*)

Initialize a Process for calculating all linear representations of a soluble group G over the field F. The field F is restricted to either a finite field or the rationals. The parameters are as described above.

NextRepresentation(P)

NextModule(P)

Return `true` and the next representation from the process P, if there is one, or just `false` if the process is exhausted.

AbsolutelyIrreducibleRepresentationProcessDelete(~P)

Free all data associated with the process P.

90.5.4 The Rational Algorithm

MAGMA V2.16 contains a new algorithm for constructing irreducible rational representations of an arbitrary finite group. This algorithm is now selected by default in the function `IrreducibleModules(G, K)` when the field K is $\mathbf{Q}$.

Given an irreducible complex character for a group G, the sum of its Galois orbit gives an *irreducible rational character* for G. The Schur index of an irreducible rational character χ is defined to be the Schur index of an irreducible complex character whose Galois orbit sum is χ.

We call the sequence of all possible irreducible rational characters for G (sorted by degree) the *rational character table* of G. The irreducible modules returned by `IrreducibleModules(G, RationalField())` always match the rational character table.

`IrreducibleModules(G, Q : parameters)`

Given a finite group G, this function returns a sequence L of all irreducible rational $K[G]$-modules, and also the rational character table of G as a sequence C, which matches L. The character of the i-th module $L[i]$ is always $s_i \cdot C[i]$, where s_i is the Schur index of $C[i]$. Thus the dimension of the irreducible module $L[i]$ is $s_i \cdot \mathrm{Deg}(C[i])$.

`MaxDegree` RNGINTELT *Default :* 0

Setting the parameter `MaxDegree` to a positive integer D instructs the algorithm not to spend effort on constructing irreducible modules whose corresponding irreducible rational characters have degree greater than D. Note that irreducible modules whose character degrees greater than D may be returned in any case, if they are easily constructed (often as side effects of operations used to construct smaller-dimensional modules).

`Characters` [ALGCHTRELT] *Default :*

Setting the parameter `Characters` to a set or sequence S of characters for the group G instructs the algorithm to attempt to construct only the irreducible modules whose characters are in S. Each character in S may either be an irreducible rational character, or a (complex) ordinary character χ, in which case the irreducible rational character corresponding to χ (the orbit sum of χ) is used. As for the parameter `MaxDegree`, irreducible modules whose characters are not in S may be returned in any case, if they are easily constructed or are needed as intermediate modules to construct the desired modules.

`CharacterDegrees` [RNGINTELT] *Default :*

Setting the parameter `CharacterDegrees` to a set or sequence I of positive integers instructs the algorithm to attempt to construct only the irreducible modules whose character degrees are in S. This is equivalent to setting the parameter `Characters` to `[c: c in RationalCharacterTable(G) | Degree(c) in I]`.

`UseInduction` BOOLELT *Default :* `true`

By default, as the algorithm proceeds, it automatically searches for subgroups H_i of G such that irreducible rational H_i-modules may be induced up to G to yield

G-modules from which irreducible G-modules may be computed. In general, this method is very effective (and often yields modules for G with small entries) but can be very slow for some groups (particularly when G has many subgroups). Thus setting the parameter `UseInduction` to `false` will force the algorithm not to use induction.

`RationalCharacterTable(G)`

Given a finite group G, return the rational character table of G as a sequence C of the irreducible rational characters of G (sorted by degree), and also a index sequence I, such that for each i, $C[i]$ is the sum of the Galois orbit of $T[I[i]]$, where T is the ordinary (complex) character table of G.

Example H90E16

We compute all irreducible rational modules for PSL(3, 3) and note that the characters of the resulting modules match the entries in the rational character table.

```
> G := PSL(3, 3); #G;
5616
> T := CharacterTable(G); [Degree(x): x in T];
[ 1, 12, 13, 16, 16, 16, 16, 26, 26, 26, 27, 39 ]
> C := RationalCharacterTable(G); C;
[
    ( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
    ( 12, 4, 3, 0, 0, 1, 0, 0, -1, -1, -1, -1 ),
    ( 13, -3, 4, 1, 1, 0, -1, -1, 0, 0, 0, 0 ),
    ( 26, 2, -1, -1, 2, -1, 0, 0, 0, 0, 0, 0 ),
    ( 27, 3, 0, 0, -1, 0, -1, -1, 1, 1, 1, 1 ),
    ( 39, -1, 3, 0, -1, -1, 1, 1, 0, 0, 0, 0 ),
    ( 52, -4, -2, -2, 0, 2, 0, 0, 0, 0, 0, 0 ),
    ( 64, 0, -8, 4, 0, 0, 0, 0, -1, -1, -1, -1 )
]
> time L := IrreducibleModules(G, RationalField());
Time: 0.760
> L;
[
    GModule of dimension 1 over Rational Field,
    GModule of dimension 12 over Rational Field,
    GModule of dimension 13 over Rational Field,
    GModule of dimension 26 over Rational Field,
    GModule of dimension 27 over Rational Field,
    GModule of dimension 39 over Rational Field,
    GModule of dimension 52 over Rational Field,
    GModule of dimension 64 over Rational Field
]
> [Character(M): M in L];
[
    ( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
```

```
    ( 12, 4, 3, 0, 0, 1, 0, 0, -1, -1, -1, -1 ),
    ( 13, -3, 4, 1, 1, 0, -1, -1, 0, 0, 0, 0 ),
    ( 26, 2, -1, -1, 2, -1, 0, 0, 0, 0, 0, 0 ),
    ( 27, 3, 0, 0, -1, 0, -1, -1, 1, 1, 1, 1 ),
    ( 39, -1, 3, 0, -1, -1, 1, 1, 0, 0, 0, 0 ),
    ( 52, -4, -2, -2, 0, 2, 0, 0, 0, 0, 0, 0 ),
    ( 64, 0, -8, 4, 0, 0, 0, 0, -1, -1, -1, -1 )
]
```

Example H90E17

For large groups, one can use the parameter `MaxDegree` to compute the irreducible modules of reasonable dimension.

```
> load m23;
Loading "/home/magma/libs/pergps/m23"
M23 - Mathieu group on 23 letters - degree 23
Order 10 200 960 = 2^7 * 3^2 * 5 * 7 * 11 * 23;  Base 1,2,3,4,5,6
Group: G
> T := CharacterTable(G); [Degree(x): x in T];
[ 1, 22, 45, 45, 230, 231, 231, 231, 253, 770, 770,
896, 896, 990, 990, 1035, 2024 ]
> C := RationalCharacterTable(G); C;
[
    ( 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ),
    ( 22, 6, 4, 2, 2, 0, 1, 1, 0, 0, 0, -1, -1, -1, -1, -1, -1 ),
    ( 90, -6, 0, 2, 0, 0, -1, -1, -2, 2, 2, 1, 1, 0, 0, -2, -2 ),
    ( 230, 22, 5, 2, 0, 1, -1, -1, 0, -1, -1, 1, 1, 0, 0, 0, 0 ),
    ( 231, 7, 6, -1, 1, -2, 0, 0, -1, 0, 0, 0, 0, 1, 1, 1, 1 ),
    ( 253, 13, 1, 1, -2, 1, 1, 1, -1, 0, 0, -1, -1, 1, 1, 0, 0 ),
    ( 462, 14, -6, -2, 2, 2, 0, 0, -2, 0, 0, 0, 0, -1, -1, 2, 2 ),
    ( 1035, 27, 0, -1, 0, 0, -1, -1, 1, 1, 1, -1, -1, 0, 0, 0, 0 ),
    ( 1540, -28, 10, -4, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1 ),
    ( 1792, 0, -8, 0, 2, 0, 0, 0, 0, -1, -1, 0, 0, 2, 2, -2, -2 ),
    ( 1980, -36, 0, 4, 0, 0, -1, -1, 0, 0, 0, -1, -1, 0, 0, 2, 2 ),
    ( 2024, 8, -1, 0, -1, -1, 1, 1, 0, 0, 0, 1, 1, -1, -1, 0, 0 )
]
> Q := RationalField();
> time L := IrreducibleModules(G, Q: MaxDegree := 253);
Time: 23.400
> L;
[
    GModule of dimension 1 over Rational Field,
    GModule of dimension 22 over Rational Field,
    GModule of dimension 90 over Rational Field,
    GModule of dimension 230 over Rational Field,
    GModule of dimension 231 over Rational Field,
    GModule of dimension 253 over Rational Field,
```

```
    undef,
    undef,
    undef,
    undef,
    GModule of dimension 1980 over Rational Field
]
```

Note that the module of dimension 1980 is included (it was easily constructed as the tensor product of modules of dimension 20 and 90).

90.6 Extensions of Modules

For $K[G]$-modules M and N, the K-vector $\mathrm{Ext}(M, N)$ of equivalence classes of $K[G]$-module extensions

$$0 \to N \to L \to M \to 0$$

of N by M can be computed, and corresponding extensions L constructed.

`Ext(M, N)`

Given $K[G]$-modules M and N, construct the K-vector space $\mathrm{Ext}(M, N)$ of equivalence classes of $K[G]$-module extensions of N by M.

`Extension(M, N, e)`

Construct a $K[G]$-module extension L of N by M corresponding to the element e of E, where E must be the vector space returned by a previous call of `Ext(M,N)`. The insertion $N \to L$ and projection $L \to M$ are also returned.

`MaximalExtension(M, N, E)`

Again E must be the vector space returned by a previous call of `Ext(M,N)`. Construct the largest possible $K[G]$-module extension L of a direct sum of copies of N by M, such that none of the submodules of L that are isomorphic to N has a complement in L.

Example H90E18

```
> G := Alt(5);
> I := IrreducibleModules(G, GF(2));
> I;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2)
]
> M1 := rep{M: M in I | Dimension(M) eq 1};
> M4 := rep{M: M in I | Dimension(M) eq 4 and not IsAbsolutelyIrreducible(M)};
> M4; assert not IsAbsolutelyIrreducible(M4);
```

```
GModule M of dimension 4 over GF(2)
> E, rho := Ext(M4, M1);
> E;
Full Vector space of degree 2 over GF(2)
> Extension(M4, M1, E.1, rho);
GModule of dimension 5 over GF(2)
[0 0 0 0 1]
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
[0 0 0 0]
> E := MaximalExtension(M4, M1, E, rho);
> E;
GModule E of dimension 6 over GF(2)
> CompositionFactors(E);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2)
]
```

90.7 The Construction of Projective Indecomposable Modules

For a finite group G and a finite field K, the projective indecomposable $K[G]$-modules are in one-one correspondence with the irreducible $K[G]$-modules, where the projective indecomposable module P corresponding to the irreducible module I has the property that Socle(P) and P/JacobsonRadical(P) are both isomorphic to I.

MAGMA functions for the construction of the irreducible modules were described in Subsection 90.5.1. The functions described in this section may be used to construct the corresponding projective indecomposables for finite groups of moderate order – up to around 10^6, depending on the example. Large-dimensional projective indecomposable modules can sometimes be constructed by using *condensation* techniques, which allow many of the necessary computations to be carried out in condensed modules, which may have significantly smaller dimension than their standard uncondensed equivalents. As with the computation of irreducible modules, these methods work best for permutation groups and PC-groups.

The verbose flag `"KGModule"` may be set to 1 or 2 to show details of the computations.

`ProjectiveIndecomposableDimensions(G, K)`

The K-dimensions of the projective indecomposable $K[G]$-modules corresponding to the irreducible $K[G]$-modules returned by `IrreducibleModules(G,K)`. (These can be computed quickly from the Brauer characters of the irreducible modules, using the Cartan matrix discussed below.)

`ProjectiveIndecomposableModule(I:` *parameters*`)`

Construct and return the projective indecomposable $K[G]$-module P corresponding to the irreducible $K[G]$-module I. Note that Socle(P) and P/JacobsonRadical(P) are both isomorphic to I.

`condensation` BoolElt *Default :* `false`

If set to `true`, then an attempt is made to find a subgroup of G which allows computations to be carried out in condensed versions of the modules involved.

`ProjectiveIndecomposableModules(G, K)`

Construct the complete list of projective indecomposable $K[G]$-modules corresponding to the irreducible $K[G]$-modules returned by `IrreducibleModules(G,K)`.

`condensation` BoolElt *Default :* `false`

If set to `true`, then an attempt is made to find a subgroup of G which allows computations to be carried out in condensed versions of the modules involved.

Example H90E19

We compute the projective indecomposable modules for the alternating group of degree 8 over the field of order 2.

```
> G := Alt(8);
> K := GF(2);
> IrreducibleModules(G, K);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 64 over GF(2)
]
> ProjectiveIndecomposableDimensions(G, K);
[ 448, 192, 192, 320, 320, 192, 192, 64 ]
> time proj := ProjectiveIndecomposables(G, K);
Time: 22.070
> proj;
[
```

```
    GModule of dimension 448 over GF(2),
    GModule of dimension 192 over GF(2),
    GModule of dimension 192 over GF(2),
    GModule of dimension 320 over GF(2),
    GModule of dimension 320 over GF(2),
    GModule of dimension 192 over GF(2),
    GModule of dimension 192 over GF(2),
    GModule of dimension 64 over GF(2)
]
> CompositionFactors(proj[1]);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
```

```
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 4 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 6 over GF(2),
    GModule of dimension 14 over GF(2),
    GModule of dimension 1 over GF(2)
]
```

`CartanMatrix(G, K)`

Let k be the number of irreducible $K[G]$-modules. The Cartan Matrix C for G over K is a $k \times k$ matrix of integers, in which the entry C_{ij} is equal to the number of times that the j-th irreducible $K[G]$-module is a constituent of the i-th projective indecomposable $K[G]$-module. This can be computed quickly from the Brauer characters of the irreducible $K[G]$-modules, and is used in `ProjectiveIndecomposableDimensions`.

(Note that, unlike the absolute Cartan matrix discussed below, C need not be symmetric.)

`AbsoluteCartanMatrix(G, K)`

This is the Cartan matrix of G over an extension L of K that is large enough for all irreducible $L[G]$-modules to be absolutely irreducible. Its rows and columns correspond to the $K[G]$-modules returned by `AbsolutelyIrreducibleModules(G,K)`.

It is a symmetric matrix with integer entries, and is equal to the Cartan matrix for G in the characteristic p of K, as defined in textbooks on modular representation theory.

`DecompositionMatrix(G, K)`

The decomposition matrix D of G in the characteristic p of K, as defined in textbooks on modular representation theory. The entry D_{ij} is equal to the number of times that the j-th absolutely irreducible $K[G]$-module occurs as a constituent of the i-th ordinary irreducible G-module over the complex numbers reduced modulo p. Note that $D^{\mathrm{T}}D$ is equal to the absolute Cartan matrix.

`ProjectiveCover(M)`

Compute the projective cover P of $K[G]$-module M together with a $K[G]$-module epimorphism $P \to M$ returned as a matrix.

If P_i is the projective indecomposable $K[G]$-module corresponding to the irreducible $K[G]$-module I_i and $M/\text{JacobsonRadical}(M)$ is isomorphic to $\oplus_{j=1}^{t} I_{i_j}$ then P is isomorphic to $\oplus_{j=1}^{t} P_{i_j}$.

`CohomologicalDimension(M, n)`

For $K[G]$-module M (with K a finite field and G a finite group), compute and return the K-dimension of the cohomology group $H^n(G, M)$ for $n \geq 0$. For $n = 0$ and 1, this is carried out by using the functions described in Chapter 68. For $n \geq 2$, it is done recursively using projective covers and dimension shifting to reduce to the case $n = 1$. (In particular, for $n = 2$, the method is different from that employed by the corresponding function for a cohomology module described in Chapter 68.)

`CohomologicalDimensions(M, n)`

For $K[G]$-module M (with K a finite field and G a finite group), compute and return the sequence of K-dimensions of the cohomology groups $H^k(G, M)$ for $1 \leq k \leq n$. On account of the recursive method used, this is quicker than computing them individually.

Example H90E20

We compute the cohomology of the irreducible modules for Alt(8) over the field of order 2 computed in the previous example.

```
> G := Alt(8);
> K := GF(2);
> irr :=  IrreducibleModules(G, K);
> [ CohomologicalDimension(I, 1) : I in irr ];
[ 0, 0, 0, 1, 1, 1, 1, 0 ]
> time [ CohomologicalDimension(I, 2) : I in irr ];
[ 1, 1, 1, 0, 2, 0, 0, 0 ]
Time: 0.440
> time [ CohomologicalDimension(I, 3) : I in irr ];
[ 2, 1, 1, 1, 1, 1, 1, 0 ]
Time: 15.070
> time [ CohomologicalDimension(I, 4) : I in irr ];
[ 2, 1, 1, 2, 3, 2, 2, 0 ]
Time: 99.500
> time CohomologicalDimensions(irr[1], 6);
[ 0, 1, 2, 2, 3, 6 ]
Time: 139.270
```

91 CHARACTERS OF FINITE GROUPS

Chapter 91
CHARACTERS OF FINITE GROUPS

Assume that G is a finite group of exponent m with k conjugacy classes of elements. The operators discussed here are concerned with the ring of *class functions on* G, defined to be the ring of complex-valued functions on G that are constant on conjugacy classes. This ring is made into a $\mathbf{C}$-algebra by identifying $c \in \mathbf{C}$ with the constant function that is c everywhere. In fact we will restrict ourselves to functions with values that are elements of cyclotomic fields.

Elements of the ring, ie. objects of type `AlgChtrElt`, are represented by the k values (elements of some cyclotomic field $\mathbf{Q}(\zeta_n)$) on the classes. The numbering of those elements matches the numbering of the classes as returned by `Classes` applied to the underlying group `G`: Thus `X[i]` is the value of the character `X` on the i-th class, ie. `Classes(G)[i]`.

91.1 Creation Functions

91.1.1 Structure Creation

`ClassFunctionSpace(G)`

`CharacterRing(G)`

Given a finite group G, create the ring of complex-valued class functions. This function will trigger the computation of the conjugacy classes of G if these are not yet known. Information about the irreducible characters is stored in the ring when it is computed.

91.1.2 Element Creation

The elementary constructions for class functions are listed. Other useful ways of defining class functions and characters are defined in sections discussing the permutation character, the (de)composition functions, and the sections on the conjugating, restricting and inducing of class functions.

`elt< R | a`$_1$`, ..., a`$_k$ *:parameters* `>`

`R ! [ a`$_1$`, ..., a`$_k$ `]`

Given the ring of class functions R of a finite group G with k conjugacy classes and k elements a_i contained in some common cyclotomic field, create a class function on G for which the value on the i-th class is equal to the i-th term a_i.

`Character` BoolElt *Default :* `false`

If `Character := true`, then the resulting character is flagged to be a proper character.

`R ! a`

Define a constant class function for the ring of class functions R of the group G. Here a is allowed to be an integer, a rational field element or a cyclotomic field element.

`Id(R)`

`Identity(R)`

`One(R)`

`PrincipalCharacter(G)`

Given the finite group G or its ring of class functions R, create the principal character (which takes on the value 1 on every element of G).

`Zero(R)`

Given a ring of class functions R create its zero element (which is the class function that takes on the value 0 on every element of the group).

91.1.3 The Table of Irreducible Characters

The function `CharacterTable` can be invoked to determine the complete or a partial table of irreducible characters on a finite group G. If necessary, an existing character table can be supplemented by another call to the function. The known irreducible characters are stored in the character ring of G.

The default algorithm for character tables is Unger's Induce/Reduce algorithm, described in detail in [Ung06]. Computing character tables assumes that the classes of the group can be computed and stored, along with a class representative for each class, as well as the power map of the group. Unger's algorithm constructs elementary subgroups of the group (where elementary means direct product of cyclic group with p-group), and constructs characters of these subgroups (using Conlon's algorithm for the character table of the p-part of the group). Following Brauer's theorem on induced characters, these characters are induced to the full group and all irreducible characters of the full group are found in the integral span of the induced characters (using LLL reduction to maintain a manageable basis of the character space found). The two potentially most time-consuming parts of the algorithm are the computation of fusion in the elementary subgroups, which is necessary for induction, and repeated use of LLL to maintain a basis of the space of generalised characters found to date. Arithmetic with generalised characters is done quickly by using a finite field representation of these characters, as in Dixon's work [Dix67], but here the prime used may be twice the length of Dixon's.

The Dixon-Schneider algorithm for computing character tables [Dix67, Sch90] is also available. See below for details on how to access it. This was the previous default algorithm for character tables. As indicated above, Conlon's algorithm for the character table of a p-group [Con90] is also implemented, and may be accessed directly as indicated below.

The functions in this section return character tables, which are enumerated sequences of characters that are flagged to allow printing in a special format.

`KnownIrreducibles(G)`

Given a finite group G, or a class function space R of G, return the table of irreducible characters currently stored. Such a table is a sequence of characters with specially formatted printing.

It should be noted that characters are stored with some information about whether they are characters, generalized characters or class functions. From this information, it is often possible to deduce with little effort that a character is, in fact, irreducible. When a new irreducible is found this way, it is immediately inserted into the table of irreducible characters.

`CharacterTable(G :`*parameters*`)`

Construct the table of ordinary irreducible characters for the group G.

`Al`	MonStgElt	*Default :* "*Default*"

This parameter controls the algorithm used. The string `"DS"` forces use of the Dixon-Schneider algorithm. The string `"IR"` forces the use of Unger's Induce/Reduce algorithm [Ung06]. The string `"Conlon"` forces the use of Conlon's algorithm. This last is only valid when the group is a p-group. The `"Default"` algorithm is to use Dixon-Schneider for groups of order ≤ 5000, Conlon's algorithm for larger p-groups and Unger's algorithm for other groups. This may change in future.

`DSSizeLimit`	RngIntElt	*Default :* 0

When the algorithm selected is either "Default" or "IR", a positive value n for `DSSizeLimit` means that before using ithe Induce/Reduce algorithm, the full possible character space is split by some passes of the Dixon-Schneider algorithm, restricted to using class matrices corresponding to conjugacy classes with size at most n.

`CharacterTableDS(G :`*parameters*`)`

`ClassMatrices`	SeqEnum	*Default :* []
`ClassMatrixLimit`	RngIntElt	*Default :* ∞
`ClassSizeLimit`	RngIntElt	*Default :* ∞
`MinChars`	RngIntElt	*Default :* ∞
`Modulus`	RngIntElt	*Default :* 0

Given a finite group G, construct the table of irreducible characters of G using the Dixon-Schneider algorithm [Dix67, Sch90]. The conjugacy classes of G will be computed if necessary; if the user wishes to exert control over the computation of the classes, the function `Classes` can be invoked on G before calling `CharacterTableDS`.

There are several optional parameters for `CharacterTableDS`. Assigning a non-negative integer value m to `MinChars`, indicates that the computation is to be terminated as soon as m or more additional characters have been found. By setting `ClassMatrixLimit` to a positive integer n, the user can limit the number of class matrices used. If the calculation of all irreducible characters (or at least m of them

if `MinChars` has been set) cannot be completed by using k class matrices, an incomplete character table is returned. Setting `ClassSizeLimit` to an integer n restricts the algorithm to computing class matrices corresponding to classes of size at most n. Again, if the calculation of all irreducible characters cannot be completed using class matrices from these small classes, an incomplete character table is returned. The optional argument `ClassMatrices` (a sequence of integers in the range $1 \dots k$, where k is the number of conjugacy classes of G) can be used to specify a preference for the order in which class matrices should be used. Finally the `Modulus` argument can be used to set which prime field is used for the internal computations of the Dixon-Schneider algorithm. The value used must be a prime equivalent to 1 modulo the group exponent and greater than twice the square root of the group order.

The second return argument is the sequence of unsplit character spaces remaining at the point the algorithm terminated. This is empty when the algorithm returns a complete character table, but may be useful when an incomplete character table is returned.

`Basis(R)`

The function `Basis` takes the character ring R of G as input and performs the same computation to find the basis for R consisting of the irreducible characters.

Both `CharacterTable` and `Basis` return a character table, which is an enumerated sequence of elements of the character ring over G (if necessary, the ring is created) that only differs from arbitrary sequences of class functions with respect to printing.

`CharacterTableConlon(G)`

Compute the character table of group G using Conlon's algorithm for p-groups. The group group G must thus be a p-group for some prime p.

`LinearCharacters(G)`

Given a finite group G, determine the (partial) character table containing only the linear characters.

`CharacterDegrees(G)`

`CharacterDegrees(G)`

`CharacterDegrees(G)`

`CharacterDegrees(G)`

Returns the degrees of the ordinary irreducible characters of the finite group G. The sequence returned has form $[< d_1, c_1 >, < d_2, c_2 >, \dots]$ where c_i is the number of characters of degree d_i. For p-groups Slattery's algorithm is used, for other soluble groups Conlon's counting algorithm is used, and for insoluble groups the character table of G is computed.

`CharacterDegrees(G, z, p)`

Returns the degrees of the absolutely irreducible characters of G lying over a faithful linear character of $\langle z \rangle$, where z is a central element of G and p is zero or a prime.

`CharacterDegreesPGroup(G)`

Returns the degrees of the ordinary irreducible characters of the finite p-group G. The sequence returned has form $[c_0, c_1, c_2 \ldots]$, where c_i is the number of characters of degree p^i. Slattery's counting algorithm is used.

`RationalCharacterTable(G)`

Returns a sequence of minimal rational characters of G. These are the sums of the Galois orbits on the character table of G.

Example H91E1

We use the `CharacterTable` function and print the resulting table. Character tables have a special print format.

```
> G := Alt(5);
> CT := CharacterTable(G);
> CT;


Character Table of Group G
--------------------------


---------------------------
Class |   1  2  3    4    5
Size  |   1 15 20   12   12
Order |   1  2  3    5    5
---------------------------
p  =  2   1  1  3    5    4
p  =  3   1  2  1    5    4
p  =  5   1  2  3    1    1
---------------------------
X.1   +   1  1  1    1    1
X.2   +   3 -1  0   Z1 Z1#2
X.3   +   3 -1  0 Z1#2   Z1
X.4   +   4  0  1   -1   -1
X.5   +   5  1 -1    0    0


Explanation of Character Value Symbols
--------------------------------------

# denotes algebraic conjugation, that is,
```

```
#k indicates replacing the root of unity w by w^k

Z1     = (CyclotomicField(5: Sparse := true)) ! [
RationalField() | 1, 0, 1, 1 ]

> CT[2];
( 3, -1, 0, zeta(5)_5^3 + zeta(5)_5^2 + 1, -zeta(5)_5^3 -
  zeta(5)_5^2 )
> CT[2]:Minimal;
( 3, -1, 0, Z1, Z1#2 )
```

The character table is represented as a sequence of characters. Non-rational values in a character table are printed symbolically. This same printing can be forced on an individual character by using `Minimal` printing.

Example H91E2

The `CharacterTable` algorithm can handle quite large groups. We illustrate this by computing the character table of the almost simple group $P\Gamma U_5(4)$.

```
> G := PGammaU(5,4);
> G;
Permutation group G acting on a set of cardinality 17425
Order = 2^22 * 3^2 * 5^5 * 13 * 17 * 41
> time CT := CharacterTable(G);
Time: 205.150
> #CT;
160
> Degree(CT[160]);
5227500
```

91.2 Character Ring Operations

91.2.1 Related Structures

`Parent(R)` `Category(R)`

`Group(R)`

Given the ring R of class functions on a finite group G, return G.

`Centre(x)`

The centre of the character x of G, i.e. the subgroup of G consisting of those classes C of G for which $|x(g)|$, g in C, is equal to the degree of x.

`CoefficientField(x)`

The (minimal) coefficient field $\mathbf{Q}_m$ of the class function x.

`Kernel(x)`

The kernel of the character x of G, i.e. the normal subgroup of G consisting of those elements g for which $x(g) = x(1)$.

91.3 Element Operations

91.3.1 Arithmetic

In the list of arithmetic operations below x and y denote class functions in the same ring, and a denotes a scalar, which is any element coercible into a cyclotomic field. Also, j denotes an integer.

`+ y` `- y`

`x + y` `x - y` `x * y`

`a * x` `x ^ j`

91.3.2 Predicates and Booleans

The following Boolean-valued functions are available. Note that with the exception of `in`, `notin`, `IsReal` and `IsFaithful`, these functions use the table of irreducible characters, which will be created if it is not yet available.

`x in y`

Returns `true` if the inner product of class functions x and y is non-zero, otherwise `false`. If x is irreducible and y is a character, this tests whether or not x is a constituent of y.

`x notin y`

Returns `true` if the inner product of class functions x and y is zero, otherwise `false`. If x is irreducible and y is a character, this tests whether or not x is not a constituent of y. Returns `true` if the character x is not a constituent of the character y, otherwise `false`.

`a in F` `a notin F`

`x eq y` `x ne y`

`IsCharacter(x)`

Returns `true` if the class function x is a character, otherwise `false`. A class function is a character if and only if all inner products with the irreducible characters are non-negative integers.

`IsGeneralizedCharacter(x)`

Returns `true` if the class function x is a generalized character, otherwise `false`. A class function is a generalized character if and only if all inner products with the irreducible characters are integers.

`IsIrreducible(x)`

Returns `true` if the character x is an irreducible character, otherwise `false`.

`IsLinear(x)`

Returns `true` if the character x is a linear character, otherwise `false`.

`IsFaithful(x)`

Returns `true` if the character x is faithful, i.e. has trivial kernel, otherwise `false`.

`IsReal(x)`

Returns `true` if the character x is a real character, i.e. takes real values on all of the classes of G, otherwise `false`.

`IsOne(x)` `IsMinusOne(x)` `IsZero(x)`

91.3.3 Accessing Class Functions

In this subsection T is a character table, and x is any class function. A character table is an enumerated sequence of characters that has a special print function attached. In particular, its entries can be accessed with the ordinary sequence indexing operations.

`T[i]`

Given the table T of ordinary characters of G, return the i-th character of G, where i is an integer in the range $[1...k]$.

`T[i][j]`

The value of the i-th irreducible character (from the character table T) on the j-th conjugacy class of G.

`#T`

Given a character table T (or any sequence of characters), return the number of entries.

`x(g)`
`g @ x`

The value of the class function x on the element g of G.

`x[i]`

The value of the class function x on the i-th conjugacy class of G.

`#x`

Given a class function x on G return its length (which equals the number of conjugacy classes of the group G).

91.3.4 Conjugation of Class Functions

`x ^ g`

Given a class function x on a normal subgroup N of the group G, and an element g of G, construct the conjugate class function x^g of x which is defined as follows: $x^g(n) = x(g^{-1}ng)$, for all n in N.

`x ^ H`

Given a class function x on a normal subgroup N of the group G, and a subgroup H of G, construct the sequence of conjugates of x under the action of the subgroup H. The action of an element of H on x is that defined in the previous function.

`GaloisConjugate(x, j)`

Let $\mathbf{Q}(x)$ be the subfield of $\mathbf{Q}(\zeta_m)$ generated by $\mathbf{Q}$ and the values of the G-character x. This function returns the Galois conjugate x^j of x under the action of the element of the Galois group $\mathrm{Gal}(\mathbf{Q}(x)/\mathbf{Q})$ determined by the integer j. The integer j must be coprime to m.

`GaloisOrbit(x)`

Let $\mathbf{Q}(x)$ be the subfield of $\mathbf{Q}(\zeta_m)$ generated by $\mathbf{Q}$ and the values of the G-character x. This function returns the sequence of Galois conjugates of x under the action of the Galois group $\mathrm{Gal}(\mathbf{Q}(x)/\mathbf{Q})$.

`ClassPowerCharacter(x, j)`

Given a class function x on the group G and a positive integer j, construct the class function x^j which is defined as follows: $x^j(g) = x(g^j)$.

91.3.5 Functions Returning a Scalar

`Degree(x)`

The degree of the class function x, i.e. the value of x on the identity element of G.

`InnerProduct(x, y)`

The inner product of the class functions x and y, where x and y are class functions belonging to the same character ring.

`Order(x)`

Given a linear character of the group G, determine the order of x as an element of the group of linear characters of G.

`Norm(x)`

Norm of the class function x (which is the inner product with itself).

`Schur(x, k)`

`Indicator(x)`

Given class function x and a positive integer k, return the generalised Frobenius–Schur indicator which is defined as follows: Suppose g is some element of G, and set $T_k(g) = |\{h \in G | h^k = g\}|$. The value of `Schur(x, k)` is the coefficient a_x in the expression $T_k = \sum_{x \in \mathrm{Irr}(G)} a_x x$.

The call `Indicator(x)` is equivalent to `Schur(x,2)`.

`StructureConstant(G, i, j, k)`

The structure constant $a_{i,j,k}$ for the centre of the group algebra of the group G. If K_i is the formal sum of the elements of the i-th conjugacy class, $a_{i,j,k}$ is defined by the equation $K_i * K_j = \sum_k a_{i,j,k} * K_k$.

91.3.6 The Schur Index

MAGMA incorporates functions for computing the Schur index of an ordinary irreducible character over various number fields and local fields. The routines below are all based on the function `SchurIndices(x)`, which computes the Schur Indices of the given character over all the completions of the rationals.

The algorithm is based on calculations with characters, groups and fields, and does not compute representations.

The algorithm was devised by Gabi Nebe and Bill Unger, with code written by Bill Unger. The extension to compute a Schur index over a number field was written by Claus Fieker.

`SchurIndex(x)`

`SchurIndex(x, F)`

The Schur index of the character x over the given field. When no field is given, the Schur index over the rationals is returned. The character x must be a complex irreducible character. The field F must be an absolute number field.

`SchurIndices(x)`

`SchurIndices(x, F)`

`SchurIndices(C, s, F)`

Compute the Schur indices of the character x over the completions of the given field. The character x must be a complex irreducible character. The field F must be an absolute number field. When no field is specified the rational field is assumed. The last form takes the character field, C, and the output from `SchurIndices(x)`, s, as well as a number field. This is sufficient to compute the Schur indices over the number field without repeating group and character computations when a number of fields are being considered for one character.

The return value is a sequence of pairs. Each pair gives a completion at which the Schur index is not 1, followed by the Schur index over the complete field. For the rational field, a completion is specified by an integer. The integer zero specifies the archimedean completion (the real numbers), while a prime p specifies the p-adic field $\mathbf{Q}_p$. When a number field is given, the completions are specified by a place of the field, an object of type `PlcNumElt`.

If the character has Schur index 1 over the given field the return value will be an empty sequence. Otherwise the Schur index over the given field is the least common multiple of the second entries of the tuples returned.

Example H91E3

We first look at the faithful irreducible character of the Dihedral group of order 8. It has Schur index 1.

```
> T := CharacterTable(SmallGroup(8, 3));
> T[5];
( 2, -2, 0, 0, 0 )
> SchurIndex(T[5]);
1
> SchurIndices(T[5]);
[]
```

The corresponding character of the quaternion group of order 8 has non-trivial Schur index.

```
> T := CharacterTable(SmallGroup(8, 4));
> T[5];
( 2, -2, 0, 0, 0 )
> SchurIndex(T[5]);
2
> SchurIndices(T[5]);
[ <0, 2>, <2, 2> ]
```

The Schur index is 2 over the real numbers and Q_2. For all odd primes p, the Schur index over Q_p is 1. We look at the Schur index of this character over some number fields. First we look at some cyclotomic fields.

```
> [SchurIndex(T[5], CyclotomicField(n)):n in [3..20]];
[ 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1 ]
> SchurIndices(T[5], CyclotomicField(7));
[ <Place at Prime Ideal
Two element generators:
  [2, 0, 0, 0, 0, 0]
  [1, 1, 0, 1, 0, 0], 2>, <Place at Prime Ideal
Two element generators:
  [2, 0, 0, 0, 0, 0]
```

```
  [1, 0, 1, 1, 0, 0], 2> ]
```

The cyclotomic field of order 7 gives Schur index 2. An archimedean completion of this field is necessarily the field of complex numbers, hence no infinite places give Schur index greater than 1. There are now two 2-adic completions which give Schur index 2.

```
> P<t> := PolynomialRing(Rationals());
> F := ext<Rationals()|t^3-2>;
> SchurIndex(T[5], F);
2
> SchurIndices(T[5], F);
[ <1st place at infinity, 2>, <Place at Prime Ideal
Two element generators:
  [2, 0, 0]
  [0, 1, 0], 2> ]
```

For the non-normal field F, one archimedean completion is real, the other complex. Thus the real field features in the output of `SchurIndices`, along with the 2-adic completion.

Example H91E4

We will use a general construction for a character with given Schur index over the rationals to construct a character with Schur index 6. Given an integer $n \geq 1$, we select a prime p such that $p = kn + 1$ where k and n are coprime. We take an integer a such that a has order n modulo p. We then consider the metacyclic group

$$G = \langle < x, y | x^{n^2}, y^p, y^x = y^a \rangle.$$

The order of G is n^2p. The subgroup of G generated by x^n and y is cyclic, normal and self-centralizing in G with order np. If λ is any faithful linear character of this subgroup, then λ^G is an irreducible character of G with Schur index n over the rational field. The correctness of this construction is proved in Lemma 3 of [Tur01].
We construct G in two stages. First as a finitely presented group as described above. Then we convert to a PC-presentation for further computations. We take $n = 6$, $p = 7$ and $a = 3$.

```
> G1 := Group<x,y|x^36, y^7, y^x = y^3>;
> G, f := SolubleQuotient(G1, 36*7);
> x := f(G1.1); y := f(G1.2);
> C := sub<G|x^6,y>;
> IsCyclic(C);
true
> IsNormal(G, C);
true
> Centralizer(G,C) eq C;
true
> exists(l){l:l in LinearCharacters(C)|IsFaithful(l)};
true;
> c := Induction(l, G);
> IsIrreducible(c);
true
> Degree(c);
```

```
6
> CharacterField(c);
Cyclotomic Field of order 3 and degree 2 in sparse
representation
> SchurIndex(c);
6
```

The construction of the previous example is used in the following two intrinsics.

`SchurIndexGroup(n: parameters)`

`Prime` RngIntElt *Default :*

Return a group having a faithful character with Schur index n over the rational field. The construction used is as in the previous example. The parameter `Prime` may be used to supply the prime p. (The necessary conditions on p are not checked. If these conditions are not met, an error is possible.) If `Prime` is not set, then the least prime meeting the conditions is used.

`CharacterWithSchurIndex(n: parameters)`

`Prime` RngIntElt *Default :*

Return a character with Schur index n over the rational field. The construction used is as in the previous example. The second return value is the group of the character, equal to `SchurIndexGroup(n)`. The parameter `Prime` is as for `SchurIndexGroup`.

91.3.7 Attribute

`AssertAttribute(x, "IsCharacter", b)`

Procedure that, given a class function x and a Boolean value b, stores with x the information that the value of the predicate `IsCharacter(x)` equals b.

91.3.8 Induction, Restriction and Lifting

`Induction(x, G)`

Given a class function x on the subgroup H of the group G, construct the class function obtained by induction of x to G. Note that if x is a character of H, then `Induction(x, G)` will return a character of G.

The `Induction` command may also be used to induce a sequence of characters of a particular subgroup (such as a character table) to the given supergroup.

`LiftCharacter(c, f, G)`

Given a class function c of the quotient group Q of the group G and the natural homomorphism $f : G \to Q$, lift c to a class function of G.

`LiftCharacters(T, f, G)`

Given a sequence T of class functions of the quotient group Q of the group G and the natural homomorphism $f : G \to Q$, lift T to a sequence of corresponding class functions of G. Since a character table is just a sequence of class functions which is printed in a special way, this intrinsic may also be applied to it.

`Restriction(x, H)`

Given a class function x on the group G and a subgroup H of G, construct the restriction of x to H (a class function). Note that if x is a character of G, then `Restriction(x, H)` will return a character of H.

91.3.9 Symmetrization

See [Mur58] or [Fra82] for more details.

`Symmetrization(x, p)`

Given a class function x and a partition p of n ($2 \leq n \leq 6$), this function returns the symmetrized character with respect to p; the partition must be specified in the form of a sequence of positive integers (adding up to n).

`OrthogonalComponent(x, p)`

Given a class function x and a partition p of n ($2 \leq n \leq 6$), this function returns the Murnaghan component of the orthogonal symmetrization of x with respect to p; the partition must be specified in the form of a sequence of positive integers (adding up to n). Here x may not be a linear character, and its Frobenius–Schur indicator must be 1.

`SymplecticComponent(x, p)`

Given a class function x and a partition p of n ($2 \leq n \leq 6$), this function returns the Murnaghan component of the symplectic symmetrization of x with respect to p; the partition must be specified in the form of a sequence of positive integers (adding up to n). Here x may not be a linear character, and its Frobenius–Schur indicator must be -1.

`SymmetricComponents(x, n)`

Given a class function x and an integer n, return the set of symmetrizations of x by all partitions of m with $2 < m \leq n \leq 5$.

`OrthogonalComponents(x, n)`

Given a class function x, return the set of Murnaghan components for orthogonal symmetrizations of x by all partitions of m with $2 < m \leq n \leq 6$. Here x may not be a linear character, and its Frobenius–Schur indicator must be 1.

`SymplecticComponents(x, n)`

Given a class function x, return the set of Murnaghan components for symplectic symmetrizations of x by all partitions of m with $2 < m \leq n \leq 5$. Here x may not be a linear character, and its Frobenius–Schur indicator must be -1.

91.3.10 Permutation Character

`PermutationCharacter(G)`

Given group G represented as a permutation group, construct the character of G afforded by the defining permutation representation of G.

`PermutationCharacter(G, H)`

Given a group G and some subgroup H of G, construct the character of G afforded by the permutation representation of G given by the action of G on the right cosets of H in G.

91.3.11 Composition and Decomposition

`Composition(T, q)`

Given a sequence or table of characters T for the group G and a sequence q of k elements of $\mathbf{Q}(\zeta_m)$ (possibly $\mathbf{Q}$), create the class function $q_1 * T_1 + \cdots + q_k * T_k$, where T_i is the i-th character in T.

`Decomposition(T, y)`

Given a sequence or table of class functions T for G of length l and a class function y on G, attempt to express y as a linear combination of the elements of T.

The function returns two values: a sequence $q = [q_1, \ldots, q_l]$ of cyclotomic field elements and a class function z. For $1 \leq i \leq l$, the i-th term of q is defined to be the ratio of inner products $(y, T_i)/(T_i, T_i)$, where T_i is the i-th entry of T. The sequence q determines a class function $x = q_1 \cdot T_1 + \cdots + q_l \cdot T_l$ which will equal y if T is the complete table of irreducible characters. The difference $z = y - x$ is the second return value. If the entries in T are mutually orthogonal, then z is the zero class function if and only if y is a linear combination of the T_i.

91.3.12 Finding Irreducibles

A common approach to finding the irreducible characters of a group is to start with an irreducible character and generate new characters by applying `SymmetricComponents`, `OrthogonalComponents` or `SymplecticComponents`. Then, by examining norms and inner products, it is often possible to identify irreducible characters or at least characters with smaller norms. There are two MAGMA intrinsics available to help with this task.

`RemoveIrreducibles(I, C)`

Remove occurrences of the irreducible characters in the sequence I from the characters in the sequence C and look for characters of norm 1 among the reduced characters. Return a sequence of new irreducibles found and the sequence of reduced characters.

`ReduceCharacters(I, C)`

Make the norms of the characters in the sequence C smaller by computing the differences of appropriate pairs. Return a sequence of new irreducibles found and a sequence of reduced characters.

Example H91E5

We conclude this section with an example, showing how the above functions can be used to construct the character table for A_5 (compare Isaacs, p64), using only characters on subgroups.

```
> A := AlternatingGroup(GrpPerm, 5);
> R := CharacterRing(A);
```

The first character will be the principal character

```
> T1 := R ! 1;
> T1;
( 1, 1, 1, 1, 1 )
```

Next construct the permutation character

```
> pc := PermutationCharacter(A);
> T2 := pc - T1;
> InnerProduct(pc, T1), InnerProduct(T2, T2);
1 1
> T2;
( 4, 0, 1, -1, -1 )
```

It follows that pc - T1 is an irreducible character

```
> B := Stabilizer(A, 5);
> r := RootOfUnity(3, CyclotomicField(3));
> S := CharacterRing(B);
> lambda := S ! [1, 1, r, r^2 ];
> IsLinear(lambda);
true
```

This defines a linear character on a subgroup of index 5 in A

```
> T3 := Induction(lambda, A);
> InnerProduct(T3, T3);
1
> T3;
```

```
( 5, 1, -1, 0, 0 )
```

Finally we use characters on the cyclic subgroup of order 5:

```
> K := sub<A |  (1,2,3,4,5) >;
> Y := CharacterTable(K);
> Y;

Character Table of Group K
--------------------------

-------------------------------
Class |   1    2    3    4    5
Size  |   1    1    1    1    1
Order |   1    5    5    5    5
-------------------------------
p  =  5   1    1    1    1    1
-------------------------------
X.1   +   1    1    1    1    1
X.2   0   1   Z1 Z1#2 Z1#3 Z1#4
X.3   0   1 Z1#2 Z1#4   Z1 Z1#3
X.4   0   1 Z1#3   Z1 Z1#4 Z1#2
X.5   0   1 Z1#4 Z1#3 Z1#2   Z1

Explanation of Symbols:
-----------------------

# denotes algebraic conjugation, that is,
# k indicates replacing the root of unity w by w^k

Z1     = -1 - zeta_5 - zeta_5^2 - zeta_5^3

> mu := Induction(Y[2], A);
```

We subtract what we already know from mu and get a new irreducible. We use decomposition with respect to a sequence.

```
> _, T4 := Decomposition([T1, T2, T3], mu);
> InnerProduct(T4, T4);
1
> T4;
( 3, -1, 0, (1 + zeta_5^2 + zeta_5^3), (-zeta_5^2 - zeta_5^3) )
> T5 := GaloisConjugate(T4, 2);
> T5;
( 3, -1, 0, (-zeta_5^2 - zeta_5^3), (1 + zeta_5^2 + zeta_5^3) )
```

Compare this to the standard character table:

```
> CharacterTable(A);

Character Table of Group A
--------------------------

--------------------------
```

```
Class |   1  2  3    4    5
Size  |   1 15 20   12   12
Order |   1  2  3    5    5
---------------------------
p  =  2   1  1  3    5    4
p  =  3   1  2  1    5    4
p  =  5   1  2  3    1    1
---------------------------
X.1   +   1  1  1    1    1
X.2   +   3 -1  0   Z1 Z1#2
X.3   +   3 -1  0 Z1#2   Z1
X.4   +   4  0  1   -1   -1
X.5   +   5  1 -1    0    0

Explanation of Symbols:
-----------------------

#  denotes algebraic conjugation, that is,
# k indicates replacing the root of unity w by w^k

Z1     =(1 + zeta_5^2 + zeta_5^3)
```

91.3.13 Brauer Characters

MAGMA has some support for the calculation of Brauer characters. These functions are noted in this section. We anticipate considerable change to the functionality described here in the near future.

A Brauer character modulo p in MAGMA is represented as a class function (that is, element of a character ring) which is zero on p-singular group elements. In this format the standard character operations of addition, multiplication, induction and restriction all apply directly to Brauer characters as they do to other class functions.

Note that problems associated with choice of lifting from finite fields to complex roots of unity have not yet been dealt with.

`BrauerCharacter(x, p)`

The Brauer character modulo the prime p obtained by setting the value of x on p-singular elements to be zero.

`Blocks(T, p)`

When T is the full ordinary character table of a group, return the partition of T into p-blocks, where p is a given prime. The partition is returned as a sequence of sets of integers which give the blocks by the positions of the characters in T. The second return value is the corresponding sequence of defects of the blocks. The blocks are ordered first by decreasing defect, second by first character in the block.

Example H91E6

We give an example of the use of these Brauer character functions. We consider the 3-modular characters of the Higman-Sims simple group.

```
> load hs176;
> T := CharacterTable(G);
> Blocks(T,3);
[
  { 1, 2, 5, 10, 18, 19, 21, 23, 24 },
  { 3, 4, 6, 7, 11, 12, 14, 15, 20 },
  { 8, 13, 16 },
  { 9 },
  { 17 },
  { 22 }
]
[ 2, 2, 1, 0, 0, 0 ]
```

The characters $T[8], T[13], T[16]$ are the ordinary irreducible characters in a 3-block of defect one. In such a small block the two ordinary irreducibles of minimal degree will restrict to modular irreducibles.

```
> [Degree(T[i]): i in [8, 13, 16]];
[ 231, 825, 1056 ]
> BrauerCharacter(T[8], 3);
( 231, 7, -9, 0, 15, -1, -1, 6, 1, 1, 0, 0, 0, -1, -1, -1,
2, 1, 0, 0, 0, 0, 0, 0 )
> BrauerCharacter(T[13], 3);
( 825, 25, 9, 0, -15, 1, 1, 0, -5, 0, 0, 0, -1, 1, 1, 1, 0,
-1, 0, 0, 0, 0, 0, 0 )
> $1 + $2 eq BrauerCharacter(T[16], 3);
true
```

The projective indecomposable characters corresponding to these Brauer irreducible characters are as follows.

```
> T[8] + T[16];
( 1287, 39, -9, 0, 15, -1, -1, 12, -3, 2, 0, 0, -1, -1, -1,
-1, 4, 1, 0, 0, 0, 0, 0, 0 )
> T[13] + T[16];
( 1881, 57, 9, 0, -15, 1, 1, 6, -9, 1, 0, 0, -2, 1, 1, 1, 2,
-1, 0, 0, 0, 0, 0, 0 )
```

91.4 Bibliography

[**Con90**] S. B. Conlon. Calculating characters of p-groups. *J. Symbolic Comp.*, 9:535–550, 1990.

[**Dix67**] J. D. Dixon. High–speed computation of group characters. *Numerische Mathematik*, 10:446–450, 1967.

[**Fra82**] J. S. Frame. Recursive computation of tensor power components. *Bayreuth. Math. Schr.*, (10):153–159, 1982.

[**Mur58**] F. D. Murnaghan. The orthogonal and symplectic groups. *Comm. Dublin Inst. Adv. Studies. Ser. A, no.*, 13:146, 1958.

[**Sch90**] G. J. A. Schneider. Dixon's Character Table Algorithm Revisited. *J. Symbolic Computation*, 9:601–606, 1990.

[**Tur01**] A. Turull. Schur indices of perfect groups. *Proc. Amer. Math. Soc.*, 130(2): 367–370, 2001.

[**Ung06**] W.R. Unger. Computing the character table of a finite group. *J. Symbolic Comp.*, 41(8):847–862, 2006.

92 REPRESENTATIONS OF SYMMETRIC GROUPS

Chapter 92
REPRESENTATIONS OF SYMMETRIC GROUPS

92.1 Introduction

This chapter describes functions available in MAGMA for computations concerning the non modular representation theory of the symmetric group and the alternating group. It is possible to compute different matrix representations, complete character tables using special routines, single irreducible characters or the value of an irreducible character on an element of the group.

92.2 Representations of the Symmetric Group

For the symmetric group of degree n the irreducible representations can be indexed by partitions of weight n. For more information on partitions see Section 145.2.

92.2.1 Integral Representations

It is possible to define representing matrices of the symmetric group over the integers.

`SymmetricRepresentation(pa, pe)`

`Al` MONSTGELT *Default :* "*JamesKerber*"

Given a partition *pa* of weight n and a permutation *pe* in a symmetric group of degree n, return an irreducible representing matrix for *pe*, indexed by *pa*, over the integers. If `Al` is set to the default `"JamesKerber"` then the method described in [JK81] is used. If `Al` is set to `"Boerner"` the method described in the book of Boerner [Boe67] is used. If `Al` is set to `"Specht"` then the method used is a direct implementation of that used by Specht in his paper from 1935 [Spe35].

Example H92E1

We compute a representing matrix of a permutation using two different algorithms and check whether the results have the same character.

```
> a:=SymmetricRepresentation([3,2],Sym(5)!(3,4,5) : Al := "Boerner");a;
[ 0  0  1 -1  0]
[ 1  0  0 -1  0]
[ 0  1  0 -1  0]
[ 0  0  0 -1  1]
[ 0  0  0 -1  0]
> b:=SymmetricRepresentation([3,2],Sym(5)!(3,4,5) : Al := "Specht");b;
```

```
[ 0  1  0 -1  0]
[ 0  0  1  0 -1]
[ 1  0  0  0  0]
[ 0  0  0  0 -1]
[ 0  0  0  1 -1]
> IsSimilar(Matrix(Rationals(), a), Matrix(Rationals(), b));
true
```

The matrices are similar as they should be.

92.2.2 The Seminormal and Orthogonal Representations

The seminormal and orthogonal representations involve matrices which are not necessarily integral. The method MAGMA uses to construct these matrices is described in [JK81, Section 3.3];

`SymmetricRepresentationSeminormal(pa, pe)`

Given a partition *pa* of weight n and a permutation *pe* in a symmetric group of degree n, return the matrix of the seminormal representation for *pe*, indexed by *pa*, over the rationals.

`SymmetricRepresentationOrthogonal(pa, pe)`

Given a partition *pa* of weight n and a permutation *pe* in a symmetric group of degree n, return the matrix of the orthogonal representation for *pe*, indexed by *pa*. An orthogonal basis is used to compute the matrix which may have entries in a cyclotomic field.

Example H92E2

We compare the seminormal and orthogonal representations of a permutation and note that they are similar.

```
> g:=Sym(5)!(3,4,5);
> a:=SymmetricRepresentationSeminormal([3,2],g);a;
[-1/2    0 -3/4    0    0]
[   0  1/2    0  3/4    0]
[   1    0 -1/2    0    0]
[   0  1/3    0 -1/6  8/9]
[   0    1    0 -1/2 -1/3]
> b:=SymmetricRepresentationOrthogonal([3,2],g);b;
[-1/2 0 zeta(24)_8^2*zeta(24)_3 + 1/2*zeta(24)_8^2 0 0]
[0 1/2 0 -zeta(24)_8^2*zeta(24)_3 - 1/2*zeta(24)_8^2 0]
[-zeta(24)_8^2*zeta(24)_3 - 1/2*zeta(24)_8^2 0 -1/2 0 0]
[0 -1/3*zeta(24)_8^2*zeta(24)_3 - 1/6*zeta(24)_8^2 0
 -1/6 2/3*zeta(24)_8^3 - 2/3*zeta(24)_8]
[0 2/3*zeta(24)_8^3*zeta(24)_3 + 1/3*zeta(24)_8^3
+ 2/3*zeta(24)_8*zeta(24)_3 + 1/3*zeta(24)_8 0
```

```
    -1/3*zeta(24)_8^3 + 1/3*zeta(24)_8 -1/3]
> IsSimilar(a,b);
true
```

They should both be of finite order, 3.

```
> IsOne(a^Order(g));
true
> IsOne(b^Order(g));
true
>
```

92.3 Characters of the Symmetric Group

92.3.1 Single Values

The method used to compute the value of a character on a permutation is the recursion formula of Murnaghan and Nakayama [JK81, p. 60], except when computing the value of a character on the identity permutation a formula for the dimension of the representation indexed by a partition is used, [JK81, p. 56].

`SymmetricCharacterValue(pa, pe)`

Computes the value of the irreducible character of the symmetric group of degree n indexed by the partition pa of weight n on the permutation pe. When computing the value of the character on the identity permutation, i.e. the degree of the character, the dimension (hook) formula is used on the partition pa.

92.3.2 Irreducible Characters

`SymmetricCharacter(pa)`

Return the character of the representation of the symmetric group of degree n indexed by the partition pa where pa has weight n.

92.3.3 Character Table

`SymmetricCharacterTable(d)`

Return the character table of the symmetric group of degree d.

92.4 Representations of the Alternating Group

92.5 Characters of the Alternating Group

There is a special routine which computes the character table of the alternating group, as well as routines which compute values of alternating characters and the characters themselves. These routines make use of the fact that in most cases the irreducible characters of the symmetric group, which can be computed quickly, are also irreducible in the alternating group. So the irreducible characters of the alternating group may be indexed by partitions in the same way as those of the symmetric group. As the restriction of the irreducible character indexed by the partition λ is equal to the restriction of the character indexed by the conjugate partition, we only need to take one from each of these pairs of partitions to form a full set of irreducible characters. When a partition is conjugate to itself the character of the symmetric group indexed by that partition is no longer irreducible but is the sum of two irreducibles. This method is described in [JK81].

92.5.1 Single Values

`AlternatingCharacterValue(pa, pe)`

Return the value of the character of the alternating group of degree n indexed by the partition pa of weight n on the permutation pe. The partition pa and its conjugate should be distinct.

`AlternatingCharacterValue(pa, i, pe)`

Return the value of the ith character of the alternating group of degree n indexed by the self conjugate partition pa of weight n on the permutation pe. Since there are two possible irreducible characters indexed by such partitions i must be either 1 or 2.

92.5.2 Irreducible Characters

`AlternatingCharacter(pa)`

Return the character of the alternating group of degree n indexed by the partition pa of weight n. The partition pa and its conjugate should be distinct.

`AlternatingCharacter(pa, i)`

Return the ith character of the alternating group of degree n indexed by the self conjugate partition pa of weight n. Since there are two possible irreducible characters indexed by such partitions i must be either 1 or 2.

92.5.3 Character Table

`AlternatingCharacterTable(d)`

Returns the character table of the alternating group of degree d.

92.6 Bibliography

[**Boe67**] H. Boerner. *Darstellungen von Gruppen. 2. Aufl.* Berlin-Heidelberg-New York: Springer-Verlag. XIV, 317 S. , 1967.

[**JK81**] Gordon James and Adalbert Kerber. *The representation theory of the symmetric group*. Addison-Wesley Publishing Co., Reading, Mass., 1981. With a foreword by P. M. Cohn, With an introduction by Gilbert de B. Robinson.

[**Spe35**] Wilhelm Specht. Die irreduziblen Darstellungen der symmetrischen Gruppe. *Math. Z.*, 39:696–711, 1935.

93 MOD P GALOIS REPRESENTATIONS

Chapter 93
MOD P GALOIS REPRESENTATIONS

93.1 Introduction

This package provides tools to work with φ-modules over $k((u))$ where k is a finite field, and representations of the absolute Galois group of $k((u))$ with coefficients in a finite field. The main functionality of the package computes the semisimplification of a given φ-module, and the semisimplification of the Galois representation that is naturally attached to it. In particular, the slopes of the φ-module, corresponding to the tame inertia weights of the Galois representation, can be computed using this package.

93.1.1 Motivation

Let K be a p-adic field and let G_K be the absolute Galois group of K. Representations of this group naturally arise from geometry, namely from the p-adic étale cohomology of a scheme over K.

The study of these representations is a central topic in arithmetic, and a motivation for creating this package is the following: let V be a $\mathbf{Q}_p$-representation of G_K, *i.e.* a $\mathbf{Q}_p$-vector space endowed with a continuous, linear action of G_K. Now let $T \subset V$ be any $\mathbf{Z}_p$-lattice stable under the action of G_K. There always exists such a lattice. Moreover, the quotient T/pT has a natural structure of $\mathbf{F}_p$-representation of G_K. This representation depends on the choice of T, but its semisimplification $(T/pT)^{ss}$ does not, according to the Brauer-Nesbitt theorem. Recall the semisimplification of a representation is the direct sum of the composition factors appearing in any Jordan-Holder sequence of this representation. Therefore, it is an interesting question to determine properties of $(T/pT)^{ss}$ in terms of V. Although the Fontaine-Laffaille theory completely addresses this question for some V, the general case remains an open question. Some computations concerning this problem can be performed in Magma using this package.

93.1.2 Definitions

Let k be a finite field of characteristic p, and let $K = k((u))$ be the field of Laurent series with coefficients in k. Let $s \geq 0$ and $b \geq 2$ be integers. We define a "Frobenius" map σ on K by the following formula:

$$\sigma\left(\sum_{i\in\mathbf{Z}} a_i u^i\right) = \sum_{i\in\mathbf{Z}} a_i^{p^s} u^{bi}.$$

A φ-module over K is the data of a finite-dimensional K-vector space D, endowed with an endomorphism $\varphi : D \to D$ that is semilinear with respect to σ. This means that for all $\lambda \in K$, $x \in D$, we have the identity $\varphi(\lambda x) = \sigma(\lambda)\varphi(x)$.

A φ-module is said to be étale if the map φ is injective. A φ-module can be described by the matrix representing the action of φ on some basis of D, and it is étale if and only if this matrix is invertible.

93.1.3 Classification of φ-modules

Some φ-modules play a crucial role in the theory because they are the simple objects in the category of étale φ-modules over the maximal unramified extension K^{ur} of K.

Let $d \geq 1$, $h \in \mathbf{Z}$, $\lambda \in \bar{k}$. We define the φ-module $D(d, s, \lambda)$ as the φ-module of dimension d whose matrix in some basis is the companion matrix of the polynomial $T^d - u^h$. We also write $D(d, h) = D(d, h, 1)$. Note that in general there are several ways to extend the action of σ on K^{ur}, but we may only distinguish the cases where σ acts as identity on k, and the case where is does not. We say that a couple (d, h) is reduced if there is no divisor d' of d (except d) such that $\frac{b^{d'}-1}{b^d-1}$ is a divisor of h. The main classification results are the following:

If $\sigma \neq id$, the simple objects of the category of étale φ-modules over K^{ur} are the $D(d, h)$ for (d, h) reduced, and if $\sigma = id$, the simple objects of the category of étale φ-modules over K^{ur} are the $D(d, h, \lambda)$ for (d, h) reduced.

By definition, the slope of a simple φ-module isomorphic to $D(d, h, \lambda)$ is the rational number $\frac{h}{b^d-1}$, up to the equivalence relation "$x \sim y \Leftrightarrow \exists$m,n$\in$ N such that $b^m x - b^n y \in \mathbf{Z}$". With this equivalence relation, the definition does not depend on the choice of (d, h).

If D is a φ-module over K, the slopes of D are the collection of the slopes of the composition factors of $K^{ur} \otimes_K D$ (this notion does not depend on how σ is extended to K^{ur}). Note that even though the algorithms that we present can give decompositions over K, for most practical uses the knowledge of the slopes should be sufficient.

93.1.4 Connection with Galois Representations

Let us explain the link between Galois representations and φ-modules over K. In this section, we assume that σ is the classical Frobenius $x \mapsto x^p$. Let K^{sep} be a separable closure of K and let $G_K = Gal(K^{sep}/K)$ be the absolute Galois group of K.

A theorem of Katz states that there is an equivalence of categories between the étale φ-modules over K and the $\mathbf{F}_p$-representations of G_K.

Under this equivalence of categories, the φ-module $D(d, h)$ corresponds to the "fundamental character of level d" to the power h, ω_d^h, seen as a $\mathbf{F}_p$-representation. The figures of h in base p are called the tame inertia weights of the representation, because they describe the action of the tame inertia group on the representation. These weights can be recovered from the slope of the φ-module. It is worth noting that if F is a p-adic field whose residue field is k, and F_∞ is the extension of F generated by a compatible sequence of p^n-th roots of the uniformizer for all n, then G_{F_∞} is isomorphic to G_K. Moreover, the tame inertia weights of a $\mathbf{F}_p$-representation of G_F are the same as the tame inertia weights of its restriction to G_{F_∞}, seen as a representation of G_K. Hence, working with φ-modules will enable us to study representations of p-adic Galois groups.

93.2 φ-modules and Galois Representations in Magma

Let us now give an overview of the functionalities of the package.

93.2.1 φ-modules

93.2.1.1 Category

In Magma, φ-modules have type `PhiMod`. Elements of φ-modules have type `PhiModElt`.

93.2.1.2 Creation functions

`PhiModule(M)`

F SeqEnum *Default :* $[1, p]$

Create the φ-module whose matrix is given by M in some basis. The optional argument F describes the action of the Frobenius on coefficients: if $F = [s, b]$ then φ acts by $a \mapsto a^{p^s}$ on the residue field and maps the variable u to u^b. The default value is $[1, p]$ where p is the characteristic of the base field, corresponding to the absolute Frobenius.

`ElementaryPhiModule(S,d,h)`

F SeqEnum *Default :* $[1, p]$

Create the φ-module $D(d, s)$ whose matrix is the companion matrix of $T^d - u^s$.

`PhiModuleElement(x,D)`

Create the element of the φ-module D whose coordinates are given by the vector x.

93.2.1.3 Attributes of φ-modules

`Dimension(D)`

The dimension of a φ-module.

`CoefficientRing(D)`

The coefficient ring of a φ-module.

`FrobeniusMatrix(D)`

Return the matrix of the action of φ on D in the current basis.

93.2.1.4 Basic operations and properties of φ-modules

`IsEtale(D)`

Return `true` if the action of φ on D is injective. This is only possible up to the precision of the coefficient ring of D.

`ChangePrecision(~D, prec)`

Change the precision of the coefficient ring of D to `prec`.

`DirectSum(D1, D2)`

The direct sum of two φ-modules. The coefficient rings and Frobenius action on the coefficients must be the same.

`BaseChange(~D, P)`

Change the basis of D. The base change matrix is P, meaning that if G is the current matrix of φ, the new matrix will be $P^{-1}G\varphi(P)$.

`RandomBaseChange(~D)`

Randomly change the basis of D.

`Phi(D, x)`

Compute the image of $x \in D$ under the action of φ.

93.2.1.5 Reduction of φ-modules and Galois Representations

`SemisimpleDecomposition(D)`

Compute a Jordan-Holder sequence for the φ-module D. The result G, P, sl, pol is as follows: G is the matrix of φ in a basis where it is block upper triangular, with diagonal blocks corresponding to simple φ-modules. The matrix P gives the corresponding basis. The list sl is the list of the slopes of D, and the list pol is a list of polynomials. The isomorphism class of a simple block of G is determined by the corresponding slope and polynomial.

`Slopes(D)`

Compute the list of slopes of D (with multiplicities).

`SSGaloisRepresentation(D)`

Compute the semisimplification of the Galois representation corresponding to D.

93.2.2 Semisimple Galois Representations

This part is dedicated to the study of representations of absolute Galois groups of fields of the form $k((u))$ with k finite, and with coefficients in finite fields. The implementation is for semisimple representations, and these are described by their tame inertia weights and polynomials giving the action of the Frobenius on the unramified part.

93.2.2.1 Category

In Magma, semisimple Galois representations have type `SSGalRep`.

93.2.2.2 Creation functions

`SSGaloisRepresentation(E,K,w,P)`

Create the semisimple representation of the absolute Galois group of K with coefficients in E, tame inertia weights given by w, and action of the Frobenius described by the elements of the list P.

93.2.2.3 Basic operations

`CoefficientRing(V)`

The coefficient ring.

`FixedField(V)`

The fixed field of the absolute Galois group of which V is a representation.

`Weights(V)`

The tame inertia weights of V.

93.2.2.4 Representation associated to a φ-module

`SSGaloisRepresentation(D)`

If D is a φ-module over a field K of Laurent series this returns the semisimplification of the representation associated to D.

93.3 Examples

Example H93E1

We create two φ-modules D_1, D_2, build their direct sum, and compute its slopes and corresponding representation.

```
> k<e9> := GF(3,2);
> S<u> := LaurentSeriesRing(k,20);
> D1 := ElementaryPhiModule(S,3,2);
> D1;
Phi-module of dimension 3 over Laurent series field in u over GF(3^2)
with fixed relative precision 20 with matrix
[ O(u^20)         O(u^20)        u^2 + O(u^20)]
[ 1 + O(u^20)     O(u^20)             O(u^20) ]
[ O(u^20)        1 + O(u^20)          O(u^20) ] and Frobenius [1,3]
> M := Matrix(S,2,2,[0,k.1*u,1,0]);
> D2 := PhiModule(M);
> D2;
Phi-module of dimension 2 over Laurent series field in u over GF(3^2)
with fixed relative precision 20 with matrix
[ O(u^20)       e9*u^2 + O(u^20)]
[ 1 + O(u^20)            O(u^20) ] and Frobenius [1,3]
> D := DirectSum(D1,D2);
> Slopes(D);
[
  [2, 1],
  [3, 2]
]
> SSGaloisRepresentation(D);
```

```
Semisimple representation of the absolute Galois group of
Laurent series field in u over GF(3^2) with fixed relative
precision 20 with coefficients in Finite field of size 3 and
components [
[3, 18],
[2, 3]
]
```

PART XIII
LIE THEORY

94 INTRODUCTION TO LIE THEORY

Chapter 94
INTRODUCTION TO LIE THEORY

A number of structures from Lie theory and the theory of Coxeter groups can be handled by Magma. Specifically, facilities are provided for:

1. Coxeter matrices, Coxeter graphs, Cartan matrices, Dynkin diagrams, and Cartan's naming system for Coxeter groups;
2. Finite root systems and finite root data;
3. Coxeter groups in three different formats: as finitely presented groups, as permutation groups, and as reflection groups;
4. Complex reflection groups;
4. Lie algebras, given as structure constant algebras, matrix algebras, or finitely generated algebras;
5. Groups of Lie type (connected reductive algebraic groups);
5. Representations of Lie algebras and groups of Lie type;
6. Universal enveloping algebras and Quantum groups.

94.1 Descriptions of Coxeter Groups

A *Coxeter system* is a group G with finite generating set $S = \{s_1, \ldots, s_n\}$, defined by the power relations $s_i^2 = 1$ for $i = 1, \ldots, n$ and braid relations

$$s_i s_j s_i \cdots = s_j s_i s_j \cdots$$

for $i, j = 1, \ldots, n$ with $i < j$, where each side of this relation has length $m_{ij} \geq 2$. Although traditionally $m_{ij} = \infty$ signifies that the corresponding relation is omitted, for technical reasons, we use $m_{ij} = 0$ instead. Set $m_{ji} = m_{ij}$ and $m_{ii} = 1$. The group G is called a *Coxeter group* and S is called the set of *Coxeter generators*. Since every group in Magma has a preferred generating set, no distinction is made between a Coxeter system and its Coxeter group.

Due to the importance and ubiquity of Coxeter groups, a number of different ways of describing these groups and their reflections have been developed. Functions for manipulating these descriptions are described in Chapter 95.

Coxeter groups are usually described by a *Coxeter matrix* $M = (m_{ij})_{i,j=1}^n$, or by a *Coxeter graph* with vertices $1, \ldots, n$ and an edge connecting i and j labeled by m_{ij} whenever $m_{ij} \geq 3$.

Coxeter systems are mainly important because they provide presentations for the real reflection groups. A *Cartan matrix* describes a particular reflection representation of a

Coxeter group. In certain cases, such a representation can be described by an integer-labelled digraph, called the *Dynkin digraph* (this is equivalent to a *Dynkin diagram*, but we have modified the definition for technical reasons).

For finite and affine Coxeter groups, the naming system due to Cartan is also used. Hyperbolic Coxeter groups of degree larger than 3 are numbered.

94.2 Root Systems and Root Data

A (real) reflection is an automorphism of a real vector space that acts as negation on a one-dimensional subspace while fixing a hyperplane pointwise. The subspace is described by a vector called the *root*, while the hyperplane is described as the kernel of an element of the dual space called the *coroot*.

A root system is a collection of root/coroot pairs that is closed under the action of the corresponding reflections. Only finite root systems are supported at the present time. A root system gives a much more detailed description of a reflection representation of a finite Coxeter group.

Root systems are used to classify the *semisimple Lie algebras*. The closely related concept of a root datum is used to classify the *groups of Lie type*.

This is described in Chapters 96 and 97.

94.3 Coxeter and Reflection Groups

Three different methods are provided for computing with a Coxeter group: the Coxeter presentation, the permutation representation on roots, or a reflection representation.

For most purposes, the presentation will be the most useful of these descriptions. The standard normal form is used for elements (the lexicographically least word of minimal length). Robert Howlett has implemented his highly efficient method for normalising and multiplying elements, based on ideas from [BH93].

If the Coxeter group is finite, it is often better to use the permutation representation. Note that elements are represented as permutations on the set of roots. This is not the minimal degree representation, but is more useful in many cases.

Finally, Coxeter groups can be represented as a reflection group over the reals (in practice over a number field, since the reals are not infinite precision). Although functions are provided for creating reflection groups over an arbitrary field, fewer facilities are available for such groups. In addition, functions are provided to construct all the finite *complex* reflection groups.

Efficient functions are provided for converting between these three forms of Coxeter group.

This is described in Chapters 98 and 99.

94.4 Lie Algebras and Groups of Lie Type

Lie algebras can be constructed in three different ways in MAGMA: as structure constant algebras, as Lie matrix algebras, or as finitely presented algebras. Most of our functionality is for algebras of finite dimension over a field. Algorithms designed and implemented by de Graaf [dG00] are available for determining the structure of a Lie algebra. In particular, if the algebra is reductive, its root system and its highest-weight representations can be determined.

We provide functionality for computing with groups of Lie type (i.e. reductive algebraic groups and their split (untwisted) groups, given by the Steinberg presentation. A canonical form for words in this group, and algorithms for computing with these words are given in [CMT04, CHM08]. Twisted groups are given by a modified version of this presentation using Galois cohomology [Hal05]. Efficient algorithms have been implemented for arithmetic with the Steinberg presentation and for converting between this presentation and matrix representations over the base field. Note that these presentations are not in the category `GrpFP` since the generators are parametrised by field elements and so the groups involved are not necessarily finitely generated.

This is described in Chapter 100.

94.5 Highest Weight Representations

The representations of Lie algebras and connected reductive Lie groups are classified by *highest weights*. In addition to being able to construct these representations [dG01], we can compute the combinatorics of their weights. This includes all the functionality of the LiE system [vLCL92]. This is described in Chapter 104.

94.6 Universal Enveloping Algebras and Quantum Groups

Given a semisimple Lie algebra over a field of characteristic zero, we can construct an integral basis for its universal enveloping algebra. Functionality for computing in these algebras is described in Section 100.17. Moreover, we provide functionality for computing in their quantised versions, which are called quantum groups. This is described in Chapter 102.

94.7 Bibliography

[**BH93**] Brigitte Brink and Robert B. Howlett. A finiteness property and an automatic structure for Coxeter groups. *Math. Ann.*, 296(1):179–190, 1993.

[**CHM08**] Arjeh M. Cohen, Sergei Haller, and Scott H. Murray. Computing in unipotent and reductive algebraic groups. *LMS J. Comput. Math.*, 11:343–366, 2008.

[**CMT04**] Arjeh M. Cohen, Scott H. Murray, and D. E. Taylor. Computing in groups of Lie type. *Math. Comp.*, 73(247):1477–1498, 2004.

[**dG00**] W.A. de Graaf. *Lie Algebras: Theory and Algorithms*. Number 56 in North-Holland Mathematical Library. Elsevier, 2000.

[**dG01**] W. A. de Graaf. Constructing representations of split semisimple Lie algebras. *J. Pure Appl. Algebra*, 164(1-2):87–107, 2001. Effective methods in algebraic geometry (Bath, 2000).

[**Hal05**] Sergei Haller. *Computing Galois Cohomology and Forms of Linear Algebraic Groups*. Phd thesis, Technical University of Eindhoven, 2005.

[**vLCL92**] M.A.A. van Leeuwen, A.M. Cohen, and B. Lisser. *LiE, A package for Lie Group Computations*. CAN, Amsterdam, 1992.

95 COXETER SYSTEMS

Chapter 95
COXETER SYSTEMS

95.1 Introduction

The functions in this chapter handle basic descriptions of Coxeter systems. A *Coxeter system* is a group G with finite generating set $S = \{s_1, \ldots, s_n\}$, defined by relations $s_i^2 = 1$ for $i = 1, \ldots, n$ and

$$s_i s_j s_i \cdots = s_j s_i s_j \cdots$$

for $i, j = 1, \ldots, n$ with $i < j$, where each side of this relation has length $m_{ij} \geq 2$. Traditionally, $m_{ij} = \infty$ signifies that the corresponding relation is omitted but for technical reasons $m_{ij} = 0$ is used in MAGMA instead. The group G is called a *Coxeter group* and S is called the set of *Coxeter generators*. Since every group in MAGMA has a preferred generating set, no distinction is made between a Coxeter system and its Coxeter group. See [Bou68] for more details on the theory of Coxeter groups.

The *rank* of the Coxeter system is $n = |S|$. A Coxeter system is said to be *reducible* if there is a proper subset I of $\{1, \ldots, n\}$ such that $m_{ij} = 2$ or $m_{ji} = 2$ whenever $i \in I$ and $j \notin I$. In this case, G is an (internal) direct product of the Coxeter subgroups $W_I = \langle s_i \mid i \in I \rangle$ and $W_{I^c} = \langle s_i \mid i \notin I \rangle$. Note that an *irreducible* Coxeter group may still be a nontrivial direct product of abstract subgroups (for example, $W(G_2) \cong S_2 \times S_3$). Two Coxeter groups are *Coxeter isomorphic* if there is a group isomorphism between them which takes Coxeter generators to Coxeter generators. In other words, the two groups are the same modulo renumbering of the generators.

Coxeter groups and their representations as reflection groups have a number of useful descriptions. In this chapter, Coxeter matrices, Coxeter graphs, Cartan matrices, and Dynkin digraphs will be discussed. The classification of finite and affine Coxeter groups provides a naming system for these groups. In Chapters 96 and 97, finite root systems and root data, which provide a more detailed description of finite Coxeter groups, are discussed. Coxeter groups themselves are discussed in Chapter 98; reflection representations of Coxeter groups are discussed in Chapter 99.

95.2 Coxeter Matrices

A Coxeter system is defined by the numbers $m_{ij} \in \{2, 3, \ldots, \infty\}$ for $i, j = 1, \ldots n$ and $i < j$, as in the previous section. Setting $m_{ji} = m_{ij}$ and $m_{ii} = 1$, yields a matrix $M = (m_{ij})_{i,j=1}^n$ that is called the *Coxeter matrix*.

Since ∞ is not an integer in MAGMA, it will be represented by 0 in Coxeter matrices.

`IsCoxeterMatrix(M)`

Returns `true` if, and only if, the matrix M is the Coxeter matrix of some Coxeter group.

`CoxeterMatrix(G)`

`CoxeterMatrix(C)`

`CoxeterMatrix(D)`

The Coxeter matrix corresponding to a Coxeter graph G, Cartan matrix C, or Dynkin digraph D.

Example H95E1

```
> M := SymmetricMatrix([1, 3,1, 2,3,1]);
> M;
[1 3 2]
[3 1 3]
[2 3 1]
> IsCoxeterMatrix(M);
true
```

`IsCoxeterIsomorphic(M1, M2)`

Returns `true` if, and only if, the Coxeter matrices M_1 and M_2 give rise to Coxeter isomorphic groups. If so, a sequence giving the permutation of the underlying basis which takes M_1 to M_2 is also returned.

`CoxeterGroupOrder(M)`

`CoxeterGroupFactoredOrder(M)`

The (factored) order of the Coxeter group with Coxeter matrix M.

Example H95E2

```
> M1 := SymmetricMatrix([1, 3,1, 2,3,1]);
> M2 := SymmetricMatrix([1, 3,1, 3,2,1]);
> IsCoxeterIsomorphic(M1, M2);
true [ 2, 1, 3 ]
>
> CoxeterGroupOrder(M1);
24
```

`IsCoxeterIrreducible(M)`

Returns `true` if, and only if, the matrix M is the Coxeter matrix of an irreducible Coxeter system. If the Coxeter matrix is reducible, this function also returns a nontrivial subset I of $\{1, \ldots, n\}$ such that $m_{ij} = 2$ whenever $i \in I$, $j \notin I$.

`IsSimplyLaced(M)`

Returns `true` if, and only if, the Coxeter matrix M is simply laced, i.e. all its entries are 1, 2, or 3.

Example H95E3

```
> M := SymmetricMatrix([1, 3,1, 2,3,1]);
> IsCoxeterIrreducible(M);
true
> M := SymmetricMatrix([1, 2,1, 2,3,1]);
> IsCoxeterIrreducible(M);
false { 1 }
```

95.3 Coxeter Graphs

A *Coxeter graph* is an undirected labelled graph describing a Coxeter system. Suppose a Coxeter system has Coxeter matrix $M = (m_{ij})_{i,j=1}^{n}$. Then the Coxeter graph has vertices $1, \ldots, n$; whenever $m_{ij} > 2$ there is an edge connecting i and j labeled by the value of m_{ij}. When $m_{ij} = 3$, the label is usually omitted.

Since ∞ is not an integer, it will be represented by 0 in our Coxeter graphs. Clearly a Coxeter graph must be standard, i.e. its vertices must be the integers $1, 2, \ldots, n$ for some n. A Coxeter system is irreducible if, and only if, its Coxeter graph is connected. Two Coxeter graphs give rise to Coxeter isomorphic groups if, and only if, they are isomorphic as labelled graphs. See Chapter 149 for more information on graphs.

`IsCoxeterGraph(G)`

Returns `true` if, and only if, the graph G is the Coxeter graph of some Coxeter group.

`CoxeterGraph(M)`
`CoxeterGraph(C)`
`CoxeterGraph(D)`

The Coxeter graph corresponding to a Coxeter matrix M, Cartan matrix C, or Dynkin digraph D.

`CoxeterGroupOrder(G)`
`CoxeterGroupFactoredOrder(G)`

The (factored) order of the Coxeter group with Coxeter graph G.

Example H95E4

```
> G := PathGraph(4);
> AssignLabel(G, 1,2, 4);
> AssignLabel(G, 3,4, 4);
> IsCoxeterGraph(G);
true
> CoxeterGroupOrder(G);
Infinity
>
> M := SymmetricMatrix([1, 3,1, 2,5,1]);
> G := CoxeterGraph(M);
> Labels(EdgeSet(G));
[ undef, 5 ]
```

`IsSimplyLaced(G)`

Returns `true` if, and only if, the Coxeter graph G is simply laced, i.e. unlabelled.

Example H95E5

```
> G := PathGraph(2);
> IsSimplyLaced(G);
true
> AssignLabel(G, 1,2, 6);
> IsSimplyLaced(G);
false
```

95.4 Cartan Matrices

A *Cartan matrix* is a real valued matrix $C = (c_{ij})_{i,j=1}^{n}$ satisfying the properties:

1. $c_{ii} = 2$;
2. $c_{ij} \leq 0$ for $i \neq j$;
3. $c_{ij} = 0$ if, and only if, $c_{ji} = 0$; and
4. if $n_{ij} := c_{ij}c_{ji} < 4$, then $n_{ij} = 4\cos^2(\pi/m_{ij})$ for some integer $m_{ij} \geq 2$.

In MAGMA, Cartan matrices can be defined over the integer ring (Chapter 18), the rational field (Chapter 20), number fields (Chapter 34), and cyclotomic fields (Chapter 36). The real field (Chapter 25) is *not* allowed since it is not infinite precision. A Cartan matrix is called *crystallographic* if all its entries are integers.

Given a Cartan matrix, the corresponding Coxeter matrix $M = (m_{ij})_{i,j=1}^{n}$ is defined by $m_{ii} = 1$; m_{ij} as in (4) if $n_{ij} < 4$; $m_{ij} = \infty$ (ie, 0) if $n_{ij} \geq 4$. The significance of Cartan matrices is due to the following construction: Let X be a real inner-product space with basis $\alpha_1, \ldots, \alpha_n$. Take the unique basis $\alpha_1^\star, \ldots, \alpha_n^\star$ for X such that $(\alpha_i, \alpha_j^\star) = c_{ij}$. Let s_i be the reflection in α_i and $\alpha_i^\star$, i.e. $s_i : V \to V$ is defined by $vs_i = v - (v, \alpha_i^\star)\alpha_i$. Then the group generated by $s_1, \ldots, s_n$ is a Coxeter group with Coxeter matrix M. In other words, a Cartan matrix specifies a faithful representation of the Coxeter group as a real reflection group. For more details on reflection groups see Chapter 99.

`IsCartanMatrix(C)`

`RealInjection`	ANY	*Default :* `false`

Returns `true` if, and only if, the matrix C is a Cartan matrix.

Number field elements and cyclotomic field elements do not have a natural identification with real numbers. The `RealInjection` flag allows the user to provide one. If the base field of C is a number field, the flag should be an injection into the real field; if the base field is cyclotomic, the flag should be an injection into the complex field taking real values on the entries of C. If no real injection is given, conditions (2) and (4) of the definition are not checked.

`CartanMatrix(M)`
`CartanMatrix(G)`

`Symmetric`	BOOLELT	*Default :* `false`
`BaseField`	MONSTGELT	*Default :* `"NumberField"`

A Cartan matrix corresponding to the Coxeter matrix M or Coxeter graph G. Note that the Cartan matrix of a Coxeter system is not unique. By default this function returns the Cartan matrix with $c_{ij} = -4\cos^2(\pi/m_{ij})$, $c_{ji} = -1$ when $m_{ij} \neq 2$ and $i < j$. This matrix is crystallographic whenever there exists a crystallographic Cartan matrix corresponding to M.

If the `Symmetric` flag is set `true`, the symmetric Cartan matrix with

$$c_{ij} = c_{ji} = -2\cos(\pi/m_{ij})$$

is returned.

The BaseField flag determines the field over which the Cartan matrix is defined. If the matrix is crystallographic however, it is defined over the integers regardless of the value of this flag. The possible values are:

1. "NumberField": An algebraic number field. This is the default. See Chapter 34.
2. "Cyclotomic" or "SparseCyclotomic": A cyclotomic field with the sparse representation for elements. See Chapter 36.
3. "DenseCyclotomic": A cyclotomic field with the dense representation for elements. See Chapter 36.

CartanMatrix(D)

The crystallographic Cartan matrix corresponding to the Dynkin digraph D.

Example H95E6

```
> C := Matrix(2,2, [ 2,-3, -1,2 ]);
> C;
> IsCartanMatrix(C);
true
> CoxeterMatrix(C);
[1 6]
[6 1]
>
> G := PathGraph(4);
> AssignLabel(G, 1,2, 4);
> AssignLabel(G, 3,4, 4);
> CartanMatrix(G);
[ 2 -2  0  0]
[-1  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
> CartanMatrix(G : Symmetric, BaseField := "Cyclotomic");
[2 zeta(8)_8^3 - zeta(8)_8 0 0]
[zeta(8)_8^3 - zeta(8)_8 2 -1 0]
[0 -1 2 zeta(8)_8^3 - zeta(8)_8]
[0 0 zeta(8)_8^3 - zeta(8)_8 2]
```

IsCoxeterIsomorphic(C1, C2)

Returns true if, and only if, the Cartan matrices C_1 and C_2 give rise to Coxeter isomorphic Coxeter groups, i.e. their Coxeter matrices are equal modulo permutation of the underlying basis. If so, a sequence giving the permutation of the underlying basis which takes the Coxeter matrix of C_1 to the Coxeter matrix of C_2 is also returned.

`IsCartanEquivalent(C1, C2)`

Returns `true` if, and only if, the crystallographic Cartan matrices C_1 and C_2 are Cartan equivalent, i.e. they are equal modulo permutation of the underlying basis. If so, a sequence giving the permutation of the underlying basis which takes C_1 to C_2 is also returned.

Example H95E7

Cartan equivalence is a stronger condition than Coxeter isomorphism.

```
> C1 := Matrix(2,2, [ 2,-2, -2,2 ]);
> C2 := Matrix(2,2, [ 2,-1, -5,2 ]);
> IsCoxeterIsomorphic(C1, C2);
true [ 1, 2 ]
> IsCartanEquivalent(C1, C2);
false
```

`NumberOfPositiveRoots(C)`

`NumPosRoots(C)`

The number of positive roots of the root system with Cartan matrix C. See Subsection 96.1.3 for the definition of positive roots.

`CoxeterGroupOrder(C)`

`CoxeterGroupFactoredOrder(C)`

The (factored) order of the Coxeter group with Cartan matrix C.

`FundamentalGroup(C)`

The fundamental group of the crystallographic Cartan matrix C, i.e. $\mathbf{Z}^n/\Gamma$ where n is the degree of C and Γ is the lattice generated by the rows of C. The natural mapping $\mathbf{Z}^n \to \mathbf{Z}^n/\Gamma$ is the second returned value.

Example H95E8

```
> C := CartanMatrix(PathGraph(4));
> FundamentalGroup(C);
Abelian Group isomorphic to Z/5
Defined on 1 generator
Relations:
    5*$.1 = 0
Mapping from: Standard Lattice of rank 4 and degree 4 to Abelian Group
isomorphic to Z/5
Defined on 1 generator
Relations:
    5*$.1 = 0
```

`IsCoxeterIrreducible(C)`

Returns `true` if, and only if, C is the Cartan matrix of an irreducible Coxeter system. If the Coxeter matrix is reducible, this function also returns a nontrivial subset I of $\{1, \ldots, n\}$ such that $m_{ij} = 2$ (i.e. $c_{ij} = 0$) whenever $i \in I$, $j \notin I$.

`IsCrystallographic(C)`

Returns `true` if, and only if, the Cartan matrix C is crystallographic, i.e. C has integral entries.

`IsSimplyLaced(C)`

Returns `true` if, and only if, the Cartan matrix C is simply laced, i.e. all the entries in its Coxeter matrix are 1, 2, or 3.

Example H95E9

```
> C := Matrix(2,2, [ 2,-2, -2,2 ]);
> IsCoxeterIrreducible(C);
true
> IsCrystallographic(C);
true
> IsSimplyLaced(C);
false
```

95.5 Dynkin Digraphs

A *Dynkin digraph* is a directed labelled graph describing a *crystallographic* Cartan matrix $C = (c_{ij})_{i,j=1}^{n}$. The Dynkin digraph has vertices $1, \ldots, n$; whenever $c_{ij} < 0$ there is an edge from i to j labeled by the value $-c_{ij}$. When $c_{ij} = -1$, the label is usually omitted.

In the literature, the term *Dynkin diagram* is used, but here this will be reserved for a printed display of the Dynkin digraph (or Coxeter graph) corresponding to a finite or affine Coxeter group (see Section 95.6 below). For convenience, Dynkin digraphs have labelled edges rather than multiple edges.

Clearly a Dynkin digraph must be standard, i.e. its vertices must be the integers $1, 2, \ldots, n$ for some n. A Dynkin digraph has an edge from i to j if, and only if, it has an edge from j to i (although the labels may be different). Hence strong and weak connectivity are equivalent for these graphs. The Coxeter system is irreducible if, and only if, the Dynkin digraph is connected. Two Dynkin digraphs give rise to Cartan equivalent Cartan matrices if they are isomorphic as labelled digraphs. See Chapter 149 for more information on digraphs.

Note that functions are not given for computing the Dynkin digraph of a Coxeter matrix or Coxeter graph, since a particular choice of crystallographic Cartan matrix is required.

`IsDynkinDigraph(D)`

Returns `true` if, and only if, the digraph D is the Dynkin digraph of some crystallographic Cartan matrix.

`DynkinDigraph(C)`

The Dynkin digraph of the crystallographic Cartan matrix C.

`CoxeterGroupOrder(D)`

`CoxeterGroupFactoredOrder(D)`

The (factored) order of the Coxeter group with Dynkin digraph D.

`FundamentalGroup(D)`

The fundamental group of the Dynkin digraph D, i.e. $\mathbf{Z}^n/\Gamma$ where n is the degree of D and Γ is the lattice generated by the rows of the corresponding Cartan matrix. The natural mapping $\mathbf{Z}^n \to \mathbf{Z}^n/\Gamma$ is the second returned value.

`IsSimplyLaced(D)`

Returns `true` if, and only if, the Dynkin digraph D is simply laced, i.e. unlabelled.

Example H95E10

```
> D := Digraph< 4 | <1,{2,3,4}>, <2,{1}>, <3,{1}>, <4,{1}> >;
> AssignLabel(D, 1,2, 2);
> AssignLabel(D, 1,3, 5);
> IsDynkinDigraph(D);
true
> CartanMatrix(D);
[ 2 -2 -5 -1]
[-1  2  0  0]
[-1  0  2  0]
[-1  0  0  2]
> FundamentalGroup(D);
Abelian Group isomorphic to Z/2 + Z/8
Defined on 2 generators
Relations:
    2*$.1 = 0
    8*$.2 = 0
Mapping from: Standard Lattice of rank 4 and degree 4 to Abelian Group
isomorphic to Z/2 + Z/8
Defined on 2 generators
Relations:
    2*$.1 = 0
    8*$.2 = 0
```

95.6 Finite and Affine Coxeter Groups

Functions related to the classification of finite and affine Coxeter groups are described in this section. This classification is due to Cartan [Car52] and Coxeter [Cox34].

An affine reflection group is a group generated by reflections in affine space (in other words, real reflections in a hyperplane that does not necessarily pass through the origin). A Coxeter group is called *affine* if it is infinite and it has a representation as a discrete, properly acting, affine reflection group (see [Bou68] for more details on discreteness and proper action). Note that a Coxeter group is finite if, and only if, it has a representation as a discrete, properly acting group of reflections of the sphere; hence finite Coxeter groups are sometimes called *spherical*.

A Coxeter group is finite if, and only if, all its irreducible components are finite; a Coxeter group is affine if, and only if, all its irreducible components are finite or affine, and at least one component is affine. So it suffices to classify irreducible Coxeter groups.

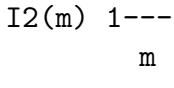
The Dynkin diagrams of the irreducible finite crystallographic Coxeter groups are:

```
An  1---2---3- ... -n        Bn  1---2- ... -(n-1)=>=n

                                                  (n-1)
                                                 /
Cn  1---2- ... -(n-1)=<=n    Dn  1---1- ... -(n-2)
                                                 \
                                                  n

E6  1---3---4---5---6        E7  1---3---4---5---6---7
            |                            |
            2                            2

E8  1---3---4---5---6---7---8
            |
            2

F4  1---2=>=3---4            G2  1=<=2
                                     3
```

Due to the difficulty of drawing a triple bond with text characters, the edge for G_2 is labelled.

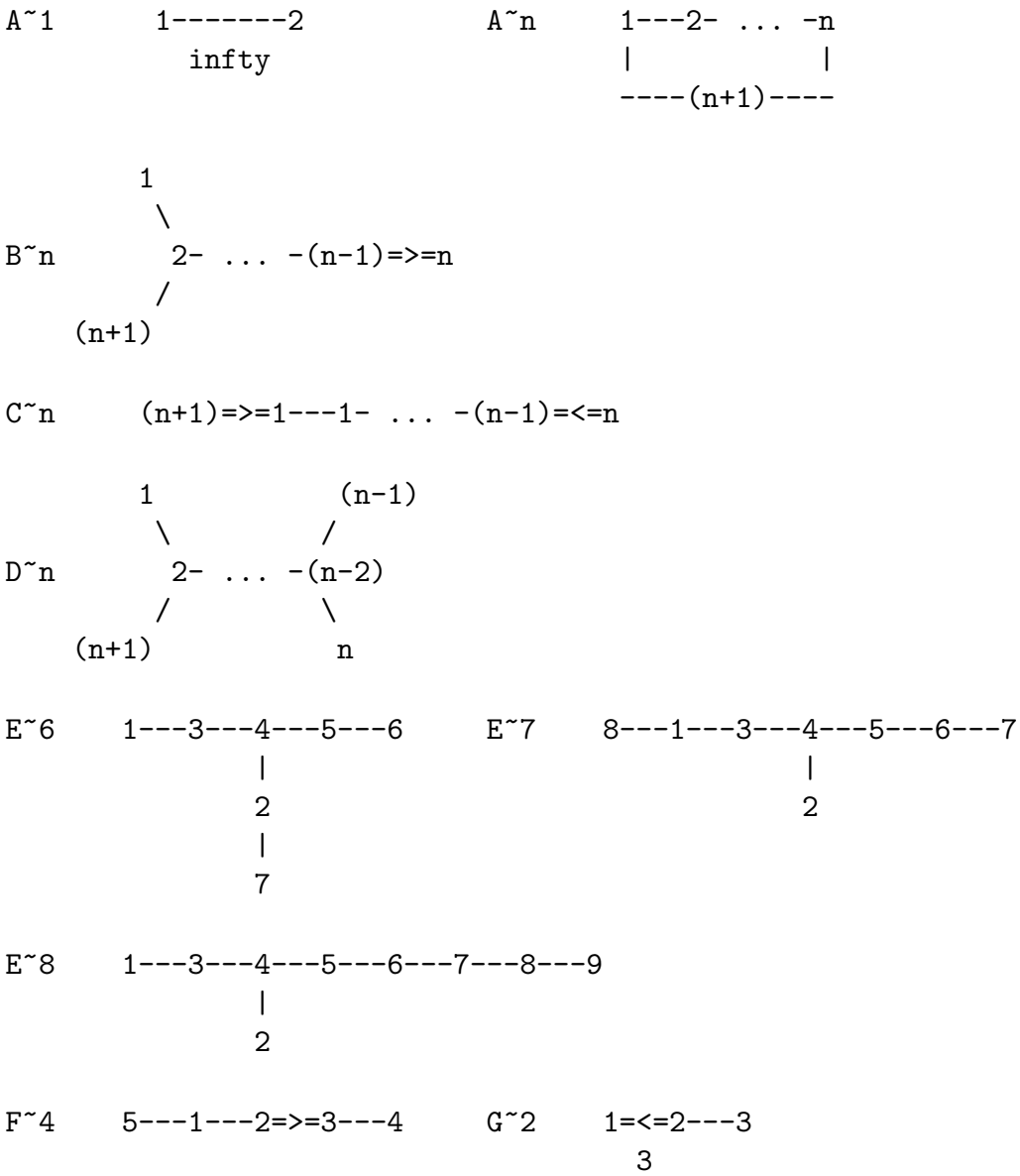
The only irreducible noncrystallographic finite Coxeter groups are H_3, H_4 and $I_2(m)$ for $m = 5$ and $m > 6$. The Coxeter graphs of these groups are:

```
H3  1---2---3                H4  1---2---3---4
      5                            5
```



Note that there is some redundancy in this classification; specifically $A_1 = B_1 = C_1 = D_1$, $A_2 = I_2(3)$, $B_2 = C_2 = I_2(4)$, $D_2 = A_1 + A_1$, $D_3 = A_3$, $G_2 = I_2(6)$. Furthermore, for $n \geq 3$, types B_n and C_n have identical Coxeter matrices but inequivalent crystallographic Cartan matrices for $n > 2$.

All irreducible affine groups are crystallographic. There is one corresponding to each irreducible crystallographic finite group. Their Dynkin diagrams are:



The labels on the vertices of these diagrams show the standard vertex order used in MAGMA, which is consistent with the order used in [Bou68].

`IsCoxeterFinite(M)`
`IsCoxeterFinite(G)`
`IsCoxeterFinite(C)`
`IsCoxeterFinite(D)`
`IsCoxeterFinite(N)`

Returns `true` if, and only if, the corresponding Coxeter group is finite. The input variable can be a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N.

`IsCoxeterAffine(M)`
`IsCoxeterAffine(G)`
`IsCoxeterAffine(C)`
`IsCoxeterAffine(D)`
`IsCoxeterAffine(N)`

Returns `true` if, and only if, the corresponding Coxeter group is affine. The input variable can be a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N.

Example H95E11

```
> IsCoxeterAffine("A~2");
true
> IsCoxeterAffine("A~2B2");
true
> IsCoxeterAffine("A2B2");
false
> IsCoxeterFinite("A2B2");
true
```

`CoxeterMatrix(N)`

The Coxeter matrix with Cartan name given by the string N.

`CoxeterGraph(N)`

The Coxeter graph with Cartan name given by the string N.

`CartanMatrix(N)`

`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* `"NumberField"`

The Cartan matrix with Cartan name given by the string N. By default, the crystallographic matrix is returned for crystallographic types; otherwise the Cartan matrix with $c_{ij} = -4\cos^2(\pi/m_{ij})$, $c_{ji} = -1$ when $m_{ij} \neq 2$ and $i < j$ is returned.

If the `Symmetric` flag is set `true`, the symmetric Cartan matrix with $c_{ij} = c_{ji} = -2\cos(\pi/m_{ij})$ is returned.

The `BaseField` flag determines the field over which the Cartan matrix is defined. If the matrix is crystallographic however, it is defined over the integers regardless of the value of this flag. The possible values are:

1. `"NumberField"`: An algebraic number field. This is the default. See Chapter 34.
2. `"Cyclotomic"` or `"SparseCyclotomic"`: A cyclotomic field with the sparse representation for elements. See Chapter 36.
3. `"DenseCyclotomic"`: A cyclotomic field with the dense representation for elements. See Chapter 36.

`DynkinDigraph(N)`

The Dynkin digraph with Cartan name given by the string N. The Cartan name must be crystallographic, i.e. it cannot involve types H_3, H_4 and $I_2(m)$.

Example H95E12

```
> CoxeterMatrix("I2(7)");
[1 7]
[7 1]
> CoxeterGraph("A3");
Graph
Vertex  Neighbours
1       2 ;
2       1 3 ;
3       2 ;
> CartanMatrix("H3" : Symmetric);
[   2 -$.1    0]
[-$.1    2   -1]
[   0   -1    2]
> DynkinDigraph("A~2");
Digraph
Vertex  Neighbours
1       2 3 ;
2       1 3 ;
```

```
3       1 2 ;
```

The code for interpreting a string as a Cartan name is quite flexible: letters and numbers must alternate, except in type I where brackets must be used.

```
> M := CoxeterMatrix("A_5B3 c2I2 (5)");
> CartanName(M);
A5 B3 B2 I2(5)
```

`IrreducibleCoxeterMatrix(X, n)`

The irreducible Coxeter matrix with Cartan name X_n (or $I_2(n)$ if $X =$`"I"`).

`IrreducibleCoxeterGraph(X, n)`

The irreducible Coxeter graph with Cartan name X_n (or $I_2(n)$ if $X =$`"I"`).

`IrreducibleCartanMatrix(X, n)`

`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* `"NumberField"`

The irreducible Cartan matrix with Cartan name X_n (or $I_2(n)$ if $X =$`"I"`).

If the `Symmetric` flag is set `true`, the symmetric Cartan matrix with $c_{ij} = c_{ji} = -2\cos(\pi/m_{ij})$ is returned.

The `BaseField` flag determines which field the Cartan matrix is defined over. If the matrix is crystallographic however, it is defined over the integers regardless of the value of this flag. The possible values are:

1. `"NumberField"`: An algebraic number field. This is the default. See Chapter 34.
2. `"Cyclotomic"` or `"SparseCyclotomic"`: A cyclotomic field with the sparse representation for elements. See Chapter 36.
3. `"DenseCyclotomic"`: A cyclotomic field with the dense representation for elements. See Chapter 36.

`IrreducibleDynkinDigraph(X, n)`

The irreducible Dynkin digraph with Cartan name X_n. The Cartan name must be crystallographic, i.e. it cannot involve types H_3, H_4 or $I_2(m)$.

Example H95E13

These functions are useful in loops.

```
> for n in [1..5] do
>     IsTree(IrreducibleCoxeterGraph("A~", n));
> end for;
true
false
false
false
false
> C := &join[ IrreducibleCoxeterGraph(t, 4) : t in ["A","B","C","D","F"] ];
```

`IsCoxeterIsomorphic(N1, N2)`

Returns `true` if, and only if, the Cartan names given by the strings N_1 and N_2 correspond to Coxeter isomorphic groups.

`IsCartanEquivalent(N1, N2)`

Returns `true` if, and only if, the Cartan names given by the strings N_1 and N_2 correspond to Cartan equivalent Cartan matrices. The Cartan names must be crystallographic, i.e. they cannot involve types H_3, H_4 and $I_2(m)$.

Example H95E14

```
> IsCoxeterIsomorphic("A1A1", "D2");
true
> IsCoxeterIsomorphic("B5", "C5");
true
> IsCartanEquivalent("B5", "C5");
false
```

`IsSimplyLaced(N)`

Returns `true` if, and only if, the Coxeter matrix with Cartan name given by the string N is simply laced, i.e. all its entries are 1, 2, or 3.

`CoxeterGroupOrder(N)`

`CoxeterGroupFactoredOrder(N)`

The (factored) order of the Coxeter group with Cartan name given by the string N.

`NumberOfPositiveRoots(N)`
`NumPosRoots(N)`

The number of positive roots of the Coxeter group with Cartan name given by the string N. See Subsection 96.1.3 for the definition of positive roots.

`FundamentalGroup(N)`

The fundamental group of the crystallographic Cartan matrix with Cartan name given by the string N, i.e. $\mathbf{Z}^n/\Gamma$ where Γ is the lattice generated by the rows of the Cartan matrix. The natural mapping $\mathbf{Z}^n \to \mathbf{Z}^n/\Gamma$ is the second returned value.

Example H95E15

```
> CoxeterGroupOrder("F4");
1152
> CoxeterGroupFactoredOrder("F4");
[ <2, 7>, <3, 2> ]
> NumPosRoots("F4");
24
> #FundamentalGroup("F4");
1
```

`CartanName(M)`
`CartanName(G)`
`CartanName(C)`
`CartanName(D)`

The Cartan name of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, or Dynkin digraph D. If the corresponding Coxeter group is neither finite nor affine, an error is flagged.

Example H95E16

```
> CartanName(SymmetricMatrix([1, 3,1, 2,3,1]));
A3
> CartanName(SymmetricMatrix([1, 3,1, 3,3,1]));
A~2
> CartanName(SymmetricMatrix([1, 3,1, 4,3,1]));
The component at rows and columns [ 1, 2, 3 ]
is not a finite or affine Coxeter matrix
> C := Matrix(4,4, [2,-2,0,0, -1,2,0,0, 0,0,2,-2, 0,0,-1,2] );
> C;
[ 2 -2  0  0]
[-1  2  0  0]
[ 0  0  2 -2]
```

```
[ 0  0 -1  2]
```

DynkinDiagram(M)
DynkinDiagram(G)
DynkinDiagram(C)
DynkinDiagram(D)
DynkinDiagram(N)

Print the Dynkin diagram of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D or Cartan name given by the string N. If the corresponding group is neither affine nor crystallographic, an error is flagged.

Example H95E17

```
> DynkinDiagram("A~5 D4 BC3");
A~5    1 - 2 - 3 - 4 - 5
       |               |
       ------- 6 -------
D4     9
      /
7 - 8
      \
       10
BC3    11 - 12 =>= 13
```

CoxeterDiagram(M)
CoxeterDiagram(G)
CoxeterDiagram(C)
CoxeterDiagram(D)
CoxeterDiagram(N)

Print the Coxeter diagram of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D or Cartan name given by the string N. If the corresponding group is not affine or is not crystallographic, an error is flagged.

Example H95E18

```
> CoxeterDiagram("A~5 D4 BC3");
A~5    1 - 2 - 3 - 4 - 5
       |               |
       ------- 6 -------
D4     9
      /
7 - 8
      \
       10
BC3    11 - 12 === 13
```

95.7 Hyperbolic Groups

A hyperbolic reflection group is a group generated by reflections in hyperbolic space. A Coxeter group is called *hyperbolic* if it is infinite, nonaffine, and it has a representation as a discrete, properly acting, hyperbolic reflection group whose Tits' cone consists entirely of vectors with negative norm (see [Bou68] for more details). A hyperbolic reflection group is *compact hyperbolic* if it is hyperbolic with a compact fundamental region.

Every infinite nonaffine Coxeter group of rank 3 is hyperbolic. There are only 72 hyperbolic groups of rank larger than 3 which, for convenience, are numbered from 1 to 72. The numbering is essentially arbitrary.

`IsCoxeterHyperbolic(M)`

`IsCoxeterCompactHyperbolic(M)`

Returns `true` if, and only if, the matrix M is the Coxeter matrix of a (compact) hyperbolic Coxeter group.

`IsCoxeterHyperbolic(G)`

`IsCoxeterCompactHyperbolic(G)`

Returns `true` if, and only if, the graph G is the Coxeter graph of a (compact) hyperbolic Coxeter group.

`HyperbolicCoxeterMatrix(i)`

The Coxeter matrix of the ith hyperbolic Coxeter group of rank larger than 3.

`HyperbolicCoxeterGraph(i)`

The Coxeter graph of the ith hyperbolic Coxeter group of rank larger than 3.

Example H95E19

```
> for i in [1..72] do
>   if IsCoxeterCompactHyperbolic(HyperbolicCoxeterMatrix(i)) then
>     printf "%o, ", i;
>   end if;
> end for;
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
```

95.8 Related Structures

In this section functions for creating other structures from Coxeter matrices, Coxeter graphs, Cartan matrices, Dynkin diagrams, and Cartan names are listed. The reader is referred to the appropriate sections of the Handbook for more details.

RootSystem(M)
RootSystem(G)
RootSystem(C)
RootSystem(D)
RootSystem(N)

The finite root system of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 96.

RootDatum(C)
RootDatum(M)
RootDatum(G)
RootDatum(D)
RootDatum(N)

The finite root datum of a crystallographic Cartan matrix C, Coxeter matrix M, Coxeter graph G, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 97.

```
CoxeterGroup(GrpFPCox, M)
CoxeterGroup(GrpFPCox, G)
CoxeterGroup(GrpFPCox, C)
CoxeterGroup(GrpFPCox, D)
CoxeterGroup(GrpFPCox, N)
```

The Coxeter group of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. See Chapter 98.

```
CoxeterGroup(GrpPermCox, M)
CoxeterGroup(GrpPermCox, G)
CoxeterGroup(GrpPermCox, C)
CoxeterGroup(GrpPermCox, D)
CoxeterGroup(GrpPermCox, N)
```

The permutation Coxeter group of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 98.

```
CoxeterGroup(M)
CoxeterGroup(G)
CoxeterGroup(C)
CoxeterGroup(D)
CoxeterGroup(N)
```

The Coxeter group of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is finite, it is returned as a permutation group; otherwise it is returned as a finitely presented group.

```
ReflectionGroup(M)
ReflectionGroup(G)
ReflectionGroup(C)
ReflectionGroup(D)
ReflectionGroup(N)
```

The reflection group of a Coxeter matrix M, Coxeter graph G, Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. See Chapter 99.

`LieAlgebra(C, k)`

`LieAlgebra(D, k)`

`LieAlgebra(N, k)`

The Lie algebra over the ring k of a crystallographic Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 100.

`MatrixLieAlgebra(C, k)`

`MatrixLieAlgebra(D, k)`

`MatrixLieAlgebra(N, k)`

The Lie algebra over the ring k of a crystallographic Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 100.

`GroupOfLieType(C, k)`

`GroupOfLieType(D, k)`

`GroupOfLieType(N, k)`

The group of Lie type over the ring k of a crystallographic Cartan matrix C, Dynkin digraph D, or Cartan name given by the string N. If the corresponding Coxeter group is infinite, an error is flagged. See Chapter 103.

95.9 Bibliography

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Car52**] Elie Cartan. *Œuvres complètes. Partie I. Groupes de Lie.* Gauthier-Villars, Paris, 1952.

[**Cox34**] H. S. M. Coxeter. Discrete groups generated by reflections. *Ann. of Math.*, 35:588–621, 1934.

96 ROOT SYSTEMS

Chapter 96
ROOT SYSTEMS

96.1 Introduction

This chapter describes Magma functions for computing with finite real root systems. A root system describes the reflections in a reflection group (Chapter 99). Root systems are essential in the theories of finite Coxeter groups (Chapter 98) and Lie algebras (Chapter 100). See [Bou68] for more details on the theory of root systems. The closely related concept of a root datum is discussed in Chapter 97.

96.1.1 Reflections

Let X and Y be vector spaces over a field k with bilinear pairing $\langle \circ, \circ \rangle : X \times Y \to k$ that identifies Y with the dual of X. Given nonzero $\alpha \in X$ and $\alpha^\star \in Y$, the linear map $s_\alpha : X \to X$ is defined by

$$x s_\alpha = x - \langle x, \alpha^\star \rangle \alpha$$

and the linear map $s_\alpha^\star : Y \to Y$ by

$$y s_\alpha^\star = y - \langle \alpha, y \rangle \alpha^\star.$$

These maps are called *reflections* if one of the following equivalent properties hold: $\langle \alpha, \alpha^\star \rangle = 2$; ${s_\alpha}^2 = 1$; $\langle x s_\alpha, y s_\alpha^\star \rangle = \langle x, y \rangle$ for all $x \in X$ and $y \in Y$; $\alpha s_\alpha = -\alpha$. The mapping $s_\alpha^\star$ is also called a *coreflection*: this just means it is a reflection defined on Y instead of X. MAGMA functions for computing with reflections are described in Section 99.2.

If X has an inner product, then we can take $Y = X$ and use the inner product as our pairing. In MAGMA, we generally take $X = Y$ to be a row space, with the bilinear pairing given by the standard inner product $\langle x, y \rangle = xy^T$. However, it is sometimes useful to allow X and Y to be distinct subspaces of a row space.

For the purposes of this chapter, k will always be the rational field (Chapter 20), a number field (Chapter 20), or a cyclotomic field (Chapter 36). The real field (Chapter 25) is *not* allowed since it is not infinite precision.

96.1.2 Definition of a Root System

Suppose Φ is a finite subset of $X \setminus \{0\}$. For each α in Φ, suppose a corresponding nonzero $\alpha^\star$ in Y is given; set $\Phi^\star = \{\alpha^\star \mid \alpha \in \Phi\}$. The tuple $R = (X, \Phi, Y, \Phi^\star)$ is called a *root system* if the following conditions are satisfied for every α in Φ

1. s_α and $s_\alpha^\star$ are reflections;
2. Φ is closed under the action of s_α; and
3. $\Phi^\star$ is closed under the action of $s_\alpha^\star$.

The set X is called the *root space* and Y is called the *coroot space*. The elements of Φ are called *roots* and the elements of $\Phi^\star$ are called *coroots*. A root system is said to be *crystallographic* if $\langle \alpha, \beta^\star \rangle$ is integral for every root α and coroot $\beta^\star$. A root system is *reduced*, if $\alpha, \beta \in \Phi$ with β a scalar product of α implies $\alpha = \pm\beta$. Note that it is possible for the set of roots to be empty, in which case the system is called *toral*.

96.1.3 Simple and Positive Roots

A subset Δ of Φ is called a set of *simple roots* if

1. Δ is a basis for the span of the roots $k\Phi \leq X$; and
2. $\Phi = \Phi^+ \cup \Phi^-$, where Φ^+ is the set of linear combinations of elements of Δ with nonnegative coefficients, and $\Phi^- = -\Phi^+$.

Every root system has a set of simple roots. Simple roots are frequently called fundamental roots. The elements of Φ^+ are called *positive roots* and the elements of Φ^- are called *negative roots*. The coroots corresponding to the simple (respectively, positive, negative) roots are the *simple* (respectively, *positive*, *negative*) *coroots*.

The *rank* of a root system is the size of Δ, i.e. the dimension of the subspace $k\Phi$. The rank cannot be larger than the *dimension* of the root system (i.e. the dimension of X); if the rank and dimension are equal, the root system is said to be *semisimple*.

Choose a basis $e_1, \ldots, e_d$ for X and a dual basis $f_1, \ldots, f_d$ for Y, so that $\langle e_i, f_j \rangle = \delta_{ij}$. A reduced root system is determined by a pair of real matrices A and B where the rows of A are the simple roots and the rows of B are the corresponding coroots; i.e. $A_{ij} = \langle \alpha_i, f_j \rangle$ and $B_{ij} = \langle e_j, \alpha_i^\star \rangle$.

96.1.4 The Coxeter Group

The group W generated by the reflections s_α, for α a simple root, is a finite Coxeter group. The *Cartan matrix* of a root system is

$$C = \left(\left\langle\, \alpha_i, \alpha_j^\star \,\right\rangle \right)_{i,j=1}^{n} = AB^t.$$

Note that the root system is crystallographic if, and only if, its Cartan matrix is crystallographic. As in Chapter 95, the Cartan matrix is used to define the Coxeter matrix, Coxeter graph, and Dynkin digraph of a root system.

The classification of Section 95.6 applies to reduced semisimple root systems. The isomorphism class of a reduced root system is determined by its Coxeter graph and its dimension.

A *Coxeter form* is a W-invariant bilinear form on X. If R is reduced and irreducible, then the roots can have at most two different lengths with respect to this form. We call the roots *long* or *short* accordingly. The Coxeter form is normalised so that the short roots in each component have length one. Note that, even if $X = Y$, this form will generally not be the same as the pairing $\langle \circ, \circ \rangle$; however it can be arranged for them to be the same (see `StandardRootSystem`).

96.1.5 Nonreduced Root Systems

A root system is *reduced*, if $\alpha, \beta \in \Phi$ with β a scalar product of α implies $\alpha = \pm\beta$. A root α with the property $2\alpha \notin \Phi$ is called *reduced*. A root α with the property $\frac{1}{2}\alpha \in \Phi$ is called *divisible*. If R is a root system, then the set R_0 of indivisible roots in R form the *indivisible subsystem*.

Let R be a nonreduced irreducible *crystallographic* root system of rank n. It can be shown that R_0 is irreducible of type of type B_n and every root is either in R_0, or is two times a short root of R_0. The Cartan type of R in this case is BC_n. For noncrystallographic root systems the situation is more complex.

Note that the Cartan matrix, Coxeter matrix, Coxeter diagram, Coxeter group and Dynkin diagram are the same for R and R_0. Thus, when creating a non-reduced crystallographic root system for a given Cartan matrix, Coxeter matrix, Coxeter diagram, Coxeter group or Dynkin diagram, one must specify the set of nonreduced simple roots. For example, let C be a cartan matrix of type $B_2 \times B_3$. Then the set of non-reduced fundamental roots can be one of $\emptyset$, $\{2\}$, $\{5\}$, or $\{2, 5\}$, in which cases the root system will be of types $B_2 \times B_3$, $BC_2 \times B_3$, $B_2 \times BC_3$, or $BC_2 \times BC_3$ respectively.

96.2 Constructing Root Systems

We first describe some optional parameters that are common to many functions described in this section.

`RealInjection` ANY *Default :* `false`

Number field elements and cyclotomic field elements do not have a natural identification with real numbers. The `RealInjection` flag allows the user to provide one. If the base field of the Cartan matrix C is a number field, the flag should be an injection into the real field; if the base field is cyclotomic, the flag should be an injection into the complex field taking real values on the entries of C (see more in Section 95.4).

`Nonreduced` SETENUM *Default :* {}

The optional argument `Nonreduced` is used to distinguish the reducedness of a root system in case the input doesn't uniquely determine it.

`Symmetric` BOOLELT *Default :* `false`

If the `Symmetric` flag is set `true`, the symmetric Cartan matrix is used. For types $I_2(m)$, H_3, H_4 the symmetric Cartan matrix is *always* used, since the root system is nonreduced otherwise.

`BaseField` MONSTGELT *Default :* `"NumberField"`

The `BaseField` flag determines the field over which the Cartan matrix is defined. The possible values are:

1. `"NumberField"`: An algebraic number field. This is the default. See Chapter 34.
2. `"Cyclotomic"` or `"SparseCyclotomic"`: A cyclotomic field with the sparse representation for elements. See Chapter 36.

3. `"DenseCyclotomic"`: A cyclotomic field with the dense representation for elements. See Chapter 36.

`RootSystem(N)`

`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* `"NumberField"`

The root system with Cartan name given by the string N. In addition to the Cartan names in Section 95.6, we allow `"BCn"` for the irreducible nonreduced system, and `"Tn"` for the n-dimensional toral subsystem. Note that `"Tn"` is used for input only and does not appear in the string returned by `CartanName` when applied to the resulting root system (see example below). For descriptions of the parameters `Symmetric` and `BaseField` see the beginning of this section

Example H96E1

```
> RootSystem("H3 E6");
Root system of type H3 E6
> RootSystem("A2 T1 I2(5)");
Root system of type A2 I2(5)
```

`RootSystem(M)`

`RootSystem(G)`

`Nonreduced`	SetEnum	*Default :* {}
`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* `"NumberField"`

The semisimple root system with Coxeter matrix M or Coxeter graph G (see Chapter 95). If the corresponding Coxeter group is infinite, an error is flagged. For descriptions of the parameters `Nonreduced`, `Symmetric`, and `BaseField` see the beginning of this section.

`RootSystem(C)`

`RealInjection`	Any	*Default :* `false`
`Nonreduced`	SetEnum	*Default :* {}

The semisimple root system with Cartan matrix C (see Chapter 95). If the corresponding Coxeter group is infinite, an error is flagged. For descriptions of the parameters `RealInjection` and `Nonreduced` see the beginning of this section.

`RootSystem(D)`

`Nonreduced` SETENUM *Default :* {}

The semisimple crystallographic root system with Cartan matrix C, or Dynkin diagram D (see Chapter 95). If the corresponding Coxeter group is infinite, an error is flagged. For a description of the parameter `Nonreduced` see the beginning of this section.

Example H96E2

```
> M := SymmetricMatrix([1, 3,1, 2,3,1]);
> RootSystem(M);
Root system of type A3
> M := SymmetricMatrix([1, 3,1, 3,3,1]);
> RootSystem(M);
>> RootSystem(M);
              ^
Runtime error in 'RootSystem': Not a finite root system in rows/columns
[ 1, 2, 3 ]
```

`RootSystem(A, B)`

`RealInjection` ANY *Default :* `false`

`Nonreduced` SETENUM *Default :* {}

The root system with simple roots given by the rows of the matrix A and simple coroots given by the rows of the matrix B. The matrices A and B must have the following properties:

1. A and B must have the same number of rows and the same number of columns; they must be defined over the same ring, which must be the integers, the rational field, a number field, or a cyclotomic field;
2. the number of columns must be at least the number of rows; and
3. AB^t must be the Cartan matrix of a finite Coxeter group.

For descriptions of the parameters `RealInjection` and `Nonreduced` see the beginning of this section.

Example H96E3

The following code creates a nonsemisimple root system of type G_2.

```
> A := Matrix(2,3, [1,-1,0, -1,1,-1]);
> B := Matrix(2,3, [1,-1,1, 0,1,-1]);
> RootSystem(A, B);
Root system of type G2
```

`IrreducibleRootSystem(X, n)`

`Symmetric`	BoolElt	*Default :* `false`
`BaseField`	MonStgElt	*Default :* `"NumberField"`

The irreducible root system with Cartan name X_n (or $I_2(n)$ if $X =$ "I") given by the string X and integer n. In addition to the Cartan names in Section 95.6, we allow `"BCn"` for the irreducible nonreduced system. For descriptions of the parameters `Symmetric` and `BaseField` see the beginning of this section.

`StandardRootSystem(X, n)`

The standard root system with Cartan name X_n (or $I_2(n)$ if $X =$ "I") given by the string X and integer n, i.e. the root system whose Coxeter form is the same as the standard inner product. In addition to the Cartan names in Section 95.6, we allow `"BCn"` for the irreducible nonreduced system. For type A_n, the standard root system is not semisimple.

Example H96E4

```
> Rs := { IrreducibleRootSystem("I", n) : n in [3..20] };
> { R : R in Rs | IsCrystallographic(R) };
{
    Root system of type I2(3) ,
    Root system of type I2(4) ,
    Root system of type I2(6)
}
```

`ToralRootSystem(n)`

The toral root system of dimension n, i.e., the n-dimensional root system with no roots or coroots.

`TrivialRootSystem()`

The trivial root system of dimension 0.

96.3 Operators on Root Systems

`R1 eq R2`

Returns `true` if, and only if, the root systems R_1 and R_2 are identical.

`IsIsomorphic(R1, R2)`

Returns `true` if, and only if, root systems R_1 and R_2 are isomorphic.

`IsCartanEquivalent(R1, R2)`

Returns `true` if, and only if, the crystallographic root systems R_1 and R_2 are Cartan equivalent, i.e. their Cartan matrices are the same modulo a permutation of the underlying basis.

Example H96E5

Note that the root systems B_n and C_n are isomorphic but not Cartan equivalent. Hence Cartan equivalence is *not* an invariant of a root system since it depends on the particular representation of the (co)roots within the (co)root space.

```
> R := RootSystem("B4");  S := RootSystem("C4");
> IsIsomorphic(R, S);
true
> IsCartanEquivalent(R, S);
false
```

`CartanName(R)`

The Cartan name of the root system R (Section 95.6).

`CoxeterDiagram(R)`

Print the Coxeter diagram of the root system R (Section 95.6).

`DynkinDiagram(R)`

Print the Dynkin diagram of the root system R (Section 95.6). If R is not crystallographic, an error is flagged.

`CoxeterMatrix(R)`

The Coxeter matrix of the root system R (Section 95.2).

`CoxeterGraph(R)`

The Coxeter graph of the root system R (Section 95.3).

`CartanMatrix(R)`

The Cartan matrix of the root system R (Section 95.4).

`DynkinDigraph(R)`

The Dynkin digraph of the root system R (Section 95.5). If R is not crystallographic, an error is flagged.

Example H96E6

```
> R := RootSystem("F4");
> DynkinDiagram(R);
F4    1 - 2 =>= 3 - 4
> CoxeterDiagram(R);
F4    1 - 2 === 3 - 4
```

`BaseField(R)`
`BaseRing(R)`

The field over which the root system R is defined.

`RealInjection(R)`

The real injection of the root system R (Section 96.2).

`Rank(R)`

The rank of the root system R, i.e. the number of simple (co)roots.

`Dimension(R)`

The dimension of the root system R, i.e. the dimension of the (co)root space. This is always at least as large as the rank, with equality when R is semisimple.

`CoxeterGroupOrder(R)`

The order of the Coxeter group of the root system R.

Example H96E7

```
> R := RootSystem("I2(7)");
> BaseField(R);
Number Field with defining polynomial x^3 - x^2 - 2*x + 1 over the
Rational Field
> Rank(R) eq Dimension(R);
true
> CoxeterGroupOrder(R);
14
```

96.4 Properties of Root Systems

`IsIrreducible(R)`

Returns `true` if, and only if, the root system R is irreducible.

`IsProjectivelyIrreducible(R)`

Returns `true` if, and only if, the root system R is a direct sum of a simple system and a toral system. This is equivalent to R having a connected Coxeter diagram.

`IsReduced(R)`

Returns `true` if, and only if, the root system R is reduced.

`IsSemisimple(R)`

Returns `true` if, and only if, the root system R is semisimple, i.e. its rank is equal to its dimension.

`IsCrystallographic(R)`

Returns `true` if, and only if, the root system R is crystallographic, i.e. its Cartan matrix is integral.

`IsSimplyLaced(R)`

Returns `true` if, and only if, the root system R is simply laced, i.e. its Coxeter graph contains no labelled edges.

Example H96E8

```
> R := RootSystem("A5 B2");
> IsIrreducible(R);
false
> IsSemisimple(R);
true
> IsCrystallographic(R);
true
> IsSimplyLaced(R);
false
```

96.5 Roots and Coroots

The roots are stored as an indexed set

$$\{@ \alpha_1, \ldots, \alpha_N, \alpha_{N+1}, \ldots, \alpha_{2N} @\},$$

where $\alpha_1, \ldots, \alpha_N$ are the positive roots (in an order compatible with height), and $\alpha_{N+1}, \ldots, \alpha_{2N}$ are the corresponding negative roots (i.e. $\alpha_{i+N} = -\alpha_i$). The simple roots are $\alpha_1, \ldots, \alpha_n$ where n is the rank.

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space (this is the default); or
2. `"Root"`: the basis of simple (co)roots.

96.5.1 Accessing Roots and Coroots

`RootSpace(R)`

`CorootSpace(R)`

The vector space containing the (co)roots of the root system R, i.e. X (respectively, Y).

`SimpleRoots(R)`

`SimpleCoroots(R)`

The simple (co)roots of the root system R as the rows of a matrix, i.e. A (respectively, B).

Example H96E9

```
> R := RootSystem("G2");
> RootSpace(R);
Full Vector space of degree 2 over Rational Field
> CorootSpace(R);
Full Vector space of degree 2 over Rational Field
> SimpleRoots(R);
[1 0]
[0 1]
> SimpleCoroots(R);
[ 2 -3]
[-1  2]
> CartanMatrix(R);
[ 2 -1]
[-3  2]
```

`NumberOfPositiveRoots(R)`

`NumPosRoots(R)`

The number of positive roots of the root system R. This is also the number of positive coroots. The total number of (co)roots is twice the number of positive (co)roots.

`Roots(R)`

`Coroots(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The indexed set of (co)roots of the root system R, i.e. $\{@ \alpha_1, \ldots \alpha_{2N} @\}$ (respectively, $\{@ \alpha_1^\star, \ldots \alpha_{2N}^\star @\}$).

`PositiveRoots(R)`

`PositiveCoroots(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The indexed set of positive (co)roots of the root system R, i.e. $\{@ \alpha_1, \ldots \alpha_N @\}$ (respectively, $\{@ \alpha_1^\star, \ldots \alpha_N^\star @\}$).

`Root(R, r)`

`Coroot(R, r)`

`Basis` MonStgElt *Default :* `"Standard"`

The rth (co)root α_r (respectively, $\alpha_r^\star$) of the root system R.

`RootPosition(R, v)`

`CorootPosition(R, v)`

`Basis` MonStgElt *Default :* `"Standard"`

If v is a (co)root in the root system R, return its index; otherwise return 0. These functions will try to coerce v, which can be a vector or a sequence representing a vector, into the appropriate vector space; v should be written with respect to the basis specified by the parameter `Basis`.

Example H96E10

```
> A := Matrix(2,3, [1,-1,0, -1,1,-1]);
> B := Matrix(2,3, [1,-1,1, 0,1,-1]);
> R := RootSystem(A, B);
> Roots(R);
{@
    (1 -1  0),
    (-1  1 -1),
    (0  0 -1),
    (1 -1 -1),
```

```
    (2 -2 -1),
    (1 -1 -2),
    (-1  1  0),
    (1 -1  1),
    (0 0 1),
    (-1  1  1),
    (-2  2  1),
    (-1  1  2)
@}
> PositiveCoroots(R);
{@
    (1 -1  1),
    (0  1 -1),
    (1  2 -2),
    (2  1 -1),
    (1 0 0),
    (1  1 -1)
@}
> #Roots(R) eq 2*NumPosRoots(R);
true
> Root(R, 4);
(1 -1 -1)
> Root(R, 4 : Basis := "Root");
(2 1)
> RootPosition(R, [1,-1,-1]);
4
> RootPosition(R, [2,1] : Basis := "Root");
4
```

`HighestRoot(R)`

`HighestCoroot(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The unique (co)root of greatest height in the irreducible root system R.

`HighestLongRoot(R)`

`HighestLongCoroot(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The unique long (co)root of greatest height in the irreducible root system R.

`HighestShortRoot(R)`

`HighestShortCoroot(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The unique short (co)root of greatest height in the irreducible root system R.

Example H96E11

```
> R := RootSystem("G2");
> HighestRoot(R);
(3 2)
> HighestLongRoot(R);
(3 2)
> HighestShortRoot(R);
(2 1)
```

`CoxeterForm(R)`

`DualCoxeterForm(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The matrix of an inner product on the (co)root space of the root system R which is invariant under the action of the (co)roots. This inner product is uniquely determined up to a constant on each irreducible component of R. The inner product is normalised so that the short roots in each crystallographic component have length one.

96.5.2 Reflections

The root α acts on the root space via the reflection s_α; the coroot $\alpha^\star$ acts on the coroot space via the coreflection $s^\star_\alpha$.

`SimpleReflectionMatrices(R)`

`SimpleCoreflectionMatrices(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The sequence of matrices giving the action of the simple (co)roots of the root system R on the (co)root space, i.e. the matrices of $s_{\alpha_1}, \ldots, s_{\alpha_n}$ (respectively, $s^\star_{\alpha_1}, \ldots, s^\star_{\alpha_n}$).

`ReflectionMatrices(R)`

`CoreflectionMatrices(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The sequence of matrices giving the action of the (co)roots of the root system R on the (co)root space, i.e. the matrices of $s_{\alpha_1}, \ldots, s_{\alpha_{2N}}$ (respectively, $s^\star_{\alpha_1}, \ldots, s^\star_{\alpha_{2N}}$).

`ReflectionMatrix(R, r)`

`CoreflectionMatrix(R, r)`

`Basis` MonStgElt *Default :* `"Standard"`

The matrix giving the action of the rth (co)root of the root system R on the (co)root space, i.e. the matrix of s_{α_r} (respectively, $s^\star_{\alpha_r}$).

`SimpleReflectionPermutations(R)`

The sequence of permutations giving the action of the simple (co)roots of the root system R on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutations(R)`

The sequence of permutations giving the action of the (co)roots of the root system R on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutation(R, r)`

The permutation giving the action of the rth (co)root of the root system R on the (co)roots. This action is the same for roots and coroots.

`ReflectionWords(R)`

The sequence of words in the simple reflections for all the reflections of the root system R. These words are given as sequences of integers. In other words, if $[a_1, \ldots, a_l] =$ `ReflectionWords(R)[r]`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

`ReflectionWord(R, r)`

The word in the simple reflections for the rth reflection of the root system R. The word is given as a sequence of integers. In other words, if $[a_1, \ldots, a_l] =$ `ReflectionWord(R,r)`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

Example H96E12

```
> R := RootSystem("B3");
> mx := ReflectionMatrix(R, 4);
> perm := ReflectionPermutation(R, 4);
> wd := ReflectionWord(R, 4);
> RootPosition(R, Root(R,2) * mx) eq 2^perm;
true
> perm eq &*[ ReflectionPermutation(R, r) : r in wd ];
true
>
> mx := CoreflectionMatrix(R, 4);
> CorootPosition(R, Coroot(R,2) * mx) eq 2^perm;
true
```

96.5.3 Operations and Properties for Roots and Coroot Indices

`Sum(R, r, s)`

The index of the sum of the rth and sth roots in the crystallographic root system R, or 0 if the sum is not a root. In other words, if $t =$ `Sum(R,`r`,`s`)` $\neq 0$ then $\alpha_t = \alpha_r + \alpha_s$. We require $\alpha_r \neq \pm\alpha_s$.

`IsPositive(R, r)`

Returns `true` if, and only if, the rth (co)root of the root system R is a positive root.

`IsNegative(R, r)`

Returns `true` if, and only if, the rth (co)root of the root system R is a negative root.

`Negative(R, r)`

The index of the negative of the rth (co)root of the root system R. In other words, if $s =$ `Negative(R,`r`)` then $\alpha_s = -\alpha_r$.

Example H96E13

```
> R := RootSystem("G2");
> Sum(R, 1, Negative(R,5));
10
> IsPositive(R, 10);
false
> Negative(R, 10);
4
> P := PositiveRoots(R);
> P[1] - P[5] eq -P[4];
true
```

`RootHeight(R, r)`

`CorootHeight(R, r)`

The height of the rth (co)root of the root system R, i.e. the sum of the coefficients of α_r (respectively, $\alpha_r^\star$) with respect to the simple (co)roots.

`RootNorms(R)`

`CorootNorms(R)`

The sequence of squares of the lengths of the (co)roots of the root system R.

`RootNorm(R, r)`

`CorootNorm(R, r)`

The square of the length of the rth (co)root of the root system R.

`IsLongRoot(R, r)`

Returns `true` if, and only if, the rth root of the root system R is long. This only makes sense for irreducible crystallographic root systems. Note that for non-reduced root systems, the roots which are not indivisible are actually longer than the long ones.

`IsShortRoot(R, r)`

Returns `true` if, and only if, the rth root of the root system R is short. This only makes sense for irreducible crystallographic root systems.

`IsIndivisibleRoot(R, r)`

Returns `true` if, and only if, the rth root of the root system R is indivisible, ie, $\alpha_r/2$ is not a root.

`LeftString(R, r, s)`

Indices in the crystallographic root system R of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s - \alpha_r, \alpha_s - 2\alpha_r, \ldots, \alpha_s - p\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_p]$ where $\alpha_{r_i} = \alpha_s - i\alpha_r$ and $\alpha_s - (p+1)\alpha_r$ is not a root. We require that $\alpha_r \neq \pm\alpha_s$.

`RightString(R, r, s)`

Indices in the crystallographic root system R of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s + \alpha_r, \alpha_s + 2\alpha_r, \ldots, \alpha_s + q\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_q]$ where $\alpha_{r_i} = \alpha_s + i\alpha_r$ and $\alpha_s + (q+1)\alpha_r$ is not a root. We require that $\alpha_r \neq \pm\alpha_s$.

`LeftStringLength(R, r, s)`

The largest p such that $\alpha_s - p\alpha_r$ is a root. We require that the root system R be crystallographic and $\alpha_s \neq \pm\alpha_r$.

`RightStringLength(R, r, s)`

The largest q such that $\alpha_s + q\alpha_r$ is a root. We require that the root system R be crystallographic and $\alpha_s \neq \pm\alpha_r$.

Example H96E14

```
> R := RootSystem("G2");
> RootHeight(R, 5);
4
> F := CoxeterForm(R);
> v := Root(R, 5);
> (v*F, v) eq RootNorm(R, 5);
true
> IsLongRoot(R, 5);
true
> LeftString(R, 1, 5);
[ 4, 3, 2 ]
> roots := Roots(R);
> for i in [1..3] do
>   RootPosition(R, roots[5]-i*roots[1]);
> end for;
4
3
2
> R := RootSystem("BC2");
> Root(R,2), IsIndivisibleRoot(R,2);
(0 1) true
> Root(R,4), IsIndivisibleRoot(R,4);
(0 2) false
```

`AdditiveOrder(R)`

An additive order on the positive roots of the root system R, ie. a sequence containing the numbers $1, \ldots, N$ in some order so that $\alpha_r + \alpha_s = \alpha_t$ implies t is between r and s. This is computed using the techniques of [Pap94].

`IsAdditiveOrder(R, Q)`

Returns `true` if, and only if, the sequence Q gives an additive order on a set of positive roots of the root system R. Q must be a sequence of integers in the range $[1..N]$, where N is the number of positive roots of R, with no gaps or repeats.

Example H96E15

```
> R := RootSystem("A5");
> a := AdditiveOrder(R);
> Position(a, 2);
6
> Position(a, 3);
10
> Position(a, Sum(R, 2, 3));
7
```

96.6 Building Root Systems

`sub< R | a >`

The root subsystem of the root system R generated by the roots $\alpha_{a_1}, \ldots, \alpha_{a_k}$ where $a = \{a_1, \ldots, a_k\}$ is a set of integers.

`sub< R | s >`

The root subsystem of the root system R generated by the roots $\alpha_{s_1}, \ldots, \alpha_{s_k}$ where $s = [s_1, \ldots, s_k]$ is a *sequence* of integers. In this version the roots must be simple in the root subsystem (i.e. none of them may be a summand of another), otherwise an error is signalled. The simple roots will appear in the subsystem in the given order.

`R1 subset R2`

Returns `true` if and only if the root system R_1 is a subset of the root system R_2. If true, returns an injection as sequence of roots as second return value.

`R1 + R2`

`DirectSum(R1, R2)`

The direct sum of the root systems R_1 and R_2. The root space of the result is the direct sum of the root spaces of R_1 and R_2.

`R1 join R2`

The union of the root systems R_1 and R_2. The root systems must have the same root space, which will also be the root space of the result.

Example H96E16

```
> R  := RootSystem("A1A1");
> R1 := sub<R|[1]>;
> R2 := sub<R|[2]>;
> R1 + R2;
Root system of dimension 4 of type A1 A1
> R1 join R2;
Root system of dimension 2 of type A1 A1

> R1 := RootSystem("A3T2B4T3");
> R2 := RootSystem("T3G2T4BC3");
> R1 + R2;
Root system of dimension 24 of type A3 B4 G2 BC3
> R1 join R2;
Root system of dimension 12 of type A3 B4 G2 BC3
```

`DirectSumDecomposition(R)`

`IndecomposableSummands(R)`

The set of irreducible direct summands of the semisimple root system R.

`Dual(R)`

The dual of the root system R, obtained by swapping the roots and coroots.

`IndivisibleSubsystem(R)`

The root system consisting of all indivisible roots of the root system R.

Example H96E17

```
> R1 := RootSystem("H4");
> R2 := RootSystem("B4");
> R1 + Dual(R2);
Root system of type H4 C4
> R := RootSystem("BC2");
> I := IndivisibleSubsystem(R); I;
I: Root system of type B2
> I subset R;
true [ 1, 2, 3, 5, 7, 8, 9, 11 ]
```

96.7 Related Structures

In this section functions for creating other structures from a root system are briefly listed. The reader is referred to the appropriate chapters of the Handbook for more details.

`RootDatum(R)`

The (split) root datum corresponding to the root system R. The coefficients of the simple roots and coroots must be integral; otherwise an error is signalled. See Chapter 97

`CoxeterGroup(GrpFPCox, R)`

The Coxeter group with root system R. See Chapter 98. The braid group and pure braid group can be computed from the Coxeter group using the commands in Section 98.12.

`CoxeterGroup(R)`

`CoxeterGroup(GrpPermCox, R)`

The permutation Coxeter group with root system R. See Chapter 98.

`ReflectionGroup(R)`

`CoxeterGroup(GrpMat, W)`

The reflection group of the root system R. See Chapter 99.

`LieAlgebra(R, k)`

The Lie algebra of the root system R over the base ring k. See Chapter 100.

`MatrixLieAlgebra(R, k)`

The matrix Lie algebra of the root system R over the base ring k. See Chapter 100.

Example H96E18

```
> R := RootSystem("b3");
> SemisimpleType(LieAlgebra(R, Rationals()));
B3
> #CoxeterGroup(R);
48
```

96.8 Bibliography

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Pap94**] Paolo Papi. A characterization of a special ordering in a root system. *Proc. Amer. Math. Soc.*, 120(3):661–665, 1994.

97 ROOT DATA

Chapter 97
ROOT DATA

97.1 Introduction

This chapter describes Magma functions for computing with (extended) root data. Root data are fundamental to Lie theory: Lie algebras (Chapter 100) and groups of Lie type (Chapter 103). Our description of split reduced root data follows [Dem65] and [Car93] except that reflections act on the right as in customary in MAGMA. Our description of extended root data follows [Sat71], [Sch69], and [Hal05]. Our description of split non-reduced root data follows [Bou68].

The closely related concept of a root system is discussed in Chapter 96. When working with Lie algebras or groups of Lie type, root data should be used. When working with Coxeter groups (Chapter 98) or reflection groups (Chapter 99), it is likely that only root systems are of interest.

97.1.1 Reflections

Let X and Y be free $\mathbf{Z}$-modules with bilinear pairing $\langle \circ, \circ \rangle : X \times Y \to \mathbf{Z}$ that identifies Y with the dual of X. Given nonzero $\alpha \in X$ and $\alpha^\star \in Y$, we define the $\mathbf{Z}$-linear map $s_\alpha : X \to X$ by

$$x s_\alpha = x - \langle x, \alpha^\star \rangle \alpha$$

and the $\mathbf{Z}$-linear map $s_\alpha^\star : Y \to Y$ by

$$y s_\alpha^\star = y - \langle \alpha, y \rangle \alpha^\star.$$

These maps are called *reflections* if one of the following equivalent properties hold: $\langle \alpha, \alpha^\star \rangle = 2$; ${s_\alpha}^2 = 1$; $\langle x s_\alpha, y s_\alpha^\star \rangle = \langle x, y \rangle$ for all $x \in X$ and $y \in Y$; $\alpha s_\alpha = -\alpha$. The map $s_\alpha^\star$ is also called a *coreflection*: this just means it is a reflection defined on Y instead of X. MAGMA functions for computing with reflections are described in Section 99.2.

If X has an inner product, then we can take $Y = X$ and use the inner product as our pairing. In MAGMA, X and Y are usually standard $\mathbf{Z}$-modules. However, it is sometimes useful to allow X and Y to be distinct sublattices of a standard lattice. The bilinear pairing is always given by the standard inner product: $\langle x, y \rangle = x y^T$.

97.1.2 Definition of a Split Root Datum

Suppose Φ is a finite subset of $X \setminus \{0\}$. For each α in Φ, suppose there is a corresponding $\alpha^\star$ in $Y \setminus \{0\}$; set $\Phi^\star = \{\alpha^\star \mid \alpha \in \Phi\}$. The datum $R = (X, \Phi, Y, \Phi^\star)$ is said to be a *(split) root datum* if the following conditions are satisfied for every α in Φ

1. s_α and $s_\alpha^\star$ are reflections;
2. Φ is closed under the action of s_α; and
3. $\Phi^\star$ is closed under the action of $s_\alpha^\star$.

The lattice X is called the *full root lattice* and Y the *full coroot lattice*. The vector space $X \otimes \mathbf{Q}$ is called the *root space* and $Y \otimes \mathbf{Q}$ the *coroot space*. The elements of Φ are called *roots* and the elements of $\Phi^\star$ are called *coroots*. A root datum is *reduced*, if $\alpha, \beta \in \Phi$ with β a scalar product of α implies $\alpha = \pm\beta$.

97.1.3 Simple and Positive Roots

A subset Δ of Φ is called a set of *simple roots* if

1. Δ is a basis for the rational span of the roots $\mathbf{Q}\Phi \leq \mathbf{Q} \otimes X$; and
2. $\Phi = \Phi^+ \cup \Phi^-$, where Φ^+ is the set of linear combinations of elements of Δ with nonnegative coefficients, and $\Phi^- = -\Phi^+$.

Every root datum has a set of simple roots. Simple roots are frequently called fundamental roots. The elements of Φ^+ are called *positive roots* and the elements of Φ^- *negative roots*. The coroots corresponding to the simple (resp. positive, negative) roots are the *simple* (respectively, *positive*, *negative*) *coroots*.

The *rank* of the root datum is the size of Δ, i.e. the dimension of the subspace $\mathbf{Q}\Phi$. The rank cannot be larger than the *dimension* of the root datum (i.e. the dimension of $\mathbf{Q} \otimes X$). If the rank and dimension are equal, the root datum is said to be *semisimple*.

Choose a basis $e_1, \ldots, e_d$ for X and a dual basis $f_1, \ldots, f_d$ for Y, so that $\langle e_i, f_j \rangle = \delta_{ij}$. A reduced root system is determined by a pair of integral matrices A and B where the rows of A are the simple roots and the rows of B are the corresponding coroots; i.e. $A_{ij} = \langle \alpha_i, f_j \rangle$ and $B_{ij} = \langle e_j, \alpha_i^\star \rangle$.

97.1.4 The Coxeter Group

The group W generated by the reflections s_α, for α a simple root, is a finite Coxeter group. The *Cartan matrix* of a root datum is

$$C = \left(\left\langle \, \alpha_i, \alpha_j^\star \, \right\rangle \right)_{i,j=1}^{n} = AB^t.$$

As in Chapter 95, the Cartan matrix is used to define the Coxeter matrix, Coxeter graph and Dynkin digraph of a root datum.

A *Coxeter form* is a W-invariant bilinear form on X. If R is reduced and irreducible, then the roots can have at most two different lengths with respect to this form. We call the roots *long* or *short* accordingly. The Coxeter form is normalised so that the short roots in each component have length one. Note that, even if $X = Y$, this form will generally not be the same as the pairing $\langle \circ, \circ \rangle$; however it can often be arranged for them to be the same (see `StandardRootSystem`).

97.1.5 Nonreduced Root Data

A root datum is *reduced*, if $\alpha, \beta \in \Phi$ with β a scalar product of α implies $\alpha = \pm\beta$. A root α with the property $2\alpha \notin \Phi$ is called *reduced*. A root α with the property $\frac{1}{2}\alpha \in \Phi$ is called *divisible*. If R is a root datum, then the set R_0 of indivisible roots in R form the *indivisible subsystem*.

Let R be a nonreduced irreducible root datum of rank n. It can be shown that R_0 is irreducible of type of type B_n and every root is either in R_0, or is two times a short root of R_0. The Cartan type of R in this case is BC_n.

Note that the Cartan matrix, Coxeter matrix, Coxeter diagram, Coxeter group and Dynkin diagram are the same for R and R_0. Thus, when creating a non-reduced root datum for a given Cartan matrix, Coxeter matrix, Coxeter diagram, Coxeter group or Dynkin diagram, one must specify the set of non-reduced fundamental roots. E.g., let C be a cartan matrix of type $B_2 \times B_3$. Then the set of nonreduced fundamental roots can be one of $\emptyset$, $\{2\}$, $\{5\}$ or $\{2, 5\}$, in which cases the root datum will be of types $B_2 \times B_3$, $BC_2 \times B_3$, $B_2 \times BC_3$ or $BC_2 \times BC_3$ respectively.

97.1.6 Isogeny of Split Reduced Root Data

The Dynkin digraph and dimension do not completely determine the isomorphism type of a split root datum, as the Coxeter graph and dimension do for a root system. Two root data with isomorphic Dynkin digraphs are said to be *Cartan equivalent*. We now describe the isomorphism classes within each Cartan equivalence class of split reduced irreducible root data. Since every semisimple reduced root datum is isogenous to a direct sum of irreducible root data, this immediately gives a classification of the split semisimple root data. Classifying nonsemisimple root data would be more complicated.

The *weights* of a root datum are the λ in $\mathbf{Q}\Phi \leq X \otimes \mathbf{Q}$ such that $\langle \lambda, \alpha^\star \rangle \in \mathbf{Z}$ for every coroot $\alpha^\star$. The weights form a lattice Λ called the *weight lattice*. We now have lattices $\mathbf{Z}\Phi \leq X \leq \Lambda$ (note that the second inclusion holds only for semisimple root data). The isomorphism class of a root datum in a fixed Cartan equivalence class is determined by the position of X between the root lattice $\mathbf{Z}\Phi$ and the weight lattice Λ. Alternatively, the isomorphism class is determined by the *isogeny group* $X/\mathbf{Z}\Phi$ within the *fundamental group* $\Lambda/\mathbf{Z}\Phi$. The fundamental group is determined by the Cartan matrix C: it is isometric to $\mathbf{Z}^n/\Theta$ where Θ is the lattice generated by the rows of C. The fundamental groups of the irreducible Cartan equivalence classes are

A_n: $\mathbf{Z}/(n+1)$;

B_n, C_n, E_7: $\mathbf{Z}/2$;

D_n: $\mathbf{Z}/4$ for n odd, $\mathbf{Z}/2 \times \mathbf{Z}/2$ for n even;

E_6: $\mathbf{Z}/3$;

E_8, F_4, G_2: trivial.

If $X = \mathbf{Z}\Phi$ the root datum is said to be *adjoint*; if $X = \Lambda$ it is said to be *simply connected*. The quotient $Y/\mathbf{Z}\Phi^\star$ is called the *coisogeny group*; in the semisimple case it is isomorphic to $\Lambda/\mathbf{Z}\Phi$.

97.1.7 Extended Root Data

An extended root datum is a split root datum $R = (X, \Phi, Y, \Phi^\star)$ and a permutation group Γ with actions on X and Y that respect the pairing $\langle \circ, \circ \rangle$.

Fix a set of simple roots Δ. Let $O(\chi)$ denote the orbit of $\chi \in X$ under the Γ-action. Then, for $\alpha \in \Phi$ either $O(\alpha)$ is contained in Φ^+, or it is contained in Φ^-, or the sum of the roots of $O(\alpha)$ is zero. We call $O(\alpha)$ a *positive*, *negative* or *zero orbit*, respectively. Put

$$X_0 := \{\chi \in X \mid \sum_{\gamma \in \Gamma} \chi^\gamma = 0\}.$$

Let $\Phi_0 := \Phi \cap X_0$ and $\Delta_0 := \Delta \cap X_0$. Then X_0 is a submodule of X, Φ_0 is a subsystem of Φ, and Δ_0 is a fundamental system of Φ_0. Note that Δ_0 is not necessarily a basis of X_0. Analogously, we define Y_0 and $\Phi_0^\star$. The subdatum $R_0 = (X_0, \Phi_0, Y_0, \Phi_0^\star)$ is called the *anisotropic subdatum* of R.

Set $\bar{X} := X/X_0$ and let $\pi : X \to \bar{X}$ be the standard projection. Then $\bar{X}$ is a free $\mathbf{Z}$-module and π is a homomorphism of modules. Let $\bar{\Phi}$ and $\bar{\Delta}$ be the images under π of $\Phi \setminus \Phi_0$ and $\Delta \setminus \Delta_0$, respectively. Then $\bar{\Phi}$ is a root system and $\bar{\Delta}$ is a fundamental system of it. We call $\bar{\Phi}$ the *relative root system* and $\bar{\Delta}$ the *relative fundamental system*. Note that $\bar{\Phi}$ need not be irreducible nor reduced even if Φ is. The rank of the relative system is $|\bar{\Delta}|$ and is called the *relative rank*, whereas the rank $|\Delta|$ of Φ is called the *absolute rank*. Let $\bar{\Phi}^+$ and $\bar{\Phi}^-$ denote the images under π of $\Phi^+ \setminus \Phi_0$ and $\Phi^- \setminus \Phi_0$. When $X_0 = X$, the relative root system is an empty set and the form is called *anisotropic*.

Each $\gamma \in \Gamma$ acts on X by $\chi \mapsto \chi^{\sigma w}$ for some unique $w \in W$ and σ a Dynkin diagram symmetry. By $\alpha \mapsto \alpha^\sigma$ for $\alpha \in \Delta$ we define the $[\Gamma]$-action on Δ. The extended root datum is called *inner* if the $[\Gamma]$-action is trivial and *outer* otherwise. The orbits of the $[\Gamma]$-action, that are not contained in X_0 are called *distinguished*.

An extended root datum is called *twisted* if the Γ-action is not trivial.

The (split) Cartan name of an extended root datum is the name of the corresponding split root datum. An extended root datum is *absolutely irreducible* if the corresponding split datum is irreducible. It is irreducible if there is no direct sum decomposition of the split datum which is preserved under the action of Γ. The *twisted Cartan name* of a root datum is the Cartan name, with extra information describing the twist. The name ${}^m X_{n,e}$ indicates a root datum with split Cartan name X_n, where the kernel of the $[\Gamma]$-action has index m in Γ, and e is the rank of the relative root system. The twisted Cartan name describes absolutely irreducible root data up to isomorphism. This is not true for simple root data however.

97.2 Constructing Root Data

We first describe some optional parameters that are common to many functions described below.

`Isogeny` ANY *Default :* `"Ad"`

The optional parameter `Isogeny` specifies the isomorphism class of the root datum within the Cartan equivalence class (see Subsection 97.1.6). For irreducible Cartan names, `Isogeny` can be one of the following:

1. A string: `"Ad"` for adjoint or `"SC"` for simply connected.
2. An integer giving the size of the isogeny subgroup within the fundamental group. The root datum must be absolutely irreducible. This does not work in type D_n with n even and `Isogeny` $= 2$, since in this case there are three distinct isomorphism classes (see the example below to create these data).
3. An injection of an abelian group into the fundamental group.

For compound Cartan names, `Isogeny` can be a string (`"Ad"` or `"SC"`); an injection into the fundamental group; or a list of strings, integers and injections (one for each direct summand).

`Signs` ANY *Default :* 1

Many of the constants associated with root data depend on the choice of the sign ϵ_{rs} for each extraspecial pair (r, s). This parameter allows the user to fix these signs for the root datum R by giving a sequence s of length `NumExtraspecialPairs(R)` consisting of integers 1 or -1. It is also possible to set `Signs` to 1 instead of a sequence of all 1 and to -1 instead of a sequence of all -1.

`Twist` ANY *Default :* 1

This optional parameter defines a Γ-action of an extended root datum and will accept the following values:

1. a homomorphism from Γ into `Sym(2*N)`, where N is the number of positive roots, specifying the action of Γ on the (co)roots. (Only for semisimple root data).
2. an integer i giving the order of Γ, e.g., $1, 2, 3, 6$ for ${}^1D_4$, ${}^2D_4$, ${}^3D_4$, ${}^6D_4$ (only if $i = 1$ or the root datum is irreducible).
3. $\langle D, i\rangle$, where D is a set of distinguished orbits as sets of integers and i (integer) is the order of the Dynkin diagram symmetry involved (only for irreducible root data).
4. $\langle \Gamma, ims\rangle$, where Γ is the acting group and ims define images either as permutations of the simple roots or as permutation of all roots (only for semisimple root data).
5. $\langle \Gamma, imsR, imsC\rangle$, where Γ is the acting group and $imsR$ ($imsC$) is a sequence of matrices defining the action of Γ on the root space (coroot space).

`Nonreduced` SETENUM *Default :* {}

The optional argument `Nonreduced` is used to give the set of indices of the nonreduced simple roots. Note that a root datum cannot be both twisted and nonreduced.

`RootDatum(N)`

`Isogeny`	ANY	*Default :* `"Ad"`
`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1

A root datum with Cartan name given by the string N (see Section 95.6). In addition to the possible Cartan names described in Section 95.6, this function will also accept `"Tn"` as a component of the Cartan name, which stands for an n-dimensional toral subdatum. Note, however, that this addition is for input only and will not appear in the string returned by `CartanName` when applied to the resulting root datum (see example below).

If the optional parameter `Isogeny` is a list, its length should be equal to the total number of components. Entries of this list corresponding to toral components will be ignored.

If the corresponding Coxeter group is infinite affine, an error is flagged.

Example H97E1

Examples of adjoint and simply connected irreducible root data.

```
> RootDatum("E6");
Adjoint root datum of type E6
> RootDatum("E6" : Isogeny := "SC");
Simply connected root datum of type E6
```

With nonirreducible root data the isogeny can be given as a list.

```
> R := RootDatum("A5 B3" : Isogeny := [* 3, "Ad" *]);
> R : Maximal;
Root datum of type A5 B3 with simple roots
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[1 2 0 1 3 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
and simple coroots
[ 2 -1  0  0  0  0  0  0]
[-1  2 -1  0 -1  0  0  0]
[ 0 -1  2 -1  1  0  0  0]
[ 0  0 -1  2 -1  0  0  0]
[ 0  0  0 -1  1  0  0  0]
[ 0  0  0  0  0  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -2  2]
>
```

```
> RootDatum("E6 A3 B4" : Isogeny := "SC");
Simply connected root datum of type E6 A3 B4
```

Nonsemisimple root data can be constructed by specifying a central torus.

```
> R := RootDatum("B3 T2 A2" : Isogeny := [* "SC", 0, "Ad" *]);
> R;
R: Root datum of type B3 A2
> Dimension(R), Rank(R);
7 5
> SimpleCoroots(R);
[ 1  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0]
[ 0  0  1  0  0  0  0]
[ 0  0  0  0  0  2 -1]
[ 0  0  0  0  0 -1  2]
```

The following code creates the three root data of type D_6 with isogeny groups of size 2 using injections into the fundamental group.

```
> G< a, b > := FundamentalGroup("D6");
> G;
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*a = 0
    2*b = 0
> _, inj1 := sub< G | a >;
> R1 := RootDatum("D6" : Isogeny := inj1);
> _, inj2 := sub< G | b >;
> R2 := RootDatum("D6" : Isogeny := inj2);
> _, inj3 := sub< G | a*b >;
> R3 := RootDatum("D6" : Isogeny := inj3);
```

Example H97E2

Examples of extended root data:

```
> R := RootDatum("A5" : Twist := 2 ); R;
R: Twisted adjoint root datum of type 2A5,3
> R eq RootDatum("A5" : Twist := < Sym(2), [Sym(5)|(1,5)(2,4)] > );
true
> R eq RootDatum("A5" : Twist := < {{1,5},{2,4},{3}}, 2 > );
true
> RootDatum("D4" : Twist := 1);
Adjoint root datum of type D4
> RootDatum("D4" : Twist := 2);
Twisted adjoint root datum of type 2D4,3
> RootDatum("D4" : Twist := 3);
Twisted adjoint root datum of type 3D4,2
```

```
> RootDatum("D4" : Twist := 6);
Twisted adjoint root datum of type 6D4,2
>
> R := RootDatum("A2");
> TwistedRootDatum(R : Twist := 2);
Twisted adjoint root datum of type 2A2,1
```

`RootDatum(C)`

`Isogeny`	ANY	*Default :* `"Ad"`
`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1
`Nonreduced`	SETENUM	*Default :* {}

A semisimple root datum with crystallographic Cartan matrix C. If the corresponding Coxeter group is infinite, an error is flagged.

`RootDatum(D)`

`Isogeny`	ANY	*Default :* `"Ad"`
`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1
`Nonreduced`	SETENUM	*Default :* {}

A semisimple root datum with Dynkin digraph D. If the corresponding Coxeter group is infinite, an error is flagged.

`RootDatum(A, B)`

`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1
`Nonreduced`	SETENUM	*Default :* {}

The root datum with simple roots given by the rows of the matrix A and simple coroots given by the rows of the matrix B. The matrices A and B must have the following properties:

1. A and B must be integral matrices with the same number of rows and the same number of columns;
2. the number of columns must be at least the number of rows; and
3. AB^t must be the Cartan matrix of a finite Coxeter group.

Example H97E3

An example of a nonsemisimple root system of type G_2:

```
> A := Matrix(2,3, [1,-1,0, -1,1,-1]);
> B := Matrix(2,3, [1,-1,1, 0,1,-1]);
> RootDatum(A, B);
Root datum of type G2
```

An example of a non-reduced root datum and usage of `Nonreduced` argument:

```
> C := CoxeterMatrix("B2B2");
> RootDatum(C);
Adjoint root datum of type B2 B2
> RootDatum(C : Nonreduced:={2});
Adjoint root datum of type BC2 B2
> RootDatum(C : Nonreduced:={4});
Adjoint root datum of type B2 BC2
> RootDatum(C : Nonreduced:={2,4});
Adjoint root datum of type BC2 BC2
```

`IrreducibleRootDatum(X, n)`

`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1

The irreducible root datum with Cartan name X_n.

`StandardRootDatum(X, n)`

`Signs`	ANY	*Default :* 1
`Twist`	ANY	*Default :* 1

The standard root datum with Cartan name X_n, i.e. the root datum with the standard inner product equal to the Coxeter form up to a constant. For technical reasons, this is only possible for the classical types, i.e. X must be `"A"`, `"B"`, `"C"`, or `"D"`. Note that the standard root datum is not semisimple for type A_n.

Example H97E4

These functions are useful in loops.

```
> for X in ["A","B","G"] do
>   print NumPosRoots(IrreducibleRootDatum(X, 2));
> end for;
3
4
6
```

`ToralRootDatum(n)`

`Twist` ANY *Default :* 1

The toral root datum of dimension n, i.e., the n-dimensional root datum with no roots or coroots.

Example H97E5

Toral root datum of dimension 3 and a twisted version of it:

```
> ToralRootDatum(3);
Toral root datum of dimension 3
> M := Matrix(Rationals(),3,3,[0,1,0,1,0,0,0,0,1]);M;
[0 1 0]
[1 0 0]
[0 0 1]
> ToralRootDatum(3 : Twist := <Sym(2),[M],[M]>);
Twisted toral root datum of dimension 3
```

`TrivialRootDatum()`

The trivial root datum of dimension 0.

97.2.1 Constructing Sparse Root Data

Sparse root data differ from the usual root data only in the internal representation of the objects. The internal representation is less memory expensive and requires less time for creation. Sparse root data have type `RootDtmSprs`, which is a subcategory of `RootDtm`.

There are some limitation on the root data which can have sparse representation. First, sparse representation only makes sense for classical root data, that is of types A, B, C and D. At the moment only root data with a connected Coxeter diagram may have sparse representation and no twisted sparse root data can be constructed. T

`SparseRootDatum(N)`

`SparseRootDatum(N)`

`SparseRootDatum(C)`

`SparseRootDatum(D)`

`SparseRootDatum(R)`

`SparseRootDatum(A, B)`

`SparseIrreducibleRootDatum(X, n)`

`SparseStandardRootDatum(X, n)`

These functions have the same syntax as their counterparts without the "Sparse" in the name (see Section 97.2). The root datum returned has sparse representation. See [CHM08] for the algorithms used to construct sparse root data.

Example H97E6

```
> SparseRootDatum("A2");
Sparse adjoint root datum of dimension 2 of type A2
> SparseStandardRootDatum("A", 2);
Sparse root datum of dimension 3 of type A2
> SparseRootDatum("A2") eq RootDatum("A2");
true
```

`SparseRootDatum(R)`

Return a sparse root datum equal to the root datum R.

`RootDatum(R)`

Return a non-sparse root datum equal to the root datum R.

Example H97E7

Due to the restrictions mentioned above, some operations that create new root data, will return a non-sparse root datum even though the input was sparse.

```
> R := SparseRootDatum("A2");
> T := ToralRootDatum(3);
> R+T;
Sparse root datum of dimension 5 of type A2
> R+R;
Adjoint root datum of dimension 4 of type A2 A2
```

97.3 Operations on Root Data

`R1 eq R2`

Returns `true` if, and only if, R_1 and R_2 are identical root data.

`IsIsomorphic(R1, R2)`

Returns `true` if, and only if, R_1 and R_2 are isomorphic root data. If `true`, the second value returned is a sequence giving the simple root of R_2 corresponding to each simple root of R_1, and the third value returned is an isomorphism $R_1 \to R_2$. This function is currently only implemented for semisimple root data.

`IsCartanEquivalent(R1, R2)`

Returns `true` if, and only if, the root data R_1 and R_2 are Cartan equivalent, i.e. they have isomorphic Dynkin diagrams. If `true`, the second value returned is a sequence giving the simple root of R_2 corresponding to each simple root of R_1.

`IsIsogenous(R1, R2)`

Returns `true` if, and only if, R_1 and R_2 are isogenous root data. If `true`, the subsequent values returned are: a sequence giving the simple root of R_2 corresponding to each simple root of R_1, the corresponding adjoint root datum R_{ad}, the morphisms $R_{ad} \to R_1$ and $R_{ad} \to R2$, the corresponding simply connected root datum R_{sc}, and the morphisms $R_1 \to R_{sc}$ and $R_2 \to R_{sc}$.

Example H97E8

An example of isogenous root data:

```
> R1 := RootDatum("A3");
> R2 := RootDatum("A3" : Isogeny := "SC");
> R1 eq R2;
false
> IsIsomorphic(R1, R2);
false
> IsCartanEquivalent(R1, R2);
true [ 1, 2, 3 ]
> IsIsogenous(R1, R2);
true [ 1, 2, 3 ]
Adjoint root datum of type A3
Mapping from: RootDtm: ad to RootDtm: ad
Mapping from: RootDtm: ad to RootDtm: sc
Simply connected root datum of type A3
Mapping from: RootDtm: ad to RootDtm: sc
Mapping from: RootDtm: sc to RootDtm: sc
```

An example of distinct isomorphic root data:

```
> C := CartanMatrix("B2");
> R1 := RootDatum(C);
```

```
> R2 := RootDatum(Transpose(C));
> R1; R2;
Adjoint root datum of type B2
Adjoint root datum of type C2
> R1 eq R2;
false
> IsIsomorphic(R1, R2);
true [ 2, 1 ]
```

`CartanName(R)`

The Cartan name of the root datum R (Section 95.6).

`TwistedCartanName(R)`

The twisted Cartan name of the root datum R. E.g., `"2A_3,2"`.

`CoxeterDiagram(R)`

Print the Coxeter diagram of the root datum R (Section 95.6).

`DynkinDiagram(R)`

Print the Dynkin diagram of the root datum R (Section 95.6).

`CoxeterMatrix(R)`

The Coxeter matrix of the root datum R (Section 95.2).

`CoxeterGraph(R)`

The Coxeter graph of the root datum R (Section 95.3).

`CartanMatrix(R)`

The Cartan matrix of the root datum R (Section 95.4).

`DynkinDigraph(R)`

The Dynkin digraph of the root datum R (Section 95.5).

Example H97E9

```
> R := RootDatum("F4");
> DynkinDiagram(R);
F4    1 - 2 =>= 3 - 4
> CoxeterDiagram(R);
F4    1 - 2 === 3 - 4
```

Example H97E10

```
> R := RootDatum("G2");
> RootSpace(R);
Standard Lattice of rank 2 and degree 2
> CorootSpace(R);
Standard Lattice of rank 2 and degree 2
> // Add RootLattice, CorootLattice.
> // and maybe move (Co)RootSpace and (Co)RootLattice
> // to after introducing them
> SimpleRoots(R);
[1 0]
[0 1]
> SimpleCoroots(R);
[ 2 -3]
[-1  2]
> CartanMatrix(R);
[ 2 -1]
[-3  2]
> Rank(R) eq Dimension(R);
true
```

`GammaAction(R)`

The Γ-action of the root datum R. This is a record consisting of four elements: `gamma` is the Group Γ acting on R, `perm_ac` is the homomorphism defining the permutation action of Γ on the set of all roots of R, finally `mats_rt` and `mats_co` are sequences of matrices defining the action of Γ on the root and coroot spaces of R.

`GammaRootSpace(R)`

`GammaCorootSpace(R)`

The fixed space Γ acting on the (co)root space of R.

`GammaOrbitOnRoots(R,r)`

The orbit through the rth root of the Γ-action on the root datum R.

`GammaOrbitsOnRoots(R)`
`PositiveGammaOrbitsOnRoots(R)`
`NegativeGammaOrbitsOnRoots(R)`
`ZeroGammaOrbitsOnRoots(R)`

The sequence of all (respectively positive, negative and zero) orbits of the Γ-action on the root datum R (Section 97.1.7).

`GammaActionOnSimples(R)`

The [Γ]-action on the simple (co)roots of the root datum R. (Section 97.1.7). This function was called `GammaActionPi` in the last release.

`OrbitsOnSimples(R)`

The sequence of all orbits of the [Γ]-action on the simple (co)roots of the root datum R (Section 97.1.7). This function was called `OrbitsPi` in the last release.

`DistinguishedOrbitsOnSimples(R)`

The sequence of distinguished orbits of the [Γ]-action on the simple (co)roots of the root datum R (Section 97.1.7). This function was called `DistinguishedOrbitsPi` in the last release.

`BaseRing(R)`

The base ring of the root datum R is the field of rational numbers.

`Rank(R)`
`AbsoluteRank(R)`

The (absolute) rank of the root datum R, i.e. the number of simple (co)roots.

`RelativeRank(R)`

The relative rank of the root datum R, i.e. the number of simple (co)roots of the relative root system. This is the same as absolute rank for split root data.

`Dimension(R)`

The dimension of the root datum R, i.e. the dimension of the (co)root space. This is at least as large as the rank, with equality when R is semisimple.

`TwistingDegree(R)`

The twisting degree of the root datum R, i.e. the order of Γ divided by the kernel of the [Γ]-action.

`AnisotropicSubdatum(R)`

The anisitropic subdatum of the root datum R.

Example H97E11

Consider the twisted root datum of type ${}^2A_{3,1}$ with distinguished orbit $\{2\}$:

```
> R := RootDatum( "A3" : Twist := < {{2}}, 2 > );
```

First, print out the action of Γ on the root datum:

```
> GammaAction(R);
rec<recformat<gamma: GrpPerm, perm_ac: HomGrp, mats_rt, mats_co> |
     gamma := Permutation group acting on a set of cardinality 4
     Order = 4 = 2^2
          (1, 2, 3, 4),
     perm_ac := Homomorphism of GrpPerm: $, Degree 4, Order 2^2 into GrpPerm: $,
     Degree 12, Order 2^10 * 3^5 * 5^2 * 7 * 11 induced by
          (1, 2, 3, 4) |--> (1, 3, 7, 9)(2, 4, 6, 5)(8, 10, 12, 11),
     mats_rt := [
          [ 0  0  1]
          [ 1  1  0]
          [-1  0  0]
     ],
     mats_co := [
          [ 0  0  1]
          [ 0  1  0]
          [-1  1  0]
     ]
     >
```

Compute the orbits of the Γ-action:

```
> PositiveGammaOrbitsOnRoots(R);
[
     GSet{ 2, 4, 5, 6 }
]
> NegativeGammaOrbitsOnRoots(R);
[
     GSet{ 8, 10, 11, 12 }
]
> ZeroGammaOrbitsOnRoots(R);
[
     GSet{ 1, 3, 7, 9 }
]
> &+[ Root(R,r) : r in ZeroGammaOrbitsOnRoots(R)[1] ];
(0 0 0)
```

Compute the $[\Gamma]$-action and its orbits:

```
> GammaActionOnSimples(R);
Homomorphism of GrpPerm: $, Degree 4, Order 2^2 into GrpPerm: $,
Degree 3, Order 2 * 3 induced by
     (1, 2, 3, 4) |--> (1, 3)
```

```
> OrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 1, 3 }
]
> DistinguishedOrbitsOnSimples(R);
[
     GSet{ 2 }
]
```

Absolute and relative rank and the twisting degree, as well as their appearance in the name of the root datum:

```
> AbsoluteRank(R);
3
> RelativeRank(R);
1
> TwistingDegree(R);
2
> R;
R: Twisted adjoint root datum of type 2A3,1
```

anisotropic subdatum:

```
> A := AnisotropicSubdatum(R); A;
A: Twisted root datum of type 2(A1 A1)2,0
> GammaAction(A)‘perm_ac;
Homomorphism of GrpPerm: $, Degree 4, Order 2^2 into GrpPerm: $,
Degree 4, Order 2^2 induced by
     (1, 2, 3, 4) |--> (1, 2, 3, 4)
```

`CoxeterGroupOrder(R)`

The order of the (split) Coxeter group of the root datum R.

`GroupOfLieTypeOrder(R, q)`

The order of the group of Lie type with split root datum R over the field of cardinality q.

`GroupOfLieTypeFactoredOrder(R, q)`

The factored order of the group of Lie type with split root datum R over the field of order q.

Example H97E12

As well as accepting a specific prime power, these functions also take an indeterminate so that the generic order formula can be computed.

```
> P<q> := PolynomialRing(Integers());
> R := RootDatum("F4");
> GroupOfLieTypeFactoredOrder(R, q);
[
    <q - 1, 4>,
    <q, 24>,
    <q + 1, 4>,
    <q^2 - q + 1, 2>,
    <q^2 + 1, 2>,
    <q^2 + q + 1, 2>,
    <q^4 - q^2 + 1, 1>,
    <q^4 + 1, 1>
]
>
> R := RootDatum("B2");
> ord := GroupOfLieTypeOrder(R, q);
> forall{ q : q in [2..200] | not IsPrimePower(q) or
>   Evaluate(ord, q) eq GroupOfLieTypeOrder(R, q) };
true
```

`FundamentalGroup(R)`

The fundamental group $\Lambda/\mathbf{Z}\Phi$ of the root datum R together with the projection $\Lambda \to \Lambda/\mathbf{Z}\Phi$. See Subsection 97.1.6.

`IsogenyGroup(R)`

The isogeny group $X/\mathbf{Z}\Phi$ of the root datum R together with the projection $X \to X/\mathbf{Z}\Phi$. If R is semisimple, the injection $X/\mathbf{Z}\Phi \to \Lambda/\mathbf{Z}\Phi$ is also returned. See Subsection 97.1.6.

`CoisogenyGroup(R)`

The coisogeny group $Y/\mathbf{Z}\Phi^\star$ of the root datum R together with the projection $Y \to Y/\mathbf{Z}\Phi^\star$. If R is semisimple, the projection $Y/\mathbf{Z}\Phi^\star \to \Lambda/\mathbf{Z}\Phi$ is also returned. See Subsection 97.1.6.

Example H97E13

In the semisimple case, the fundamental group contains the isogeny group, with quotient isomorphic to the coisogeny group.

```
> R := RootDatum("A5" : Isogeny := 3);
> F := FundamentalGroup(R);
> G := IsogenyGroup(R);
> H := CoisogenyGroup(R);
> #G * #H eq #F;
true
```

Nonsemisimple root data have infinite isogeny groups.

```
> R := StandardRootDatum("A", 5);
> IsogenyGroup(R);
Abelian Group isomorphic to Z
Defined on 1 generator (free)
```

97.4 Properties of Root Data

`IsFinite(R)`

Returns `true` for any root datum R.

`IsIrreducible(R)`

Returns `true` if, and only if, the root datum R is irreducible.

`IsAbsolutelyIrreducible(R)`

Returns `true` if, and only if, the split version of the root datum R is irreducible.

`IsProjectivelyIrreducible(R)`

Returns `true` if, and only if, the quotient of the root datum R modulo its radical is irreducible. This is equivalent for R to have a connected Coxeter diagram.

`IsReduced(R)`

Returns `true` if, and only if, the root datum R is reduced.

`IsSemisimple(R)`

Returns `true` if, and only if, the root datum R is semisimple, i.e. its rank is equal to its dimension.

`IsCrystallographic(R)`

Returns `true` for any root datum R.

`IsSimplyLaced(R)`

Returns `true` if, and only if, the root datum R is simply laced, i.e. its Dynkin diagram contains no multiple bonds.

`IsAdjoint(R)`

Returns `true` if, and only if, the root datum R is adjoint, i.e. its isogeny group is trivial.

`IsWeaklyAdjoint(R)`

Returns `true` if, and only if, the root datum R is weakly adjoint, i.e. its isogeny group is isomorphic to $\mathbf{Z}^n$, where n is $\dim(R) - \mathrm{rk}(R)$. Note that if R is semisimple then this function is identical to IsAdjoint.

`IsSimplyConnected(R)`

Returns `true` if, and only if, the root datum R is simply connected, i.e. its isogeny group is equal to the fundamental group, i.e. its coisogeny group is trivial.

`IsWeaklySimplyConnected(R)`

Returns `true` if, and only if, the root datum R is weakly simply connected, i.e. its coisogeny group is isomorphic to $\mathbf{Z}^n$, where n is $\dim(R) - \mathrm{rk}(R)$. Note that if R is semisimple then this function is identical to IsSimplyConnected.

Example H97E14

```
> R := RootDatum("A5 B2" : Isogeny := "SC");
> IsIrreducible(R);
false
> IsSimplyLaced(R);
false
> IsSemisimple(R);
true
> IsAdjoint(R);
false
```

For some of the exceptional isogeny classes, there is only one isomorphism class of root data, which is both adjoint and simply connected.

```
> R := RootDatum("G2");
> IsAdjoint(R);
true
> IsSimplyConnected(R);
true
```

There exist root data that are neither adjoint nor simply connected.

```
> R := RootDatum("A3" : Isogeny := 2);
> IsAdjoint(R), IsSimplyConnected(R);
```

```
false false
```

Finally, we demonstrate a case where the root datum is not adjoint, but is weakly adjoint.

```
> R := RootDatum("A2T1");
> IsAdjoint(R), IsWeaklyAdjoint(R);
false true
> Dimension(R), Rank(R);
3 2
> G := IsogenyGroup(R); G;
Abelian Group isomorphic to Z
Defined on 1 generator (free)
```

`IsReduced(R)`

Returns `true` if, and only if, the root datum R is reduced.

`IsSplit(R)`

Returns `true` if, and only if, the root datum R is split, i.e. the Γ-action is trivial.

`IsTwisted(R)`

Returns `true` if, and only if, the root datum R is twisted, i.e. the Γ-action is not trivial.

`IsQuasisplit(R)`

Returns `true` if, and only if, the root datum R is quasisplit, i.e. the anisotropic subdatum is trivial.

`IsInner(R)`
`IsOuter(R)`

Returns `true` if, and only if, the root datum R is inner (resp. outer).

`IsAnisotropic(R)`

Returns `true` if, and only if, the root datum R is anisotropic, i.e. when $X = X_0$.

97.5 Roots, Coroots and Weights

The roots are stored as an indexed set

$$\{@\ \alpha_1, \ldots, \alpha_N, \alpha_{N+1}, \ldots, \alpha_{2N}\ @\},$$

where $\alpha_1, \ldots, \alpha_N$ are the positive roots in an order compatible with height; and $\alpha_{N+1}, \ldots, \alpha_{2N}$ are the corresponding negative roots (i.e. $\alpha_{i+N} = -\alpha_i$). The simple roots are $\alpha_1, \ldots, \alpha_n$ where n is the rank.

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space. This is the default.
2. `"Root"`: the basis of simple (co)roots.
3. `"Weight"`: the basis of fundamental (co)weights (see Subsection 99.8.3 below).

97.5.1 Accessing Roots and Coroots

`RootSpace(R)`

`CorootSpace(R)`

The vector space containing the (co)roots of the root datum R, i.e. $X \otimes \mathbf{Q}$ (respectively, $Y \otimes \mathbf{Q}$).

`FullRootLattice(R)`

`FullCorootLattice(R)`

The lattice containing the (co)roots of the root datum R, i.e. X (respectively, Y). An inclusion map into the (co)root space of R is returned as the second value.

`RootLattice(R)`

`CorootLattice(R)`

The lattice spanned by the (co)roots of the root datum R. An inclusion map into the (co)root space of R is returned as the second value.

Example H97E15

The root space, full root lattice and the root lattice of the standard root datum of type A_2:

```
> R := StandardRootDatum("A",2);
> V := RootSpace(R);
> FullRootLattice(R);
Standard Lattice of rank 3 and degree 3
Mapping from: Standard Lattice of rank 3 and degree 3 to ModTupFld: V
> RootLattice(R);
Lattice of rank 2 and degree 3
Basis:
( 1 -1  0)
( 0  1 -1)
Mapping from: Lattice of rank 2 and degree 3 to ModTupFld: V
```

`IsRootSpace(V)`

`IsCorootSpace(V)`

Return `true` if, and only if, V is the (co)root space of some root datum.

`IsInRootSpace(v)`

`IsCorootSpace(v)`

Return `true` if, and only if, V is an element of the (co)root space of some root datum.

`RootDatum(V)`

If V is the (co)root space of some root datum, this returns the datum.

Example H97E16

```
> R := RootDatum("a3");
> V := RootSpace(R);
> v := V.1;
> IsRootSpace(V);
true
> RootDatum(V);
R: Adjoint root datum of dimension 3 of type A3
> IsInRootSpace(v);
true
```

`ZeroRootLattice(R)`

`ZeroRootSpace(R)`

For the given root datum R, return the lattice X_0 and the vector space $X_0 \otimes \mathbf{Q}$, respectively (see Section 97.1.7).

`RelativeRootSpace(R)`

For the given root datum R, return the vector space $\bar{X} = (X \otimes \mathbf{Q})/(X_0 \otimes \mathbf{Q})$ containing the relative roots (see Section 97.1.7). The projection from $X \otimes \mathbf{Q}$ onto $\bar{X}$ is returned as second return value.

`SimpleRoots(R)`

`SimpleCoroots(R)`

The simple (co)roots of the root datum R as the rows of a matrix, i.e. A (respectively, B).

`NumberOfPositiveRoots(R)`

`NumPosRoots(R)`

The number of positive roots of the root datum R. This is also the number of positive coroots. The total number of (co)roots is twice the number of positive (co)roots.

`Roots(R)`

`Coroots(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The indexed set of (co)roots of the root datum R, i.e. $\{@\, \alpha_1, \ldots \alpha_{2N}\, @\}$ (respectively, $\{@\, \alpha_1^\star, \ldots \alpha_{2N}^\star\, @\}$).

`PositiveRoots(R)`

`PositiveCoroots(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The indexed set of positive (co)roots of the root datum R, i.e. $\{@\, \alpha_1, \ldots \alpha_N\, @\}$ (respectively, $\{@\, \alpha_1^\star, \ldots \alpha_N^\star\, @\}$).

`Root(R, r)`

`Coroot(R, r)`

`Basis` MonStgElt *Default :* `"Standard"`

The rth (co)root α_r (respectively, $\alpha_r^\star$) of the root datum R.

`RootPosition(R, v)`

`CorootPosition(R, v)`

`Basis` MonStgElt *Default :* `"Standard"`

If v is a (co)root in the root datum R, return its index; otherwise return 0. These functions will try to coerce v into the appropriate lattice; v should be written with respect to the basis specified by the parameter `Basis`.

`BasisChange(R,v)`

`InBasis` MonStgElt *Default :* `"Standard"`

`OutBasis` MonStgElt *Default :* `"Standard"`

`Coroots` BoolElt *Default :* `false`

Changes the basis of the vector v contained in the space spanned by the (co)roots of the root datum R. The vector v is considered as an element of the root space by default. If the parameter `Coroots` is set to `true`, v is considered as an element of the coroot space. The optional arguments `InBasis` and `OutBasis` may take the same values as the parameter `Basis` as described at the beginning of the current section.

Example H97E17

```
> R := RootDatum("A3" : Isogeny := 2);
> Roots(R);
{@
    (1 0 0),
    (0 1 0),
    (1 0 2),
    (1 1 0),
    (1 1 2),
    (2 1 2),
    (-1  0  0),
    (0 -1  0),
    (-1  0 -2),
    (-1 -1  0),
    (-1 -1 -2),
    (-2 -1 -2)
@}
> PositiveCoroots(R);
{@
    (2 -1 -1),
    (-1  2  0),
    (0 -1  1),
    (1  1 -1),
    (-1  1  1),
    (1 0 0)
@}
> #Roots(R) eq 2*NumPosRoots(R);
true
> Coroot(R, 4);
(1  1 -1)
> Coroot(R, 4 : Basis := "Root");
(1 1 0)
> CorootPosition(R, [1,1,-1]);
4
> CorootPosition(R, [1,1,0] : Basis := "Root");
4
> BasisChange(R, [1,0,0] : InBasis:="Root");
(1 0 0)
> BasisChange(R, [1,0,0] : InBasis:="Root", Coroots);
( 2 -1 -1)
```

`IsInRootSpace(R,v)`

`IsInCorootSpace(R,v)`

Returns `true` if and only if the vector v is contained in the (co)root space of the root datum R.

`HighestRoot(R)`

`HighestCoroot(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The unique (co)root of greatest height in the irreducible root datum R.

`HighestLongRoot(R)`

`HighestLongCoroot(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The unique long (co)root of greatest height in the irreducible root datum R.

`HighestShortRoot(R)`

`HighestShortCoroot(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The unique short (co)root of greatest height in the irreducible root datum R.

Example H97E18

```
> R := RootDatum("G2");
> HighestRoot(R);
(3 2)
> HighestLongRoot(R);
(3 2)
> HighestShortRoot(R);
(2 1)
```

`RelativeRoots(R)`

`PositiveRelativeRoots(R)`

`NegativeRelativeRoots(R)`

`SimpleRelativeRoots(R)`

The indexed set of all (resp. positive, negative, simple) relative roots of the root datum R. Note that the relative roots are returned in the order induced by the standard ordering on the (nonrelative) roots of R.

`RelativeRootDatum(R)`

The relative root datum of the root datum R.

`GammaOrbitsRepresentatives(R, delta)`

The preimage of a relative root δ is a disjoint union of Γ-orbits on the set of all roots of the root datum R. This intrinsic returns a sequence of representatives of these orbits.

Example H97E19

We first consider the twisted root datum of type ${}^2E_6$, which is quasisplit:

```
> DynkinDiagram(RootDatum("E6"));
E6    1 - 3 - 4 - 5 - 6
              |
              2
>
> R := RootDatum("E6" : Twist:=2 ); R;
R: Twisted adjoint root datum of type 2E6,4
> OrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 4 },
     GSet{ 1, 6 },
     GSet{ 3, 5 }
]
> DistinguishedOrbitsOnSimples(R) eq OrbitsOnSimples(R);
true
> AnisotropicSubdatum(R);
Twisted toral root datum of dimension 6
[]
> RR := RelativeRootDatum(R);RR;
RR: Adjoint root datum of type F4
> _,pi := RelativeRootSpace(R);
> DynkinDiagram(RR);
F4    1 - 2 =>= 4 - 3
> [ Position(Roots(RR), pi(Root(R,i)) ) : i in [1,6, 3,5, 4, 2]];
[ 3, 3, 4, 4, 2, 1 ]
```

now one with distinguished orbits $\{2\}$ and $\{4\}$:

```
> R := RootDatum("E6" : Twist := <{{2},{4}},2> ); R;
R: Twisted adjoint root datum of type 2E6,2
> OrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 4 },
     GSet{ 1, 6 },
     GSet{ 3, 5 }
```

```
]
> DistinguishedOrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 4 }
]
> AnisotropicSubdatum(R);
Twisted root datum of type 2(A2 A2)4,0
[ 1, 3, 5, 6, 7, 11, 37, 39, 41, 42, 43, 47 ]
> RR := RelativeRootDatum(R);RR;
RR: Adjoint root datum of type G2
> DynkinDiagram(RR);
G2    2 =<= 1
        3
> _,pi := RelativeRootSpace(R);
> [ Position(Roots(RR), pi(Root(R,i)) ) : i in [2,4]];
[ 1, 2 ]
```

and now the one with distinguished orbits $\{2\}$ and $\{1,6\}$, which has a non-reduced relative root datum:

```
> R := RootDatum("E6" : Twist := <{{2},{1,6}},2> ); R;
R: Twisted adjoint root datum of dimension 6 of type 2E6,2
> OrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 4 },
     GSet{ 1, 6 },
     GSet{ 3, 5 }
]
> DistinguishedOrbitsOnSimples(R);
[
     GSet{ 2 },
     GSet{ 1, 6 }
]
> AnisotropicSubdatum(R);
Twisted root datum of type 2A3,0
[ 3, 4, 5, 9, 10, 15, 39, 40, 41, 45, 46, 51 ]
> RR := RelativeRootDatum(R);RR;
RR: Adjoint root datum of type BC2
> DynkinDiagram(RR);
BC2    1 =>= 2
> _,pi := RelativeRootSpace(R);
> [ Position(Roots(RR), pi(Root(R,i)) ) : i in [2, 1,6]];
[ 1, 2, 2 ]
```

Finally, the twisted root Datum of type ${}^6D_{4,1}$:

```
> T := RootDatum( "D4" : Twist:=<{{2}},6> );
> T;
```

```
T: Twisted adjoint root datum of dimension 4 of type 6D4,1
> RelativeRootDatum(T);
Adjoint root datum of type BC1
> GammaOrbitsRepresentatives(T,1);
[ 11, 5 ]
> GammaOrbitsRepresentatives(T,2);
[ 12 ]
```

`CoxeterForm(R)`

`DualCoxeterForm(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The matrix of an inner product on the (co)root space of the root datum R which is invariant under the action of the (co)roots. The inner product is normalised so that the short roots in each irreducible component have length one.

97.5.2 Reflections

The root α acts on the root space via the reflection s_α; the coroot $\alpha^\star$ acts on the coroot space via the coreflection $s^\star_\alpha$.

`SimpleReflectionMatrices(R)`

`SimpleCoreflectionMatrices(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The sequence of matrices giving the action of the simple (co)roots of the root datum R on the (co)root space, i.e. the matrices of $s_{\alpha_1}, \ldots, s_{\alpha_n}$ (respectively, $s^\star_{\alpha_1}, \ldots, s^\star_{\alpha_n}$).

`ReflectionMatrices(R)`

`CoreflectionMatrices(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The sequence of matrices giving the action of the (co)roots of the root datum R on the (co)root space, i.e. the matrices of $s_{\alpha_1}, \ldots, s_{\alpha_{2N}}$ (respectively, $s^\star_{\alpha_1}, \ldots, s^\star_{\alpha_{2N}}$).

`ReflectionMatrix(R, r)`

`CoreflectionMatrix(R, r)`

`Basis` MonStgElt *Default :* `"Standard"`

The matrix giving the action of the rth (co)root of the root datum R on the (co)root space, i.e. the matrix of s_{α_r} (respectively, $s^\star_{\alpha_r}$).

`SimpleReflectionPermutations(R)`

The sequence of permutations giving the action of the simple (co)roots of the root datum R on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutations(R)`

The sequence of permutations giving the action of the (co)roots of the root datum R on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutation(R, r)`

The permutation giving the action of the rth (co)root of the root datum R on the (co)roots. This action is the same for roots and coroots.

`ReflectionWords(R)`

The sequence of words in the simple reflections for all the reflections of the root datum R. These words are given as sequences of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWords(R)[r]`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

`ReflectionWord(R, r)`

The word in the simple reflections for the rth reflection of the root datum R. The word is given as a sequence of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWord(R,r)`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

Example H97E20

```
> R := RootDatum("A3" : Isogeny := 2);
> mx := ReflectionMatrix(R, 4);
> perm := ReflectionPermutation(R, 4);
> wd := ReflectionWord(R, 4);
> RootPosition(R, Root(R,2) * mx) eq 2^perm;
true
> perm eq &*[ ReflectionPermutation(R, r) : r in wd ];
true
>
> mx := CoreflectionMatrix(R, 4);
> CorootPosition(R, Coroot(R,2) * mx) eq 2^perm;
true
```

97.5.3 Operations and Properties for Root and Coroot Indices

`Sum(R, r, s)`

The index of the sum of the rth and sth roots in the root datum R, or 0 if the sum is not a root. In other words, if $t =$ `Sum(R,`r`,`s`)` $\neq 0$ then $\alpha_t = \alpha_r + \alpha_s$. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied.

`IsPositive(R, r)`

Returns `true` if, and only if, the rth (co)root of the root datum R is a positive root.

`IsNegative(R, r)`

Returns `true` if, and only if, the rth (co)root of the root datum R is a negative root.

`Negative(R, r)`

The index of the negative of the rth (co)root of the root datum R. In other words, if $s =$ `Negative(R,`r`)` then $\alpha_s = -\alpha_r$.

`LeftString(R, r, s)`

Indices in the root datum R of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s - \alpha_r, \alpha_s - 2\alpha_r, \ldots, \alpha_s - p\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_p]$ where $\alpha_{r_i} = \alpha_s - i\alpha_r$ and $\alpha_s - (p+1)\alpha_r$ is not a root. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied.

`RightString(R, r, s)`

Indices in the root datum R of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s + \alpha_r, \alpha_s + 2\alpha_r, \ldots, \alpha_s + q\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_q]$ where $\alpha_{r_i} = \alpha_s + i\alpha_r$ and $\alpha_s + (q+1)\alpha_r$ is not a root. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied.

`LeftStringLength(R, r, s)`

The largest p such that $\alpha_s - p\alpha_r$ is a root of the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`RightStringLength(R, r, s)`

The largest q such that $\alpha_s + q\alpha_r$ is a root of the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

Example H97E21

```
> R := RootDatum("G2");
> Sum(R, 1, Negative(R,5));
10
> IsPositive(R, 10);
false
> Negative(R, 10);
4
> P := PositiveRoots(R);
> P[1] - P[5] eq -P[4];
true
```

`RootHeight(R, r)`

`CorootHeight(R, r)`

The height of the rth (co)root of the root datum R, i.e. the sum of the coefficients of α_r (respectively, $\alpha_r^\star$) with respect to the simple (co)roots.

`RootNorms(R)`

`CorootNorms(R)`

The sequence of squares of the lengths of the (co)roots of the root datum R.

`RootNorm(R, r)`

`CorootNorm(R, r)`

The square of the length of the rth (co)root of the root datum R.

`IsLongRoot(R, r)`

Returns `true` if, and only if, the rth root of the root datum R is long. This only makes sense for irreducible crystallographic root data. Note that for non-reduced root data, the roots which are not indivisible, are actually longer than the long ones.

`IsShortRoot(R, r)`

Returns `true` if, and only if, the rth root of the root datum R is short. This only makes sense for irreducible crystallographic root data.

`IsIndivisibleRoot(R, r)`

Returns `true` if, and only if, the rth root of the root system R is indivisible.

Example H97E22

Note that the Coxeter form is defined over the rationals. Since it is not possible to multiply a lattice element by a rational matrix, (co)roots must be coerced into a rational vector space first.

```
> R := RootDatum("G2");
> RootHeight(R, 5);
4
> F := CoxeterForm(R);
> v := VectorSpace(Rationals(),2) ! Root(R, 5);
> (v*F, v) eq RootNorm(R, 5);
true
> IsLongRoot(R, 5);
true
> LeftString(R, 1, 5);
[ 4, 3, 2 ]
> roots := Roots(R);
> for i in [1..3] do
>   RootPosition(R, roots[5]-i*roots[1]);
> end for;
4
3
2
```

`RootClosure(R, S)`

The closure in the root datum R of the set S of root indices. That is the indices of every root that can be written as a sum of roots with indices in S.

`AdditiveOrder(R)`

An additive order on the positive roots of the root datum R, ie. a sequence containing the numbers $1, \ldots, N$ in some order such that $\alpha_r + \alpha_s = \alpha_t$ implies t is between r and s. This is computed using the techniques of [Pap94]

`IsAdditiveOrder(R, Q)`

Returns `true` if, and only if, Q gives an additive order on a set of positive roots of R. Q must be a sequence of integers in the range $[1..N]$ with no gaps or repeats.

Example H97E23

```
> R := RootDatum("A5");
> a := AdditiveOrder(R);
> Position(a, 2);
6
> Position(a, 3);
10
> Position(a, Sum(R, 2, 3));
7
```

97.5.4 Weights

`WeightLattice(R)`

The weight lattice Λ of the root datum R. i.e. the λ in $\mathbf{Q}\Phi \leq \mathbf{Q} \otimes X$ such that $\langle \lambda, \alpha^\star \rangle \in \mathbf{Z}$ for every coroot $\alpha^\star$.

`CoweightLattice(R)`

The coweight lattice $\Lambda^\star$ of the root datum R, i.e. the $\lambda^\star$ in $\mathbf{Q}\Phi^\star \leq \mathbf{Q} \otimes Y$ such that $\langle \alpha, \lambda^\star \rangle \in \mathbf{Z}$ for every root α.

`FundamentalWeights(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The fundamental weights $\lambda_1, \ldots, \lambda_n$ of the root datum R given as the rows of a matrix. This is the basis of the weight lattice Λ dual to the simple coroots, i.e. $\langle \lambda_i, \alpha_j^\star \rangle = \delta_{ij}$.

`FundamentalCoweights(R)`

`Basis` MonStgElt *Default :* `"Standard"`

The fundamental coweights $\lambda_1^\star, \ldots, \lambda_n^\star$ of the root datum R given as the rows of a matrix. This is the basis of the coweight lattice $\Lambda^\star$ dual to the simple roots, i.e. $\langle \alpha_i, \lambda_j^\star \rangle = \delta_{ij}$.

Example H97E24

```
> R := RootDatum("E6");
> WeightLattice(R);
Lattice of rank 6 and degree 6
Basis:
(4  3  5  6  4  2)
(3  6  6  9  6  3)
(5  6 10 12  8  4)
(6  9 12 18 12  6)
(4  6  8 12 10  5)
(2  3  4  6  5  4)
Basis Denominator: 3
> FundamentalWeights(R);
[ 4/3    1  5/3    2  4/3  2/3]
[   1    2    2    3    2    1]
[ 5/3    2 10/3    4  8/3  4/3]
[   2    3    4    6    4    2]
[ 4/3    2  8/3    4 10/3  5/3]
[ 2/3    1  4/3    2  5/3  4/3]
```

`IsDominant(R, v)`

`Basis` MonStgElt *Default :* `"Standard"`

Returns `true` if, and only if, v is a dominant weight for the root datum R, ie, a nonnegative integral linear combination of the fundamental weights.

`DominantWeight(R, v)`

`Basis` MonStgElt *Default :* `"Standard"`

The unique dominant weight in the same W-orbit as the weight v, where W is the Weyl group of the root datum R. The second value returned is a Weyl group element taking v to the dominant weight. The weight v can be given either as a vector or as a sequence representing the vector and is coerced into the weight lattice first.

`WeightOrbit(R, v)`

`Basis` MonStgElt *Default :* `"Standard"`

The W-orbit of the weight v as an indexed set, where W is the Weyl group of the root datum R. The first element in the orbit is always dominant. The second value returned is a sequence of Weyl group elements taking the dominant weight to the corresponding element of the orbit. The weight v can be given either as a vector or as a sequence representing the vector and is coerced into the weight lattice first.

Example H97E25

```
> R := RootDatum("B3");
> DominantWeight(R, [1,-1,0] : Basis:="Weight");
(1 0 0)
[ 2, 3, 2, 1 ]
> #WeightOrbit(R, [1,-1,0] : Basis:="Weight");
6
```

97.6 Building Root Data

`sub< R | a >`

The root subdatum of the root datum R generated by the roots $\alpha_{a_1}, \ldots, \alpha_{a_k}$ where $a = \{a_1, \ldots, a_k\}$ is a set of integers.

`sub< R | s >`

The root subdatum of the root datum R generated by the roots $\alpha_{s_1}, \ldots, \alpha_{s_k}$ where $s = [s_1, \ldots, s_k]$ is a *sequence* of integers. In this version the roots must be simple in the root subdatum (i.e. none of them may be a summand of another) otherwise an error is signalled. The simple roots will appear in the subdatum in the given order.

Example H97E26

```
> R := RootDatum("A4");
> PositiveRoots(R);
{@
    (1 0 0 0),
    (0 1 0 0),
    (0 0 1 0),
    (0 0 0 1),
    (1 1 0 0),
    (0 1 1 0),
    (0 0 1 1),
    (1 1 1 0),
    (0 1 1 1),
    (1 1 1 1)
@}
> s := sub< R | [6,1,4] >;
> s;
Root datum of type A3
> PositiveRoots(s);
{@
    (0 1 1 0),
    (1 0 0 0),
```

```
    (0 0 0 1),
    (1 1 1 0),
    (0 1 1 1),
    (1 1 1 1)
@}
> s := sub< R | [1,5] >;
Error: The given roots are not simple in a subdatum
> s := sub< R | {1,5} >;
> s;
Root datum of type A2
> PositiveRoots(s);
{@
    (1 0 0 0),
    (0 1 0 0),
    (1 1 0 0)
@}
```

`R1 subset R2`

Returns `true` if and only if the root datum R_1 is a subset of the root datum R_2. If true, returns an injection as sequence of roots as second return value.

`R1 + R2`

`DirectSum(R1, R2)`

The external direct sum of the root data R_1 and R_2. The full (co)root space of the result is the direct sum of the full (co)root spaces of R_1 and R_2.

`R1 join R2`

The internal direct sum of the root data R_1 and R_2. The root data must have the same full (co)root space, which will also be the full (co)root space of the result. The root data must have disjoint (co)root spaces.

Example H97E27

```
> R  := RootDatum("A1A1");
> R1 := sub<R|[1]>;
> R2 := sub<R|[2]>;
> R1 + R2;
Root datum of dimension 4 of type A1 A1
> R1 join R2;
R: Adjoint root datum of dimension 2 of type A1 A1
```

`DirectSumDecomposition(R)`
`IndecomposableSummands(R)`

Returns a sequence Q of irreducible root data, a root datum S which is the direct sum of the terms of Q, and an isogeny map $\phi : S \to R$. The root datum R must be semisimple. Note that a semisimple root datum R need not be a direct sum of simple root data, but it is isogenous to a direct sum of root data S.

Example H97E28

If the root datum in adjoint or simply connected, then it is a direct sum of simples. In this case we get $S = R$.

```
> R := RootDatum("A4B5" : Isogeny:="SC");
> Q, S := DirectSumDecomposition( R );
> R eq S;
true
> R eq Q[1] join Q[2];
true
```

The join of the summands of the direct sum decomposition is the original root datum again:

```
> R eq &join DirectSumDecomposition(R);
true
> R eq &+    DirectSumDecomposition(R);
false

> R1 := RootDatum("A3T2B4T3");
> R2 := RootDatum("T3G2T4BC3");
> R1 + R2;
Adjoint root datum of dimension 24 of type A3 B4 G2 BC3
> R1 join R2;
Root datum of dimension 12 of type A3 B4 G2 BC3
```

Here is an example of a semisimple root datum which is not a direct sum of simple subdata. Note that a simple root datum of type A_1 is either simply connected or adjoint.

```
> G<a,b>:=FundamentalGroup("A1A1");
> _,inj:=sub<G|a*b>;
> R:=RootDatum("A1A1":Isogeny:=inj);
> ad := RootDatum( "A1" : Isogeny:="Ad" );
> sc := RootDatum( "A1" : Isogeny:="SC" );
> IsIsomorphic( R, DirectSum(ad,ad) );
false
> IsIsomorphic( R, DirectSum(ad,sc) );
false
> IsIsomorphic( R, DirectSum(sc,sc) );
false
> Q, S := DirectSumDecomposition( R );
> R eq S;
```

```
false
```

`Dual(R)`

The dual of the root datum R, obtained by swapping the roots and coroots. The second value returned is the dual morphism from R to its dual.

`SimplyConnectedVersion(R)`

The simply connected version of the root datum R. If R is semisimple then the injection of the simply connected version into R is returned as the second value.

`AdjointVersion(R)`

The adjoint version of the root datum R. If R is semisimple then the projection from R to its adjoint version is returned as the second value.

`IndivisibleSubdatum(R)`

The root datum consisting of all indivisible roots of the root datum R.

`Radical(R)`

The radical of the root datum R, ie, the toral subdatum whose root (resp. coroot) space consists of the vectors perpendicular to every coroot (resp. root).

Example H97E29

An adjoint or simply connect root datum is always a direct sum of irreducible subdata. In these cases we take $S = R$.

```
> R1 := RootDatum("A5");
> R2 := RootDatum("B4");
> R := DirectSum(R1, Dual(R2));
> DirectSumDecomposition(R);
{
    Root datum of type A5 ,
    Root datum of type C4
}

> R := RootDatum("BC2");
> I := IndivisibleSubdatum(R); I;
I: Root datum of type B2
> I subset R;
true [ 1, 2, 3, 5, 7, 8, 9, 11 ]

> R := StandardRootDatum("A", 3);
> Radical(R);
Toral root datum of dimension 1
```

TwistedRootDatum(R)

TwistedRootDatum(N)

Twist ANY *Default :* 1

Create a twisted root datum from the root datum R, or from the semisimple root datum with Cartan name N. The twist may be specified in any of the following ways:

- An integer, specifying the order of the twist;
- A permutation, specifying the action of the primitive roots;
- A pair $< D, i >$, where D is a set of distinguished orbits as sets of integers, and i is the order of the Dynkin diagram symmetry;
- A pair $< \Gamma, Q >$, where Γ is the acting group, and Q is a sequence containing the permutation of the primitive roots for each of the generators of Γ;
- A homomorphism from Γ to the symmetric group whose order is the number of roots of R, describing how the acting group Γ acts on the roots.

Example H97E30

We construct a twisted root datum in a number of ways.

```
> S := TwistedRootDatum("D4" : Twist := 3);
> S;
S: Twisted adjoint root datum of dimension 4 of type 3D4,2

> R := RootDatum("A1A3");
> DynkinDiagram(R);

A1    1

A3    2 - 3 - 4
> S := TwistedRootDatum(R : Twist := Sym(4)!(2,4));
> S;
S: Twisted adjoint root datum of dimension 4 of type 2(A1 A3)4,3

> S := TwistedRootDatum("A4" : Twist := <{{1,4},{2,3}}, 2>);
> S;
S: Twisted adjoint root datum of dimension 4 of type 2A4,2

> R := RootDatum("E6" : Isogeny := "SC");
> DynkinDiagram(R);

E6    1 - 3 - 4 - 5 - 6
              |
              2
> S := TwistedRootDatum(R : Twist := <Sym(2) ,[ Sym(6)!(1,6)(3,5) ]>);
> S;
```

```
S: Twisted simply connected root datum of dimension 6 of type 2E6,4

> R := RootDatum("D4");
> DynkinDiagram(R);

D4    3
      /
1 - 2
      \
       4
> Gamma := Sym(3);
> Gamma.1, Gamma.2;
(1, 2, 3)
(1, 2)
> S := TwistedRootDatum(R : Twist := <Gamma, [ Sym(4) | (1,3,4), (1,4) ]>);
> S;
S: Twisted adjoint root datum of dimension 4 of type 6D4,2

> R := RootDatum("A2");
> DynkinDiagram(R);

A2    1 - 2
> Roots(R);
{@
    (1 0),
    (0 1),
    (1 1),
    (-1  0),
    ( 0 -1),
    (-1 -1)
@}
> S6 := Sym(#Roots(R));
> phi := hom<Sym(2) -> S6 | S6!(1,2)(4,5)>;
> S := TwistedRootDatum(R : Twist := phi);
```

`UntwistedRootDatum(R)`

`SplitRootDatum(R)`

The split version of the (twisted) root datum R.

97.7 Morphisms of Root Data

Morphisms are currently only defined for split root data. Let $R_i = (X_i, \Phi_i, Y_i, \Phi_i^\star)$ be a root datum for $i = 1, 2$. A *morphism* of root data $\phi : R_1 \to R_2$ consists of a pair of $\mathbf{Z}$-linear maps $\phi_X : X_1 \to X_2$ and $\Phi_Y : Y_1 \to Y_2$ satisfying

1. $\phi_X(\Phi_1) \subseteq \Phi_2 \cup \{0\}$; and
2. $\phi_Y(\alpha^\star) = \phi_X(\alpha)^\star$ (with the convention that $0^\star = 0$).

A *fractional morphism* is similar, except that it consists of $\mathbf{Q}$-linear maps on the (co)root spaces $X_1 \otimes \mathbf{Q} \to X_2 \otimes \mathbf{Q}$ and $Y_1 \otimes \mathbf{Q} \to Y_2 \otimes \mathbf{Q}$. The main examples of fractional morphisms are isogeny maps (Section 97.1.6). A *dual morphism* is similar, except that the maps are $X_1 \to Y_2$ and $Y_1 \to X_2$. This is clearly equivalent to a morphism from R_1 to the dual of R_2. Finally we define a *dual fractional morphism* in the obvious way.

A (fractional) morphism $\phi : R_1 \to R_2$ also stores a sign corresponding to each simple root of R_1. This has no effect on the action of ϕ on roots or coroots, but does effect the definition of the corresponding homomorphisms of Lie algebras and groups of Lie type.

```
hom< R -> S | phiX, phiY >
```

```
hom< R -> S | phiX, phiY >
```

Construct a (fractional) morphism of root data $R \to S$ with the given linear maps or matrices of linear maps.

```
hom< R -> S | Q >
```

Construct a (fractional) morphism of root data $R \to S$ with the given sequence of root images. The sequence Q must have length $2N$ and consist of elements in the range $[0, \ldots, 2M]$, where N is the number of positive roots of R and M is the number of positive roots of S. The domain R must be semisimple.

```
Morphism(R, S, phiX, phiY)
```

```
Morphism(R, S, phiX, phiY)
```

`SimpleSigns`	.	*Default :* 1
`Check`	BoolElt	*Default :* `true`

Construct a (fractional) morphism of root data $R \to S$ with the given sequence of root images. The sequence Q must have length $2N$ and consist of elements in the range $[0, \ldots, 2M]$, where N is the number of positive roots of R and M is the number of positive roots of S. The domain R must be semisimple.

`SimpleSigns` is a sequence of signs corresponding to the simple roots, or ± 1 to indicate a constant sequence. If `Check` is set to `false`, the function does not check that the maps send (co)roots to (co)roots. This function is the same as the constructor `hom`, except for these optional parameters.

`Morphism(R, S, Q)`

`SimpleSigns` . *Default :* 1

`Check` BoolElt *Default :* `true`

Construct a (fractional) morphism of root data $R \to S$ with the given sequence of root images. The sequence Q must have length $2N$ and consist of elements in the range $[0, \ldots, 2M]$, where N is the number of positive roots of R and M is the number of positive roots of S. The domain R must be semisimple.

`SimpleSigns` is a sequence of signs corresponding to the simple roots, or ± 1 to indicate a constant sequence. If `Check` is set to `false`, the function does not check that the maps send (co)roots to (co)roots. This function is the same as the constructor `hom`, except for these optional parameters.

`DualMorphism(R, S, phiX, phiY)`

`DualMorphism(R, S, phiX, phiY)`

`Check` BoolElt *Default :* `true`

Construct a (fractional) dual morphism of root data $R \to S$ with the given linear maps or matrices of linear maps. If `Check` is set to `false`, the function does not check that the maps send (co)roots to (co)roots.

`DualMorphism(R, S, Q)`

`Check` BoolElt *Default :* `true`

Construct a (fractional) dual morphism of root data $R \to S$ with the given sequence of root images. The sequence Q must have length $2N$ and consist of elements in the range $[0, \ldots, 2M]$, where N is the number of positive roots of R and M is the number of positive roots of S. The domain R must be semisimple. If `Check` is set to `false`, the function does not check that the maps send (co)roots to (co)roots.

`RootImages(phi)`

The indices of the root images of the (dual) (fractional) morphism ϕ.

`RootPermutation(phi)`

The indices of the root images of the automorphism ϕ.

`IsIsogeny(phi)`

Returns `true` if the morphism ϕ is an isogeny, ie, ϕ_Y is onto with finite kernel.

`IdentityMap(R)`

`IdentityAutomorphism(R)`

The identity morphism $R \to R$.

Example H97E31

We construct the fractional morphism from the standard root datum of type A_3 onto the adjoint root datum of type A_3. This will allow us to construct the algebraic projection $GL_4 \to PGL_4$ in Section 103.11.

```
> RGL := StandardRootDatum( "A", 3 );
> RPGL := RootDatum( "A3" );
> A := VerticalJoin( SimpleRoots(RGL), Vector([Rationals()|1,1,1,1]) )^-1 *
>   VerticalJoin( SimpleRoots(RPGL), Vector([Rationals()|0,0,0]) );
> B := VerticalJoin( SimpleCoroots(RGL), Vector([Rationals()|1,1,1,1]) )^-1 *
>   VerticalJoin( SimpleCoroots(RPGL), Vector([Rationals()|0,0,0]) );
> phi := hom< RGL -> RPGL | A, B >;
> v := Coroot(RGL,1);
> v; phi(v);
( 1 -1  0  0)
( 2 -1  0 )
```

97.8 Constants Associated with Root Data

In this section functions for a number of constants associated with root data will be described. These constants are needed to define Lie algebras and groups of Lie type. The notation of [Car72] will be used, except that the constants are defined for right actions rather than left actions [CMT04].

`ExtraspecialPairs(R)`

The sequence of extraspecial pairs of the root datum R (see [Car72, page 58]). That is the sequence $[(r_i, s_i)]_{i=1}^{N-n}$ where r_i is minimal such that $\alpha_{r_i} + \alpha_{s_i} = \alpha_{i+n}$ (n is the rank of R and N is the number of positive roots).

`NumExtraspecialPairs(R)`

The number of extraspecial pairs of the root datum R. This function doesn't actually compute the extraspecial pairs, thus is much more efficient than calling `#ExtraspecialPairs(R)` in case extraspecial pairs are not yet computed.

`ExtraspecialPair(R,r)`

The extraspecial pair of the rth root in the root datum R. That is the pair (s, t) where s is minimal such that $\alpha_s + \alpha_t = \alpha_r$.

`ExtraspecialSigns(R)`

Return the sequence of extraspecial signs of the root datum R.

`LieConstant_p(R, r, s)`

The constant p_{rs} for the root datum R, i.e. the largest p such that $\alpha_s - p\alpha_r$ is a root. This is the same as `LeftStringLength`. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`LieConstant_q(R, r, s)`

The constant q_{rs} for the root datum R, i.e. the largest q such that $\alpha_s + q\alpha_r$ is a root. This is the same as `RightStringLength`. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`CartanInteger(R, r, s)`

The Cartan integer $\langle \alpha_r, \alpha_s^\star \rangle$ for the root datum R.

`LieConstant_N(R, r, s)`

The Lie algebra structure constant N_{rs} for the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`LieConstant_epsilon(R, r, s)`

The constant $\epsilon_{rs} = \mathrm{Sign}(N_{rs})$ for the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`LieConstant_M(R, r, s, i)`

The constant $M_{rsi} = \frac{1}{i!} N_{s_0 r} \cdots N_{s_{i-1} r}$ where $\alpha_{s_i} = i\alpha_r + \alpha_s$ for the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`LieConstant_C(R, i, j, r, s)`

The Lie group structure constant C_{ijrs} for the root datum R. The conditions $\alpha_s \neq \pm\alpha_r$ and $\alpha_r + \alpha_s \in \Phi$ must be satisfied.

`LieConstant_eta(R, r, s)`

The constant

$$\eta_{rs} = (-1)^{p_{rs}} \frac{\epsilon_{r,s-pr} \cdots \epsilon_{r,s-r}}{\epsilon_{r,s-pr} \cdots \epsilon_{r,s+(q-p-1)r}}$$

for the root datum R. The condition $\alpha_s \neq \pm\alpha_r$ must be satisfied.

`StructureConstants(R)`

The Lie algebra structure constants for the reductive Lie algebra with root datum R in the sparse format described in Section 100.2.

Example H97E32

The code below verifies some standard formulas in the root datum of type F_4:

```
> R := RootDatum("F4");
> N := NumPosRoots(R);
> r := Random([1..N]);
> s := Random([1..r-1] cat [r+1..r+N-1] cat [r+N+1..2*N]);
```

1. Agreement of the Cartan matrix with the Cartan integers.

```
> C := CartanMatrix(R);
> C[2,3] eq CartanInteger(R,2,3);
true
```

2. p_{rs} is the length of the left string through α_s in the direction of α_r.

```
> LieConstant_p(R,r,s) eq #LeftString(R,r,s);
true
```

3. q_{rs} is the length of the right string through α_s in the direction of α_r.

```
> LieConstant_q(R,r,s) eq #RightString(R,r,s);
true
```

4. $\langle \alpha_s, \alpha_r^\star \rangle = p_{rs} - q_{rs}$.

```
> CartanInteger(R,s,r) eq
> LieConstant_p(R,r,s) - LieConstant_q(R,r,s);
true
```

5. $N_{rs} = \epsilon_{rs}(p_{rs} + 1)$.

```
> LieConstant_N(R,r,s) eq
> LieConstant_epsilon(R,r,s) * (LieConstant_p(R,r,s) + 1);
true
```

97.9 Related Structures

In this section functions for creating other structures from a root datum are briefly listed. See the appropriate chapters of the Handbook for more details.

`RootSystem(R)`

The root system corresponding to the root datum R. See Chapter 96.

`CoxeterGroup(GrpFPCox, R)`

The (split) Coxeter group with root datum R. See Chapter 98. The braid group and pure braid group can be computed from the Coxeter group using the commands described in Section 98.12.

`CoxeterGroup(R)`

`CoxeterGroup(GrpPermCox, R)`

The permutation Coxeter group with root datum R. See Chapter 98.

`ReflectionGroup(R)`

`CoxeterGroup(GrpMat, R)`

The reflection group of the root datum R. See Chapter 99.

`LieAlgebraHomorphism(phi,k)`

The homomorphism of reductive Lie algebras over the ring k corresponding to the root datum morphism ϕ. See Chapter 100.

`LieAlgebra(R, k)`

The reductive Lie algebra over the ring k with root datum R. See Chapter 100.

`GroupOfLieType(R, k)`

The group of Lie type over the ring k with root datum R. See Chapter 103.

`GroupOfLieTypeHomomorphism(phi, k)`

The algebraic homomorphism of groups of Lie type over the ring k corresponding to the root datum morphism ϕ. See Chapter 103.

Example H97E33

```
> R := RootDatum("b3");
> SemisimpleType(LieAlgebra(R, Rationals()));
B3
> #CoxeterGroup(R);
48
> GroupOfLieType(R, Rationals());
$: Group of Lie type B3 over Rational Field
```

97.10 Bibliography

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Car72**] Roger W. Carter. *Simple groups of Lie type.* John Wiley & Sons, London-New York-Sydney, 1972. Pure and Applied Mathematics, Vol. 28.

[**Car93**] Roger W. Carter. *Finite groups of Lie type.* John Wiley & Sons, Chichester, 1993. Conjugacy classes and complex characters, Reprint of the 1985 original, A Wiley-Interscience Publication.

[**CHM08**] Arjeh M. Cohen, Sergei Haller, and Scott H. Murray. Computing in unipotent and reductive algebraic groups. *LMS J. Comput. Math.*, 11:343–366, 2008.

[**CMT04**] Arjeh M. Cohen, Scott H. Murray, and D. E. Taylor. Computing in groups of Lie type. *Math. Comp.*, 73(247):1477–1498, 2004.

[**Dem65**] M. Demazure. Donées radicielles. In *Schémas en Groupes (Sém. Géométrie Algébrique, Inst. Hautes études Sci., 1964), Fasc. 6, Exposé 21*, page 85. Inst. Hautes Études Sci., Paris, 1965.

[**Hal05**] Sergei Haller. *Computing Galois Cohomology and Forms of Linear Algebraic Groups.* Phd thesis, Technical University of Eindhoven, 2005.

[**Pap94**] Paolo Papi. A characterization of a special ordering in a root system. *Proc. Amer. Math. Soc.*, 120(3):661–665, 1994.

[**Sat71**] I. Satake. *Classification theory of semi-simple algebraic groups.* Marcel Dekker Inc., New York, 1971. With an appendix by M. Sugiura, Notes prepared by Doris Schattschneider, Lecture Notes in Pure and Applied Mathematics, 3.

[**Sch69**] Doris J. Schattschneider. On restricted roots of semi-simple algebraic groups. *J. Math. Soc. Japan*, 21:94–115, 1969.

98 COXETER GROUPS

Chapter 98
COXETER GROUPS

98.1 Introduction

This chapter describes Magma functions for computing with Coxeter groups. A *Coxeter system* is a group G with finite generating set $S = \{s_1, \ldots, s_n\}$, defined by relations $s_i^2 = 1$ for $i = 1, \ldots, n$ and

$$s_i s_j s_i \cdots = s_j s_i s_j \cdots$$

for $i, j = 1, \ldots, n$ with $i < j$, where each side of this relation has length $m_{ij} \geq 2$. Traditionally, $m_{ij} = \infty$ signifies that the corresponding relation is omitted but, for technical reasons, $m_{ij} = 0$ is used in MAGMA instead. The group G is called a *Coxeter group* and S is called the set of *Coxeter generators*. Since every group in MAGMA has a preferred generating set, no distinction is made between a Coxeter system and its Coxeter group. See [Bou68] for more details on the theory of Coxeter groups.

The *rank* of the Coxeter system is $n = |S|$. A Coxeter system is said to be *reducible* if there is a proper subset I of $\{1, \ldots, n\}$ such that $m_{ij} = 2$ or $m_{ji} = 2$ whenever $i \in I$ and $j \notin I$. In this case, G is an (internal) direct product of the Coxeter subgroups $W_I = \langle s_i \mid i \in I \rangle$ and $W_{I^c} = \langle s_i \mid i \notin I \rangle$. Note that an *irreducible* Coxeter group may still be a nontrivial direct product of abstract subgroups (for example, $W(G_2) \cong S_2 \times S_3$). Two Coxeter groups are *Coxeter isomorphic* if there is a group isomorphism between them which takes Coxeter generators to Coxeter generators. In other words, the two groups are the same modulo renumbering of the generators.

MAGMA provides three methods for working with Coxeter groups:

1. As a finitely presented group with the standard presentation given above. These groups have type `GrpFPCox`. See Chapter 70 for general functions for finitely presented groups.
2. As a permutation group acting on the roots of the root system. Clearly the group must be finite. These groups have type `GrpPermCox`. See Chapter 58 for general functions for permutation groups.
3. As a reflection group, i.e. a matrix group generated by reflections. These groups have the same type as general matrix groups (`GrpMat`). They can be distinguished with the `IsReflectionGroup` function.

The first two methods are described in this chapter. The third is described in Chapter 99.

A permutation Coxeter group always has an underlying root system or root datum, and so many commands involving roots also work for these groups. A finitely presented Coxeter group does not have such an underlying structure.

The code for Coxeter groups as permutation groups was originally modelled on the corresponding part of the Chevie package of GAP [GHL+96] by Meinholf Geck, Frank Lübeck, Jean Michel and Götz Pfeiffer.

98.1.1 The Normal Form for Words

Every element w of a Coxeter group W can be written as a word

$$w = r_1 r_2 \cdots r_l$$

with each r_i in S. A *reduced expression* for w is such a word with l minimal; in this case, l is defined to be the *length* of w.

An ordering on words in S is obtained by taking the *lexicographic (alphabetic) order* induced by the existing ordering on S. The *normal form* for w in W is the smallest reduced expression for w with respect to this ordering. Algorithms for efficiently computing this normal form have been developed and implemented by R. B. Howlett. These algorithms are based on the concept of a minimal root [Bri98, BH93].

The main difference of the category of Coxeter groups (GrpFPCox) from the category of finitely presented groups (GrpFP) is that that all words are automatically put into this normal form. In particular, this means that two words are equal if, and only if, they are equal as group elements. Coxeter groups can also be constructed in the category GrpFP if the user wishes to avoid automatic normalisation of elements (see Section 98.3).

98.2 Constructing Coxeter Groups

It is possible to specify the category GrpFPCox or GrpPermCox when constructing a Coxeter group. If the category is not specified, then a GrpPermCox is returned for finite groups and a GrpFPCox is returned for infinite groups. If the category GrpPermCox is specified for an infinite group, an error is signalled.

```
CoxeterGroup(GrpFPCox, N)
```

```
CoxeterGroup(GrpPermCox, N)
```

```
CoxeterGroup(N)
```

The finite or affine Coxeter group with Cartan name given by the string N (see Section 95.6).

```
IrreducibleCoxeterGroup(GrpFPCox, X, n)
```

The finite or affine irreducible Coxeter group with Cartan name X_n, or $I_2(n)$ if $X =$"I" (see Section 95.6).

Example H98E1

```
> CoxeterGroup(GrpFPCox, "B3");
Coxeter group: Finitely presented group on 3 generators
Relations
    $.1 * $.2 * $.1 = $.2 * $.1 * $.2
    $.1 * $.3 = $.3 * $.1
    ($.2 * $.3)^2 = ($.3 * $.2)^2
```

```
    $.1^2 = Id($)
    $.2^2 = Id($)
    $.3^2 = Id($)
> CoxeterGroup("A2B2");
Coxeter group: Permutation group acting on a set of cardinality 14
Order = 48 = 2^4 * 3
    (1, 8)(2, 5)(9, 12)
    (1, 5)(2, 9)(8, 12)
    (3, 10)(4, 6)(11, 13)
    (3, 7)(4, 11)(10, 14)
```

CoxeterGroup(GrpFPCox, M)
CoxeterGroup(GrpPermCox, M)
CoxeterGroup(M)

The Coxeter group with Coxeter matrix M (see Chapter 95).

CoxeterGroup(GrpFPCox, G)
CoxeterGroup(GrpPermCox, G)
CoxeterGroup(G)

The Coxeter group with Coxeter graph G (see Chapter 95).

CoxeterGroup(GrpFPCox, C)
CoxeterGroup(GrpPermCox, C)
CoxeterGroup(C)

The Coxeter group with Cartan matrix C (see Chapter 95).

CoxeterGroup(GrpFPCox, D)
CoxeterGroup(GrpPermCox, D)
CoxeterGroup(D)

The Coxeter group with Dynkin digraph D (see Chapter 95).

Example H98E2

```
> M := SymmetricMatrix([ 1, 4,1, 3,4,1 ]);
> G<a,b,c> := CoxeterGroup(M);
> G;
Coxeter group: Finitely presented group on 3 generators
Relations
    (a * b)^2 = (b * a)^2
    a * c * a = c * a * c
    (b * c)^2 = (c * b)^2
    a^2 = Id($)
    b^2 = Id($)
    c^2 = Id($)
> M := SymmetricMatrix([ 1, 3,1, 2,3,1 ]);
> G<a,b,c> := CoxeterGroup(M);
> G;
Coxeter group: Permutation group G acting on a set of cardinality 12
Order = 24 = 2^3 * 3
    (1, 7)(2, 4)(5, 6)(8, 10)(11, 12)
    (1, 4)(2, 8)(3, 5)(7, 10)(9, 11)
    (2, 5)(3, 9)(4, 6)(8, 11)(10, 12)
> G<a,b,c> := CoxeterGroup(GrpFPCox, M);
> G;
Coxeter group: Finitely presented group on 3 generators
Relations
    a * b * a = b * a * b
    a * c = c * a
    b * c * b = c * b * c
    a^2 = Id($)
    b^2 = Id($)
    c^2 = Id($)
```

Note that a Coxeter group does not have a unique Cartan matrix.

```
> C := CartanMatrix("G2");
> W := CoxeterGroup(GrpFPCox, C);
>  CartanMatrix(W);
>> CartanMatrix(W);
               ^
Runtime error in 'CartanMatrix': Bad argument types
Argument types given: GrpFPCox
```

`CoxeterGroup(GrpFPCox, R)`

`CoxeterGroup(GrpPermCox, R)`

`CoxeterGroup(R)`

The finite Coxeter group with root system or root datum R (see Chapters 96 and 97).

`CoxeterGroup(A, B)`

The permutation Coxeter group with roots given by the rows of the matrix A and coroots given by the rows of the matrix B. The matrices A and B must have the following properties:

1. A and B must have same number of rows and the same number of columns; they must be defined over the same field, which must be the rational field, a number field, or a cyclotomic field; the entries must be real;
2. the number of columns must be at least the number of rows; and
3. AB^t must be the Cartan matrix of a finite Coxeter group.

Example H98E3

```
> R := RootDatum("A3" : Isogeny := 2);
> CoxeterGroup(R);
Coxeter group: Permutation group acting on a set of cardinality 12
Order = 24 = 2^3 * 3
    (1, 7)(2, 4)(5, 6)(8, 10)(11, 12)
    (1, 4)(2, 8)(3, 5)(7, 10)(9, 11)
    (2, 5)(3, 9)(4, 6)(8, 11)(10, 12)
```

98.3 Converting Between Types of Coxeter Group

In this section, we describe functions for converting between the various descriptions of Coxeter groups available in MAGMA.

Since a finitely presented Coxeter group W does not come with an in-built reflection representation, the optional parameters A, B, and C can be used to specify the representation. They are respectively the matrix whose rows are the simple roots, the matrix whose rows are the simple coroots, and the Cartan matrix. These must have the following properties:

1. A and B must have same number of rows and the same number of columns; they must be defined over the same field, which must be the rational field, a number field, or a cyclotomic field; the entries must be real;
2. the number of columns must be at least the number of rows; and
3. $C = AB^t$ must be a Cartan matrix for W.

It is not necessary to specify all three matrices, since any two of them will determine the third. If these matrices are not given, the default is to take A to be the identity and to take C to be the standard Cartan matrix described in Section 95.4.

`CoxeterGroup(GrpFPCox, W)`

The finitely presented Coxeter group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$.

`CoxeterGroup(GrpFPCox, W)`

The finitely presented Coxeter group W' isomorphic to the real reflection group W (see Chapter 99).

`CoxeterGroup(GrpPermCox, W)`

A	MTRX	*Default :*
B	MTRX	*Default :*
C	MTRX	*Default :*

The permutation Coxeter group W' isomorphic to the finitely presented Coxeter group W, together with the isomorphism $W \to W'$. If W is infinite, an error is flagged.

`CoxeterGroup(GrpPermCox, W)`

The permutation Coxeter group W' isomorphic to the real reflection group W, together with the isomorphism $W \to W'$ (see Chapter 99). If W is infinite, an error is flagged.

Example H98E4

```
> W<a,b> := CoxeterGroup(GrpFPCox, "G2");
> Wp, h := CoxeterGroup(GrpPermCox, W);
> a*b;
a * b
> h(a*b);
(1, 11, 12, 7, 5, 6)(2, 4, 3, 8, 10, 9)
```

`ReflectionGroup(W)`
`CoxeterGroup(GrpMat, W)`

A	MTRX	*Default :*
B	MTRX	*Default :*
C	MTRX	*Default :*

A reflection group W' of the Coxeter group W, together with the isomorphism $W \to W'$.

`ReflectionGroup(W)`
`CoxeterGroup(GrpMat, W)`

The reflection group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$. There are no optional parameters A, B, and C in this case because every permutation Coxeter group has a root system, and this determines the reflection representation.

Example H98E5

```
> W<a,b,c> := CoxeterGroup(GrpFPCox, "B3");
> G, h := CoxeterGroup(GrpMat, W);
> a*b; h(a*b);
a * b
[-1 -1  0]
[ 1  0  0]
[ 0  1  1]
```

`CoxeterGroup(GrpFP, W)`

The finitely presented group W' isomorphic to the finitely presented Coxeter group W, together with the isomorphism $W \to W'$.

`CoxeterGroup(GrpFP, W)`

The finitely presented group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$.

`CoxeterGroup(GrpFP, W)`

The finitely presented group W' isomorphic to the real reflection group W, together with the isomorphism $W \to W'$ (see Chapter 99).

`CoxeterGroup(GrpPerm, W)`

The permutation group W' isomorphic to the finitely presented Coxeter group W, together with the isomorphism $W \to W'$. If W is infinite, an error is flagged.

`CoxeterGroup(GrpPerm, W)`

The permutation group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$.

`CoxeterGroup(GrpPerm, W)`

The permutation group W' isomorphic to the real reflection group W, together with the isomorphism $W \to W'$ (see Chapter 99). If W is infinite, an error is flagged.

98.4 Operations on Coxeter Groups

See Chapter 70 for general functions for finitely presented groups, or Chapter 58 for general functions for permutation groups.

`IsIsomorphic(W1, W2)`

Returns `true` if, and only if, W_1 and W_2 are isomorphic as abstract groups. This is only implemented for permutation Coxeter groups.

`IsCoxeterIsomorphic(W1, W2)`

Returns `true` if, and only if, W_1 and W_2 are isomorphic as Coxeter groups. If `true`, a sequence giving the permutation of the generators which takes W_1 to W_2 is also returned.

`IsCartanEquivalent(W1, W2)`

Returns `true` if, and only if, the crystallographic Coxeter groups W_1 and W_2 have Cartan equivalent Cartan matrices. This only makes sense for permutation Coxeter groups.

Example H98E6

```
> W1 := CoxeterGroup(GrpFPCox, "B4");
> W2 := CoxeterGroup(GrpFPCox, "C4");
> IsCoxeterIsomorphic(W1, W2);
true [ 1, 2, 3, 4 ]
```

An example of abstractly isomorphic Coxeter groups which are not Coxeter isomorphic:

```
> W1 := CoxeterGroup("G2");
> W2 := CoxeterGroup("A1A2");
> IsIsomorphic(W1, W2);
true
> IsCoxeterIsomorphic(W1, W2);
false
```

An example of Coxeter isomorphic groups which are not Cartan equivalent:

```
> W1 := CoxeterGroup("B3");
> W2 := CoxeterGroup("C3");
> IsIsomorphic(W1, W2);
true
> IsCoxeterIsomorphic(W1, W2);
true [ 1, 2, 3 ]
> IsCartanEquivalent(W1, W2);
false
```

`RootSystem(W)`

The underlying root system of the permutation Coxeter group W.

`RootDatum(W)`

The root datum of the permutation Coxeter group W. If W does not have a root datum, an error is flagged.

Example H98E7

```
> W := CoxeterGroup("C5");
> RootSystem(W);
Root system of type C5
> RootDatum(W);
Root datum of type C5
>
> W := CoxeterGroup("H4");
> RootSystem(W);
Root system of type H4
>  RootDatum(W);
Error: This group does not have a root datum
```

`CartanName(W)`

The Cartan name of the finite or affine Coxeter group W (Section 95.6).

`CoxeterDiagram(W)`

Print the Coxeter diagram of the finite or affine Coxeter group W (Section 95.6).

`DynkinDiagram(W)`

Print the Dynkin diagram of the permutation Coxeter group W. If W is not crystallographic, an error is flagged.

Example H98E8

```
> W := CoxeterGroup("F4");
> CartanName(W);
F4
> DynkinDiagram(W);
F4    1 - 2 =>= 3 - 4
> CoxeterDiagram(W);
F4    1 - 2 === 3 - 4
```

`CoxeterMatrix(W)`

The Coxeter matrix of the Coxeter group W.

`CoxeterGraph(W)`

The Coxeter graph of the Coxeter group W.

`CartanMatrix(W)`

The Cartan matrix of the permutation Coxeter group W.

`DynkinDigraph(W)`

The Dynkin digraph of the permutation Coxeter group W.

`Rank(W)`
`NumberOfGenerators(W)`

The rank of the Coxeter group W.

`NumberOfPositiveRoots(W)`
`NumPosRoots(W)`

The number of positive roots of the Coxeter group W.

`Dimension(W)`

The dimension of the permutation Coxeter group W, ie. the dimension of the root space.

Example H98E9

```
> R := StandardRootSystem("A", 4);
> W := CoxeterGroup(R);
> Rank(W);
4
> Dimension(W);
5
```

`ConjugacyClasses(W)`

The conjugacy classes of the finite Coxeter group W. This uses the algorithm of [GP00].

`FundamentalGroup(W)`

The fundamental group of the permutation Coxeter group W. The roots and coroots of W must have integral components.

`IsogenyGroup(W)`

The isogeny group of the permutation Coxeter group W. The roots and coroots of W must have integral components.

`CoisogenyGroup(W)`

The coisogeny group of the permutation Coxeter group W. The roots and coroots of W must have integral components.

`BasicDegrees(W)`

The degrees of the basic invariant polynomials of the Coxeter group W. These are computed using the table in [Car72, page 155].

`BasicCodegrees(W)`

The basic codegrees of the Coxeter group W. These are computed using the algorithm in [LT09].

Example H98E10

The product of the basic degrees is the order of the Coxeter group; the sum of the basic degrees is the sum of the rank and the number of positive roots.

```
> W := CoxeterGroup("E6");
> degs := BasicDegrees(W);
> degs;
[ 2, 5, 6, 8, 9, 12 ]
> &*degs eq #W;
true
> &+degs eq NumPosRoots(W) + Rank(W);
true
```

`BruhatLessOrEqual(x, y)`

If Coxeter group element x is less than or equal to y in the Bruhat order [Deo77]. Suppose x is an element of the Coxeter group W. The Bruhat order is the partial order generated by the relations: $x \leq xw$ if $l(x) < l(xw)$, and $xw \leq x$ if $l(xw) < l(x)$, for $x \in W$ and w a reflection. If $l(xw) = l(x)+1$, then x is called a *Bruhat descendant* of xw. The algorithm used is a straightforward recursive procedure.

`BruhatDescendants(x)`

z GrpPermElt *Default :*

Let x be an element of the Coxeter group W, then the returned set S contains the Bruhat descendants of x. If $l(yw) = l(y) + 1$, then y is called a *Bruhat descendant* of yw. If the optional parameter z is set, only those descendants y with $z \leq y$ are returned. Algorithm: For each fundamental reflection in x it is tested whether leaving it out decreases the length of x by exactly 1. If so, it is included in the result. In particular, this algorithm does not use `BruhatLessOrEqual`.

`BruhatDescendants(X)`

z GrpPermElt *Default :*

Let X consist of elements of the Coxeter group W, then the returned set S contains the Bruhat descendants of every element of X.

If the optional parameter z is set, only those w are returned for which $z \leq w$ in the Bruhat ordering.

Example H98E11

Bruhat descendants:

```
> R := RootDatum("D4" : Isogeny := "SC");
> W := CoxeterGroup(GrpPermCox, R);
> Wfp,phi := CoxeterGroup(GrpFPCox, W);
> x := W.1*W.3*W.2*W.4*W.2*W.2*W.2*W.1;
> Eltseq(phi(x));
[ 1, 3, 2, 4, 2, 1 ]
> S := BruhatDescendants(x);
> { Eltseq(phi(w)) : w in S };
{
     [ 1, 3, 2, 4, 2 ],
     [ 3, 2, 4, 2, 1 ],
     [ 1, 2, 4, 2, 1 ],
     [ 1, 3, 2, 1, 4 ],
     [ 1, 3, 4, 2, 1 ]
}
```

98.5 Properties of Coxeter Groups

`IsFinite(W)`

Returns `true` if, and only if, the Coxeter group W is finite.

`IsAffine(W)`

Returns `true` if, and only if, the Coxeter group W is affine (Section 95.6).

`IsHyperbolic(W)`

Returns `true` if, and only if, the Coxeter group W is hyperbolic (Section 95.7).

`IsCompactHyperbolic(W)`

Returns `true` if, and only if, the Coxeter group W is compact hyperbolic (Section 95.7).

`IsIrreducible(W)`

Returns `true` if, and only if, the Coxeter group W is irreducible.

`IsSemisimple(W)`

Returns `true` if, and only if, the permutation Coxeter group W is semisimple, i.e. its rank is equal to its dimension.

`IsCrystallographic(W)`

Returns `true` if, and only if, the permutation Coxeter group W is crystallographic, i.e. if the corresponding reflection representation is defined over the integers.

`IsSimplyLaced(W)`

Returns `true` if, and only if, the Coxeter group W is simply laced, i.e. its Coxeter graph has no labels.

Example H98E12

```
> W := CoxeterGroup(GrpFPCox, HyperbolicCoxeterMatrix(22));
> IsFinite(W);
false
> IsAffine(W);
false
> IsHyperbolic(W);
true
> IsCompactHyperbolic(W);
true
> IsIrreducible(W);
true
> IsSimplyLaced(W);
true
```

```
> W := CoxeterGroup("A2 D4");
> IsIrreducible(W);
false
> IsSemisimple(W);
true
> IsCrystallographic(W);
true
> IsSimplyLaced(W);
true
```

98.6 Operations on Elements

See Chapter 70 for general functions for finitely presented groups or Chapter 58 for general functions for permutation groups.

Unlike groups of type GrpFP, elements of a group of type GrpFPCox are always converted into the normal form of Section 98.1.1.

Example H98E13

Arithmetic with words.

```
> W<[s]> := CoxeterGroup(GrpFPCox, "G2");
> w1 := W![2,1,2,1,2] ;
> w1;
s[2] * s[1] * s[2] * s[1] * s[2]
> w2 := W![1,2,2,1,2,1];
> w2;
s[2] * s[1]
> w1 * w2;
s[1] * s[2] * s[1]
> W![1,2,1,2,1,2] eq W![2,1,2,1,2,1];
true
```

`#w`

`Length(w)`

`CoxeterLength(W, w)`

The length of w as an element of the Coxeter group W, ie. the number of positive roots of W which become negative under the action of w. The # operator does not work for permutation Coxeter group elements.

`LongestElement(W)`

The unique longest element of the Coxeter group W.

`CoxeterElement(W)`

The Coxeter element of the Coxeter group W, ie. the product of the generators of W.

`CoxeterNumber(W)`

The Coxeter number of the irreducible Coxeter group W (see [Car93, page 20]).

Example H98E14

```
> W<[s]> := CoxeterGroup(GrpFPCox, "F4");
> LongestElement(W);
s[1] * s[2] * s[1] * s[3] * s[2] * s[1] * s[3] * s[2] * s[3] * s[4] * s[3] *
s[2] * s[1] * s[3] * s[2] * s[3] * s[4] * s[3] * s[2] * s[1] * s[3] * s[2] *
s[3] * s[4]
> CoxeterElement(W);
s[1] * s[2] * s[3] * s[4]
> W := CoxeterGroup("E8");
> Length(W, LongestElement(W));
120
> Length(W, CoxeterElement(W));
8
```

The Coxeter number can be described in a variety of ways.

```
> W := CoxeterGroup("D5");
> CoxeterNumber(W) eq Order(CoxeterElement(W));
true
> CoxeterNumber(W) eq #Roots(W) / Rank(W);
true
> R := RootDatum(W);
> CoxeterNumber(W) eq &+Eltseq(HighestRoot(R)) + 1;
true
```

`LeftDescentSet(W, w)`

The set of indices r of simple roots of the Coxeter group W such that the length of the product $s_r w$ is less than that of the element w.

`RightDescentSet(W, w)`

The set of indices r of simple roots of the Coxeter group W such that the length of the product $w s_r$ is less than that of the element w.

Example H98E15

```
> W := CoxeterGroup("A5");
> x := W.1*W.2*W.4*W.5;
> LeftDescentSet(W, x);
{ 1, 4 }
> RightDescentSet(W, x);
{ 2, 5 }
```

98.7 Roots, Coroots and Reflections

The functions in this section give access to the underlying root system (or datum) of a permutation Coxeter group. These functions do not apply to finitely presented Coxeter groups

Roots are stored as an indexed set

$$\{@ \alpha_1, \ldots, \alpha_N, \alpha_{N+1}, \ldots, \alpha_{2N} @\},$$

where $\alpha_1, \ldots, \alpha_N$ are the positive roots in an order compatible with height; and $\alpha_{N+1}, \ldots, \alpha_{2N}$ are the corresponding negative roots (i.e. $\alpha_{i+N} = -\alpha_i$). The simple roots are $\alpha_1, \ldots, \alpha_n$ where n is the rank.

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space. This is the default.
2. `"Root"`: the basis of simple (co)roots.
3. `"Weight"`: the basis of fundamental (co)weights (see Subsection 99.8.3 below).

98.7.1 Accessing Roots and Coroots

`RootSpace(W)`

`CorootSpace(W)`

The (co)root space of the Coxeter group W. This can be a vector space over a field of characteristic zero (Chapter 28), or an integer lattice in the crystallographic case (Chapter 30). The (co)reflection group of W acts on the (co)root space.

`SimpleRoots(W)`

`SimpleCoroots(W)`

The simple (co)roots of the Coxeter group W as the rows of a matrix.

Example H98E16

```
> W := CoxeterGroup("G2");
> RootSpace(W);
Full Vector space of degree 2 over Rational Field
> CorootSpace(W);
Full Vector space of degree 2 over Rational Field
> SimpleRoots(W);
[1 0]
[0 1]
> SimpleCoroots(W);
[ 2 -3]
[-1  2]
> CartanMatrix(W);
[ 2 -1]
[-3  2]
```

`NumberOfPositiveRoots(W)`

`NumPosRoots(W)`

The number of positive roots of the Coxeter group W.

`Roots(W)`

`Coroots(W)`

`Basis` MonStgElt *Default :* "*Standard*"

An indexed set containing the (co)roots of the Coxeter group W.

`PositiveRoots(W)`

`PositiveCoroots(W)`

`Basis` MonStgElt *Default :* "*Standard*"

An indexed set containing the positive (co)roots of the Coxeter group W.

`Root(W, r)`

`Coroot(W, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The rth (co)root of the Coxeter group W.

`RootPosition(W, v)`

`CorootPosition(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

If v is a (co)root of the Coxeter group W, this returns its position; otherwise it returns 0. These functions will try to coerce v, which can be a vector or a sequence representing a vector, into the appropriate vector space; v should be written with respect to the basis specified by the parameter `Basis`.

Example H98E17

```
> A := Matrix(2,3, [1,-1,0, -1,1,-1]);
> B := Matrix(2,3, [1,-1,1, 0,1,-1]);
> W := CoxeterGroup(A, B);
> Roots(W);
{@
    (1 -1  0),
    (-1  1 -1),
    (0  0 -1),
    (1 -1 -1),
    (2 -2 -1),
    (1 -1 -2),
    (-1  1  0),
    (1 -1  1),
    (0 0 1),
    (-1  1  1),
    (-2  2  1),
    (-1  1  2)
@}
> PositiveCoroots(W);
{@
    (1 -1  1),
    (0  1 -1),
    (1  2 -2),
    (2  1 -1),
    (1 0 0),
    (1  1 -1)
@}
> #Roots(W) eq 2*NumPosRoots(W);
true
> Root(W, 4);
(1 -1 -1)
> Root(W, 4 : Basis := "Root");
(2 1)
> RootPosition(W, [1,-1,-1]);
4
> RootPosition(W, [2,1] : Basis := "Root");
4
```

`HighestRoot(W)`

`HighestLongRoot(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique (long) root of greatest height of the irreducible Coxeter group W.

`HighestShortRoot(W)`

`Basis` MonStgElt *Default : "Standard"*

The unique short root of greatest height of the irreducible Coxeter group W.

Example H98E18

```
> W := RootDatum("G2");
> HighestRoot(W);
(3 2)
> HighestLongRoot(W);
(3 2)
> HighestShortRoot(W);
(2 1)
```

`CoxeterForm(W)`

`DualCoxeterForm(W)`

`Basis` MonStgElt *Default : "Standard"*

The matrix of an inner product on the (co)root space of the finite Coxeter group W which is invariant under the action of W. This inner product is uniquely determined up to a constant on each irreducible component of W. The inner product is normalised so that the short roots in each crystallographic component have length one.

`AdditiveOrder(W)`

An additive order on the positive roots of the finite Coxeter group W, i.e. a sequence containing the numbers $1, \ldots, N$ in some order such that $\alpha_r + \alpha_s = \alpha_t$ implies t is between r and s. This is computed using the techniques of [Pap94].

98.7.2 Operations and Properties for Root and Coroot Indices

`Sum(W, r, s)`

The index of the sum of the rth and sth roots in the Coxeter group W, or 0 if the sum is not a root. In other words, if $t =$ `Sum(W,`r`,`s`)` $\neq 0$ then $\alpha_t = \alpha_r + \alpha_s$. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

`IsPositive(W, r)`

Returns `true` if, and only if, the rth (co)root of the Coxeter group W is a positive root.

`IsNegative(W, r)`

Returns `true` if, and only if, the rth (co)root of the Coxeter group W is a negative root.

`Negative(W, r)`

The index of the negative of the rth (co)root of the Coxeter group W. In other words, if $s =$ `Negative(W,`r`)` then $\alpha_s = -\alpha_r$.

`LeftString(W, r, s)`

Root indices in the Coxeter group W of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s - \alpha_r, \alpha_s - 2\alpha_r, \ldots, \alpha_s - p\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_p]$ where $\alpha_{r_i} = \alpha_s - i\alpha_r$ and $\alpha_s - (p+1)\alpha_r$ is not a root. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

`RightString(W, r, s)`

Root indices of the Coxeter group W of the left string through α_s in the direction of α_r, i.e. the indices of $\alpha_s + \alpha_r, \alpha_s + 2\alpha_r, \ldots, \alpha_s + q\alpha_r$. In other words, this returns the sequence $[r_1, \ldots, r_q]$ where $\alpha_{r_i} = \alpha_s + i\alpha_r$ and $\alpha_s + (q+1)\alpha_r$ is not a root. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

`LeftStringLength(W, r, s)`

The largest p such that $\alpha_s - p\alpha_r$ is a root of the Coxeter group W. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

`RightStringLength(W, r, s)`

The largest q such that $\alpha_s + q\alpha_r$ is a root of the Coxeter group W. The condition $\alpha_r \neq \pm\alpha_s$ must be satisfied. If W is noncrystallographic, an error is flagged.

Example H98E19

```
> W := RootDatum("G2");
> Sum(W, 1, Negative(W,5));
10
> IsPositive(W, 10);
false
> Negative(W, 10);
4
> P := PositiveRoots(W);
> P[1] - P[5] eq -P[4];
true
```

`RootHeight(W, r)`

`CorootHeight(W, r)`

The height of the rth (co)root of the Coxeter group W, i.e. the sum of the coefficients of α_r (respectively, $\alpha_r^\star$) with respect to the simple (co)roots.

`RootNorms(W)`

`CorootNorms(W)`

The sequence of squares of the lengths of the (co)roots of the Coxeter group W.

`RootNorm(W, r)`

`CorootNorm(W, r)`

The square of the length of the rth (co)root of the Coxeter group W.

`IsLongRoot(W, r)`

Returns `true` if, and only if, the rth root of the Coxeter group W is long, i.e. the rth coroot is short. An error is flagged unless W is irreducible and crystallographic.

`IsShortRoot(W, r)`

Returns `true` if, and only if, the rth root of the Coxeter group W is short, i.e. the rth coroot is long. An error is flagged unless W is irreducible and crystallographic.

Example H98E20

```
> W := RootDatum("G2");
> RootHeight(W, 5);
4
> F := CoxeterForm(W);
> v := VectorSpace(Rationals(),2) ! Root(W, 5);
> (v*F, v) eq RootNorm(W, 5);
true
> IsLongRoot(W, 5);
true
> LeftString(W, 1, 5);
[ 4, 3, 2 ]
> roots := Roots(W);
> for i in [1..3] do
>   RootPosition(W, roots[5]-i*roots[1]);
> end for;
4
3
2
```

98.7.3 Weights

`WeightLattice(W)`

`CoweightLattice(W)`

The (co)weight lattice of the Coxeter group W. The roots and coroots of W must have integral components.

`FundamentalWeights(W)`

`FundamentalCoweights(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The fundamental (co)weights of the Coxeter group W. The roots and coroots of W must have integral components.

`IsDominant(R, v)`

`Basis` MonStgElt *Default :* "*Standard*"

Returns `true` if, and only if, v is a dominant weight for the root datum R, ie, a nonnegative integral linear combination of the fundamental weights.

`DominantWeight(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique element in the W-orbit of the weight v which lies in the fundamental Weyl chamber, and the word in the generators which sends v to this element. The Coxeter group W must have a root datum. The weight v can be given either as a vector or as a sequence representing the vector and is coerced into the weight lattice first.

`WeightOrbit(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The orbit of the weight v under the action of W. The Coxeter group W must have a root datum. The weight v can be given either as a vector or as a sequence representing the vector and is coerced into the weight lattice first.

Example H98E21

```
> W := CoxeterGroup("B3");
> DominantWeight(W, [1,-1,0] : Basis:="Weight");
(1 0 0)
[ 2, 3, 2, 1 ]
> #WeightOrbit(W, [1,-1,0] : Basis:="Weight");
6
```

98.8 Reflections

An element of a Coxeter group is called a *reflection* if it is conjugate to one of the Coxeter generators.

In a permutation Coxeter group, the root α acts on the root space via the reflection s_α; the coroot $\alpha^\star$ acts on the coroot space via the coreflection $s_\alpha^\star$.

`IsReflection(w)`

Returns `true` if, and only if, w is a reflection, i.e. w is conjugate to a Coxeter generator. If w is in a permutation Coxeter group, the root, coroot and root index are also returned.

`Reflections(W)`

The sequence of reflections in the finite Coxeter group W. If W is a permutation Coxeter group, the rth reflection in the sequence corresponds to the rth (co)root.

Example H98E22

```
> W<a,b> := CoxeterGroup(GrpFPCox, "A2");
> Reflections(W);
[ a, b, a * b * a, a, b, a * b * a ]
> IsReflection(a*b);
false
```

`SimpleReflections(W)`

The sequence of simple reflections in the Coxeter group W, ie, the generators of W.

`SimpleReflectionPermutations(W)`

The sequence of simple reflections in the permutation Coxeter group W, ie, the generators of W.

`Reflection(W, r)`

`ReflectionPermutation(W, r)`

The reflection in permutation Coxeter group W corresponding to the rth (co)root. If $r = 1, \ldots, n$, this is a generator of W.

`SimpleReflectionMatrices(W)`

`SimpleCoreflectionMatrices(W)`

`Basis` MonStgElt *Default : "Standard"*

The matrices giving the action of the simple (co)roots on the (co)root space of the permutation Coxeter group W.

`ReflectionMatrices(W)`

`CoreflectionMatrices(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrices giving the action of the (co)roots on the (co)root space of the permutation Coxeter group W.

`ReflectionMatrix(W, r)`

`CoreflectionMatrix(W, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrix giving the action of the rth (co)root on the (co)root space of the permutation Coxeter group W.

`ReflectionWords(W)`

The sequence of words in the simple reflections for all the reflections of the Coxeter group W. These words are given as sequences of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWords(W)[r]`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

`ReflectionWord(W, r)`

The word in the simple reflections for the rth reflection of the Coxeter group W. The word is given as a sequence of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWord(W,r)`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

Example H98E23

```
> W := CoxeterGroup("B3");
> IsReflection(W.1*W.2);
false
> mx := ReflectionMatrix(W, 4);
> perm := Reflection(W, 4);
> wd := ReflectionWord(W, 4);
> rt := VectorSpace(Rationals(), 3) ! Root(W,2);
> RootPosition(W, rt * mx) eq 2^perm;
true
> perm eq &*[ Reflection(W, r) : r in wd ];
true
>
> mx := CoreflectionMatrix(W, 4);
> CorootPosition(W, Coroot(W,2) * mx) eq 2^perm;
true
```

98.9 Reflection Subgroups

A *reflection subgroup* of a Coxeter group is a subgroup which is generated by a set of reflections. Note that reflection subgroups are also Coxeter groups. The most important class of reflection subgroups are the *standard parabolic subgroups*, which are generated by a subset of the simple roots. Given a set of indices $J \subseteq \{1, \ldots, \texttt{Rank(W)}\}$, the corresponding standard parabolic is denoted W_J. A *parabolic subgroup* is a subgroup which is conjugate to a standard parabolic subgroup. Note that in a reflection subgroup, the elements are given as permutations of the roots of the *larger* group.

Most of the functions in this section are currently only implemented for permutation Coxeter groups with a root *datum* (rather than a root system).

`ReflectionSubgroup(W, a)`

The reflection subgroup of the permutation Coxeter group W generated by the roots $\alpha_{a_1}, \ldots, \alpha_{a_k}$ where $a = \{a_1, \ldots, a_k\}$ is a set of integers. This only works if W has an underlying root datum.

`ReflectionSubgroup(W, s)`

The reflection subgroup of the permutation Coxeter group W generated by simple roots $\alpha_{s_1}, \ldots, \alpha_{s_k}$ where $s = [s_1, \ldots, s_k]$ is a *sequence* of integers. In this version the roots must be simple in the root subdatum (ie. none of them may be a summand of another) otherwise an error is signalled. The simple roots will appear in the reflection subgroup in the given order. This only works if W has an underlying root datum.

`StandardParabolicSubgroup(W, J)`

The standard parabolic subgroup of the Coxeter group W generated by the simple roots $\alpha_{j_1}, \ldots, \alpha_{j_k}$ where $J = \{j_1, \ldots, j_k\} \subseteq \{1, \ldots, \texttt{Rank(W)}\}$. This function works for both finitely presented and permutation Coxeter groups.

`IsReflectionSubgroup(W, H)`

Returns `true` if, and only if, H is a reflection subgroup of the permutation Coxeter group W.

`IsParabolicSubgroup(W, H)`

Returns `true` if, and only if, H is a parabolic subgroup of the permutation Coxeter group W.

`IsStandardParabolicSubgroup(W, H)`

Returns `true` if, and only if, H is a standard parabolic subgroup of the permutation Coxeter group W.

`Overgroup(H)`

The overgroup of H, ie. the Coxeter group whose roots are permuted by the elements of the permutation Coxeter subgroup H.

`Overdatum(H)`

The root datum whose roots are permuted by the elements of the permutation Coxeter subgroup H.

`LocalCoxeterGroup(H)`

Given a Coxeter subgroup H this returns the Coxeter group L isomorphic to H but acting on the roots of H itself rather than the roots of its overgroup, together with the isomorphism $L \to H$.

Example H98E24

```
> W := CoxeterGroup("A4");
> P := StandardParabolicSubgroup(W, {1,2});
> Overgroup(P) eq W;
true
> L, h := LocalCoxeterGroup(P);
> hinv := Inverse(h);
> L.1;
(1, 4)(2, 3)(5, 6)
> h(L.1);
(1, 11)(2, 5)(6, 8)(9, 10)(12, 15)(16, 18)(19, 20)
> hinv(h(L.1));
(1, 4)(2, 3)(5, 6)
```

`Transversal(W, H)`

The indexed set of (right) coset representatives of the reflection subgroup H of the Coxeter group W. This contains the unique element of shortest length in each coset. The algorithm is due to Don Taylor (personal communication).

`TransversalWords(W, H)`

The indexed set of words of (right) coset representatives of the reflection subgroup H of the Coxeter group W. The algorithm is due to Don Taylor (personal communication).

`TransversalElt(W, H, x)`

The representative of the coset Hx in the Coxeter group W. This is the unique element of Hx of shortest length in W and also the unique element of Hx which sends every positive root of H to another positive root. The algorithm is due to Don Taylor (personal communication).

Example H98E25

```
> W := CoxeterGroup("A4");
> P := StandardParabolicSubgroup(W, {1,2});
> x := W.1 * W.2 * W.3;
> x := TransversalElt(W, P, x);
> x eq W.3;
true
> x in Transversal(W, P);
true
```

`TransversalElt(W, x, H)`

The representative of the coset xH in the Coxeter group W. This is the unique element of xH of shortest length in W and also the unique element of xH which sends every positive root of H to another positive root.

`TransversalElt(W, H, x, J)`

The representative of the coset HxJ in the Coxeter group W. This is the unique element of HxJ of shortest length in W and also the unique element of HxJ which sends every positive root of HJ to another positive root.

`Transversal(W, J)`

`Transversal(W, J, L)`

The set of right coset representatives of minimal length for the standard parabolic subgroup $W_J \leq W$. In the first form W must be finite and the result is a full transversal. In the second form W may be infinite, but the transversal produced is limited to words of length at most L.

`Transversal(W, J, K)`

The sequence of W_J, W_K-double cosets representatives of minimal length in W. Restricted to W finite. The second return value gives the generators of the standard parabolic subgroup $W_J \cap W_K^d$ for each double coset representative d.

`DirectProduct(W1, W2)`

The direct product of the Coxeter groups W_1 and W_2.

`Dual(W)`

The dual of the Coxeter group W, obtained by swapping the roots and coroots.

Example H98E26

```
> W1 := CoxeterGroup("G2");
> W2 := CoxeterGroup("C3");
> DirectProduct(W1, Dual(W2));
Coxeter group: Permutation group acting on a set of cardinality 30
Order = 576 = 2^6 * 3^2
    (1, 7)(2, 5)(3, 4)(8, 11)(9, 10)
    (1, 3)(2, 8)(5, 6)(7, 9)(11, 12)
    (13, 22)(14, 16)(17, 20)(19, 21)(23, 25)(26, 29)(28, 30)
    (13, 16)(14, 23)(15, 17)(18, 21)(22, 25)(24, 26)(27, 30)
    (14, 19)(15, 24)(16, 21)(23, 28)(25, 30)
> W1 := CoxeterGroup(GrpFPCox, "G2");
> W2 := CoxeterGroup(GrpFPCox, "A2");
> DirectProduct(W1, W2);
Coxeter group: Finitely presented group on 4 generators
Relations
    $.1 * $.2 * $.1 = $.2 * $.1 * $.2
    $.1 * $.3 = $.3 * $.1
    $.1 * $.4 = $.4 * $.1
    $.2 * $.3 = $.3 * $.2
    $.2 * $.4 = $.4 * $.2
    $.3 * $.4 * $.3 = $.4 * $.3 * $.4
    $.1^2 = Id($)
    $.2^2 = Id($)
    $.3^2 = Id($)
    $.4^2 = Id($)
```

98.10 Root Actions

The functions in this section give access to the action on the underlying root system (or datum) of a permutation Coxeter group. These functions do not apply to finitely presented Coxeter groups

In the following functions, the optional parameter `Basis` determines which basis the roots are given with respect to: `"Standard"` for the standard basis of the root space; `"Root"` for the basis of simple (co)roots; `"Weight"` for the basis of simple (co)weights.

`RootGSet(W)`

`CorootGSet(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The G-set of the Coxeter group W acting on the (co)roots.

Example H98E27

```
> W := CoxeterGroup("B3");
> X := RootGSet(W);
> r := Root(W, 5);
> r;
(0 1 1)
> Image(W.1, X, r);
(1 1 1)
```

`RootAction(W)`

`CorootAction(W)`

Basis MonStgElt *Default :* "*Standard*"

The map $X \times W \to X$ giving the action of the Coxeter group W on the (co)root space X.

Example H98E28

```
> W := CoxeterGroup("B3");
> act := CorootAction(W);
> act([1,-2,1], W.1);
(-1 -1  1)
```

`ReflectionGroup(W)`

`CoreflectionGroup(W)`

Basis MonStgElt *Default :* "*Standard*"

The Coxeter group W as a real reflection group (ie. as a matrix group over some subfield of $\mathbf{R}$) acting on the (co)root space, and the isomorphism from W to the (co)reflection group.

Example H98E29

```
> W := CoxeterGroup("B3");
> _, h := ReflectionGroup(W);
> W.1*W.3;
(1, 10)(2, 8)(3, 12)(4, 7)(5, 6)(11, 17)(13, 16)(14, 15)
> h(W.1*W.3);
[-1  0  0]
[ 1  1  2]
[ 0  0 -1]
```

98.11 Standard Action

Every finite Coxeter group W has a standard action. For example, the standard action group of a Coxeter group of type A_n is the symmetric group of degree $n+1$ acting on $\{1, \ldots, n\}$.

`StandardAction(W)`

The standard action of the finite Coxeter group W.

`StandardActionGroup(W)`

The group G of the standard action of the finite Coxeter group W, together with an isomorphism $W \to G$.

Example H98E30

```
> W := CoxeterGroup("A3");
> G, h := StandardActionGroup(W);
> IsSymmetric(G);
true
> h(W.1); h(W.2); h(W.3);
(1, 2)
(2, 3)
(3, 4)
```

98.12 Braid Groups

`BraidGroup(W)`

The braid group B of the Coxeter group W as an finitely presented group, together with the natural map $W \to B$. Words in the braid group are not automatically normalised. However, the braid group of type A_n with normalisation can be constructed with the command `BraidGroup(n+1)` (see Chapter 73).

`PureBraidGroup(W)`

Returns the pure braid group of the Coxeter group W, ie. the kernel of the epimorphism from the braid group of W to W. Words in the pure braid group are not automatically normalised.

Example H98E31

```
> W<a,b,c> := CoxeterGroup(GrpFPCox, "B3");
> W;
Coxeter group: Finitely presented group on 3 generators
Relations
    a * b * a = b * a * b
    a * c = c * a
    (b * c)^2 = (c * b)^2
    a^2 = Id($)
    b^2 = Id($)
    c^2 = Id($)
> B<x,y,z> := BraidGroup(W);
> B;
Finitely presented group B on 3 generators
Relations
    x * y * x = y * x * y
    x * z = z * x
    (y * z)^2 = (z * y)^2
> P := PureBraidGroup(W);
> P;
Finitely presented group P on 3 generators
Generators as words in group B
    P.1 = x^2
    P.2 = y^2
    P.3 = z^2
```

98.13 W-graphs

Given a Coxeter system (W, S), a W-graph is a (directed or undirected) graph with vertex labels and edge weights. The label attached to a vertex v is a subset of S (called the *descent set* of v) and the edge weights are scalars (usually integers).

A W-graph must determine a representation of the Hecke algebra $H = H\langle q\rangle$ of the associated Coxeter system. The vertices of the W-graph can be identified with basis elements of the representation space, and by the conventions adopted here the action of the generator T_s of H associated with an element $s \in S$ on a basis element v is given by

$$v * T_s = \begin{cases} (-q^{-1}) * v & \text{if } s \text{ is in the descent set of } v, \\ q * v + \sum'(m * u) & \text{if } s \text{ is not in the descent set of } v, \end{cases}$$

where $\sum'$ indicates the sum over all edges with terminal vertex equal to v for which s is in the descent set of the initial vertex u, and m is the weight of the edge.

For the Coxeter group calculations involved in these functions we need to know how the generators $s \in S$ act on the set of elementary roots (see [Bri98]).

MAGMA has a function `ReflectionTable` that provides the necessary information. Specifically, let W be a finitely presented Coxeter group with N elementary roots (numbered from 1 to N) and r simple reflections (numbered 1 to r). If we define

```
eltroots:=ReflectionTable(W);
```

then for $i \in \{1, \ldots, r\}$ and $j \in \{1, \ldots, N\}$, `eltroots[i,j] = k` if the i-th simple reflection takes the j-th elementary root to the k-th elementary root, or to a non-elementary root if $k = 0$, or to a negative root if $k < 0$. (This last alternative occurs if and only if $j = i$ and $k = -i$.) Knowing the table `eltroots` makes it quick and easy to do symbolic computation with elements of W, represented as sequences of integers in $\{1, \ldots, r\}$ (corresponding to words in S).

`SetVerbose("WGraph", v)`

Set the verbose printing to level v for all W-graph related functions. A level of 2 means that informative messages and progress information will be printed durng a computation.

Sometimes it is convenient to use 'mij-sequences' to specify Coxeter groups. The mij-sequence consists of the on or below diagonal entries in the Coxeter matrix. Thus if `seq` is the mij-sequence and M the Coxeter matrix then

```
M := SymmetricMatrix(seq);
```

and

```
seq := &cat[[M[i,j] :  j in [1..i]] :  i in [1..Rank(W)]];
```

`Mij2EltRootTable(seq)`

Return the elementary root action table for the Coxeter group defined by the given mij-sequence.

`Name2Mij(name)`

The mij-sequence of the Coxeter groups of type name.

Example H98E32

```
> e6:=[1,3,1,2,3,1,2,3,2,1,2,2,2,3,1,2,2,3,2,2,1];
> E6 := CoxeterGroup(GrpFPCox, SymmetricMatrix(e6) );
> ReflectionTable(E6) eq Mij2EltRootTable(e6);
true
```

The functions defined in this section are mainly concerned with W-graph posets. The motivating example for this concept is the set of all standard tableaux corresponding to a given partition, the partial order being dominance. By definition, if P is a W-graph poset then P must be in one-to-one correspondence with a basis for an H-module V (where H is the Hecke algebra associated with the given Coxeter system). In the standard tableaux example, this module is the Specht module; hence in the general case we refer to the

module V as GSM(P) (for generalized Specht module). For each $v \in P$ the set S must be the disjoint union of two sets $A(v)$ and $D(v)$, the ascents and descents of v. There must be a function $(s, v) \mapsto sv$ from $S \times P$ to P such that the action of H on GSM(P) satisfies the following rules (for all $s \in S$ and $v \in P$):

$$v * T_s = \begin{cases} sv & \text{if } sv > v, \\ sv + (q - q^{-1}) * v & \text{if } sv < v, \\ -q^{-1} * v & \text{if } sv = v \text{ and } s \in D(v), \\ q * v + q * \langle \text{earlier} \rangle & \text{if } sv = v \text{ and } s \in A(v), \end{cases}$$

where $\langle$earlier$\rangle$ denotes a linear combination of $\{u \in P \mid u < v\}$ with coefficients that are polynomials in q. For each $s \in A(v)$ either $sv = v$ or $sv > v$, and for each $s \in D(v)$ either $sv < v$ or $sv = v$. This (admittedly strange) definition is motivated by the fact that Specht modules satisfy it. If v is a standard tableau corresponding to a partition of n then a number i in $\{1, \ldots, n-1\}$ is an ascent of v if $i+1$ is in a later column of t than i, and is a descent of v if $i+1$ is in a lower row of t than i. The fact that Specht modules satisfy the formulas above is proved in the literature (e.g. Mathas' book), except that in the "weak ascent" case ($sv = v$ and $s \in A(v)$) it is not proved that the polynomial coefficients of $\{u \in P \mid u < v\}$ are all divisible by q. The fact that they are is a theorem of V. M. Nguyen (PhD thesis, University of Sydney, 2010). It turns out that there is an algorithm by which a W-graph may be constructed from a W-graph poset, the W-graph being uniquely determined by the function $(s, v) \mapsto sv$ from $S \times P \to P$ and the descent/ascent sets. The polynomial coefficients in the weak ascent case are not required. Of course the H-module determined by the resulting W-graph is isomorphic to GSM(P).

`Partition2WGtable(pi)`

Returns the W-graph table and the Weyl group for the partition `pi`, where `pi` is a nonincreasing sequence $[a_1, a_2, \ldots, a_k]$ of positive integers. It returns the table corresponding to the W-graph poset of standard tableaux of the given shape and the finitely presented Coxeter group of type A_n, where $n + 1 = \sum a_i$.

`WGtable2WG(table)`

Convert a W-graph table to a W-graph.

`TestWG(W,wg)`

This procedure can be used to test whether a presumed undirected or directed W-graph is indeed a W-graph. Two input values are required: the Coxeter group W and the W-graph. When applied to the W-graph produced by the `WGtable2WG` function, this tests whether the input table did genuinely correspond to a W-graph poset.

For example,

Example H98E33

```
> wtable, W :=Partition2WGtable([4,4,3,1]);
> wg := WGtable2WG(wtable);
> TestWG(W,wg);
```

which should cause the word `true` to be printed 66 times (as the defining relations of the Hecke algebra are checked).

Given a Coxeter system (W, S) and an element $w \in W$, let P be the set $\{x \in W \mid \text{length}(wx^{-1}) = \text{length}(w) - \text{length}(x)\}$, considered as a poset under the Bruhat order on W. Given also a subset J of $\{t \in S \mid \text{length}(wt) > \text{length}(w)\}$, for each $x \in P$ we define $D(x)$ to be union of $\{s \in S \mid \text{length}(sx) < \text{length}(x)\}$ and $\{s \in S \mid sx = xt \text{ for some } t \in J\}$. If P is now a W-graph poset with the sets $D(x)$ as the descent sets then we say that w is a W-graph determining element relative to J.

For example, suppose that (W, S) is of type A_n, and given a partition of $n+1$ let t be the (unique) standard tableau whose column group is generated by a subset of S. Let w be the maximal length element such that the tableau wt is standard. Then w is a W-graph determining element with respect to the set J consisting of those $s \in S$ that are in the column stabilizer of t.

Other examples (for any Coxeter system with finite W) are provided by the distinguished left coset representatives of maximal length for standard parabolic subgroups W_K (where the set J may be taken to be either K or the empty set).

`WGelement2WGtable(g,K)`

Returns the W-graph table and W-graph ideal of a W-graph determining element g, subset K.

Example H98E34

```
> b5 := [1,4,1,2,3,1,2,2,3,1,2,2,2,3,1];
> b5mat := SymmetricMatrix(b5);
> W := CoxeterGroup(GrpFPCox, b5mat );
> table, _ := WGelement2WGtable(W![5,4,3,2,1,2,3,4,5],{});
> wg := WGtable2WG(table);
> TestWG(W,wg);
true <1, 2> 4
true <2, 3> 3
true <3, 4> 3
true <4, 5> 3
```

`GetCells(wg)`

Return the cells of the W-graph.

`InduceWG(W,wg,seq)`

Induce a W-graph from a standard parabolic subgroup.

`InduceWGtable(J, table, W)`

Returns the table of the W-graph induced from the table of a parabolic subgroup defined by J.

`IsWGsymmetric(dwg)`

Test a W-graph for symmetry. If the graph is symmetric the second return value is the undirected version of the W-graph.

`MakeDirected(uwg)`

Convert an undirected W-graph to a directed W-graph.

`TestHeckeRep(W,r)`

Tests whether the matrices in r satisfy the defining relations of the Hecke algebra of the Coxeter group W.

`WG2GroupRep(wg)`

The matrix representation of a W-graph.

`WG2HeckeRep(W,wg)`

Returns a sequence of sparse matrices that satisfy the defining relations of the Hecke algebra.

`WGidealgens2WGtable(dgens,K)`

Returns the W-graph table and W-graph ideal of a W-graph determining generators dgens and subset K.

Example H98E35

In type E_6 we start with a rank 3 standard parabolic subgroup. The set of minimal coset representatives is a (single-generator) W-graph ideal, corresponding to the representation induced from the trivial representation of the parabolic. We compute the W-graph and find the cells. The bottom cell is necessarily an ideal in the weak order. It turns out that 3 elements are required to generate it; we can use them to test the function `WGidealgens2WGtable`.

```
> mij:=[1,3,1,2,3,1,2,3,2,1,2,2,2,3,1,2,2,3,2,2,1];
> E6 := CoxeterGroup(GrpFPCox, SymmetricMatrix(mij) );
> J := {1,3,5};
> drs := Transversal(E6,J);
> ttt := WGidealgens2WGtable([drs[1398],drs[156],drs[99]],J);
> nwg := WGtable2WG(ttt);
> TestWG(E6,nwg);
true <1, 2> 3
true <2, 3> 3
```

```
true <2, 4> 3
true <4, 5> 3
true <3, 6> 3
```

`WriteWG(file,uwg)`

`WriteWG(file,dwg)`

Writes the W-graph to a file.

98.14 Related Structures

In this section functions for creating other structures from a permutation Coxeter group are briefly listed. See the appropriate chapters of the Handbook for more details.

`CoxeterGroup(GrpFP, W)`

`Presentation(W)`

The finitely presented group isomorphic to the permutation Coxeter group W. See Chapter 70.

`ReflectionGroup(W)`

`CoxeterGroup(GrpMat, W)`

The reflection group isomorphic to the Coxeter group W. See Chapter 99.

`LieAlgebra(W, R)`

The reductive Lie algebra over the ring R with Weyl group W. If W is noncrystallographic, an error is flagged. See Section 100.5.1.

`GroupOfLieType(W, R)`

The group of Lie type over the ring R with Weyl group W. The roots and coroots of W must have integral components. See Chapter 103.

98.15 Bibliography

[**BH93**] Brigitte Brink and Robert B. Howlett. A finiteness property and an automatic structure for Coxeter groups. *Math. Ann.*, 296(1):179–190, 1993.

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Bri98**] Brigitte Brink. The set of dominance-minimal roots. *J. Algebra*, 206(2):371–412, 1998.

[**Car72**] Roger W. Carter. *Simple groups of Lie type.* John Wiley & Sons, London-New York-Sydney, 1972. Pure and Applied Mathematics, Vol. 28.

[**Car93**] Roger W. Carter. *Finite groups of Lie type.* John Wiley & Sons, Chichester, 1993. Conjugacy classes and complex characters, Reprint of the 1985 original, A Wiley-Interscience Publication.

[**Deo77**] V.V. Deodhar. Some Characteristics of Bruhat Ordering on a Coxeter group and determination of the Relative Möbius Function. *Inventiones Math.*, 39:179–198, 1977.

[**GHL+96**] Meinolf Geck, Gerhard Hiss, Frank Lübeck, Gunter Malle, and Götz Pfeiffer. CHEVIE—a system for computing and processing generic character tables. *Appl. Algebra Engrg. Comm. Comput.*, 7(3):175–210, 1996. Computational methods in Lie theory (Essen, 1994).

[**GP00**] Meinolf Geck and Götz Pfeiffer. *Characters of finite Coxeter groups and Iwahori-Hecke algebras*, volume 21 of *London Mathematical Society Monographs. New Series.* The Clarendon Press Oxford University Press, New York, 2000.

[**LT09**] G. I. Lehrer and D. E. Taylor. *Unitary Reflection Groups*, volume 20 of *Australian Mathematical Society Lecture Series.* Cambridge University Press, Cambridge, 2009.

[**Pap94**] Paolo Papi. A characterization of a special ordering in a root system. *Proc. Amer. Math. Soc.*, 120(3):661–665, 1994.

99 REFLECTION GROUPS

Chapter 99
REFLECTION GROUPS

99.1 Introduction

A reflection is a diagonalisable linear transformation of finite order whose space of fixed points is a hyperplane. A reflection group is a finite dimensional linear group over a field F, which is generated by a finite number of reflections.

There are no restrictions on the field F and there is no requirement for a reflection to be a transformation of order two. However, if F is a real field, every reflection does have order two and there is a much richer theory. In particular, every Coxeter group is a real reflection group (see Chapter 98).

The books [LT09], [Bro10] or [Kan01] are useful references for complex reflection groups. Standard references for the theory of real reflection groups include [Bou68, Chapters 4, 5, 6] and [Hum90].

99.2 Construction of Pseudo-reflections

Let V be a vector space of dimension n over a field F. As defined in Bourbaki [Bou68], a *pseudo-reflection* in MAGMA is a linear transformation of V whose space of fixed points is a subspace of dimension $n-1$, namely a hyperplane. (Some authors require a pseudo-reflection to be invertible and diagonalisable.)

A reflection, as defined above, is a pseudo-reflection and so too is a transvection. The MAGMA package described in this chapter includes code for the construction of transvections but the emphasis is on groups generated by reflections.

If r is a pseudo-reflection, then $\dim(\mathrm{im}(1-r)) = 1$ and a basis element of $\mathrm{im}(1-r)$ is called a *root* of r.

Let a be a root of the pseudo-reflection r and let $H = \ker(1-r)$ be the hyperplane of fixed points of r. For all $v \in V$ there exists $\phi(v) \in F$ such that $v - vr = \phi(v)a$. Then $\phi \in V^*$ and $\ker\phi = H$. This means that every pseudo-reflection has the form

$$vr = v - \phi(v)a$$

and its determinant is $1 - \phi(a)$. The linear functional ϕ is a *coroot* of r.

- If $\phi(a) = 1$, then r is not invertible; it is the *projection* of V onto H along a.
- If $\phi(a) = 0$ (equivalently, $a \in H$), then r is by definition a *transvection*.
- If $\phi(a) \neq 0, 1$, then r is called a *reflection*. For the most part we consider only reflections of finite order, but not necessarily of order two.

In MAGMA both V and its dual space V^* are identified with the space F^n of row vectors of length n and the standard bilinear pairing between V and V^* is $(a, b) \mapsto ab^{\mathrm{tr}}$, where b^{tr} denotes the column vector which is the *transpose* of b.

The row vector b which represents the coroot ϕ is also called a *coroot* of the pseudo-reflection; it is uniquely determined by r and a. The matrix of r is

$$I - b^{\mathrm{tr}}a$$

and, in particular, $ar = (1 - ab^{\mathrm{tr}})a$. Thus r is a reflection of finite order d if and only if $ab^{\mathrm{tr}} \neq 0, 1$ and $1 - ab^{\mathrm{tr}}$ is a d-th root of unity.

`PseudoReflection(a, b)`

The matrix of the pseudo-reflection with root a and coroot b.

`Transvection(a, b)`

The matrix of the transvection with root a and coroot b. The input is checked to ensure that the root and coroot define a transvection.

`Reflection(a, b)`

The matrix of the reflection with root a and coroot b. The input is checked to ensure that the root and coroot define a reflection.

`IsPseudoReflection(r)`

Returns `true` if r is the matrix of a pseudo-reflection, in which case a root and a coroot are returned as well.

`IsTransvection(r)`

Returns `true` if r is the matrix of a transvection, in which case a root and a coroot are returned as well.

`IsReflection(r)`

Returns `true` if r is the matrix of a reflection, in which case a root and a coroot are returned as well.

`IsReflectionGroup(G)`

Returns `true` if G is a group generated by reflections.

Example H99E1

Create a pseudo-reflection directly and then check that it is a transvection.

```
> V := VectorSpace(GF(5), 3);
> t := PseudoReflection(V![1,0,0],V![0,1,0]);
> t;
[1 0 0]
[4 1 0]
[0 0 1]
> IsTransvection(t);
true (1 0 0)
(0 1 0)
> IsReflection(t);
false
```

Example H99E2

An example of a group which can be generated by reflections even though not every given generator is a reflection.

```
> F<omega> := CyclotomicField(3);
> r := Matrix(F,2,2,[1,omega^2,0,omega]);
> IsReflection(r);
true (         0 -omega + 1)
(1/3*(2*omega + 1)                  1)
> s := Matrix(F,2,2,[0,-1,1,0]);
> IsReflection(s);
false
> G := MatrixGroup<2,F | r,s >;
> IsReflectionGroup(G);
true
> #G;
24
```

To find reflection generators for this group we look for a reflection which, together with the reflection r, generates G. (This is a rather special example; not every finite reflection group of rank two can be generated by two reflections.)

```
> exists(t){ t : t in G | IsReflection(t) and G eq sub<G|r,t> };
true
> t;
[        0 omega + 1]
[        1    -omega]
>
```

Example H99E3

The groups $\mathrm{SL}(n, q)$ are generated by transvections. To illustrate this we find representatives for the conjugacy classes of $\mathrm{GL}(3, 25)$ which are transvections and then check that the normal closure is $\mathrm{SL}(3, 25)$.

```
> G := GL(3,25);
> ccl := Classes(G);
> T := [ c : c in ccl | IsTransvection(c[3]) ];
> #T;
1
> t := T[1][3]; t;
[     1      0      0]
[     0      1      1]
[     0      0      1]
> S := ncl< G | t >;
> S eq SL(3,25);
true
```

99.2.1 Pseudo-reflections Preserving Reflexive Forms

Let J be the matrix of a non-degenerate reflexive bilinear or sesquilinear form β on the vector space V over a field F. Then β is either a symmetric, alternating or hermitian form.

We may assume that F is equipped with an automorphism σ such that $\sigma^2 = 1$. If β is a symmetric or alternating form, σ is the identity; if β is hermitian, the order of $\sigma : \alpha \mapsto \bar{\alpha}$ is two and $J = \bar{J}^{\mathrm{tr}}$. If a is the row vector $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, define $\sigma(a) = (\sigma(\alpha_1), \sigma(\alpha_2), \ldots, \sigma(\alpha_n))$.

If a is a root of a pseudo-reflection r and if r preserves β, then the coroot of r is $\alpha\sigma(a)J^{\mathrm{tr}}$ for some $\alpha \in F$. Thus the matrix of r is $I - \alpha J^{\mathrm{tr}}\sigma(a)^{\mathrm{tr}}a$.

`SymplecticTransvection(a, alpha)`

The symplectic transvection with root a and multiplier α with respect to the form attached to the parent of a. If the form is not alternating a runtime error is generated.

If β is a non-degenerate alternating form preserved by a pseudo-reflection r, then the dimension of V is even and r must be a transvection. If a is a root of r, the coroot is $\alpha a J^{\mathrm{tr}}$ and the matrix of r is $I - \alpha J a^{\mathrm{tr}}a$, for some $\alpha \neq 0$ in F.

`UnitaryTransvection(a, alpha)`

The unitary transvection with root a and multiplier α with respect to the hermitian form attached to the parent of a.

The matrix of the unitary transvection is $I - \alpha J\bar{a}^{\mathrm{tr}}a$, where a is isotropic and the trace of α is 0; that is, $aJ\bar{a}^{\mathrm{tr}} = 0$ and $\alpha + \bar{\alpha} = 0$.

A runtime error is generated if the form is not hermitian, if a is not isotropic, or if the trace of α is not 0.

`UnitaryReflection(a, zeta)`

The unitary reflection with root a and determinant ζ, where ζ is a root of unity. The reflection preserves the hermitian form attached to the ambient space of a and sends a to ζa.

In the case of a unitary reflection r with matrix $I - \alpha J^{\mathrm{tr}}\sigma(a)^{\mathrm{tr}}a$, the root a must be non-isotropic and $ar = \zeta a$, where ζ is a root of unity. Therefore, $\alpha = (1-\zeta)/aJ\bar{a}^{\mathrm{tr}}$.

The vector $a^\vee = \bar{\alpha}a$ is the *coroot* of a and the definition of r becomes

$$vr = v - \beta(v, a^\vee)a.$$

`OrthogonalReflection(a)`

The reflection determined by a non-singular vector a of a quadratic space.

A *quadratic space* is a vector space V equipped with a quadratic form Q (see Chapter 88 for more details). The *polar form* of Q is the symmetric bilinear form $\beta(u, v) = Q(u+v) - Q(u) - Q(v)$. Thus $\beta(v, v) = 2Q(v)$ and therefore, if the characteristic of F is not two, Q is uniqely determined by β.

If a is non-singular (that is, $Q(a) \neq 0$), the formula

$$vr = v - Q(a)^{-1}\beta(v, a)a$$

defines a pseudo-reflection. If the characteristic of F is 2, this is a transvection; in all other cases it is a reflection. However, in characteristic 2 there is a certain ambivalence in the literature and the pseudo-reflections just defined are often called reflections.

The *coroot* of a is $a^\vee = Q(a)^{-1}a$. If the characteristic of F is not two, then $a^\vee = 2a/\beta(a, a)$ and this coincides with the usual notion of coroot, as found in [Hum90], for example. In particular, if $\beta(u, v)$ is the standard inner product $(u, v) = uv^{\mathrm{tr}}$, then the inner product and the pairing between V and its dual are essentially the same and the concepts of coroot and coroot coincide.

Example H99E4

We create an hermitian space by attaching an hermitian form J to a vector space V over a field with complex conjugation. The vector $a = (1, 0, 0, 0)$ is isotropic with respect to this form and therefore we can use it to create a unitary transvection.

```
> K<i> := CyclotomicField( 4 );
> sigma := hom< K -> K | x :-> ComplexConjugate(x) >;
> J := Matrix(4,4,[K|0,0,0,1, 0,0,1,0, 0,1,0,0, 1,0,0,0]);
> V := UnitarySpace(J,sigma);
> a := V![1,0,0,0];
> t := UnitaryTransvection(a,i);
> t;
[ 1  0  0  0]
[ 0  1  0  0]
[ 0  0  1  0]
[-i  0  0  1]
```

Continuing the previous example we note that $b = (1, 1, 1, 1)$ is non-isotropic and we create a unitary reflection of order 4 with b as root.

```
> b := V![1,1,1,1];
> InnerProduct(b,b);
4
> r := UnitaryReflection(b,i);
> r, Eigenvalues(r);
[1/4*(i + 3) 1/4*(i - 1) 1/4*(i - 1) 1/4*(i - 1)]
[1/4*(i - 1) 1/4*(i + 3) 1/4*(i - 1) 1/4*(i - 1)]
[1/4*(i - 1) 1/4*(i - 1) 1/4*(i + 3) 1/4*(i - 1)]
[1/4*(i - 1) 1/4*(i - 1) 1/4*(i - 1) 1/4*(i + 3)]
{
    <i, 1>,
    <1, 3>
}
```

99.3 Construction of Reflection Groups

In MAGMA a *pseudo-reflection group* is a group generated by a finite set of invertible pseudo-reflections. A convenient way to provide the generators for a pseudo-reflection group W is via a finite collection of roots and coroots. In this context the roots and coroots of the generators are called the *basic roots* and *basic coroots* of W.

In the most general case, even when the pseudo-reflection group W is generated by reflections, there are no known distinguished generating reflections whose roots have properties analogous to simple roots in Weyl groups or Coxeter groups. Therefore, one should be careful to distinguish between the basic roots as defined here and the simple (or fundamental) roots of real reflection groups

See Section 99.4 for the construction of real reflection groups and Section 99.5 for the construction of finite complex reflection groups.

`PseudoReflectionGroup(A, B)`

The pseudo-reflection group with the basic roots and corresponding coroots given by the rows of the matrices A and B.

Example H99E5

A direct construction of the Shephard and Todd group $G(14, 1, 2)$ with user supplied roots and coroots.

```
> F<z> := CyclotomicField(7);
> A := Matrix(F,2,3,[[z,0,1],[0,1,0]]);
> B := Matrix(F,2,3,[[1,1,1],[1,2,1]]);
> G<x,y> := PseudoReflectionGroup(A,B);
> IsReflectionGroup(G);
true
> Order(x),Order(y),Order(x*y);
14 2 28
> #G;
392
```

99.4 Construction of Real Reflection Groups

The only root of unity in the real field is -1, hence every pseudoreflection over the real field is a reflection. We call a reflection group *real* if it is defined over the reals and its simple roots and simple coroots are linearly independent. We allow real reflection groups to be defined as matrix groups over the integer ring (Chapter 18), the rational field (Chapter 20), number fields (Chapter 34), and cyclotomic fields (Chapter 36); the real field (Chapter 25) is *not* allowed since it is not infinite precision.

The real reflection groups are just the reflection representations of the Coxeter groups (Chapter 98). This allows us to compute many more properties for these groups than for general reflection groups. Note that the classification of finite real reflection groups is given in Section 95.6.

`ReflectionGroup(M)`

`ReflectionGroup(G)`

`ReflectionGroup(C)`

`ReflectionGroup(D)`

The reflection group with Coxeter matrix M, Coxeter graph G, Cartan matrix C, or Dynkin digraph D (see Chapter 95).

`ReflectionGroup(N)`

The finite or affine reflection group with Cartan name given by the string N (see Section 95.6).

`IrreducibleReflectionGroup(X, n)`

The finite or affine irreducible reflection group with Cartan name X_n (see Section 95.6).

Example H99E6

```
> C := CartanMatrix("B3" : Symmetric);
> G := ReflectionGroup(C);
> G;
MatrixGroup(3, Number Field with defining polynomial x^2 - 2 over the Rational
Field) of order 48 = 2^4 * 3
Generators:
    [-1  0  0]
    [ 1  1  0]
    [ 0  0  1]

    [  1   1   0]
    [  0  -1   0]
    [  0 $.1   1]

    [  1   0   0]
    [  0   1 $.1]
    [  0   0  -1]
```

`ReflectionGroup(R)`

The finite reflection group with root system or root datum R (see Chapters 96 and 97).

Example H99E7

```
> R := RootDatum("B3");
> ReflectionGroup(R);
MatrixGroup(3, Integer Ring) of order 48 = 2^4 * 3
Generators:
    [-1  0  0]
    [ 1  1  0]
    [ 0  0  1]

    [ 1  1  0]
    [ 0 -1  0]
    [ 0  1  1]

    [ 1  0  0]
    [ 0  1  2]
    [ 0  0 -1]
```

`ReflectionGroup(W)`

A	Mtrx	*Default :*
B	Mtrx	*Default :*
C	Mtrx	*Default :*

A reflection group W' of the Coxeter group W, together with the isomorphism $W \to W'$ (see Chapter 98). Since a Coxeter group W does not come with an in-built reflection representation, the optional parameters A, B, and C can be used to specify the representation. They are respectively the matrix whose rows are the simple roots, the matrix whose rows are the simple coroots, and the Cartan matrix. These must have the following properties:

1. A and B must have same number of rows and the same number of columns; they must be defined over the same field, which must be the rational field, a number field, or a cyclotomic field; the entries must be real;
2. the number of columns must be at least the number of rows; and
3. $C = AB^{\mathrm{tr}}$ must be a Cartan matrix for W.

It is not necessary to specify all three matrices: any two of them will determine the third. If C is not determined, it is taken to be the standard matrix described in Section 95.4.

`ReflectionGroup(W)`

The reflection group W' isomorphic to the permutation Coxeter group W, together with the isomorphism $W \to W'$ (see Chapter 98). There are no optional parameters A, B, and C in this case because every permutation Coxeter group has a root system, and this determines the reflection representation.

Example H99E8

```
> W<a,b,c> := CoxeterGroup(GrpFPCox, "B3");
> G, h := CoxeterGroup(GrpMat, W);
> a*b; h(a*b);
a * b
[-1 -1  0]
[ 1  0  0]
[ 0  1  1]
```

99.5 Construction of Finite Complex Reflection Groups

In this section, we describe the classification and construction of finite complex reflection groups.

A finite complex reflection group has a finite root system but there is no known analogue of a set of simple roots as in the theory of finite Coxeter groups. To illustrate the difficulty, one of the examples in this section constructs a complex reflection group of rank 4 which cannot be generated by fewer than 5 generators.

Nevertheless, it is possible to generalise the concept of *root datum* to the complex case and construct all complex reflection groups via their root data.

Let D be the ring of integers of a number field F which admits a well-defined operation of complex conjugation (which in the case of a real number field will be the identity automorphism). Let $\mu(D)$ be the group of roots of unity in D and let $V = F \times_D L$.

A *complex root datum* is a 4-tuple (L, L^*, Φ, ρ), where

- L and L^* are free D-modules of rank n which are in duality via a pairing $L \times L^* \to D : (a, \phi) \mapsto \langle\, a, \phi \,\rangle$;
- Φ is a finite subset of L and $\rho : \Phi \to L^*$.

For all $a \in \Phi$ we have:

1. for all $\lambda \in F$, we have $\lambda a \in \Phi$ if and only if $\lambda \in \mu(D)$;
2. for all $\lambda \in D$, we have $\rho(\lambda a) = \overline{\lambda}\rho(a)$;
3. $f(a) = 1 - \langle\, a, \rho(a) \,\rangle \in \mu(D) \setminus \{1\}$;
4. the reflection r_a of V defined by $vr_a = v - \langle\, v, \rho(a) \,\rangle a$ and the reflection r_a^* of V^* defined by $\phi r_a^* = \phi - \langle\, a, \phi \,\rangle \rho(a)$ satisfy:
   - $\Phi r_a \subseteq \Phi$ and $\Phi^* r_a^* \subseteq \Phi^*$, where $\Phi^* = \rho(\Phi)$.
   - $f(ar_b) = f(a)$ for all $a, b \in \Phi$.

Put $a^* = \rho(a)$ and $V^* = F \otimes_D L^*$. Then $\rho : \Phi \to \Phi^*$ is a bijection and the map

$$p := V \to V^* : v \mapsto \sum_{a \in \Phi} \overline{\langle\, v, a^* \,\rangle} a^*$$

is semilinear. Furthermore, $\beta(u, v) = \langle\, u, p(v) \,\rangle$ defines a non-degenerate hermitian form on the span of Φ.

The group W generated by the reflections $\{r_a \mid a \in \Phi\}$ is the *Weyl group* of the root datum. For any set $\{r_1, r_2, \ldots, r_k\}$ of reflections that generate W, every reflection in W is conjugate to a power of some r_i. The set Φ is a *root system* for W and Φ^* is the set of *coroots*.

If a_1, a_2, ..., $a_k \in \Phi$ are roots of the reflections r_1, r_2, ..., r_n which generate W, then $C = (\langle a_i, a_j^* \rangle)$ is a *complex Cartan matrix* and the a_i and a_j^* are *basic* roots and coroots of W.

Even though there is no satisfactory notion of 'simple roots', a complex reflection group can nevertheless be described by means of a complex Cartan matrix. In MAGMA if the roots are the rows of a matrix A and if the coroots are the rows of a matrix B, then $C = AB^{\mathrm{tr}}$. The matrices A and B are called *basic* root and coroot matrices.

The complex Cartan matrix can be described by a diagram similar to the Dynkin diagram of a Coxeter group. This notation was suggested by Coxeter and used by Cohen in [Coh76]. (There is a different type of diagram used by Broué, Malle and others.)

Cohen's naming scheme for the diagrams extends the standard notation A_n, B_n, ..., H_3, H_4 used for Coxeter groups. MAGMA uses a slight variation of Cohen's scheme; that is, in MAGMA, Cohen's group EN_4 is referred to as O_4.

The original numbering system for the primitive complex reflection groups is due to Shephard and Todd [ST54].

`ShephardTodd(n)`

`NumFld` BOOLELT *Default :* `false`

This function returns the primitive reflection group G_n using the Shephard and Todd numbering.

By default the matrices are written over the ring of integers of the smallest cyclotomic field which contains the character values of the reflections. If the parameter `NumFld` is set to `true`, the number field generated by the character values of the reflections is used.

The groups available via this function include all finite primitive complex reflection groups other than the symmetric groups $\mathrm{Sym}(n)$ for $n \geq 5$. The groups are listed below.

Nineteen 2-dimensional primitive complex reflection groups:

Tetrahedral family: G_4, ..., G_7
Octahedral family: G_8, ..., G_{15}
Icosahedral family: G_{16}, ..., G_{22}

Five 3-dimensional complex reflection groups:

G_{23}: $W(H_3) = \mathbf{Z}_2 \times PSL(2,5)$, order 120.
G_{24}: $W(J_3(4)) = \mathbf{Z}_2 \times PSL(2,7)$, order 336.
G_{25}: $W(L_3) = 3^{1+2} \cdot SL(2,3)$, order 648; Hessian group.
G_{26}: $W(M_3) = \mathbf{Z}_2 \times 3^{1+2} \cdot SL(2,3)$, order 1296; Hessian group.
G_{27}: $W(J_3(5)) = \mathbf{Z}_2 \times (\mathbf{Z}_3 \cdot \mathrm{Alt}(6))$, order 2160, where $\mathbf{Z}_3 \cdot \mathrm{Alt}(6)$ denotes the non-split extension of $\mathbf{Z}_3$ by $\mathrm{Alt}(6)$.

Five 4-dimensional complex reflection groups in addition to Sym(5):

G_{28}: $W(F_4) = (SL(2,3) \circ SL(2,3)) \cdot (\mathbf{Z}_2 \times \mathbf{Z}_2)$, order 1152.
G_{29}: $W(N_4) = (\mathbf{Z}_4 \circ 2^{1+4}) \cdot \mathrm{Sym}(5)$, order 7680 (splits).
G_{30}: $W(H_4) = (SL(2,5) \circ SL(2,5)) \cdot \mathbf{Z}_2$, order 14 400.
G_{31}: $W(O_4) = (\mathbf{Z}_4 \circ 2^{1+4}) \cdot Sp(4,2)$, order 46 080 (non-split) 5 generators.
G_{32}: $W(L_4) = \mathbf{Z}_3 \times Sp(4,3)$, order $155\,520 = 2^7 \times 3^5 \times 5$.

One 5-dimensional complex reflection group in addition to Sym(6):

G_{33}: $W(K_5) = \mathbf{Z}_2 \times \Omega(5,3) = \mathbf{Z}_2 \times PSp(4,3) = \mathbf{Z}_2 \times PSU(4,2)$, order $51\,840 = 2^7 \times 3^4 \times 5$.

Two 6-dimensional complex reflection groups in addition to Sym(7):

G_{34}: $W(K_6) = \mathbf{Z}_3 \cdot \widehat{\Omega}^-(6,3)$, order $39\,191\,040 = 2^9 \times 3^7 \times 5 \times 7$ (non-split), where $\widehat{\Omega}^-(6,3)$ is a semidirect product of $\Omega^-(6,3)$ by $\mathbf{Z}_2$.

G_{35}: $W(E_6) = SO(5,3) = O^-(6,2) = PSp(4,3) \cdot \mathbf{Z}_2 = PSU(4,2) \cdot \mathbf{Z}_2$, order $51\,840 = 2^7 \times 3^4 \times 5$.

One 7-dimensional complex reflection group in addition to Sym(8):

G_{36}: $W(E_7) = \mathbf{Z}_2 \times Sp(6,2)$, order $2\,903\,040 = 2^{10} \times 3^4 \times 5 \times 7$.

One 8-dimensional complex reflection group in addition to Sym(9):

G_{37}: $W(E_8) = \mathbf{Z}_2 \cdot O^+(8,2)$, order $696\,729\,600 = 2^{14} \times 3^5 \times 5^2 \times 7$ (non-split).

Example H99E9

We verify that the complex reflection group G_{24} is isomorphic to $\mathbf{Z}_2 \times \Omega(3,7)$.

```
> W := ShephardTodd(24);
> G := sub<GL(3,7) | Omega(3,7), -GL(3,7)!1>;
> IsIsomorphic(W,G):Minimal;
true Homomorphism of MatrixGroup(3, Cyclotomic Field of order 7 and degree 6)
of order 2^4 * 3 * 7 into MatrixGroup(3, GF(7)) of order 2^4 * 3 * 7
```

`ComplexReflectionGroup(C)`

`Reduced` BoolElt *Default :* `true`

This function returns the complex reflection group defined by the (complex) Cartan matrix C. When the optional parameter `Reduced` is `true` (the default), the roots and coroots are computed modulo the null space of C.

`ComplexReflectionGroup(X, n)`

`NumFld` BoolElt *Default :* `false`

This function returns the primitive reflection group of type X and rank n, using the Cohen/Coxeter naming scheme.

By default the matrices are written over the ring of integers of the smallest cyclotomic field which contains the character values of the reflections. If the parameter `NumFld` is set to `true`, the number field generated by the character values of the reflections is used.

Example H99E10

In this example we find (up to conjugacy) all subgroups of $G = W(O_4) = G_{31}$ that are generated by reflections. This shows that G cannot be generated by fewer than 5 reflections.
We begin by checking that G has only one class of reflections.

```
> G := ComplexReflectionGroup("O",4);
> print #[c[3] : c in Classes(G) | IsReflection(c[3])];
1
> R := Class(G,G.1); #R;
60
> #G;
46080
```

We proceed by building the list of reflection subgroups in 'layers', where the n-th layer consists of representatives of the subgroups generated by n reflections.

```
> L := [sub<G|G.1>];
> layers := [L];
> n := 0;
> while true do
>   n +:= 1;
>   nextlayer := [];
```

We extend each group in layer n by adjoining one additional reflection. The resulting subgroup will be generated by n or $n+1$ reflections. If we haven't seen it before we add it to the list.

```
>   for H in layers[n] do
>     for A in {sub<G|H,s> : s in R | s notin H} do
>       if forall{B : B in L | not IsConjugate(G,A,B)} then
>         Append(~nextlayer,A);
>         Append(~L,A);
>       end if;
>     end for;
>   end for;
>   if IsEmpty(nextlayer) then break; end if;
>   Append(~layers,nextlayer);
```

After the construction of each layer we print the orders of the subgroups.

```
>   print n+1,"generators";
```

```
>   print [#A : A in nextlayer];
> end while;
2 generators
[ 4, 4, 8, 6 ]
3 generators
[ 8, 8, 16, 24, 12, 16, 24, 48, 16, 96 ]
4 generators
[ 192, 16, 32, 96, 16, 192, 32, 48, 1536, 384, 64, 64, 32,
 7680, 1152, 36, 384, 120, 120, 192, 192 ]
5 generators
[ 64, 384, 3072, 128, 46080 ]
6 generators
[ 256 ]
```

Looking at the orders we see that the first time the group G_{31} appears is in layer 5. That is, it cannot be generated by 4 or fewer reflections. It is interesting to note that there is one subgroup which requires 6 generators; namely the imprimitive group $G(4,2,2) \times G(4,2,2)$.

`ShephardTodd(m, p, n)`

`ImprimitiveReflectionGroup(m, p, n)`

`NumFld` BoolElt *Default :* `false`

Let B be the direct product of n copies of the cyclic group C_m of order m and represent the elements of B by diagonal matrices $\mathrm{diag}(\theta_1, \theta_2, \ldots, \theta_n)$. The elements of the symmetric group $\mathrm{Sym}(n)$ can be represented by $n \times n$ permutation matrices and in this guise it acts on the group B; the resulting semidirect product is also known as the *wreath product* $C_m \wr \mathrm{Sym}(n)$.

For each divisor p of m define

$$A(m,p,n) := \{\, \mathrm{diag}(\theta_1, \theta_2, \ldots, \theta_n) \in B \mid (\theta_1\theta_2 \cdots \theta_n)^{m/p} = 1 \,\}.$$

It is immediately clear that $A(m,p,n)$ is a subgroup of index p in B that is invariant under the action of $\mathrm{Sym}(n)$. The semidirect product of $A(m,p,n)$ by the symmetric group $\mathrm{Sym}(n)$ is the group $G(m,p,n)$. These groups are imprimitive when $m \geq 2$. The group $G(1,1,n)$ is the symmetric group $\mathrm{Sym}(n)$ acting as permutation matrices.

Shephard and Todd proved that every irreducible imprimitive complex reflection subgroup of $GL(n, \mathbf{C})$ is conjugate to $G(m,p,n)$ for some m and p.

This function returns the Shephard and Todd group $G(m,p,n) \subset \mathrm{GL}(n,F)$, where p divides m. In general, $G(m,p,n)$ is irreducible but if $m = p = 1$, the function returns $\mathrm{Sym}(n)$ in its natural permutation representation, which is not irreducible.

By default the matrices are written over the ring of integers of the smallest cyclotomic field which contains the character values of the reflections. If the parameter `NumFld` is set to `true`, the number field generated by the character values of the reflections is used.

Example H99E11

```
> ShephardTodd(6, 3, 3);
MatrixGroup(3, Cyclotomic Field of order 6 and degree 2)
Generators:
     [0 1 0]
     [1 0 0]
     [0 0 1]

     [1 0 0]
     [0 0 1]
     [0 1 0]

     [1      0      0]
     [0      0      z]
     [0 -z + 1      0]

     [1  0   0]
     [0  1   0]
     [0  0  -1]
Mapping from: MatrixGroup(3, Cyclotomic Field of order 6 and degree 2) to GL(3,
CyclotomicField(6))
```

`ComplexRootMatrices(k)`

`ComplexRootMatrices(m, p, n)`

`NumFld` BoolElt *Default :* `false`

Given an integer n ($4 \leq n \leq 37$) the first form of the function returns basic root and coroot matrices for the primitive Shephard-Todd group G_n.

Given three positive integers m, p and n such that p divides m, the second version returns basic root and coroot matrices for the imprimitive complex reflection group $G(m, p, n)$. In both cases the functions return an invariant hermitian form, a generator for the group of roots of unity of the ring of definition, and the order of the generator.

By default the matrices are written over the ring of integers of the smallest cyclotomic field which contains the character values of the reflections. If the parameter `NumFld` is set to `true`, the number field generated by the character values of the reflections is used.

Example H99E12

```
> A,B,J,gen,ordgen := ComplexRootMatrices(13);
> A,B;
[            1               0]
[ -z^2 - z - 1               1]
[z^2 + 2*z + 1    -z^2 - z - 1]

[             2     -z^3 + z + 2]
[           z^2    z^3 + z^2 + 1]
[-z^3 + z^2 - 1          z^2 - 1]
> gen,ordgen;
-z^3
8
> G := PseudoReflectionGroup(A,B);
> #G;
96
```

`ComplexCartanMatrix(k)`

`ComplexCartanMatrix(m, p, n)`

`NumFld` BoolElt *Default :* `false`

If A and B are the basic root and coroot matrices returned by `ComplexRootMatrices` above, then this function returns AB^{tr}, where B^{tr} is the transpose of B. The meaning of the optional parameter `NumFld` has been described above.

`BasicRootMatrices(C)`

`Reduced` BoolElt *Default :* `true`

This function returns a matrix A of roots and a matrix B of coroots such that $C = AB^{\mathrm{tr}}$. The default, when the optional parameter `Reduced` is `true`, is to compute the roots and coroots modulo the null space of C.

`CohenCoxeterName(k)`

Cohen's string name and rank of the Shephard and Todd group G_k. This is an extension of the naming scheme for Coxeter groups. For example, the Shephard and Todd group G_{37} is the Coxeter group of type E_8 whereas the Shephard and Todd group G_{32} has Cohen name L_4.

`ShephardToddNumber(X, n)`

Given a string X and an integer n, this function returns the Shephard and Todd number of the complex reflection group $W(X_n)$ of type X and rank n. The rank is the dimension of the space on which the group acts; it is not always the number of generators.

The Shepard and Todd numbers range from 1 to 37. All symmetric groups (type A) have Shephard and Todd number 1, all imprimitive groups $G(m, p, n)$ have Shephard and Todd number 2, and all cyclic groups have Shephard and Todd number 3. The primitive complex reflection groups of rank 2 have Shepard and Todd numbers in the range 4 to 22. Except for the group G_4 which has type L_2, the rank 2 groups do not have Cohen–Coxeter names.

The Shephard and Todd numbers in the range 23 to 37 refer to the Cohen–Coxeter groups $W(E_6)$, $W(E_7)$, $W(E_8)$, $W(F_4)$, $W(H_3)$, $W(H_4)$, $W(J_3(4))$, $W(J_3(5))$, $W(K_5)$, $W(K_6)$, $W(L_3)$, $W(L_4)$, $W(M_3)$, $W(N_4)$, and $EW(N_4)$. Note that in MAGMA the types of the rank 3 groups $W(J_3(4))$ and $W(J_3(5))$ are $J4$ and $J5$; and the type of the rank 4 group $EW(N_4)$ is O.

As a matrix group the Coxeter group of type A is returned by the function `CoxeterGroup(GrpMat,"A",n)`, where n is the rank. The groups of types B, C and D are Coxeter groups and imprimitive complex reflection groups. Thus, as matrix groups, they can be obtained via the function `ShephardTodd(2,p,n)`, where $p = 1$ for type B or C and $p = 2$ for type D.

Example H99E13

The type of the group G_{31} is O and its rank is 4. This is the notation used in [LT09].

```
> ShephardToddNumber("J5",3);
27
> CohenCoxeterName(31);
O 4
```

Example H99E14

To construct a complex reflection group with a given name, first convert the name to its Shephard and Todd number.

```
> G := ShephardTodd(ShephardToddNumber("L",4));
> G;
MatrixGroup(4, Cyclotomic Field of order 3 and degree 2)
Generators:
    [     omega           0           0           0]
    [-omega - 1           1           0           0]
    [         0           0           1           0]
    [         0           0           0           1]

    [         1 omega + 1           0           0]
    [         0     omega           0           0]
```

```
[        0 omega + 1          1          0]
[        0          0          0          1]

[         1          0          0          0]
[         0          1 -omega - 1          0]
[         0          0      omega          0]
[         0          0 -omega - 1          1]

[        1          0          0          0]
[        0          1          0          0]
[        0          0          1 omega + 1]
[        0          0          0      omega]
```

`ComplexRootDatum(k)`

`ComplexRootDatum(m, p, n)`

`NumFld` BoolElt *Default :* `false`

A root datum for the Shephard and Todd group G_k or, in the second form of the function, the imprimitive group $G(m, p, n)$. This is returned as a 5-tuple Φ, Φ^*, ρ, W, J, where Φ is the sequence of roots, Φ^* the sequence of coroots, $\rho : A \to B$ is a bijective map, W is the complex reflection group of the root datum, and J is an hermitian form preserved by W.

99.6 Operations on Reflection Groups

See Chapter 59 for general functions for matrix groups. Note that most of the functions in this section only work for real reflection groups.

`IsCoxeterIsomorphic(W1, W2)`

Returns `true` if, and only if, the real reflection groups W_1 and W_2 are isomorphic as Coxeter groups.

`IsCartanEquivalent(W1, W2)`

Returns `true` if, and only if, the crystallographic real reflection groups W_1 and W_2 have Cartan equivalent Cartan matrices.

Example H99E15

```
> W1 := ReflectionGroup("B3");
> W2 := ReflectionGroup("C3");
> IsCoxeterIsomorphic(W1, W2);
true [ 1, 2, 3 ]
> IsCartanEquivalent(W1, W2);
false
```

`CartanName(W)`

The Cartan name of the finite or affine real reflection group W (Section 95.6).

`CoxeterDiagram(W)`

A display of the Coxeter diagram of the real reflection group W (Section 95.6). If W is not affine or finite, an error is flagged.

`DynkinDiagram(W)`

A display of the Coxeter diagram of the real reflection group W (Section 95.6). If W is not affine or finite, or if W is not crystallographic, an error is flagged.

Example H99E16

```
> G := CompleteGraph(3);
> W := ReflectionGroup(G);
> CartanName(W);
A~2
> CoxeterDiagram(W);
A~2    1 - 2
       |   |
       - 3 -
```

`RootSystem(W)`

The root system of the finite real reflection group W (Chapter 96). If W is infinite, an error is flagged.

`RootDatum(W)`

The root datum of the finite real reflection group W (Chapter 97). The roots and coroots of W must have integral components, and W must be finite.

`CoxeterMatrix(W)`

The Coxeter matrix of the real reflection group W (Section 95.2).

`CoxeterGraph(W)`

The Coxeter graph of the real reflection group W (Section 95.3).

`CartanMatrix(W)`

The Cartan matrix of the real reflection group W (Section 95.4).

`DynkinDigraph(W)`

The Dynkin digraph of the real reflection group W (Section 95.5).

`Rank(W)`

`NumberOfGenerators(W)`

The rank of the reflection group W.

Example H99E17

```
> R := StandardRootSystem("A", 4);
> W := ReflectionGroup(R);
> Rank(W);
4
> Dimension(W);
5
```

`FundamentalGroup(W)`

The fundamental group of the real reflection group W (Subsection 97.1.6). The roots and coroots of W must have integral components.

`IsogenyGroup(W)`

The isogeny group of the real reflection group W, together with the injection into the fundamental group (Subsection 97.1.6). The roots and coroots of W must have integral components.

`CoisogenyGroup(W)`

The fundamental group of the real reflection group W together with the projection onto the fundamental group (Subsection 97.1.6). The roots and coroots of W must have integral components.

`BasicDegrees(W)`

The degrees of the basic invariant polynomials of the reflection group W. These are computed using the table in [Car72, page 155] if the group is real, and using the algorithm of [LT09] in other cases. If W is infinite, an error is flagged.

`BasicCodegrees(W)`

The basic codegrees of the reflection group W. These are computed using the algorithm of [LT09]. If W is infinite, an error is flagged.

Example H99E18

The product of the basic degrees is the order of the Coxeter group; the sum of the basic degrees is the sum of the rank and the number of positive roots.

```
> W := ReflectionGroup("E6");
> degs := BasicDegrees(W);
> degs;
[ 2, 5, 6, 8, 9, 12 ]
> &*degs eq #W;
true
> &+degs eq NumPosRoots(W) + Rank(W);
true
```

`LongestElement(W)`

The unique longest element in the finite real reflection group W.

`CoxeterElement(W)`

The Coxeter element in the reflection group W, ie. the product of the generators.

`CoxeterNumber(W)`

The order of the Coxeter element in the real reflection group W.

Example H99E19

Operations on groups.

```
> W := ReflectionGroup("A4");
> LongestElement(W);
[ 0  0  0 -1]
[ 0  0 -1  0]
[ 0 -1  0  0]
[-1  0  0  0]
> CoxeterElement(W);
[-1 -1 -1 -1]
[ 1  0  0  0]
[ 0  1  0  0]
[ 0  0  1  0]
```

`LeftDescentSet(W, w)`

The set of indices r of simple roots of the finite real reflection group W such that the length of the product $s_r w$ is less than that of the element w.

`RightDescentSet(W, w)`

The set of indices r of simple roots of the finite real reflection group W such that the length of the product ws_r is less than that of the element w.

Example H99E20

```
> W := ReflectionGroup("A5");
> x := W.1*W.2*W.4*W.5;
> LeftDescentSet(W, x);
{ 1, 4 }
> RightDescentSet(W, x);
{ 2, 5 }
```

99.7 Properties of Reflection Groups

See Chapter 59 for general functions for matrix groups.

`IsReflectionGroup(G)`

Returns `true` if G is a group generated by reflections. It need not be the case that all the group elements returned by `Generators(G)` are reflections.

`RootsAndCoroots(G)`

Returns the orders of the reflections, the roots and the coroots of the reflection group G.

`IsRealReflectionGroup(G)`

Returns `true` if, and only if, the matrix group G is a real reflection group. If `true`, the simple orders, roots, and coroots are also returned.

Example H99E21

```
> W := ComplexReflectionGroup("A", 4);
> IsReflectionGroup(W);
true
> IsRealReflectionGroup(W);
true

[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
> W := ComplexReflectionGroup("M", 3);
> IsReflectionGroup(W);
true
>  IsRealReflectionGroup(W);
                          ^
Runtime error in 'IsRealReflectionGroup': The group must be defined over the
reals
```

`IsCrystallographic(W)`

Returns `true` if, and only if, the real reflection group W is crystallographic, i.e. its Cartan matrix has integral entries.

`IsSimplyLaced(W)`

Returns `true` if, and only if, the real reflection group W is simply laced, i.e. its Coxeter graph has no labels.

Example H99E22

```
> W := ReflectionGroup("A~2 D4");
> IsFinite(W);
false
> IsCrystallographic(W);
true
> IsSimplyLaced(W);
true
```

`Dual(G)`

The dual of the reflection group G, ie, the reflection group gotten by swapping roots with coroots.

`Overgroup(H)`

The overgroup of H, ie. the reflection group whose roots are permuted by the elements of the reflection subgroup H.

`Overdatum(H)`

The root datum whose roots are permuted by the elements of the reflection subgroup H.

Every Coxeter group W has a standard action. For example, the standard action group of a Coxeter group of type A_n is the symmetric group of degree $n+1$ acting on $\{1, \ldots, n\}$.

`StandardAction(W)`

The standard action of the reflection group W.

`StandardActionGroup(W)`

The group G of the standard action of the reflection group W, together with an isomorphism $W \to G$.

99.8 Roots, Coroots and Reflections

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space. This is the default.
2. `"Root"`: the basis of simple (co)roots.
3. `"Weight"`: the basis of fundamental (co)weights (see Subsection 99.8.3 below).

99.8.1 Accessing Roots and Coroots

`RootSpace(W)`

`CorootSpace(W)`

The base space of the reflection group W. If W is not a reflection group, an error occurs.

Example H99E23

```
> W := ComplexReflectionGroup("M", 3);
> RootSpace(W);
Full Vector space of degree 3 over Cyclotomic Field of order 24 and degree 8
```

`SimpleOrders(W)`

The sequence of simple orders of the reflection group W. If W is not a reflection group, an error is flagged.

`SimpleRoots(W)`

`SimpleCoroots(W)`

The simple (co)roots of the reflection group W as the rows of a matrix, i.e. A (resp. B).

`NumberOfPositiveRoots(W)`

`NumPosRoots(W)`

The number of positive roots of the real reflection group W. This is also the number of positive coroots. The total number of (co)roots is twice the number of positive (co)roots. This number is finite if, and only if, W is finite.

`Roots(W)`

`Coroots(W)`

`Basis` MonStgElt *Default : "Standard"*

The indexed set of (co)roots of the real reflection group W, i.e. $\{@\,\alpha_1, \ldots \alpha_{2N}\,@\}$ (resp. $\{@\,\alpha_1^\star, \ldots \alpha_{2N}^\star\,@\}$). If W is infinite, an error is flagged.

`PositiveRoots(W)`

`PositiveCoroots(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The indexed set of positive (co)roots of the real reflection group W, that is, $\{@ \alpha_1, \ldots \alpha_N @\}$ (resp. $\{@ \alpha_1^\star, \ldots \alpha_N^\star @\}$). If W is infinite, an error is flagged.

`Root(W, r)`

`Coroot(W, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The rth (co)root α_r (resp. $\alpha_r^\star$) of the real reflection group W. If W is infinite, an error is flagged.

`RootPosition(W, v)`

`CorootPosition(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

If v is a (co)root in the finite real reflection group W, return its index; otherwise return 0. These functions will try to coerce v into the appropriate lattice; v should be written with respect to the basis specified by the parameter `Basis`. If W is infinite, an error is flagged.

Example H99E24

```
> W := ReflectionGroup("A3");
> Roots(W);
{@
    (1 0 0),
    (0 1 0),
    (0 0 1),
    (1 1 0),
    (0 1 1),
    (1 1 1),
    (-1  0  0),
    (0 -1  0),
    (0  0 -1),
    (-1 -1  0),
    (0 -1 -1),
    (-1 -1 -1)
@}
> PositiveCoroots(W);
{@
    (2 -1  0),
    (-1  2 -1),
    (0 -1  2),
    (1  1 -1),
```

```
    (-1  1  1),
    (1 0 1)
@}
> #Roots(W) eq 2*NumPosRoots(W);
true
> Root(W, 4);
(1 1 0)
> Root(W, 4 : Basis := "Root");
(1 1 0)
> RootPosition(W, [1,1,0]);
4
> W := ReflectionGroup("A3");
> Roots(W);
{@
    (1 0 0),
    (-1 -1 -3),
    (1 2 4),
    (0 -1 -3),
    (0 1 1),
    (1 1 1),
    (-1  0  0),
    (1 1 3),
    (-1 -2 -4),
    (0 1 3),
    (0 -1 -1),
    (-1 -1 -1)
@}
> PositiveCoroots(W);
{@
    (2 -1  0),
    (-1  2 -1),
    (0 1 0),
    (1  1 -1),
    (-1  3 -1),
    (1  2 -1)
@}
> #Roots(W) eq 2*NumPosRoots(W);
true
> Root(W, 4);
(0 -1 -3)
> Root(W, 4 : Basis := "Root");
(1 1 0)
> RootPosition(W, [0,-1,-3]);
4
```

99.8.2 Reflections

The root α acts on the root space via the reflection s_α; the coroot $\alpha^\star$ acts on the coroot space via the coreflection $s_\alpha^\star$.

`ReflectionMatrices(W)`

`CoreflectionMatrices(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The sequence of reflections in the finite real reflection group W. The rth reflection in the sequence corresponds to the rth (co)root.

`SimpleReflectionMatrices(W)`

`SimpleCoreflectionMatrices(W)`

`Basis` MonStgElt *Default :* "*Standard*"

The matrices giving the action of the simple (co)roots on the (co)root space of the finite real reflection group W.

`ReflectionMatrix(W, r)`

`CoreflectionMatrix(W, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The reflection in finite real reflection group W corresponding to the rth (co)root. If $r = 1, \ldots, n$, this is a generator of W.

`SimpleReflectionPermutations(W)`

The sequence of permutations giving the action of the simple (co)roots of the finite reflection group W on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutations(W)`

The sequence of permutations giving the action of the (co)roots of the finite reflection group W on the (co)roots. This action is the same for roots and coroots.

`ReflectionPermutation(W, r)`

The permutation giving the action of the rth (co)root of the finite reflection group W on the (co)roots. This action is the same for roots and coroots.

`ReflectionWords(W)`

The sequence of words in the simple reflections for all the reflections of the real reflection group W. These words are given as sequences of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWords(W)[r]`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

`ReflectionWord(W, r)`

The word in the simple reflections for the rth reflection of the real reflection group W. The word is given as a sequence of integers. In other words, if $a = [a_1, \ldots, a_l] =$ `ReflectionWord(W,r)`, then $s_{\alpha_r} = s_{\alpha_{a_1}} \cdots s_{\alpha_{a_l}}$.

Example H99E25

```
> Q := RationalField();
> W := ReflectionGroup("A3");
> mx := ReflectionMatrix(W, 4);
> perm := ReflectionPermutation(W, 4);
> RootPosition(W, Vector(Q, Eltseq(Root(W,2))) * mx) eq 2^perm;
true
> mx := CoreflectionMatrix(W, 4);
> CorootPosition(W, Coroot(W,2) * mx) eq 2^perm;
true
```

Length(w)

CoxeterLength(w)

The length of w as an element of the Coxeter group W, ie. the number of positive roots of W which become negative under the action of w.

99.8.3 Weights

WeightLattice(W)

CoweightLattice(W)

The (co)weight lattice of the real reflection group W. The roots and coroots of W must have integral components.

FundamentalWeights(W)

FundamentalCoweights(W)

Basis MonStgElt *Default : "Standard"*

The fundamental weights of the real reflection group W given as the rows of a matrix. The roots and coroots of W must have integral components.

Example H99E26

```
> W := ReflectionGroup("E6");
> WeightLattice(W);
Lattice of rank 6 and degree 6
Basis:
(4  3  5  6  4  2)
(3  6  6  9  6  3)
(5  6 10 12  8  4)
(6  9 12 18 12  6)
(4  6  8 12 10  5)
(2  3  4  6  5  4)
Basis Denominator: 3
```

```
> FundamentalWeights(W);
[ 4/3    1  5/3    2  4/3  2/3]
[   1    2    2    3    2    1]
[ 5/3    2 10/3    4  8/3  4/3]
[   2    3    4    6    4    2]
[ 4/3    2  8/3    4 10/3  5/3]
[ 2/3    1  4/3    2  5/3  4/3]
```

`IsDominant(R, v)`

`Basis` MonStgElt *Default :* "*Standard*"

Returns `true` if, and only if, v is a dominant weight for the root datum R, ie, a nonnegative integral linear combination of the fundamental weights.

`DominantWeight(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique dominant weight in the same W-orbit as v, where W is a real reflection group and v is a weight given as a vector or a sequence representing a vector. The second value returned is a Coxeter group element taking v to the dominant weight.

`WeightOrbit(W, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The W-orbit of v as an indexed set, where W is a real reflection group and v is a weight given as a vector or a sequence representing a vector. The first element in the orbit is always dominant. The second value returned is a sequence of Coxeter group elements taking the dominant weight to the corresponding element of the orbit.

Example H99E27

```
> W := CoxeterGroup("B3");
> DominantWeight(W, [1,-1,0] : Basis:="Weight");
(1 0 0)
$.2 * $.3 * $.2 * $.1
> #WeightOrbit(W, [1,-1,0] : Basis:="Weight");
6
```

99.9 Related Structures

In this section we briefly list functions for creating other structures from a reflection group. See the appropriate chapters of the Handbook for more details.

`CoxeterGroup(GrpFPCox, W)`

The Coxeter group isomorphic to the real reflection group W. See Chapter 98.

`CoxeterGroup(GrpPermCox, W)`

The permutation Coxeter group isomorphic to the finite real reflection group W. See Chapter 98.

`LieAlgebra(W, R)`

The reductive Lie algebra over the ring R with Weyl group W. Unless W is finite, real, and crystallographic, an error is flagged. See Section 100.5.1.

`GroupOfLieType(W, k)`

The group of Lie type over the field k with Weyl group W. Unless W is finite, real, and crystallographic, an error is flagged. The roots and coroots of W must also have integral components. See Chapter 103.

99.10 Bibliography

[**Bou68**] N. Bourbaki. *Éléments de mathématique. Fasc. XXXIV. Groupes et algèbres de Lie. Chapitre IV: Groupes de Coxeter et systèmes de Tits. Chapitre V: Groupes engendrés par des réflexions. Chapitre VI: Systèmes de racines.* Hermann, Paris, 1968.

[**Bro10**] M. Broué. *Introduction to Complex Reflection Groups and their Braid Groups*, volume 1988 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 2010.

[**Car72**] Roger W. Carter. *Simple groups of Lie type*. John Wiley & Sons, London-New York-Sydney, 1972. Pure and Applied Mathematics, Vol. 28.

[**Coh76**] Arjeh M. Cohen. Finite complex reflection groups. *Ann. Sci. École Norm. Sup. (4)*, 9(3):379–436, 1976.

[**Hum90**] James E. Humphreys. *Reflection groups and Coxeter groups*, volume 29 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1990.

[**Kan01**] Richard Kane. *Reflection Groups and Invariant Theory*, volume 5 of *CMS Books in Mathematics/Ouvrages de Mathématiques de la SMC*. Springer-Verlag, New York, 2001.

[**LT09**] G. I. Lehrer and D. E. Taylor. *Unitary Reflection Groups*, volume 20 of *Australian Mathematical Society Lecture Series*. Cambridge University Press, Cambridge, 2009.

[**ST54**] G. C. Shephard and J. A. Todd. Finite unitary reflection groups. *Canadian J. Math.*, 6:274–304, 1954.

100 LIE ALGEBRAS

Chapter 100
LIE ALGEBRAS

100.1 Introduction

This chapter is concerned with finite dimensional Lie algebras. A large number of specialised functions are provided for these algebras. We refer to [dG00] for a general introduction to the theory of Lie algebras and their algorithms. For some of the functions described here that rely on a non-trivial algorithm we will indicate a precise reference.

Lie algebras are viewed as free modules over a base ring R with a multiplication satisfying the usual Lie axioms. Some functions require additional conditions on the base ring; for example, many functions require that the base ring be a field.

The main computational machinery in MAGMA for Lie algebras assumes that they are given either as structure constant algebras or as matrix algebras. Functions are provided which, given a finitely presented finite dimensional Lie algebra, will attempt to construct an isomorphic structure constant Lie algebra. As a structure constant algebra, the Lie algebra L of dimension n over a ring R is defined in MAGMA by giving the n^3 *structure constants* $a_{ij}^k \in R (1 \leq i, j, k \leq n)$ such that, if $\{e_1, e_2, \ldots, e_n\}$ is the basis of L, $e_i * e_j = \sum_{k=1}^{n} a_{ij}^k * e_k$.

In MAGMA V2.19 there is more functionality for Lie algebras defined by structure constants than for matrix Lie algebras. Throughout this chapter the algebra representation appropriate for a given intrinsic will be noted. For information on matrix algebras considered as associative algebras see Chapter 83.

In addition, some functions are provided for finitely presented Lie algebras and Lie algebras generated by extremal elements, and databases of solvable Lie algebras, nilpotent Lie algebras, and nilpotent orbits in simple Lie algebras are available.

100.1.1 Guide for the Reader

As mentioned above, the most extensively supported Lie algebras in MAGMA are structure constant Lie algebras and matrix Lie algebras. The methods for constructing these (by explicitly specifying structure constants or matrices) are described in Sections 100.2.

Well known simple finite Lie algebras can be more easily constructed by specifying the type. The classical, reductive, Lie algebras (A_n, B_n, C_n, D_n, E_6, E_7, E_8, F_4, G_2) and their twisted variants are described in Section 100.5.1, the Witt Lie algebras and its derivatives (of *Cartan-Type*) in Section 100.5.2, and the Melikian Lie algebras in Section 100.5.3. Those interested primarily in classical Lie algebras may want to skip to Section 100.5.1, which includes a number of examples to get started.

Elementary properties of these Lie algebras (bases, types, Weyl group, etc.) are described in Section 100.8. This section also contains information about isomorphism testing. Other properties (nilpotency, simplicity) are discussed in Section 100.10. Some further operations that only apply to matrix Lie algebras may be found in Section 100.13.

Constructors such as direct sums, subalgebras, centralisers, Cartan subalgebras, derived series, etc. are treated in Sections 100.7 and 100.9; construction of homomorphisms can be found in Section 100.14. The construction of elements of Lie algebras is described in Section 100.6, and operations on them in Section 100.11.

In addition to these Lie algebras MAGMA supports two other types of Lie algebras. Firstly finitely presented Lie algebras, as free Lie algebras modulo a set of relations, described in Section 100.3. Secondly Lie algebras generated by extremal elements, described in Section 100.4.

More specialistic functions for structure constant Lie algebras are described in Sections 100.12 (the natural module), 100.15 (automorphisms of classical-type Lie algebras), 100.16 (restrictable Lie algebras), and 100.17 (universal enveloping algebras).

MAGMA also provides databases and recognition procedures for small-dimensional solvable and nilpotent Lie algebras (described in Section 100.18), a database of semisimple subalgebras of simple Lie algebras (described in Section 100.19), and a database of nilpotent orbits in simple Lie algebras (described in Section 100.20).

Other chapters that may be of interest are Chapter 101 on Kac-Moody Lie algebras and Chapter 102 on Quantum Groups. Furthermore, Chapter 104 deals with representations of Lie algebras and groups of Lie type. Of particular importance is Section 104.3.1, dealing with the construction of representations of Lie algebras.

100.2 Constructors for Lie Algebras

The construction of a Lie algebra defined by structure constants is identical to that of a general structure constant algebra. Most constructors take two optional parameters: `Check` and `Rep`.

By default, the conditions for the algebra to be a Lie algebra are checked. If the user decides to omit this check, by setting the parameter `Check` to `false`, and the algebra is not actually Lie then functions in this section will fail or give incorrect answers.

The optional parameter `Rep` can be used to select the internal representation of the structure constants. The possible values for `Rep` are "`Dense`", "`Sparse`" and "`Partial`", with the default being "`Dense`". In the dense format, the n^3 structure constants are stored as n^2 vectors of length n. This is the best representation if most of the structure constants are non-zero. The sparse format, intended for use when most structure constants are zero, stores the positions and values of the non-zero structure constants. The partial format stores the vectors, but records for efficiency the positions of the non-zero structure constants.

`LieAlgebra< R, n | Q : parameters >`

`LieAlgebra< M | Q : parameters >`

`Check`	BOOLELT	*Default :* `true`
`Rep`	MONSTGELT	*Default :* "*Dense*"

This function creates the Lie structure constant algebra L over the free module $M = R^n$, with standard basis $\{e_1, e_2, \ldots, e_n\}$, and structure constants a_{ij}^k being given by the sequence Q. The sequence Q can be of any of the following three forms. Note that in all cases the actual ordering of the structure constants is the same: the only difference is that their partitioning into blocks varies.

(i) A sequence of n sequences of n sequences of length n. The j-th element of the i-th sequence is the sequence $[a_{ij}^1, \ldots, a_{ij}^n]$, or the element $(a_{ij}^1, \ldots, a_{ij}^n)$ of M, giving the coefficients of the product $e_i * e_j$.

(ii) A sequence of n^2 sequences of length n, or n^2 elements of M. Here the coefficients of $e_i * e_j$ are given by position $(i-1)n + j$ of Q.

(iii) A sequence of n^3 elements of the ring R. The sequence elements are the structure constants themselves, in the order $a_{11}^1, a_{11}^2, \ldots, a_{11}^n, a_{12}^1, a_{12}^2, \ldots, a_{nn}^n$. So a_{ij}^k lies in position $(i-1)n^2 + (j-1)n + k$ of Q.

`LieAlgebra< R, n | T :` *parameters* `>`

`Check`	BoolElt	*Default :* `true`
`Rep`	MonStgElt	*Default :* "*Dense*"

This function creates the Lie structure constant algebra L with standard basis $\{e_1, e_2, \ldots, e_n\}$ over the ring R. The sequence T contains quadruples $< i, j, k, a_{ij}^k >$ giving the non-zero structure constants. All other structure constants are defined to be 0.

`LieAlgebra< t | T :` *parameters* `>`

`Check`	BoolElt	*Default :* `true`
`Rep`	MonStgElt	*Default :* "*Dense*"

This function creates the Lie structure constant algebra L over the integers, with standard basis $\{e_1, e_2, \ldots, e_n\}$. The sequence T contains quadruples $< i, j, k, a_{ij}^k >$ (where the a_{ij}^k are integers) giving the non-zero structure constants. All other structure constants are defined to be 0. The argument t is a sequence of length n consisting of nonnegative integers giving the moduli of the basis elements. Thus let t_i denote the i-th element of t; then $t_i e_i = 0$. So if $t_i = 0$, then $ke_i \neq 0$ for all integers k.

`LieAlgebra(A)`

Given an associative structure-constant algebra A, create the Lie algebra L consisting of the elements in A with the induced Lie product $(x, y) \to x * y - y * x$. As a second value the map identifying the elements of L and A is returned.

`LieAlgebra(A)`

Given an associative matrix algebra A, create a structure-constant Lie algebra L isomorphic to A with the induced Lie product $(x, y) \to x * y - y * x$.

`AbelianLieAlgebra(R, n)`

`Rep` MonStgElt *Default : "Sparse"*

Create the abelian Lie algebra of dimension n over the ring R.

Example H100E1

We construct the Heisenberg Lie algebra, then a Lie algebra from an associative algebra, and finally a Lie algebra over the integers (also called a Lie ring).

```
> T:= [ <1,2,3,1>, <2,1,3,-1> ];
> LieAlgebra< Rationals(), 3 | T >;
Lie Algebra of dimension 3 with base ring Rational Field
> A:= Algebra( GF(27), GF(3) );
> LieAlgebra(A);
Lie Algebra of dimension 3 with base ring GF(3)
> T:= [ <1,2,2,2>, <2,1,2,2> ];
> t:= [0,4];
> K:= LieAlgebra< t | T : Rep:= "Dense" >; K;
Lie Algebra of dimension 2 with base ring Integer Ring
Column moduli: [0, 4]
> LowerCentralSeries( K );
[
    Lie Algebra of dimension 2 with base ring Integer Ring
    Column moduli: [0, 4],
    Lie Algebra of dimension 1 with base ring Integer Ring
    Column moduli: [2],
    Lie Algebra of dimension 0 with base ring Integer Ring
]
```

`ChangeBasis(L, B)`

`Rep` MonStgElt *Default : "Dense"*

Create a new Lie structure constant algebra L', isomorphic to L, by recomputing the structure constants with respect to the basis B. The basis B can be specified as a set or sequence of elements of L, a set or sequence of vectors, or a matrix. The second returned value is the isomorphism from L to L'.

As above, the optional parameter `Rep` can be used to select the internal representation of the structure constants. Note that the default is dense representation, regardless of the representation used by L.

`MatrixLieAlgebra(R, n)`

Given a ring R and an integer n, create the full Lie algebra of matrices of degree d over R.

`MatrixLieAlgebra(A)`

Given an associative matrix algebra A, create the matrix Lie algebra L consisting of the elements in A with the induced Lie product $(x, y) \to x * y - y * x$.

`Algebra(M)`

`LieAlgebra(M)`

Return a structure-constant Lie algebra isomorphic to the matrix Lie algebra M.

Example H100E2

We construct the subalgebra of the matrix Lie algebra of 2×2 matrices, consisting of upper triangular matrices.

```
> L:= MatrixLieAlgebra( Rationals(), 2 );
> a:= L!Matrix( [[1,0],[0,0]] );
> b:= L!Matrix( [[0,0],[1,0]] );
> c:= L!Matrix( [[0,0],[0,1]] );
> K:= sub< L | [ a, b, c ] >;
> Dimension(K);
3
> IsSolvable(K);
true
> IsNilpotent(K);
false
```

100.3 Finitely Presented Lie Algebras

A finitely presented Lie algebra is constructed as the quotient of a free Lie algebra on a finite number of generators. Denote the set of generators by $X = \{x_1, \ldots, x_n\}$. Let F denote the base ring. Then the free Lie algebra generated by the x_i over the ring F is denoted by $L_F(X)$. The free magma on X is the set of the x_i together with all bracketed expressions in the x_i, e.g., $((x_1, x_2), ((x_1, x_3), x_2)))$. The free Lie algebra $L_F(X)$ is spanned by $M(X)$. However, the elements of this set are not linearly independent. It is a nontrivial problem to describe a basis of the free Lie algebra. One of several possibilities is the well-known *Hall basis*. Currently MAGMA does not support calculations involving bases of the free Lie algebra, as they are of little use for our main problem: the construction of a basis and multiplication table for a finitely-presented Lie algebra.

It is convenient to define an ordering on the elements of $M(X)$. First of all, each generator is assigned a degree. Usually, the degree of all x_i is taken to be one, but it is also possible to assign different degrees. The degree of a bracket (a, b) is defined to be the sum of the degrees of a and b. Let m, m' be two elements of $M(X)$. Then define $m < m'$ if the degree of m is less than the degree of m'. If their degrees are equal, then define $m < m'$ if $m = x_i$ and $m' = (a', b')$, for some a', b' in $M(X)$. If both m and m' are generators of the same degree, so that $m = x_i$, $m' = x_j$, then define $m < m'$ if $i < j$. Finally, if both m and

m' are bracketed expressions, that is, $m = (a, b)$ and $m' = (a', b')$, then define $m < m'$ if $a < a'$ or $a = a'$ and $b < b'$.

In the free Lie algebra, the relations $(a, b) = -(b, a)$, and $(a, a) = 0$ hold. In MAGMA this is used to rewrite an arbitrary element as a linear combination of elements of the form (a, b) with $a < b$. If instead we were to work relative to a basis for $L_F(X)$, then the use of the Jacobi identity when rewriting elements can lead to rather large expressions. Thus, mathematically speaking, in MAGMA rather than work in the free Lie algebra, we actually work in the free nonassociative anticommutative algebra. However, as our main interest lies in finitely-presented Lie algebras, this is usually not a problem.

100.3.1 Construction of the Free Lie Algebra

`FreeLieAlgebra(F, n)`

Given a ring F and a positive integer n, this function creates the free n-generator Lie algebra over the ring F. The generators are ordered, with the first generator being the biggest in the ordering, and the last generator the smallest. The angle bracket notation can be used to assign names to the generators.

Example H100E3

The following statement creates the MAGMA object corresponding to the free Lie algebra on three generators over the field $\mathbf{F}_2$.

```
> L<a,b,c>:= FreeLieAlgebra(GF(2), 3);
```

100.3.2 Properties of the Free Lie Algebra

`Rank(L)`

The number of generators of the free Lie algebra L.

`CoefficientRing(L)`

`BaseRing(L)`

The coefficient ring of L.

100.3.3 Operations on Elements of the Free Lie Algebra

Once a free Lie algebra has been created the user can construct a bracketed expression (a, b), either by simply typing it literally as `(a,b)`, or by using the multiplication operator as in `a*b`. Recall that MAGMA rewrites elements so they are in the form (a, b) with $a < b$. On some occasions this can lead to the introduction of a minus sign. Also, if an element contains a subexpression of the form (a, a), it will be rewritten to 0.

We can multiply and add elements, and multiply them by scalars.

```
x + y
```

```
x - y
```

```
x * y
```

```
L ! 0
```

```
Zero(L)
```

The zero element of the free Lie algebra L.

Example H100E4

```
> L<z,y,x> := FreeLieAlgebra(Rationals(), 3);
> x*y;
(x, y)
> (x, y);
(x, y)
> ((x*y)*z);
-(z, (x, y))
> ((x, y), z);
-(z, (x, y))
> ((x, y),(y, x));
0
> 2*((x, y), z) - ((x, z), (y, z)) + 1/2*(x, (x, (y, z)));
-((x, z), (y, z)) + 1/2*(x, (x, (y, z))) - 2*(z, (x, y))
```

```
IsLeaf(m)
```

Given a monomial element m of the free Lie algebra L, return `true` if m is a generator and `false` otherwise. If the result is `true` then the second return value is an integer i such that m is `L.i`. If the result is `false` then $a, b \in L$ are also returned such that m is a multiple of (a, b).

Note that in the latter case m is not equal to (a, b), but merely equal to a scalar multiple of (a, b). See the example for a possible method of retrieving the appropriate scalar.

Example H100E5

```
> L<z,y,x>:= FreeLieAlgebra(Rationals(), 3);
> IsLeaf(x);
true 3
> m := 2*((x, y), z);
> m;
-2*(z, (x, y))
> il, a, b := IsLeaf(m);
> il, a, b;
false z
(x, y)
> m eq (a, b);
false
> m eq LeadingCoefficient(m)*(a,b);
true
```

100.3.4 Construction of a Finitely-Presented Lie Algebra

`LieAlgebra(R)`

Given a set or sequence R of elements of a free Lie algebra L, let I be the ideal of L generated by the elements of R. It is assumed that the quotient algebra $Q = L/I$ is finite dimensional. This function returns the structure constant Lie algebra K isomorphic to the quotient Q. If the quotient Q is infinite dimensional then the program will not terminate. (The question of determining whether the quotient is finite dimensional is known to be undecidable.) The function can be interrupted by pressing `Ctrl-C`. The elements of R are referred to as *relations*.

This function works if the base ring is either a field or equal to the ring of integers. In these two cases slightly different objects are returned.

If the base ring is a field then four values are returned:

(a) A structure constant algebra K isomorphic to the quotient Q;

(b) A sequence G comprising sequences of integers;

(c) A sequence B of elements of the free Lie algebra L;

(d) A map $f : B \times B \to L$.

The sequence B maps to a basis of the quotient algebra, so it is a basis of a complement of the ideal I in the free Lie algebra L. The elements of B are in one-to-one correspondence with the basis elements of K.

If all the relations of R are homogeneous (i.e., if they are linear combinations of elements of the same degree), then Q is graded. The sequence G contains information about the grading. It consists of sequences of length two. The first element of each subsequence is the degree of a homogeneous subspace H, while the second element is the dimension of H. The basis elements of K are ordered with respect to increasing

degree. So from G it is straightforward to read off the degree of each basis element. If the relations are not homogeneous then the sequence G is empty. Finally, f is a map that takes two elements from B as arguments, and returns their product (in L) modulo the ideal I. The algorithm used is described in [dG00], §7.4.

Secondly, in the case in which the base ring is the ring of integers, four values are returned:

(a) A structure constant algebra K isomorphic to the quotient Q;

(b) A sequence G comprising sequences of integers;

(c) A sequence B that is always empty;

(d) A map $f : K \to L$.

Here the structure constant algebra is defined over the ring of integers, so it may have torsion. The sequence G is nonempty only if the input relations are homogeneous in which case it contains the dimensions of the homogeneous components. The function f is a map that takes an element u of K and returns an element of the free algebra L that maps to u under the projection map (from the free algebra to the quotient).

Example H100E6

In this example we compute the subalgebra K of E_7 spanned by the positive root spaces.

```
> L<x7,x6,x5,x4,x3,x2,x1>:= FreeLieAlgebra(RationalField(), 7);
> pp:= { [1,3], [3,4], [2,4], [4,5], [5,6], [6,7] };
> R:= [ ];
> g:= [ L.i : i in [1..7] ];
> for i in [1..7] do
>     for j in [i+1..7] do
>         if [i,j] in pp then
>            a:= (g[i],(g[i],g[j]));
>            Append( ~R, a );
>            Append( ~R, (g[j],(g[j],g[i])) );
>         else
>            Append( ~R, (g[i],g[j]) );
>         end if;
>     end for;
> end for;
> R;
[
    -(x6, x7), -(x7, (x5, x7)), (x5, (x5, x7)), -(x4, x7),
    -(x3, x7), -(x2, x7), -(x1, x7), -(x5, x6), -(x6, (x4, x6)),
    (x4, (x4, x6)), -(x3, x6), -(x2, x6), -(x1, x6), -(x5, (x4, x5)),
    (x4, (x4, x5)), -(x3, x5), -(x2, x5), -(x1, x5), -(x4, (x3, x4)),
    (x3, (x3, x4)), -(x2, x4), -(x1, x4), -(x3, (x2, x3)), (x2, (x2, x3)),
    -(x1, x3), -(x2, (x1, x2)), (x1, (x1, x2))
]
> time K, G, B, f := LieAlgebra(R);
```

```
Time: 0.280
> K;
Lie Algebra of dimension 63 with base ring Rational Field
> #B;
63
> B[63];
(x7, (x5, (x4, (x3, (x2, (x1, (x6, (x4, (x3, (x2, (x5, (x4,
    (x3, (x6, (x4, (x5, x7))))))))))))))))
```

Example H100E7

In this example we construct a finitely presented Lie ring (i.e., Lie algebra over the integers).

```
> L<y,x>:= FreeLieAlgebra( Integers(), 2 );
> R:= [ x*(x*(x*y))-2*x*y, 2*y*(x*(x*y)), 3*y*(y*(x*y))-x*(x*y),
>  x*(y*(x*(y*(x*y)))) ];
> K,g,b,f:= LieAlgebra( R );
> K;
Lie Algebra of dimension 8 with base ring Integer Ring
Column moduli: [2, 2, 2, 8, 8, 8, 0, 0]
> f(K.4);
(y, (x, y))
> LowerCentralSeries( K );
[
    Lie Algebra of dimension 8 with base ring Integer Ring
    Column moduli: [2, 2, 2, 8, 8, 8, 0, 0],
    Lie Algebra of dimension 6 with base ring Integer Ring
    Column moduli: [2, 2, 2, 8, 8, 8],
    Lie Algebra of dimension 6 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4, 8, 8],
    Lie Algebra of dimension 6 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4, 4, 8],
    Lie Algebra of dimension 6 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4, 4, 4],
    Lie Algebra of dimension 5 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4, 4],
    Lie Algebra of dimension 4 with base ring Integer Ring
    Column moduli: [2, 2, 2, 4],
    Lie Algebra of dimension 3 with base ring Integer Ring
    Column moduli: [2, 2, 2],
    Lie Algebra of dimension 2 with base ring Integer Ring
    Column moduli: [2, 2],
    Lie Algebra of dimension 1 with base ring Integer Ring
    Column moduli: [2],
    Lie Algebra of dimension 0 with base ring Integer Ring
]
```

`quo< L | R >`

This function is similar to the function LieAlgebra in that it constructs a structure constant Lie algebra K isomorphic to the quotient L/I, where I is the ideal of L generated by the elements (relations) of the sequence R. In addition to K, an invertible map from L to K is returned.

Example H100E8

In this example we demonstrate the use of the quotient constructor for finitely presented Lie algebras.

```
> L<x,y> := FreeLieAlgebra(Rationals(), 2);
> R := [ x*(x*y)-2*x, y*(y*x)-2*y ];
> K, phi := quo<L | R>;
> K;
Lie Algebra of dimension 3 with base ring Rational Field
> SemisimpleType(K);
A1
> [ b @@ phi : b in Basis(K) ];
[
   y,
   x,
   (y, x)
]
> phi(x*y);
( 0  0 -1)
```

`NilpotentQuotient(R, d)`

Given a set or sequence R of elements of a free Lie algebra L, let I be the ideal of L generated by the elements of R. Let d be a positive integer or `Infinity()`. This function constructs the class d nilpotent quotient of the Lie algebra L/I, a finite dimensional algebra. The function returns the same values as LieAlgebra.

This function is similar to the function LieAlgebra except that the quotient is constructed in the free nilpotent Lie algebra of class d. All elements of degree strictly larger than `d` will be added to the ideal, so the quotient will be finite-dimensional and nilpotent of class at most `d`.

Example H100E9

In this example, we compute a nilpotent quotient.

```
> L<y,x> := FreeLieAlgebra(Rationals(), 2);
> R := [(x, (x, (x, y))) - (y, (y, (x, y)))];
> time K, G, B, f := NilpotentQuotient(R, 10);
Time: 0.040
> K;
```

```
Lie Algebra of dimension 109 with base ring Rational Field
> #B;
109
> B[100];
(y, (x, (x, (y, (x, (y, (x, (x, (x, y)))))))))
> G;
[
    [ 1, 2 ],
    [ 2, 1 ],
    [ 3, 2 ],
    [ 4, 2 ],
    [ 5, 4 ],
    [ 6, 5 ],
    [ 7, 10 ],
    [ 8, 15 ],
    [ 9, 26 ],
    [ 10, 42 ]
]
> f(B[3], B[13]);
-(y, (x, (x, (x, (x, (y, (x, y)))))))) + (x, (y, (x, (x, (x, (y, (x, y)))))))
```

100.3.5 Homomorphisms of the Free Lie Algebra

`hom< L -> M | Q >`

Given a free Lie algebra L of dimension n over R and either a Lie algebra M over R or a module M over R, construct the homomorphism from L to M specified by Q. The sequence Q must have length `Rank(L)` and be of the form $[m_1, \ldots, m_n]$ ($m_i \in M$) indicating that the i-th generator of L maps to m_i.

Note that this is in general only a module homomorphism, and it is not checked whether it is an algebra homomorphism.

Example H100E10

We construct the Lie algebra of type A_1 as quotient of a free Lie algebra, using homomorphisms between a free Lie algebra and a structure constant Lie algebra. First, we construct the free Lie algebra and a structure constant Lie algebra of type A_1. The elements of a Chevalley basis are obtained by a call to `ChevalleyBasis`.

```
> L<e,f> := FreeLieAlgebra(Rationals(), 2);
> M := LieAlgebra("A1", Rationals() : Isogeny := "SC");
> pos, neg, cart := ChevalleyBasis(M);
> pos, neg, cart;
[ (0 0 1) ]
[ (1 0 0) ]
```

```
[ (0 1 0) ]
```

Next, we construct a homomorphism from L to M that sends e to the positive root element, and f to the negative root element. We construct a map from M to L that sends the positive root to e, the negative root to f, and the Cartan element to $-(e, f)$.

```
> phi := hom<L -> M | [ pos[1], neg[1] ]>;
> phi(e), phi(f), phi(e*f);
(0 0 1) (1 0 0) ( 0 -1  0)
> imgs := [ L | f, (f,e), e];
> psi := map<M -> L | x :-> &+[ x[i]*imgs[i] : i in [1..3] ]>;
> psi(cart[1]);
-(f, e)
> psi(phi((e,(e,f))));
-2*e
> assert forall{b : b in Basis(M) | phi(psi(b)) eq b };
```

Finally, we create a sequence of relations showing that the maps `phi` and `psi` are each others inverses for a small set of elements of L. We then compute the quotient of the free Lie algebra with respect to these relations.

```
> R := { x - psi(phi(x)) : x in {e, f, (e,f), (e,(e,f)), (f,(f,e))} };
> L2 := quo<L | R>;
> L2;
Lie Algebra of dimension 3 with base ring Rational Field
> SemisimpleType(L2);
A1
```

100.4 Lie Algebras Generated by Extremal Elements

A non-zero element x of a Lie algebra L over the field K is extremal if $[x, [x, y]] \in Kx$ for all $y \in L$. If x is extremal, the existence of a linear map $f_x : L \to K$ such that $[x, [x, y]] = f_x(y)x$ for all $y \in L$ immediately follows from linearity of $[\cdot, \cdot]$.

In this section we describe functions for computing with Lie algebras generated by such extremal elements. For a simple connected undirected finite graph Γ we consider an algebraic variety X over K whose K-points parametrize Lie algebras generated by extremal elements. Here the generators of the Lie algebras correspond to the vertices of the graph, and we prescribe commutation relations corresponding to the nonedges of Γ.

Details of the setup may be found in [Roo11]; we describe the essential ingredients here.

Assume that Γ is a connected undirected finite graph with n vertices, without loops or multiple bonds, and that K is a field of characteristic distinct from 2. We let Π be the vertex set of Γ and denote adjacency of two vertices $x, y \in \Pi$ by $i \sim j$.

We denote by $F(K, \Gamma)$ the quotient of the free Lie algebra over K generated by Π modulo the relations $[x, y] = 0$ for all $x, y \in \Pi$ with $x \not\sim y$. We write F^* for the space of all K-linear functions on F. For every $f \in (F^*)^\Pi$ we denote by $L(K, \Gamma, f)$ (often abbreviated to

$L(f)$) the quotient of $F(K,\Gamma)$ by the ideal $I(f)$ generated by the infinitely many elements $[x,[x,y]] - f_x(y)x$ for $x \in \Pi$, $y \in F$.

By construction $L(f)$ is a Lie algebra generated by $|\Pi| = n$ extremal elements, the extremal generators corresponding to the vertices of Γ and commuting whenever they are not adjacent. The element $f_x \in F^*$ is a parameter expressing the extremality of $x \in \Pi$.

In the Lie algebra $L(0)$ the elements of Π map to sandwich elements. This algebra is finite-dimensional, by [ZK90] this Lie algebra is finite-dimensional; for general $f \in (F^*)^\Pi$ we have $\dim(L(f)) \leq \dim(L(0))$ by [CSUW01, Lemma 4.3]. It is therefore natural to focus on the Lie algebras $L(f)$ of maximal possible dimension, i.e., those of dimension $\dim(L(0))$. We define the set $X := \{f \in (F^*)^\Pi \mid \dim(L(f)) = \dim(L(0))\}$, the parameter space for all maximal-dimensional Lie algebras of the form $L(f)$.

The functions currently implemented in MAGMA allow computation of X and $L(f)$, for any f, for the cases where X is an affine space (which is unproven, but true in all currently known cases).

Lie algebras generated by extremal elements are of type `AlgLieExtr`. The verbose flag `"AlgLieExtr"` may be set between 1 and 5 to show details and progress of the various computations.

100.4.1 Constructing Lie Algebras Generated by Extremal Elements

`ExtremalLieAlgebra(K, n)`

`CommGens`	SEQENUM	*Default :* []
`HeisenbergPairs`	SEQENUM	*Default :* []

Construct the Lie algebra over the field K generated by n extremal elements. The characteristic of K must be distinct from 2.

The optional argument `CommGens` contains pairs of integers (i,j), with $1 \leq i,j \leq n$, describing that generators x_i and x_j commute, i.e., $[x_i,x_j] = 0$.

The optional argument `HeisenbergPairs` contains pairs of integers (i,j), with $1 \leq i \leq n$ and $1 \leq j \leq \dim(L(0))$, describing that $f_{x_i}(b_j)$ should be taken equal to 0. (Note that if it is required to have $j > n$ it would be necessary to have prior knowledge about the basis of $L(0)$).

`ExtremalLieAlgebra(K, G)`

`HeisenbergPairs`	SEQENUM	*Default :* []

Construct the Lie algebra over the field K whose extremal generators are described by the graph G, i.e., with $|V(G)|$ generators, and x_i and x_j commute whenever vertices x_i and x_j of G are not adjacent.

See `ExtremalLieAlgebra` above for a description of the optional argument `HeisenbergPairs`.

100.4.2 Properties of Lie Algebras Generated by Extremal Elements

`NumberOfGenerators(L)`

The number of generators of L.

`CoefficientRing(L)`

`BaseRing(L)`

The coefficient ring of L. Immediately after construction, this is equal to the field K provided as argument to `ExtremalLieAlgebra`. However, after the multiplication table has been computed (see below), the coefficient ring would in general be a multivariate polynomial ring over K describing the parameter space.

`CommutatorGraph(L)`

The graph describing the extremal generators of L and their commutator relations.

Example H100E11

We construct a Lie algebra generated by 4 extremal elements in two different manners.

```
> QQ := Rationals();
> L := ExtremalLieAlgebra(QQ, BipartiteGraph(2,2));
> Ngens(L), CoefficientRing(L);
4 Rational Field
> G := CommutatorGraph(L); G;
Graph
Vertex  Neighbours
1       3 4 ;
2       3 4 ;
3       1 2 ;
4       1 2 ;
> L := ExtremalLieAlgebra(QQ, 4 : CommGens := [<1,2>,<3,4>]);
> Ngens(L), CoefficientRing(L);
4 Rational Field
> G := CommutatorGraph(L); G;
Graph
Vertex  Neighbours
1       3 4 ;
2       3 4 ;
3       1 2 ;
4       1 2 ;
```

`Basis(L)`

Compute a monomial basis for $L(0)$ (this is also a monomial basis for $L(f)$ for any $f \in X$; see the introduction of Section 100.4).

The first return value is a sequence consisting of monomials of the free Lie algebra over K with n generators, where K is the coefficient ring of L and n is the number of generators. The second return value is a sequence consisting of functions c. Each of these functions may be applied to a sequence of generators and a composition function. These may be used to construct the basis elements in other environments.

The algorithm used in this function is due to W. de Graaf.

`ZBasis(L)`

For L a Lie algebra generated by extremal elements over the field of rational numbers, compute a basis of the corresponding Lie ring over the integers.

This function returns three sequences B, T, C, respectively, describing bases for $L(0)$ over **any** field K. B is a not necessarily monomial basis, with torsion described by T. It is such that if $T[i]$ is nonzero, m say, then $B[i]$ is zero unless the characteristic of K divides m.

The third sequence, C, is a sequence of monomials that linearly span $L(0)$ over any field K. Note, however, that if T contains nonzero elements, then C would in general contain superfluous elements and therefore not be a basis.

The algorithm used in this function is due to W. de Graaf. The only currently known case with nontrivial torsion is for $\Gamma(L) = K_5$.

`Dimension(L)`

The dimension of $L(0)$. This value is computed via a basis computation, so potentially quite time-consuming.

Example H100E12

We continue the previous example H100E11 and demonstrate the computation of a basis of $L(0)$.

```
> B, C := Basis(L);
> B;
[
    $.1,
    $.2,
    $.3,
    $.4,
    ($.4, $.2),
    ($.4, $.1),
    ($.3, $.2),
    ($.3, $.1),
    ($.4, ($.3, $.2)),
    ($.4, ($.3, $.1)),
    ($.2, ($.4, $.1)),
    ($.2, ($.3, $.1)),
```

```
    ($.4, ($.2, ($.3, $.1))),
    ($.3, ($.2, ($.4, $.1))),
    ($.2, ($.4, ($.3, $.1)))
]
> [ c(["x","y","z","u"], func<i,j|i cat j>) : c in C ];
[ x, y, z, u, uy, ux, zy, zx, uzy, uzx, yux, yzx, uyzx, zyux, yuzx ]
> A := FreeAlgebra(Rationals(), 4);
> [ c([A.1,A.2,A.3,A.4], func<x,y|x*y>) : c in C ];
[
    $.1,
    $.2,
    $.3,
    $.4,
    $.4*$.2,
    $.4*$.1,
    $.3*$.2,
    $.3*$.1,
    $.4*$.3*$.2,
    $.4*$.3*$.1,
    $.2*$.4*$.1,
    $.2*$.3*$.1,
    $.4*$.2*$.3*$.1,
    $.3*$.2*$.4*$.1,
    $.2*$.4*$.3*$.1
]
> #B, #C, Dimension(L);
15 15 15
```

`MultiplicationTable(~L)`

`HowMuch`	MonStgElt	*Default :* "*Auto*"
`MemLimit`	RngIntElt	*Default :* ∞
`FullJacobi`	BoolElt	*Default :* `false`

Force computation of a general multiplication table for L, i.e., one that may be used for constructing $L(f)$ for any $f \in X$ (see the introduction to this section 100.4). This computation is necessary for constructing instances as described in Section 100.4.3, but it will be done automatically if needed. Data about the variety X is computed concurrently and stored internally; see Section 100.4.4 for the relevant functions in accessing that information.

The optional parameters may be used to influence the computation, although the defaults should generally work well. `HowMuch` may be set to "Auto" (the default), "Top" or "Full" and prescribes whether only the first `Ngens(L)` rows of the multiplication table are computed ("Top"), or all entries ("Full"). If set to "Auto" some fraction of the multiplication table is computed depending on the dimension of L and the other parameters.

`MemLimit` may be set to a positive integer m, and if given MAGMA will attempt to limit its memory usage to m MB, by limiting the portion of the multiplication table that is being computed.

`FullJacobi` may be set to `true` in order to force checking the Jacobi identity for all basis elements, thus providing more certainty with regards to the information about the parameter space X. Note that even if this parameter is set to `true` a heuristic (Monte-Carlo) method is used, as considering all $\dim(L(0))^3$ triples quickly becomes infeasible as the dimension grows.

The verbose flag `"AlgLieExtr"` may be set to 3 or more to obtain some information about the default choices MAGMA makes with regards to these parameters.

`MultiplicationTable(L)`

`Rep`	MONSTGELT	*Default :* "*Auto*"
`Check`	BOOLELT	*Default :* `true`

A general multiplication table for L.

If `Rep` is set to "Dense" it will be returned as a sequence of sequences of vectors over `CoefficientRing(L)`. If `Rep` is set to "Sparse" it will be returned as a sequence of 4-tuples. If `Rep` is set to "Auto" a choice between these representations is made depending on $\dim(L)$. Both these representations may be used on the right hand side of the `LieAlgebra` constructor.

The optional parameter `Check` controls whether the Jacobi identity is verified for all triples (if `true` it will actually be checked for all $\dim(L(0))^3$ triples, as opposed to the behaviour of the procedural version, `MultiplicationTable(~L)`, described above).

Note that this function is impractical in terms of CPU time and memory usage once $\dim(L)$ exceeds approximately 50. In such cases, the Lie algebra is more easily studied using the functions described in Section 100.4.3.

Example H100E13

We construct the generic Lie algebra generated by 3 extremal elements and construct a structure constant Lie algebra using the multiplication table.

```
> L := ExtremalLieAlgebra(Rationals(), 3);
> L:Maximal;
Lie algebra generated by 3 extremal elements, defined over Rational
Field
> MultiplicationTable(~L);
> L:Maximal;
Lie algebra generated by 3 extremal elements, originally defined over
Rational Field
  Now living over Polynomial ring of rank 4 over Rational Field
  Dimension: 8
  Picked 4 f-values:
    f(2, [1]) = f21
    f(3, [1]) = f31
```

```
    f(3, [2]) = f32
    f(1, [32]) = f132
> Dimension(L);
8
> MT := MultiplicationTable(L);
> MT[4][8];
(0   -1/2*f31*f32   0   -1/2*f132   0   0   0   1/2*f32)
> M := LieAlgebra<CoefficientRing(L), 8 | MT>;
> M;
Lie Algebra of dimension 8 with base ring Polynomial ring of rank 4
over Rational Field
> M.4*M.8;
(0   -1/2*f31*f32   0   -1/2*f132   0   0   0   1/2*f32)
> M.1*(M.1*M.2);
(f21   0   0   0   0   0   0   0)
```

100.4.3 Instances of Lie Algebras Generated by Extremal Elements

`Instance(L)`

`Rep`	MonStgElt	*Default :* "*Auto*"
`Check`	BoolElt	*Default :* `true`

The Lie algebra $L(f)$ for general f. The Lie algebra returned will in general be defined over a multivariate polynomial ring.

This function is identical to `MultiplicationTable`, except that it returns a Lie algebra rather than a multiplication table. Please refer to that function for information on the optional arguments `Rep` and `Check`. Note that this function also is impractical in terms of CPU time and memory usage once $\dim(L)$ exceeds approximately 50. In such cases, the Lie algebra is more easily studied by constructing particular instances of $L(f)$ individually, as described below.

`Instance(L, Q)`

`Rep`	MonStgElt	*Default :* "*Auto*"
`Check`	BoolElt	*Default :* `true`

Construct $L(f)$ where the i-th free parameter of X is set to Q[i]. Consult `L:Maximal` or `FreefValues` to obtain information about the free parameters. The coefficient ring of the Lie algebra M returned will be equal to `Universe(Q)`. As a second return value, an invertible map from M to the free Lie algebra of rank `Ngens(L)` is returned.

The optional argument `Rep` may be "Auto", "Dense" or "Sparse" (refer to the documentation at `MultiplicationTable` for more information). `Check` may be set to `true` or `false` and determines whether the Jacobi identity is checked on the Lie algebra returned.

Example H100E14

We construct the generic Lie algebra generated by 3 extremal elements and study one of its instances.

```
> L := ExtremalLieAlgebra(Rationals(), 3);
> MultiplicationTable(~L);
> L:Maximal;
Lie algebra generated by 3 extremal elements, originally defined over
Rational Field
  Now living over Polynomial ring of rank 4 over Rational Field
  Dimension: 8
  Picked 4 f-values:
    f(2, [1]) = f21
    f(3, [1]) = f31
    f(3, [2]) = f32
    f(1, [32]) = f132
 > M := Instance(L); M;
Lie Algebra of dimension 8 with base ring Polynomial ring of rank 4
over Rational Field
> M.1*(M.1*M.2);
(f21   0   0   0   0   0   0   0)
```

So in the most general case, $[x_1, [x_1, x_2]] = f_{x_2}(x_1)x_1$. Next, we consider an instance where we set $f_{x_2}(x_1) = 1/7$, $f_{x_3}(x_1) = 1/5$, $f_{x_3}(x_2) = 1/3$ and $f_{x_1}([x_3, x_2]) = 1$.

```
> N, phi := Instance(L, [Rationals()|1/7,1/5,1/3,1]);
> N;
Lie Algebra of dimension 8 with base ring Rational Field
> SemisimpleType(N);
A2
> N.1*(N.1*N.2);
(1/7   0   0   0   0   0   0   0)
> y := phi(N.2); z := phi(N.3);
> Parent(y):Minimal;
Free Lie algebra of rank 3 over Rational Field
> (y,(y,z));
-($.2, ($.3, $.2))
> (y,(y,z)) @@ phi;
(  0 1/3   0   0   0   0   0   0)
> (y,(y,z)) @@ phi @ phi;
1/3*$.2
```

100.4.4 Studying the Parameter Space

`FreefValues(L)`

The values $f_x(b)$ generating the parameter space X (see the introduction to this section 100.4 for details). This function returns two sequences: the first of the $f_x(b)$ as elements of `CoefficientRing(L)` and the second of the pairs (x, b) as two-tuples of integers.

`fValue(L, x, b)`

The value $f_x(b)$ as an element of `CoefficientRing(L)`.

`fValueProof(L, x, b)`

Print a proof of correctness for the value $f_x(b)$.

Example H100E15

We consider the generic Lie algebra generated by 4 extremal elements.

```
> L := ExtremalLieAlgebra(Rationals(), 4);
> vals, pairs := FreefValues(L);
> vals;
[
    f21,
    f31,
    f41,
    f32,
    f42,
    f43,
    f143,
    f243,
    f142,
    f132,
    f1432,
    f2431
]
> #vals;
12
> pairs;
[ <2, 1>, <3, 1>, <4, 1>, <3, 2>, <4, 2>, <4, 3>, <1, 5>, <2, 5>, <1,
6>, <1, 8>, <1, 11>, <2, 12> ]
```

This shows that $\dim(X) = 12$. We compute some values $f_x(b)$.

```
> fValue(L, 1, 5);
f143
> fValue(L, 4, 17);
-f41*f42
> fValueProof(L, 4, 17);
f(4, [241]) -> -f(4,[2])*f(4,[1]) {f(x,[y,[x,N]]) = -f(x,y)f(x,N) by
```

```
  assoc. of f and anti-comm. of L}
    f(4, [2]) = f42 {Free}
    f(4, [1]) = f41 {Free}
= -f41*f42
```

`DimensionsEstimate(L, g)`

`NumSamples`	RngIntElt	*Default :* ∞
`Check`	BoolElt	*Default :* `true`
`Rep`	MonStgElt	*Default :* "*Auto*"
Verbose	`AlgLieExtr`	*Maximum :* 10

Estimate the dimensions of the subvarieties of the parameter space X of L giving rise to irreducible Lie algebra modules of different dimensions.

This procedure repeatedly (exactly `NumSamples` times) invokes `Instance(L, g())` to produce a Lie algebra M. The composition series of M are computed, and the dimension e of its simple factor is stored. Then, for each of these e encountered, the dimension of the subvariety (inside the algebraic variety X) that contains Lie algebras whose top factor has dimension e is estimated using the dimension d of the full f-variety. (Here d is taken to be the number of free f-values computed; see `FreefValues`).

If the verbose flag `"AlgLieExtr"` is set 3 or more, then after each step the estimate is printed as a sequence of triples (e, n, s): n is the number of times dimension e was encountered, and s the estimate for the dimension of the subvariety.

Upon finishing (which will only happen if `NumSamples` is set to some finite number) that sequence of triples is returned. The second return value is a multiset containing the dimensions encountered in the search.

Note that this procedure assumes that X itself is an affine variety (which has been proved if `CommutatorGraph(L)` is a connected simply laced Dynkin diagram of finite or affine type) and that `g` produces uniformly random elements of X. If either of these two is not the case, the estimates produced are likely wrong. Moreover, `g` must produce sequences of elements of a finite field.

The optional argument `Rep` may be "Auto", "Dense" or "Sparse" (refer to the documentation at `MultiplicationTable` for more information). `Check` may be set to `true` or `false` and determines whether the Jacobi identity is checked on the Lie algebras constructed.

`InstancesForDimensions(L, g, D)`

`Check` BoolElt *Default :* `true`

For each $d \in D$ attempt to find an instance of L whose simple factor has dimension d, by repeatedly invoking `Instance(L, g())`. The result is returned in the form of an associative array A such that, for all $d \in D$, $A[d]$ is a triple (v, M, ϕ) where v is such that `Instance(L, v)` is M, and ϕ is an invertible map from M to the free Lie algebra.

See DimensionsEstimate for the required properties of `g`. The optional parameter `Check` may be set to `true` or `false` and determines whether the Jacobi identity is checked on the Lie algebras constructed.

Example H100E16

We consider the generic Lie algebra generated by 3 extremal elements.

```
> L := ExtremalLieAlgebra(Rationals(), 3);
> FreefValues(L);
[
    f21,
    f31,
    f32,
    f132
]
[ <2, 1>, <3, 1>, <3, 2>, <1, 4> ]
```

So $\dim(X) = 4$. We create a function `g` used to construct random instances of L over GF(5).

```
> g := func< | [ Random(GF(5)) : i in [1..4] ]>;
> M := Instance(L, g()); M;
Lie Algebra of dimension 8 with base ring GF(5)
> SemisimpleType(M);
A2
```

So in this case `g()` yielded a Lie algebra of type A_2. We use `g` to obtain information about X, using 500 random instances.

```
> DimensionsEstimate(L, g : NumSamples := 500);
[ <3, 121, "3.12">, <8, 379, "3.83"> ]
{* 3^^121, 8^^379 *}
```

This shows that 379 instances were found where M was simple of dimension 8, and 121 cases where M had a simple factor of dimension 3. Using this result one might conjecture that there is a codimension 1 subspace of X with Lie algebras whose simple factor has dimension 3.

```
> A := InstancesForDimensions(L, g, {3,8} : Check := false);
> A[3];
<[ 2, 1, 4, 4 ], Lie Algebra of dimension 8 with base ring GF(5),
Mapping from: Lie Algebra of dimension 8 with base ring GF(5) to Free
Lie algebra of rank 3 over GF(5) given by a rule>
> M := A[3][2]; MM := M/SolvableRadical(M); MM;
```

```
Lie Algebra of dimension 3 with base ring GF(5)
> SemisimpleType(MM);
A1
> M := A[8][2]; IsSimple(M);
true
> SemisimpleType(M);
A2
```

100.5 Families of Lie Algebras

The radical of a Lie algebra is the maximal soluble ideal. A Lie algebra is called *reductive* if its radical is equal to its centre, and *semisimple* if its radical is trivial. A Lie algebra is *almost reductive (resp. simple, semisimple)* if the corresponding group of Lie type is reductive (resp. simple, semisimple). Note that these concepts are equivalent if the field has characteristic zero.

The commands in this section construct almost reductive Lie algebras over an arbitrary field. Such Lie algebras have a corresponding root datum. The matrix versions of these commands give the standard matrix representation, which is the smallest degree representation (with a few exceptions for small characteristic fields).

100.5.1 Almost Reductive Lie Algebras

The intrinsics `LieAlgebra` and `MatrixLieAlgebra` described below take as first argument an object which describes the type of the reductive Lie algebra to be constructed. Specifically, it may be one of the five following types:

(a) A string describing the Cartan type;

(b) A root datum (see Chapter 97);

(c) A crystallographic root system (see Chapter 96);

(d) A Dynkin diagraph (see Section 95.5);

(e) A crystallographic Cartan matrix C (see Section 95.4).

In the cases (a), (d), and (e) these intrinsics take an optional argument `Isogeny`. See Section 103.2 for the possible values of this flag.

`LieAlgebra(T, k)`

`Isogeny` . *Default :* "*Ad*"

Construct the reductive Lie algebra of type T over the ring k.

`MatrixLieAlgebra(T, k)`

`Isogeny` . *Default :* "*Ad*"

Construct the reductive matrix Lie algebra of type T over the ring k.

Example H100E17

We construct some (semi)simple Lie algebras.

```
> LieAlgebra("D7", RationalField());
Lie Algebra of dimension 91 with base ring Rational Field
> LieAlgebra("G2", GF(5));
Lie Algebra of dimension 14 with base ring GF(5)
> L := LieAlgebra( "G2 B3", Rationals() );
> L;
Lie Algebra of dimension 35 with base ring Rational Field
> DirectSumDecomposition(L);
[
    Lie Algebra of dimension 14 with base ring Rational Field,
    Lie Algebra of dimension 21 with base ring Rational Field
]
> LieAlgebra( "E8", GF(2) );
Lie Algebra of dimension 248 with base ring GF(2)
```

Example H100E18

This example demonstrates the use of the `Isogeny` option. Over a field of characteristic zero, this option only effects the basis used. In characteristic p, it sometimes effects the isomorphism type of the algebra. For type A_n with $p|(n+1)$, the default Isogeny is "Ad" (adjoint), which gives an algebra with nontrivial derived subalgebra but no centre:

```
> L := LieAlgebra("A4", GF(5));
> Dimension(L);
24
> Dimension(L*L);
23
> Dimension(Centre(L));
0
```

If you take Isogeny to be "SC" (simply connected), you get a perfect algebra with a nontrivial centre.

```
> L := LieAlgebra("A4", GF(5) : Isogeny:="SC");
> Dimension(L);
24
> Dimension(L*L);
24
> Dimension(Centre(L));
1
```

If $p^2|(n+1)$ there is an intermediate isogeny type which has both a centre and a nontrivial derived algebra:

```
> L := LieAlgebra("A24", GF(5) : Isogeny:=5);
> Dimension(L);
624
```

```
> Dimension(L*L);
623
> Dimension(Centre(L));
1
```

Similar results can be obtained by constructing the Lie algebra from a root datum. This kind of phenomenon happens whenever the characteristic divides the order of the fundamental group of your root datum. See [Hog82] for more details.

```
> R := RootDatum("E6");
> #FundamentalGroup(R);
3
> L := LieAlgebra(R,GF(3));
> L;
Lie Algebra of dimension 78 with base ring GF(3)
> L*L;
Lie Algebra of dimension 77 with base ring GF(3)
```

`LieAlgebra(N, k, p)`

`LieAlgebra(R, k, p)`

The twisted (almost) semisimple Lie algebra over the finite field k with Cartan type N given as a string or root datum R, with twist given by the permutation p. The twist should either be a permutation of the indices of the simple roots, or of the indices of all roots.

`TwistedLieAlgebra(R, k)`

Given a twisted root datum R and a finite field k, construct the twisted Lie algebra $L = R(k)$.

This variant has 5 return values. First, the twisted Lie algebra L. Second, a homomorphism ϕ from L into the split Lie algebra L' (over a suitable field extension of k); Third, L'; Fourth, a split toral subalgebra H of L, and, fifth, a split toral subalgebra H' of L', such that $\phi(H) \subseteq H'$.

See also TwistedBasis.

Example H100E19

We construct two twisted Lie algebras.

```
> DynkinDiagram("E6");
E6    1 - 3 - 4 - 5 - 6
              |
              2
> LieAlgebra( "E6", GF(5), Sym(6)!(1,6)(3,5) );
Lie Algebra of dimension 78 with base ring GF(5)
> Rt := TwistedRootDatum(RootDatum("D4") : Twist := 3);
> k := GF(7);
```

```
> L, phi, Lp, H, Hp := TwistedLieAlgebra(Rt, k);
> L;
Lie Algebra of dimension 28 with base ring GF(7)
> Lp;
Lie Algebra of dimension 28 with base ring GF(7^3)
> phi(L.3);
(0 0 ksi^49 ksi^7 ksi 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> IsSplitToralSubalgebra(L, H);
true
> IsSplitToralSubalgebra(Lp, Hp);
true
> forall{b : b in Basis(H) | phi(b) in Hp};
true
```

100.5.2 Cartan-Type Lie Algebras

Simple Lie algebras over fields of characteristic 0 have been classified and are precisely the twisted forms of Lie algebras of types $A_l, B_l, C_l, D_l, E_6, E_7, E_8, F_4$ and G_2 (see previous Subsection). Over fields of finite characteristic p, the analogues of these algebras are called *classical-type* (including the exceptional algebras). Over such fields there are other simple Lie algebras, the first of them found by Witt sometimes before 1937. For $p \geq 7$, the only non-classical simple Lie algebras are the Lie algebras of *Cartan-type*, which we discuss in this section. For $p = 5$, one further class of simple Lie algebras occurs: *Melikian* algebras, which are discussed in the next section. In characteristic 2 and 3, the classification of simple Lie algebras is not yet complete.

Cartan-type Lie algebras are non-classical Lie algebras which arise from infinite dimensional algebras of differential operators over $\mathbf{C}$:

- (generalised) Witt algebras,
- special and conformal special Lie algebras,
- Hamiltonian and conformal Hamiltonian Lie algebras,
- and contact Lie algebras.

The notation and the description of these Lie algebras closely follow Strade and Farnsteiner [Str04] and [SF88]. Where the notation of the two books differs, we follow [Str04].

Let F be a finite field of characteristic $p > 0$ and m a positive integer. We refer for the definition of $O(m)$ and $x^{(a)}$ to [Str04, 2.1]. The basis of $O(m)$ is $\{x^{(a)} | 0 \leq a, a \in \mathrm{N}^m\}$.

Let n be a sequence of positive integers of length m and set $N := \sum_{i=1}^{m} n_i$. Define

$$O(m, n) := \langle x^{(a)} | 0 \leq a_i < p^{n_i} \rangle$$

For $i = 1, \ldots, m$ denote by ∂_i the derivation of $O(m)$ defined by

$$\partial_i(x_j^{(r)}) = \delta_{i,j} x_j^{(r-1)}.$$

Now define

$$W(m,n) := \sum_{i=1}^{m} O(m)\partial_i.$$

The algebra $W(m,n)$ is the *Witt* algebra and has dimension mp^N over F. In particular, $W(1,[1])$ is the standard p-dimensional Witt algebra.

The Witt algebra $W(m,n)$ is simple unless $p=2$ and $m=1$ ([SF88, 4.2.4(1)]) and is restrictable if and only if $n=[1,\ldots,1]$ ([SF88, 4.2.4(2)]).

Further define

$$\Omega^0(m,n) := O(m,n),$$

$$\Omega^1(m,n) := \mathrm{Hom}_{O(m,n)}(W(m,n), O(m,n)),$$

$$\Omega^r(m,n) := \bigwedge^r \Omega^1(m,n),$$

$$\Omega(m,n) := \bigoplus \Omega^r(m,n).$$

Let $m \geq 2$ and $\omega_S = dx_1 \wedge \ldots \wedge dx_m$. Define the following subalgebras of $W(m,n)$:

$$S(m,n) := \{D \in W(m,n) | D(\omega_S) = 0\},$$

$$CS(m,n) := \{D \in W(m,n) | D(\omega_S) \in F\omega_S\}.$$

The algebra $S(m,n)$ is the *special* and $CS(m,n)$ is the *conformal special* Lie algebra. The dimension of $S(m,n)$ over F is $(m-1)p^N + 1$ and the dimension of $CS(m,n)$ is $\dim S(m,n) + 1$.

Suppose $m \geq 3$. Then the algebra $S(m,n)^{(1)}$ is simple ([SF88, 4.3.5(1)]) and is restrictable if and only if $n=[1,\ldots,1]$ ([SF88, 4.3.5(2)]).

Let $p>2$, $m=2r\geq 2$ and let $\omega_H = \sum_{i=1}^{r} dx_i \wedge dx_{i+r}$. Define the following subalgebras of $W(m,n)$:

$$H(m,n) := \{D \in W(m,n) | D(\omega_H) = 0\},$$

$$CH(m,n) := \{D \in W(m,n) | D(\omega_H) \in F\omega_H\}.$$

The algebra $H(m,n)$ is the *Hamiltonian* and $CH(m,n)$ is the *conformal Hamiltonian* Lie algebra. The dimension of $H(m,n)$ over F is $p^N - 1$ and the dimension of $CH(m,n)$ is $\dim H(m,n) + 1$.

The algebra $H(m,n)^{(2)}$ is simple ([SF88, 4.4.5(1)]) and is restrictable if and only if $n=[1,\ldots,1]$ ([SF88, 4.4.5(2)]). And, if $m>2$, then $H(m,n)^{(2)} = H(m,n)^{(1)}$.

Let $p>2$, $m=2r+1\geq 3$ and let $\omega_K = dx_m + \sum_{i=1}^{r}(x_i dx_{i+r} - x_{i+r}dx_i)$. Define the following subalgebra of $W(m,n)$:

$$K(m,n) := \{D \in W(m,n) | D(\omega_K) \in O(m,n)\omega_K\},$$

The algebra $K(m,n)$ is the *contact* Lie algebra. The dimension of $K(m,n)$ over F is p^N.

The algebra $K(m,n)^{(1)}$ is simple ([SF88, 4.5.5(1)]) and is restrictable if and only if $n=[1,\ldots,1]$ ([SF88, 4.5.6]). If $m+3 \not\equiv 0 \bmod p$, then $K(m,n)^{(1)} = K(m,n)$.

```
WittLieAlgebra(F, m, n)
```

Check BoolElt *Default :* false

The Witt algebra $W(m, n)$ is constructed over the finite field F, where m must be a positive integer and n a sequence of positive integers of length m. If the optional argument Check is true, the algebra is checked to be Lie upon construction.

An invertible map from the polynomial ring P over F of degree $2m$ to $W(m, n)$ is returned as second value, to assist in identifying the elements of $W(m, n)$. For $1 \leq i \leq m$ the i-th generator of P maps to x_i in $W(m, n)$, and for $m + 1 \leq i \leq 2m$ the i-th generator of P maps to δ_{i-m} in $W(m, n)$.

Example H100E20

We compute the Witt algebra $W(2, [2, 1])$ over $GF(9)$ and verify the multiplication of $x_1^{(1)}\delta_1$ and $x_1^{(2)}x_2^{(1)}\delta_2$.

```
> W, phi := WittLieAlgebra(GF(9), 2, [2,1]);
> W;
Lie Algebra of dimension 54 with base ring GF(3^2)
> IsSimple(W);
true
> P<x1, x2, d1, d2> := Domain(phi);
> phi(x1*d1);
(0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> (phi(x1*d1)*phi(x1^2*x2*d2)) @@ phi;
2*x1^2*x2*d2
```

and the standard Witt algebra $W(1, [1])$ over $GF(2)$:

```
> W := WittLieAlgebra(GF(2), 1, [1]);
> W;
Lie Algebra of dimension 2 with base ring GF(2)
> IsSimple(W);
false
> IsRestrictedLieAlgebra(W);
true [ (0 0), (0 1) ]
```

```
SpecialLieAlgebra(F, m, n)
ConformalSpecialLieAlgebra(F, m, n)
```

Check BoolElt *Default :* false

The (conformal) special Lie algebra $(C)S(m, n)$ is constructed over the finite field F, where $m \geq 2$ must be an integer and n a sequence of positive integers of length m. If the optional argument Check is true, Magma checks that the algebra constructed is a Lie algebra.

The intrinsic `SpecialLieAlgebra` returns the Witt algebra $W(m, n)$ in which it is embedded as the second return value. In addition, similarly to `WittLieAlgebra`, a map from the polynomial ring P of degree $2m$ over F to $S(m, n)$ is returned as the third return value, and a map from P to $W(m, n)$ as the fourth return value.

Similarly, `ConformalSpecialLieAlgebra` returns the special Lie algebra $S(m, n)$ which it contains and the Witt Lie algebra $W(m, n)$ in which it is embedded in as second and third return values. Maps from P to $CS(m, n)$, $S(m, n)$, and $W(m, n)$ are returned as fourth, fifth, and sixth return values, respectively.

Example H100E21

We compute both $S(3, [1, 2, 1])$ and $CS(3, [1, 2, 1])$ over $GF(9)$:

```
> CS,S,W := ConformalSpecialLieAlgebra( GF(9), 3, [1,2,1] );
> CS;S;W;
Lie Algebra of dimension 164 with base ring GF(3^2)
Lie Algebra of dimension 163 with base ring GF(3^2)
Lie Algebra of dimension 243 with base ring GF(3^2)
> IsSimple(S);
false
> IsSimple(S*S);
true
> IsRestrictedLieAlgebra(S*S);
false []
```

`HamiltonianLieAlgebra(F, m, n)`

`ConformalHamiltonianLieAlgebra(F, m, n)`

`Check` BoolElt *Default :* `false`

The (conformal) Hamiltonian Lie algebra $(C)H(m, n)$ is constructed over the finite field F of characteristic at least 3, where $m \geq 2$ must be even and n a sequence of positive integers of length m. If the optional argument `Check` is `true`, the algebra is checked to be Lie upon construction.

The intrinsic `HamiltonianlLieAlgebra` returns the Witt Lie algebra $W(m, n)$ in which it is embedded as the second return value. Additionally, similarly to `WittLieAlgebra`, a map from the polynomial ring P of degree $2m$ over F to $H(m, n)$ is returned as the third return value, and a map from P to $W(m, n)$ as the fourth return value.

Similarly, `ConformalHamiltonianLieAlgebra` returns the Hamiltonian Lie algebra $H(m, n)$ it contains and the Witt Lie algebra $W(m, n)$ in which it is embedded as the second and third return values. Maps from P to $CH(m, n)$, $H(m, n)$, and $W(m, n)$ are returned as the fourth, fifth, and sixth return values, respectively.

Example H100E22

We compute both $H(2,[2,2])$ and $CH(2,[2,2])$ over $GF(9)$:

```
> CH,H,W := ConformalHamiltonianLieAlgebra( GF(9), 2, [2,2] );
> CH;H;W;
Lie Algebra of dimension 81 with base ring GF(3^2)
Lie Algebra of dimension 80 with base ring GF(3^2)
Lie Algebra of dimension 162 with base ring GF(3^2)
> IsSimple(H);
false
> IsSimple(H*H);
true
> IsSimple(H*H*H);
true
> IsRestrictedLieAlgebra(H*H*H);
false []
```

`ContactLieAlgebra(F, m, n)`

`Check` BoolElt *Default :* `false`

The contact Lie algebra $K(m,n)$ is constructed over the finite field F of characteristic at least 3, where $m \geq 3$ must be odd and n a sequence of positive integers of length m. If the optional argument `Check` is `true`, the algebra is checked to be Lie upon construction.

The intrinsic `ContactLieAlgebra` returns the Witt Lie algebra $W(m,n)$ in which it is embedded as the second return value. Additionally, similarly to `WittLieAlgebra`, a map from the polynomial ring P of degree $2m$ over F to $K(m,n)$ is returned as the third return value, and a map from P to $W(m,n)$ as the fourth return value.

Example H100E23

We compute the contact Lie algebra $K(3,[1,1,1])$ over $GF(5)$:

```
> K,W := ContactLieAlgebra( GF(5), 3, [1,1,1] );
> K;W;
Lie Algebra of dimension 125 with base ring GF(5)
Lie Algebra of dimension 375 with base ring GF(5)
> K*K eq K;
true
> IsSimple(K);
true
```

100.5.3 Melikian Lie Algebras

The Melikian Lie Algebras are a class of simple Lie algebras over finite fields of characteristic 5, parameterized by two positive integers n_1, n_2. We follow the explicit construction by Strade [Str04, Section 4.3].

Let F be a field of characteristic $p = 5$ and recall the definition of $O(m, n)$ and $W(m, n)$ from Section 100.5.2. Define $W = W(2, [n_1, n_2])$, $O = O(2, [n_1, n_2])$, and take W' to be a copy of W. We equip the vector space $W \oplus O \oplus W'$ with a bilinear product $[\cdot, \cdot]$ that is defined by the following equations, where $D, E \in W$ and $f, f_1, f_2, g, g_1, g_2 \in O$.

- On $W \times W$, the usual multiplication in W.
- On $W \times O$: $[D, f] = D(f) - 2\mathrm{div}(D)f$.
- On $W \times W'$: $[D, E'] = ([D, E])' + 2\mathrm{div}(D)E'$.
- On $O \times O$: $[f, g] = 2(g\delta_2(f) - f\delta_2(g))\delta_1' + 2(f\delta_1(g) - g\delta_1(f))\delta_2'$.
- On $O \times W'$: $[f, E'] = fE$.
- On $W' \times W'$: $[f_1\delta_1' + f_2\delta_2', g_1\delta_1' + g_2\delta_2'] = f_1g_2 - f_2g_1$.

Here div is the linear map defined by $\mathrm{div}(f\delta_i) = \delta_i f$. It follows that $M(n_1, n_2)$, of dimension $5^{n_1+n_2+1}$, is a simple Lie algebra [Str04, Lemma 4.3.1, Theorem 4.3.3].

`MelikianLieAlgebra(F, n1, n2)`

`Check` BoolElt *Default :* `false`

The Melikian Lie algebra $M = M(n_1, n_2)$ over F. An invertible map from the polynomial ring P of degree 6 over F to M is returned as second value, to assist in identifying the elements of M. Here the six generators of P represent x_1, x_2, δ_1, δ_2, δ_1', δ_2', respectively.

Example H100E24

We construct $M(2, 1)$ over $\mathbf{F}_5$ and inspect some of its properties.

```
> M, phi := MelikianLieAlgebra(GF(5), 2, 1);
> M;
Lie Algebra of dimension 625 with base ring GF(5)
> IsSimple(M);
true
```

Next, we construct subspaces (not subalgebras) W, O, W' of M.

```
> P<x1, x2, d1, d2, dp1, dp2> := Domain(phi);
> V := VectorSpace(GF(5), Dimension(M));
> W := sub<V | [ V | phi(x1^i*x2^j*d) : i in [0..24], j in [0..4],
>                                                    d in [d1,d2] ]>;
> O := sub<V | [ V | phi(x1^i*x2^j) : i in [0..24], j in [0..4] ]>;
> Wp := sub<V | [ V | phi(x1^i*x2^j*d) : i in [0..24], j in [0..4],
>                                                     d in [dp1,dp2] ]>;
> Dimension(W), Dimension(O), Dimension(Wp);
```

```
250 125 250
> Dimension(W meet O), Dimension(W meet Wp), Dimension(O meet Wp);
0 0 0
```

Finally, we verify that these subspaces multiply as required by the definition.

```
> m := func< A, B | sub<V | [ V | M!a*M!b : a in Basis(A), b in Basis(B) ]> >;
> WxWp := m(W, Wp); [ WxWp subset VV : VV in [W, O, Wp] ];
[ false, false, true ]
```

So indeed $[W, W'] \subseteq W'$.

```
> VV := [W, O, Wp]; VVnm := ["W", "O", "W'" ];
> mm := function(A, B)
>   AB := m(A, B);
>   for i in [1..#VV] do
>     if AB eq VV[i] then return VVnm[i]; end if;
>   end for;
>   return "??";
> end function;
> mm(W, Wp);
W'
> for i,j in [1..#VV] do
>   printf "[ %2o, %2o ] = %2o%o", VVnm[i], VVnm[j], mm(VV[i], VV[j]),
>     (j eq 3) select "\n" else ", ";
> end for;
[  W,  W ] =  W, [  W,  O ] =  O, [  W, W' ] = W'
[  O,  W ] =  O, [  O,  O ] = W', [  O, W' ] =  W
[ W',  W ] = W', [ W',  O ] =  W, [ W', W' ] =  O
```

100.6 Construction of Elements

`Zero(L)`

`L ! 0`

The zero element of the Lie algebra L.

`Random(L)`

Given a Lie algebra L defined over a finite ring, a random element is returned.

100.6.1 Construction of Elements of Structure Constant Algebras

`elt< L | r1,r2,...,rn >`

Given a Lie algebra L of dimension n over a ring R, and ring elements $r_1, r_2, \ldots, r_n \in R$ construct the element $r_1 * e_1 + r_2 * e_2 + \ldots + r_n * e_n$ of L.

`L ! Q`

Given a Lie algebra L of dimension n and a sequence $Q = [r_1, r_2, \ldots, r_n]$ of elements of the base ring R of L, the element $r_1 * e_1 + r_2 * e_2 + \ldots + r_n * e_n$ of L is constructed.

`BasisProduct(L, i, j)`

Returns the product of the i-th and j-th basis element of the Lie algebra L.

`BasisProducts(L)`

`Rep` MonStgElt *Default : "Dense"*

Returns the products of all basis elements of the Lie algebra L.

The optional parameter `Rep` may be used to specify the format of the result. If `Rep` is set to "Dense", the products are returned as a sequence Q of n sequences of n elements of L, where n is the dimension of L. The element $Q[i][j]$ is the product of the i-th and j-th basis elements.

If `Rep` is set to "Sparse", the products are returned as a sequence Q containing quadruples (i, j, k, a_{ijk}) signifying that the product of the i-th and j-th basis elements is $\sum_{k=1}^{n} a_{ijk} b_k$, where b_k is the k-th basis element and $n = \dim(L)$.

100.6.2 Construction of Matrix Elements

Matrix Lie elements can be constructed using the functions below. For more information on constructing matrices see Section 83.2.2.

`elt< R | L >`
`R ! L`

Create the element of the matrix Lie algebra R of degree n whose entries are the n^2 elements of the sequence L.

`DiagonalMatrix(L, Q)`

Diagonal matrix in the matrix Lie algebra L, given by the sequence Q of ring elements.

`ScalarMatrix(L, r)`

Scalar matrix in the matrix Lie algebra L, defined by the ring element r.

100.7 Construction of Subalgebras, Ideals and Quotients

If the coefficient ring R of a Lie algebra L is a Euclidean domain, then submodules and ideals can be constructed in MAGMA; if R is a field then quotients can be constructed in MAGMA. Note that left, right, and two-sided ideals are identical in a Lie algebra.

`sub< L | A >`

Creates the subalgebra S of the Lie algebra L that is generated by the elements defined by A, where A is a list of one or more items of the following types:

(a) An element of L;

(b) A set or sequence of elements of L;

(c) A subalgebra or ideal of L;

(d) A set or sequence of subalgebras or ideals of L.

As well as the subalgebra S itself, the constructor returns the inclusion homomorphism $f : S \to L$.

`ideal< L | A >`

Creates the ideal I of the Lie algebra L generated by the elements defined by A, where A is a list of one or more items of the following types:

(a) An element of L;

(b) A set or sequence of elements of L;

(c) A subalgebra or ideal of L;

(d) A set or sequence of subalgebras or ideals of L.

As well as the ideal I itself, the constructor returns the inclusion homomorphism $f : I \to L$.

`quo< L | A >`

Forms the quotient algebra L/I, where I is the two-sided ideal of L generated by the elements defined by A, where A is a list of one or more items of the following types:

(a) An element of L;

(b) A set or sequence of elements of L;

(c) A subalgebra or ideal of L;

(d) A set or sequence of subalgebras or ideals of L.

As well as the quotient L/I itself, the constructor returns the natural homomorphism $f : L \to L/I$.

`L / S`

The quotient of the Lie algebra L by the ideal closure of the subalgebra S.

Example H100E25

We construct the quotient of the matrix Lie algebra of 2×2 matrices, by the ideal spanned by the identity matrix.

```
> L := MatrixLieAlgebra( Rationals(), 2 );
> Dimension(L);
4
> I := ideal< L | L!Matrix([[1,0],[0,1]]) >;
> Dimension(I);
1
> K := L/I;
> Dimension(K);
3
> SemisimpleType( K );
A1
```

`QuotientWithPullback(L, I)`

Given a Lie algebra L and an ideal I of L, the quotient L/I is constructed. As second return value, the natural homomorphism $f : L \to L/I$ is returned.

As third return value, a function g is returned. This g takes an element $y \in I$ and returns an $x \in L$ and a vector space V such that $f(x + v) = y$ for all $v \in V$. As fourth return value, a function h is returned. This h takes an element $y \in I$ and returns the subalgebra of L generated by x and V, with x and V as above.

Example H100E26

We consider an ideal of the Lie algebra of type G_2 over the field with 3 elements.

```
> R := RootDatum("G2");
> L := LieAlgebra(R, GF(3));
> pos,neg,cart := StandardBasis(L);
> shrt := [ i : i in [1..NumPosRoots(R)] | IsShortRoot(R, i) ];
> shrt;
[ 1, 3, 4 ]
> I := ideal<L | pos[shrt]>;
> _, str1 := ReductiveType(I); str1;
The 7-dim simple constituent of a Lie algebra of type A2
```

So apparently I is isomorphic to the 7-dimensional simple constituent of a Lie algebra of type A_2. We will now use `QuotientWithPullback` to construct L/I.

```
> LI, proj, pb, pbsub := QuotientWithPullback(L, I);
> _, str2 := ReductiveType(LI); str2;
The 7-dim simple constituent of a Lie algebra of type A2
```

So apparently $I \simeq L/I$! Finally, we will demonstrate the use of the additional return values. First, we verify that an element of I maps to 0 in L/I:

```
> proj(pos[1]);
```

```
(0 0 0 0 0 0 0)
```

And then we consider the preimage in L of a randomly chosen element of L/I.

```
> y := LI![0,1,1,1,1,0,1];
> y;
(0 1 1 1 1 0 1)
> x, V := pb(y);
> x;
(0 1 0 0 1 0 0 1 0 1 0 0 0 1)
> #V;
2187
> assert #V eq #I;
> {* proj(x + v) eq y : v in V *};
{* true^^2187 *}
```

So indeed $x + v$ is a preimage of y for all $v \in V$.

```
> M := pbsub(y);
> M, M meet I;
Lie Algebra of dimension 8 with base ring GF(3)
Lie Algebra of dimension 7 with base ring GF(3)
> _,str3 := ReductiveType(M);
> str3;
Twisted Lie algebra of type 2A2 [Ad]
```

100.8 Operations on Lie Algebras

`L eq K`

Returns `true` if, and only if, the Lie algebras L and K are equal.

`L ne K`

Returns `true` if, and only if, the Lie algebras L and K are not equal.

`L subset K`

Returns `true` if, and only if, the Lie algebra L is contained in the Lie algebra K.

`L notsubset K`

Returns `true` if, and only if, the Lie algebra L is not contained in the Lie algebra K.

`L meet M`

The intersection of the Lie algebras L and M is returned. Note that L and M have a common superalgebra.

`L * M`

The Lie algebra product $[L, M]$ of the algebras L and M is returned. Note that L and M must have a common superalgebra.

`L ^ n`

The (left-normed) n-th power of the (structure constant) Lie algebra L, i.e., $((\ldots(L*L)*\ldots)*L)$ is constructed.

`Morphism(L, M)`

The map giving the morphism from the (structure constant) Lie algebra L to M is constructed. Either L is a subalgebra of M, in which case the embedding of L into M is returned, or M is a quotient algebra of L, in which case the natural epimorphism from L onto M is returned.

`IsIsomorphic(L, M)`

`HL`	AlgLie	*Default :* `false`
`HM`	AlgLie	*Default :* `false`

Returns `true` if the Lie algebras L and M are isomorphic. It is currently implemented for trivial cases (such as when the dimensions differ), reductive Lie algebras, solvable Lie algebras up to dimension 4, nilpotent Lie algebras up to dimension 6 (some special cases excluded). The solvable and nilpotent cases are handled using the databases for such algebras described in Section 100.18).

In the case of reductive Lie algebras, split maximal toral subalgebras for L and M may be provided in the optional arguments HL and HM, respectively. If these are not provided an attempt is made to compute them, a process which may fail, particularly in characteristic 0.

This intrinsic has two return values: the first a boolean describing whether L and M are isomorphic. If so, the second is an isomorphism from L to M, otherwise the second is a string describing the reason for non-isomorphism.

An error is thrown if isomorphism cannot be determined.

`IsKnownIsomorphic(L, M)`

`HL`	AlgLie	*Default :* `false`
`HM`	AlgLie	*Default :* `false`

Returns `true` if Magma can determine isomorphism between Lie algebras L and M. If so, the second return value is whether L and M are isomorphic, and the third is an isomorphism or a string (describing the reason for non-isomorphism). Refer to IsIsomorphic for more details on applicability and the meanings of the return values.

`IsIsomorphism(m)`

Returns true if the mapping m between two Lie algebras is an isomorphism of Lie algebras.

Example H100E27

We demonstrate that B_2 and C_2 are isomorphic over $\mathbf{Q}$.

```
> k := Rationals();
> L := LieAlgebra("B2", k); M := LieAlgebra("C2", k);
> b, c := IsIsomorphic(L, M);
> b;
true
> IsIsomorphism(c);
true
> c(L.1);
(0 0 1 0 0 0 0 0 0 0)
```

We demonstrate that B_3 and C_3 are non-isomorphic over $\mathbf{Q}$.

```
> L := LieAlgebra("B3", k); M := LieAlgebra("C3", k);
> b, c := IsIsomorphic(L, M);
> b;
false
> c;
21-dim component of L1 of type R1: Adjoint root datum of dimension 3 of type B3
didn't match R2:
Adjoint root datum of dimension 3 of type C3
```

We demonstrate that two distinct isogenies of B_2 are isomorphic over $\mathbf{Q}$.

```
> L := LieAlgebra("B2", k : Isogeny := "Ad");
> M := LieAlgebra("B2", k : Isogeny := "SC");
> b, c := IsIsomorphic(L, M);
> b;
true
```

For larger nilpotent algebras MAGMA cannot decide on the isomorphism question.

```
> L := LieAlgebra("B4", k);
> pL, _, _ := StandardBasis(L);
> subL := sub<L | pL>;
> subL;
Lie Algebra of dimension 16 with base ring Rational Field
> M := LieAlgebra("C4", k);
> pM, _, _ := StandardBasis(M);
> subM := sub<M | pM>;
> subL;
Lie Algebra of dimension 16 with base ring Rational Field
> IsNilpotent(subL), IsNilpotent(subM);
true true
> a,b,c := IsKnownIsomorphic(subL, subM);
> a;
false
```

Example H100E28

We demonstrate that in characteristic 3 the Lie algebras of type G_2 and A_2 have isomorphic nontrivial ideals.

```
> k := GF(3);
> CSL := CompositionSeries(LieAlgebra("G2", k));
> CSL;
[
    Lie Algebra of dimension 7 with base ring GF(3),
    Lie Algebra of dimension 14 with base ring GF(3)
]
> L := CSL[1];
> CSM := CompositionSeries(LieAlgebra("A2", k));
> CSM;
[
    Lie Algebra of dimension 7 with base ring GF(3),
    Lie Algebra of dimension 8 with base ring GF(3)
]
> M := CSM[1];
> a,b,c := IsKnownIsomorphic(L, M);
> a;
true
> b, c;
true Mapping from: AlgLie: L to AlgLie: M given by a rule
> IsIsomorphism(c);
true
```

100.8.1 Basic Invariants

`CoefficientRing(L)`

`BaseRing(L)`

The coefficient ring (or base ring) over which the Lie algebra L is defined.

`Dimension(L)`

The dimension of the Lie algebra L.

`#L`

The cardinality of the Lie algebra L, if the coefficient ring is finite.

`Moduli(L)`

This returns a sequence of integers, of length equal to the dimension of L. If the i-th element of this sequence is a_i then a_i is the minimal non-negative integer such that $a_i e_i = 0$. So if L is defined over a field, then the sequence consists of zeros.

Example H100E29

```
> T:= [ <1,2,2,2>, <2,1,2,2> ];
> t:= [0,4];
> L:= LieAlgebra< t | T : Rep:= "Dense" >;
> Moduli(L);
[ 0, 4 ]
```

100.8.2 Changing Base Rings

`ChangeRing(L, S)`

Given a Lie algebra L with base ring R, together with a ring S, this function constructs the Lie algebra M with base ring S obtained by coercing the coefficients of elements of L into S. The homomorphism from L to M is produced as second return value.

`ChangeRing(L, S, f)`

Given a Lie algebra L with base ring R, together with a ring S and a map $f : R \to S$, this function constructs the Lie algebra M with base ring S obtained by mapping the coefficients of elements of L into S via f. The homomorphism from L to M is produced as the second return value.

100.8.3 Bases

`BasisElement(A, i)`

`A . i`

The i-th basis element of the algebra L.

`Basis(A)`

The basis of the algebra L, as a sequence of elements of L.

`IsIndependent(Q)`

Given a set or sequence Q of elements of the R-algebra L, this functions returns `true` if these elements are linearly independent over R; otherwise `false`.

`ExtendBasis(S, L)`

`ExtendBasis(Q, L)`

Given an algebra L and either a subalgebra S of dimension m of L or a sequence Q of m linearly independent elements of L, this function returns a sequence containing a basis of L such that the first m elements are the basis of S resp. the elements in Q.

100.8.4 Operations for Semisimple and Reductive Lie Algebras

`SemisimpleType(L)`

`CartanName(L)`

Let L be a Lie algebra. If L has a nondegenerate Killing form, then (over some algebraic extension of the ground field) L is the direct sum of absolutely simple Lie algebras. These Lie algebras have been classified and the classes are named A_n, B_n, C_n, D_n, E_6, E_7, E_8, F_4 and G_2. This function returns a single string containing the types of the direct summands of L.

For a description of the algorithm used in the general case we refer to [dG00], §5.17.1. For Lie algebras over fields of characteristic 2 and 3 the algorithm used is described in [Roo10], Chapter 5.

Example H100E30

We compute the semisimple type of the Levi subalgebra of a subalgebra of the simple Lie algebra of type D_7.

```
> L := LieAlgebra("D7", RationalField());
> L;
Lie Algebra of dimension 91 with base ring Rational Field
> K := Centralizer(L, sub<L | [L.1,L.2,L.3,L.4]>);
> K;
Lie Algebra of dimension 41 with base ring Rational Field
> _,S := HasLeviSubalgebra(K);
> S;
Lie Algebra of dimension 6 with base ring Rational Field
> SemisimpleType(S);
A1 A1
```

`ReductiveType(L)`

`ReductiveType(L, H)`

`AssumeAlmostSimple` BoolElt *Default :* `false`

Let L be a Lie algebra of a reductive algebraic group, and H a split maximal toral subalgebra of L. This function identifies the isomorphism type of L.

This function has four return values. The first is the appropriate root datum and the second return value a textual description of L. The third return value is a sequence Q, containing a decomposition of L into direct summands. Finally, the fourth return value is a sequence P of records, such that $P[i]$ contains additional information (often a proof of correctness) of the identification of $Q[i]$.

If a split maximal toral subalgebra H is not given, an attempt is made to compute one by calling `SplitMaximalToralSubalgebra` if the characteristic of the base field k is at least 5, or `SplitToralSubalgebra` if $\mathrm{char}(k)$ is 2 or 3. Note that, if k is

infinite, such a subalgebra cannot in general be computed so the second parameter H must be supplied for this function to work.

If the optional parameter `AssumeAlmostSimple` is set to true, the (possibly time consuming) step of computing a direct sum decomposition of L is skipped.

Moreover, note that if L is the Lie algebra of a simple algebraic group but itself non-simple (such as for example A_n of intermediate type in characteristic $n+1$), the third return value Q may not be the direct sum decomposition of L but simply $[L]$.

Example H100E31

We consider a particular Lie algebra of type A_3 over $k = \mathrm{GF}(2)$.

```
> RA3 := RootDatum("A3" : Isogeny := 2);
> L := LieAlgebra(RA3, GF(2));
> D := DirectSumDecomposition(L);
> D;
[
    Lie Algebra of dimension 14 with base ring GF(2),
    Lie Algebra of dimension 1 with base ring GF(2)
]
> R, str, Q, _ := ReductiveType(L);
> R;
RA3: Root datum of dimension 3 of type A3
> str;
Lie algebra of type A3[ 2]
> Q;
[
    Lie Algebra of dimension 15 with base ring GF(2)
]
```

Note that this is an example where Q is not the direct sum decomposition of L. Instead, L in its whole is recognised as the Lie algebra of a simple algebraic group. In the remainder of the example, we investigate the 14-dimensional ideal of L.

```
> M := D[1]; M;
Lie Algebra of dimension 14 with base ring GF(2)
> R, _, _, P := ReductiveType(M);
> R;
R: Adjoint root datum of dimension 2 of type G2
```

So this computation claims that $L \simeq M \oplus k$, where M is of type G_2. Let us use the additional return values to verify that fact.

```
> pos  := P[1]‘ChevBasData‘BasisPos;
> neg  := P[1]‘ChevBasData‘BasisNeg;
> cart := P[1]‘ChevBasData‘BasisCart;
> IsChevalleyBasis(M, RootDatum("G2"), pos, neg, cart);
```

```
true [ <1, 2, 0>, <1, 3, 0>, <1, 4, 0>, <2, 5, 0> ]
```

This demonstrates the fact that the Lie algebra of type G_2 is a constituent of the Lie algebra of type A_3 over fields of characteristic 2.

`RootSystem(L)`

Given a semisimple Lie algebra L with a split Cartan subalgebra, this function computes the root system of L. This function returns four values:

(a) The roots of L with respect to the Cartan subalgebra which is output by `CartanSubalgebra(L)`. This is a sequence of vectors where the positive roots come first, followed by the negative roots.

(b) A sequence of elements of L which are the root vectors corresponding to the roots of L (so the first element corresponds to the first root and so on).

(c) A sequence of simple roots.

(d) The Cartan matrix of the root system with respect to the sequence of simple roots.

Example H100E32

We compute the root system of the simple Lie algebra of type G_2 over the rational field.

```
> L := LieAlgebra("G2", RationalField());
> R, Rv, fund, C:=RootSystem(L);
> R;
[
    (1 0),
    (0 1),
    (1 1),
    (2 1),
    (3 1),
    (3 2),
    (-1  0),
    ( 0 -1),
    (-1 -1),
    (-2 -1),
    (-3 -1),
    (-3 -2)
]
> Rv;
[ (0 0 0 0 0 0 0 0 1 0 0 0 0 0), (0 0 0 0 0 0 0 0 0 1 0 0 0 0),
(0 0 0 0 0 0 0 0 0 0 1 0 0 0), (0 0 0 0 0 0 0 0 0 0 0 1 0 0),
(0 0 0 0 0 0 0 0 0 0 0 0 1 0), (0 0 0 0 0 0 0 0 0 0 0 0 0 1),
(0 0 0 0 0 1 0 0 0 0 0 0 0 0), (0 0 0 0 1 0 0 0 0 0 0 0 0 0),
(0 0 0 1 0 0 0 0 0 0 0 0 0 0), (0 0 1 0 0 0 0 0 0 0 0 0 0 0),
(0 1 0 0 0 0 0 0 0 0 0 0 0 0), (1 0 0 0 0 0 0 0 0 0 0 0 0 0) ]
```

`RootDatum(L)`

Here L is a semisimple Lie algebra. This function returns the root datum D of L with respect to the Cartan subalgebra which is output by `CartanSubalgebra(L)`. We note that the order of the positive roots in D is not necessarily the same as the order in which they appear in the root system of L.

Example H100E33

We set up the root datum of a Lie algebra, and extract the Cartan matrix.

```
> L:= LieAlgebra("F4", Rationals());
> rd := RootDatum(L);
> rd;
Root datum of type F4
> CartanMatrix(rd);
[ 2  0 -1  0]
[ 0  2  0 -1]
[-1  0  2 -1]
[ 0 -1 -2  2]
```

`ChevalleyBasis(L)`

`ChevalleyBasis(L, H)`

`AssumeAlmostSimple` BoolElt *Default :* `false`

Given a semisimple Lie algebra L with a split maximal toral subalgebra H, this function returns three sequences, x, y and h of elements of L. They form a Chevalley basis of L. The first sequence gives basis elements corresponding to positive roots, the second to the negative roots and the third to basis elements in a Cartan subalgebra. If a split maximal toral subalgebra H is not given, an attempt is made to compute one.

For Lie algebras over fields of characteristic 2 and 3 the algorithm used is described in [CR09]. In particular, this involves computing a direct sum decomposition of L, which can be quite time consuming. If there is reason to believe that L is (almost) simple, the optional parameter `AssumeAlmostSimple` should be set to `true`.

Example H100E34

We construct a Chevalley basis for two Lie algebras.

```
> L := LieAlgebra("A2", RationalField());
> x, y, h:= ChevalleyBasis(L);
> x; y; h;
[ (0 0 0 0 0 1 0 0), (0 0 0 0 0 0 1 0), (0 0 0 0 0 0 0 1) ]
[ (0 0 1 0 0 0 0 0), (0 1 0 0 0 0 0 0), (1 0 0 0 0 0 0 0) ]
[ (0 0 0 1 0 0 0 0), (0 0 0 0 1 0 0 0) ]
> L := LieAlgebra("A3", Rationals());
```

```
> print RootDatum(L) : Maximal;
Root datum of type A3 with simple roots
[ 1  0  1]
[ 1 -2  1]
[ 0  1 -2]
and simple coroots
[ 1  1  1]
[ 0 -1  0]
[ 0  0 -1]
```

`ChevalleyBasis(L, H, R)`

Given a semisimple Lie algebra L with a split maximal toral subalgebra H, and an irreducible root datum R, this function computes a Chevalley basis of L with respect to H and R. This basis is returned in the form of three sequences, x, y and h of elements of L, where the first sequence gives basis elements corresponding to positive roots, the second to the negative roots and the third to basis elements in the toral subalgebra H.

`IsChevalleyBasis(L, R, x, y, h)`

Returns `true` if x, y and h form a Chevalley basis of the Lie algebra L with respect to the root datum R. If so, return a sequence describing the extraspecial signs as second return value.

Example H100E35

We compute a Chevalley basis for a Lie algebra of type E_6 inside one of type E_7.

```
> R := RootDatum("E7");
> L1 := LieAlgebra(R, GF(2));
> p1,n1,c1 := StandardBasis(L1);
> L1;
Lie Algebra of dimension 133 with base ring GF(2)
> DynkinDiagram(R);
E7    1 - 3 - 4 - 5 - 6 - 7
              |
              2
> S, proj := sub<R | [1..6]>;
> S;
S: Root datum of dimension 7 of type E6
> #proj;
72
> projpos := [i : i in proj | i le NumPosRoots(R)];
> #projpos;
36
> L2 := sub<L1 | p1[projpos], n1[projpos]>;
> L2;
```

```
Lie Algebra of dimension 78 with base ring GF(2)
> H2 := L2 meet SplitMaximalToralSubalgebra(L1);
> H2;
Lie Algebra of dimension 6 with base ring GF(2)
> p2,n2,c2 := ChevalleyBasis(L2, H2, RootDatum("E6"));
> ok := IsChevalleyBasis(L2, RootDatum("E6"), p2, n2, c2);
> ok;
true
```

`TwistedBasis(L, H, R)`

For a Lie algebra L, a split toral subalgebra H of L, and a twisted root datum R, the function constructs a "twisted basis" of L.

Let k be the coefficient ring of L and K an extension field of k of degree equal to the twisting degree of R. This function has 4 return values. First, $L' = L \otimes K$; second, a homomorphism ϕ from L to L', third, a record containing a Chevalley basis of L' with respect to the untwisted root datum of R; fourth, a matrix describing the action of the Frobenius automorphism of K on the positive roots of the Chevalley basis of L'.

Such a basis constitutes a proof that L' is of type R. Consult [Roo10], Chapter 5.3, for more details on such twisted bases.

Example H100E36

We investigate a twisted basis of the Lie algebra of type ${}^2\mathrm{A}_2$ over the field with 5 elements. Let δ be the automorphism of the root system of type A_2, let k =GF(5), and let K =GF(5^2).

```
> R := TwistedRootDatum(RootDatum("A2") : Twist := 2);
> L := TwistedLieAlgebra(R, GF(5));
> H := SplitToralSubalgebra(L);
> LK, phi, ChevBas, m := TwistedBasis(L, H, R);
> m;
[     0      1]
[     1      0]
```

This matrix `m` shows that δ acts as expected on the Chevalley basis elements of `LK` $= L \otimes K$. We verify the correctness of `m`.

```
> K := CoefficientRing(LK);
> simp := ChevBas`BasisPos[[1..Rank(R)]];
> simp;
[ (     0      0      0      0      0      1  ksi^8      0),
  (     0      0      0      0      0      1 ksi^16      0) ]
> fr := FrobeniusMap(K);
> frv := func<x | Vector([ fr(i) : i in Eltseq(x)])>;
> [ Position(simp, frv(x)) : x in simp ];
```

```
[ 2, 1 ]
```

So indeed the Frobenius map (acting on the coordinates of LK) acts as δ. This is equivalent [Roo10, Lemma 5.3] to the basis elements of L being stable under the composition of the Frobenius map (this time acting on the Chevalley basis of $L \otimes K$) and the root system automorphism δ. We verify this assertion explicitly for this example.

```
> p := ChevBas‘BasisPos;
> n := ChevBas‘BasisNeg;
> c := ChevBas‘BasisCart;
> pi := Sym(6)!(1, 2)(4, 5);
> ChevBasLK := VectorSpaceWithBasis([ Vector(x) : x in p cat n cat c]);
> piL := DiagramAutomorphism(LK, pi);
```

Now δ acts on $L \otimes K$ as `T`, and `fr` is still the Frobenius automorphism of the field K. The images of the basis elements of L under `delta` composed with `fr` are as follows:

```
> for i in [1..Dimension(L)] do
>    b := phi(L.i);
>    printf "i = %o, b =    %o\n", i, Coordinates(ChevBasLK, Vector(b));
>    printf "   pi(b)^fr = %o\n", [ fr(i) : i in
>                              Coordinates(ChevBasLK, Vector(piL(b))) ];
> end for;
i = 1, b =    [ 0, 0, 0, 0, 0, ksi^9, 0, 0 ]
   (b*T)^fr = [ 0, 0, 0, 0, 0, ksi^9, 0, 0 ]
i = 2, b =    [ 0, 0, 0, ksi^5, ksi, 0, 0, 0 ]
   (b*T)^fr = [ 0, 0, 0, ksi^5, ksi, 0, 0, 0 ]
i = 3, b =    [ 0, 0, 0, ksi^9, ksi^21, 0, 0, 0 ]
   (b*T)^fr = [ 0, 0, 0, ksi^9, ksi^21, 0, 0, 0 ]
i = 4, b =    [ 0, 0, 0, 0, 0, 0, ksi^5, ksi ]
   (b*T)^fr = [ 0, 0, 0, 0, 0, 0, ksi^5, ksi ]
i = 5, b =    [ 0, 0, 0, 0, 0, 0, ksi, ksi^5 ]
   (b*T)^fr = [ 0, 0, 0, 0, 0, 0, ksi, ksi^5 ]
i = 6, b =    [ ksi, ksi^5, 0, 0, 0, 0, 0, 0 ]
   (b*T)^fr = [ ksi, ksi^5, 0, 0, 0, 0, 0, 0 ]
i = 7, b =    [ ksi^21, ksi^9, 0, 0, 0, 0, 0, 0 ]
   (b*T)^fr = [ ksi^21, ksi^9, 0, 0, 0, 0, 0, 0 ]
i = 8, b =    [ 0, 0, ksi^9, 0, 0, 0, 0, 0 ]
   (b*T)^fr = [ 0, 0, ksi^9, 0, 0, 0, 0, 0 ]
```

Thus, all the basis elements of L are stable under the composition of the diagram automorphism δ and the Frobenius automorphism.

The `WeylGroup` functions are only available for structure constant Lie algebras.

`WeylGroup(L)`

`WeylGroup(GrpPermCox, L)`

The Weyl group of the reductive Lie algebra L, as a permutation Coxeter group (see Chapter 98).

`WeylGroup(GrpFPCox, L)`

The Weyl group of the reductive Lie algebra L, as a Coxeter group (see Chapter 98).

`WeylGroup(GrpMat, L)`

The Weyl group of the reductive Lie algebra L, as a reflection group (see Chapter 98).

100.9 Operations on Subalgebras and Ideals

`DirectSum(L, M)`

Given Lie algebras L and M, this intrinsic constructs a Lie algebra of dimension $n + m$, where n and m are the dimensions of L and M, respectively. The basis of the new algebra is the concatenation of the bases of L and M and the products $a * b$ where $a \in L$ and $b \in M$ are defined to be zero.

`IndecomposableSummands(L)`

`DirectSumDecomposition(L)`

Given a Lie algebra L, the function returns the direct sum decomposition of L as a sequence of ideals of L whose sum is L and each of which cannot be further decomposed into a direct sum of ideals.

The algorithms used for this function are described in [dG00], §4.12 (semisimple case), §1.15 (general case).

Example H100E37

We compute the direct sum decomposition of the simple Lie algebra of type D_2 over the rational field.

```
> L := LieAlgebra("D2", RationalField());
> L;
Lie Algebra of dimension 6 with base ring Rational Field
> D := DirectSumDecomposition(L);
> D;
[
    Lie Algebra of dimension 3 with base ring Rational Field,
    Lie Algebra of dimension 3 with base ring Rational Field
]
> Morphism(D[1], L);
[ 0  1  0  0  0  0]
[ 0  0  1 -1  0  0]
[ 0  0  0  0  1  0]
> Morphism(D[2], L);
[1 0 0 0 0 0]
[0 0 1 1 0 0]
[0 0 0 0 0 1]
```

100.9.1 Standard Ideals and Subalgebras

`Centre(L)`
`Center(L)`

Given a Lie algebra L, returns the centre of L.

`Centraliser(L, K)`
`Centralizer(L, K)`

Given a Lie algebra L and a subalgebra K of L, returns the centraliser of K in L, and its injection into L.

`Centraliser(L, x)`
`Centralizer(L, x)`

Given a Lie algebra L and an element x of L, returns the centraliser of x in L, and its injection into L.

`Normaliser(L, K)`
`Normalizer(L, K)`

Given a Lie algebra L and a subalgebra K of L, returns the normaliser of K in L, and its injection into L.

`SolubleRadical(L)`
`SolvableRadical(L)`

Given a Lie algebra L, returns the soluble radical of L.

We refer to [dG00], §2.6 for the algorithm used to implement this function.

`Nilradical(L)`

Given a Lie algebra L, returns the nilradical of L.

The algorithm makes use of Cartan subalgebras. We refer to [dG00], pp. 84, 85 for its description.

Example H100E38

We demonstrate the functions for performing basic operations with Lie algebras such as centre, normalizer etc.

```
> L := LieAlgebra("D4", RationalField());
> L;
Lie Algebra of dimension 28 with base ring Rational Field
> Centre(L);
Lie Algebra of dimension 0 with base ring Rational Field
> K := sub< L | [L.1, L.2, L.3] >;
> Centralizer(L, K);
Lie Algebra of dimension 10 with base ring Rational Field
> Normalizer(L, K);
```

```
Lie Algebra of dimension 19 with base ring Rational Field
> M := Centralizer(L, K);
> S := SolvableRadical(M);
> S;
Lie Algebra of dimension 10 with base ring Rational Field
> Morphism(S, L);
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 -1 0 0 0 -1 0 0 -1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
> Nilradical(M);
Lie Algebra of dimension 9 with base ring Rational Field
```

100.9.2 Cartan and Toral Subalgebras

`CartanSubalgebra(L)`

Given a Lie algebra L, this function returns a Cartan subalgebra of L. The algorithm works for Lie algebras L defined over a field F such that $|F| > \dim L$ and for restricted Lie algebras of characteristic p. If the Lie algebra does not fit into one of these classes then the correctness of the output is not guaranteed.

The algorithm used is described in [dG00], §3.2.

`IsCartanSubalgebra(L, H)`

The intrinsic returns `true` if H is a Cartan subalgebra of L, i.e., whether H is nilpotent and $N_L(H) = 0$.

Example H100E39

We compute a Cartan subalgebra of the simple Lie algebra of type A_4 over the rational field.

```
> L := LieAlgebra("F4", RationalField());
> L;
Lie Algebra of dimension 52 with base ring Rational Field
> H := CartanSubalgebra(L);
Lie Algebra of dimension 4 with base ring Rational Field
> H*H;
Lie Algebra of dimension 0 with base ring Rational Field
> Normalizer(L, H);
Lie Algebra of dimension 4 with base ring Rational Field
```

`SplittingCartanSubalgebra(L)`

`SplitMaximalToralSubalgebra(L)`

Given a Lie algebra L over a field k of characteristic at least 5, a split Cartan subalgebra (equivalently, a split maximal toral subalgebra) is computed for L.

The algorithm used is discussed in [CM09], Sections 5 and 6. This algorithm is proved to work ([CM09, Theorem 6.7]) if L is the Lie algebra of a k-split connected reductive group. In other cases, should the algorithm terminate, the output is guaranteed to be correct.

`IsSplittingCartanSubalgebra(L, H)`

Determine whether H is a splitting Cartan subalgebra of L, i.e., whether H is a Cartan subalgebra and the adjoint action of H on L splits completely over the coefficient ring of L.

`SplitToralSubalgebra(L)`

`TryMaximal` . *Default :* `true`

The intrinsic attempts to compute a split toral subalgebra of a Lie algebra L defined over a finite field k. This procedure uses a heuristic algorithm, described in [Roo10, Chapter 3], that works in many cases even if the characteristic of k is small. Moreover, it attempts to compute a split toral subalgebra of maximal size.

If the function returns without error, the resulting subalgebra H is a split toral subalgebra that does not lie inside a split toral subalgebra H' of larger dimension. It is, however, not guaranteed that H is of maximal dimension among all split toral subalgebras.

The optional parameter `TryMaximal` may be used as follows. If set to `true` (the default) the reductive rank r of L is computed first, and the algorithm attempts to compute a split toral subalgebra of dimension r. If set to `false`, the first split toral subalgebra found is returned. Finally, if `TryMaximal` is set to an integer $n \geq 1$, the algorithm attempts to find a split toral subalgebra of dimension n. In the latter case, if no split toral subalgebra of dimension n can be found, the biggest that has been found is returned; if on the other hand a split toral subalgebra of dimension *larger* than n is encountered, that is returned.

`IsSplitToralSubalgebra(L, H)`

Given a restrictable Lie algebra L over a finite field, the function returns `true` is H is a split toral subalgebra of L, i.e., whether $[H, H] = 0$, all elements of H are semisimple, and the basis elements are invariant under the q-map associated to L.

Example H100E40

We construct a twisted Lie algebra L of type ${}^3\mathrm{D}_4$ over the field $k = \mathrm{GF}(3^3)$ and verify that the subalgebra H returned by `SplitToralSubalgebra` is indeed a split toral subalgebra. Then, we test whether $C = C_L(H)$ is a (split) toral subalgebra of L.

```
> k := GF(3, 3);
```

```
> L, phi := TwistedLieAlgebra(TwistedRootDatum("D4" : Twist := 3), k);
> H := SplitToralSubalgebra(L);
> H;
Lie Algebra of dimension 2 with base ring GF(3^3)
> IsSplitToralSubalgebra(L, H);
true
> C := Centraliser(L,H); C;
Lie Algebra of dimension 4 with base ring GF(3^3)
> IsToralSubalgebra(L,C), IsSplitToralSubalgebra(L, C);
true false
```

Now we let K be the big field, $\mathrm{GF}(3^9)$, and test if $C \otimes K$ is a split toral subalgebra of $L \otimes K$.

```
> LK := Codomain(phi);
> LK;
Lie Algebra of dimension 28 with base ring GF(3^9)
> CK := sub<LK | [ phi(b) : b in Basis(C) ]>;
> IsSplitToralSubalgebra(LK, CK);
true
```

100.9.3 Standard Series

`CompositionSeries(L)`

A composition series is computed for the (structure constant) Lie algebra L. The function returns three values:

(a) a sequence containing the composition series as an ascending chain of subalgebras such that the successive quotients are irreducible L-modules;

(b) a sequence containing the composition factors as structure constant algebras;

(c) a transformation matrix to a basis compatible with the composition series, that is, the first basis elements form a basis of the first term of the composition series, the next extend these to a basis for the second term etc.

`CompositionFactors(L)`

Compute the composition factors of a composition series for the Lie algebra L. This function returns the same as the second return value of `CompositionSeries` above, but will often be very much quicker.

`MinimalIdeals(L : parameters)`

`Limit` RngIntElt *Default :* ∞

Returns the minimal left/right/two-sided ideals of the (structure constant) Lie algebra L (in non-decreasing size). If `Limit` is set to n, at most n ideals are calculated and the second return value indicates whether all of the ideals were computed.

`MaximalIdeals(L : parameters)`

`Limit` RngIntElt *Default :* ∞

Returns the maximal left/right/two-sided ideals of the (structure constant) Lie algebra L (in non-decreasing size). If `Limit` is set to n, at most n ideals are calculated and the second return value indicates whether all of the ideals were computed.

`DerivedSeries(L)`

Given a Lie algebra L, this function returns a sequence of ideals of L that form its derived series.

`LowerCentralSeries(L)`

Given a Lie algebra L, this function returns a sequence of ideals of L that form its lower central series.

`UpperCentralSeries(L)`

Given a Lie algebra L, this function returns a sequence of ideals of L that form the upper central series of L. The function repeatedly uses the algorithm for computing centres while keeping track of the pre-images of the ideals factored out.

Example H100E41

We compute each of the type of series of a particular subalgebra of the simple Lie algebra of type F_4 over the rational field.

```
> L:=LieAlgebra("F4", RationalField());
> L;
Lie Algebra of dimension 52 with base ring Rational Field
> K:=sub< L | [L.1, L.12, L.23, L.34, L.45] >;
> DerivedSeries(K);
[
    Lie Algebra of dimension 20 with base ring Rational Field,
    Lie Algebra of dimension 16 with base ring Rational Field,
    Lie Algebra of dimension 7 with base ring Rational Field,
    Lie Algebra of dimension 0 with base ring Rational Field
]
> LowerCentralSeries(K);
[
    Lie Algebra of dimension 20 with base ring Rational Field,
    Lie Algebra of dimension 16 with base ring Rational Field,
    Lie Algebra of dimension 12 with base ring Rational Field,
    Lie Algebra of dimension 8 with base ring Rational Field,
    Lie Algebra of dimension 5 with base ring Rational Field,
    Lie Algebra of dimension 2 with base ring Rational Field,
    Lie Algebra of dimension 1 with base ring Rational Field,
    Lie Algebra of dimension 0 with base ring Rational Field
]
> UpperCentralSeries(K);
```

```
[
    Lie Algebra of dimension 2 with base ring Rational Field,
    Lie Algebra of dimension 3 with base ring Rational Field,
    Lie Algebra of dimension 5 with base ring Rational Field,
    Lie Algebra of dimension 8 with base ring Rational Field,
    Lie Algebra of dimension 12 with base ring Rational Field,
    Lie Algebra of dimension 16 with base ring Rational Field,
    Lie Algebra of dimension 20 with base ring Rational Field
]
```

100.9.4 The Lie Algebra of Derivations

`LieAlgebraOfDerivations(L)`

Given a Lie algebra L, this function constructs its Lie algebra of derivations $\mathrm{Der}(L)$. As second return value, a record containing maps from L to $\mathrm{Der}(L)$ and vice versa, and from $\mathrm{Der}(L)$ to the matrix Lie algebra acting on L is returned.

Example H100E42

We consider the Lie algebra of derivations of D_4 in characteristic 2 or, more precisely, the 26-dimensional simple constituent L that exists in all varieties of D_4 in characteristic 2.

```
> SetSeed(1);
> R := RootDatum("D4");
> D4 := LieAlgebra(R, GF(2));
> pos,neg,cart := StandardBasis(D4);
> L := D4*D4; L;
Lie Algebra of dimension 26 with base ring GF(2)
> IsSimple(L);
true
> DerL, maps := LieAlgebraOfDerivations(L);
> DerL;
Lie Algebra of dimension 52 with base ring GF(2)
> SemisimpleType(DerL);
F4
```

So the Lie algebra of derivations is of type F_4. Let us consider one of the maps that was returned as second value.

```
> maps;
rec<recformat<mp_DerL_to_L: Map, mp_L_to_DerL: Map, mp_DerL_to_mats:
Map, mp_mats_to_DerL: Map> |
    mp_DerL_to_L := Mapping from: AlgLie: DerL to AlgLie: L given by a
        rule [no inverse],
    mp_L_to_DerL := Mapping from: AlgLie: L to AlgLie: DerL given by a
        rule [no inverse],
    mp_DerL_to_mats := Mapping from: AlgLie: DerL to Matrix Lie
```

```
        Algebra given by a rule [no inverse],
    mp_mats_to_DerL := Mapping from: Matrix Lie Algebra to AlgLie:
        DerL given by a rule [no inverse]>
> adL := AdjointRepresentation(L);
> f := maps‘mp_DerL_to_mats;
> [ f(b) in Image(adL) : b in Basis(DerL) ];
[ false, true, true, true, true, true, true, true, false, false, true,
false, true, false, false, false, true, false, true, true, true,
false, false, false, true, true, false, false, false, true, true,
false, false, false, true, true, false, false, true, false, true,
false, true, false, false, false, true, false, true, false, false,
false ]
```

So, unsurprisingly, some of the basis elements of $\mathrm{Der}(L)$ are actually elements from L, but others are not. We consider one more of these maps and investigate how L lies in $\mathrm{Der}(L)$.

```
> g := maps‘mp_L_to_DerL;
> I := ideal<DerL | [ g(b) : b in Basis(L) ]>; I;
Lie Algebra of dimension 26 with base ring GF(2)
> pos2, neg2, cart2 := ChevalleyBasis(DerL, SplitToralSubalgebra(DerL));
> [i : i in [1..#pos2] | pos2[i] in I ];
[ 3, 4, 6, 7, 8, 10, 12, 13, 15, 17, 19, 21 ]
> RF4 := RootDatum("F4");
> [ i : i in [1..NumPosRoots(RF4)] | IsShortRoot(RF4, i) ];
[ 3, 4, 6, 7, 8, 10, 12, 13, 15, 17, 19, 21 ]
```

So we conclude that the original Lie algebra L of type D_4 exists as the short roots of the Lie algebra of derivations $\mathrm{Der}(L)$ of type F_4.

100.10 Properties of Lie Algebras and Ideals

`KillingMatrix(L)`

Given a Lie algebra L such that $\{x_1, \ldots, x_n\}$ is a basis of L, return the Killing matrix of L, which is defined to be the matrix $(\mathrm{Tr}(\mathrm{ad}x_i \cdot \mathrm{ad}x_j))$.

Example H100E43

```
> L:=LieAlgebra("B2",RationalField());
> KillingMatrix(L);
[ 0  0  0 -6  0  0  0  0  0  0]
[ 0  0 -6  0  0  0  0  0  0  0]
[ 0 -6  0  0  0  0  0  0  0  0]
[-6  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  6]
[ 0  0  0  0  0  0  0  0  6  0]
[ 0  0  0  0  0  0  6  0  0  0]
```

```
[ 0  0  0  0  0  0  0  6  0  0]
[ 0  0  0  0  0  6  0  0  0  0]
[ 0  0  0  0  6  0  0  0  0  0]
```

`IsAbelian(L)`

Given a Lie algebra L, return `true` if L is abelian.

`IsSoluble(L)`
`IsSolvable(L)`

Given a Lie algebra L, return `true` if L is soluble.

`IsNilpotent(L)`

Given a Lie algebra L, return `true` if L is nilpotent.

`IsCentral(L, M)`

Given a subalgebra M of the Lie algebra L, return `true` if M is central in L.

`IsSimple(L)`

Given a Lie algebra L, return `true` if L is simple.

`IsSemisimple(L)`

Given a Lie algebra L, return `true` if L is semisimple.

`IsReductive(L)`

Given a Lie algebra L, return `true` if L is reductive.

`HasLeviSubalgebra(L)`

Given a Lie algebra L, this function determines whether L has a Levi subalgebra. If the result is `true`, then the function also returns a semisimple subalgebra (complement to the solvable radical) of L. If L is defined over a field of characteristic 0, then it always has a Levi subalgebra. However, if L is a Lie algebra of characteristic $p > 0$ then L need not have a Levi subalgebra but the function will always find one if it exists.

A description of the algorithm used is contained in [dG00], §4.13.

`IsClassicalType(L)`

Determines if the reductive Lie algebra L is of classical-type. Note that all reductive Lie algebras over fields of characteristic 0 are considered to be classical-type.

Example H100E44

We test various predicates in the context of the simple Lie algebra of type D_3 over the rational field.

```
> L:=LieAlgebra("D3",RationalField());
> L;
Lie Algebra of dimension 15 with base ring Rational Field
> K:=sub< L | [L.1,L.2,L.3] >;
> M:=Centralizer(L, K);
> M;
Lie Algebra of dimension 4 with base ring Rational Field
> R:=SolvableRadical(M);
> R;
Lie Algebra of dimension 4 with base ring Rational Field
> HasLeviSubalgebra(M);
true Lie Algebra of dimension 0 with base ring Rational Field
> K:=Centralizer(L, sub< L | [L.1,L.2,L.3] >);
> K;
Lie Algebra of dimension 4 with base ring Rational Field
> IsSolvable(K);
true
> IsNilpotent(K);
false
> R:= SolvableRadical(K);
> IsSolvable(R);
true
> IsNilpotent(R);
true
> N:= Nilradical(K);
> IsNilpotent(N);
true
```

100.11 Operations on Elements

`x + y` `x - y` `x * y`

`IsCentral(L, M)`

Given an element x of the Lie algebra L, return `true` if x is central in L.

`NonNilpotentElement(L)`

Given a (structure constant) Lie algebra L, this function returns an element of L that is *not* nilpotent, or the zero element of L if no such element exists.

The algorithm follows [dG00], §2.7.

Example H100E45

We construct a non-nilpotent element of a Lie algebra.

```
> L:=LieAlgebra("G2",RationalField());
> NonNilpotentElement(L);
(0 0 0 0 0 1 0 0 0 0 0 0 0 0)
```

`AdjointMatrix(L, x)`

`RightAdjointMatrix(L, x)`

Given a (structure constant) Lie algebra L and an element x of a subalgebra or ideal of L, return the matrix of $\mathrm{ad}x$ as an element of a matrix Lie algebra.

Example H100E46

```
> L:=LieAlgebra("B2",RationalField());
> AdjointMatrix(L, L.1);
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 1  0  0  0  0  0  0  0  0  0]
[ 2  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0 -1  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  0 -1  0  0  0  0]
```

100.11.1 Indexing

`a[i]`

If a is an element of a structure constant Lie algebra L of dimension n and $1 \leq i \leq n$ is a positive integer, then the i-th component of the element a is returned (as an element of the base ring R of L).

If a is an element of a matrix Lie algebra L of degree n and $1 \leq i \leq n$ then the ith row of the matrix a is returned.

`a[i] := r`

Given an element a belonging to a structure constant Lie algebra of dimension n over R, a positive integer $1 \leq i \leq n$ and an element $r \in R$, the i-th component of the element a is redefined to be r.

If a is an element of a matrix Lie algebra L of degree n over R and $1 \leq i \leq n$, the ith row of the matrix a is redefined to be the vector r over R.

`a[i, j]`

`a[i, j] := r`

For an element a of a matrix Lie algebra L of degree n and integers $1 \leq i, j \leq n$ return the element in the ith row and jth column of a or set this element to be r where r is an element of the coefficient ring of L.

100.12 The Natural Module

`Module(L)`

The module R^n underlying the Lie algebra L.

`RModule(L)`

The module R^n acted on by the matrix Lie algebra L.

`BaseModule(L)`

The space R^n acted on by the matrix Lie algebra L.

`Degree(L)`

The degree of the Lie algebra L. If L is a structure constant algebra, this is just the dimension of L. If L is a matrix Lie algebra, this is the degree of the matrices in L.

`Degree(a)`

Given an element a belonging to the Lie algebra L, the dimension of L is returned.

`ElementToSequence(a)`

`Eltseq(a)`

The sequence of coefficients of the Lie element a.

`Coordinates(M, a)`

Let a be an element of a Lie algebra L and let M be a subalgebra of L containing a. This function returns the coefficients of a with respect to the basis of L.

`InnerProduct(a, b)`

The (Euclidean) inner product of the coefficient vectors of a and b, where a and b are elements of some Lie algebra.

`Support(a)`

The support of the Lie algebra element a; i.e. the set of indices of the non-zero components of a.

100.13 Operations for Matrix Lie Algebras

This section describes the functionality provided for matrix Lie algebras which is additional to that provided for structure constant Lie algebras. For further information see Chapter 83.

`BaseModule(M)`

The natural module on which the matrix Lie algebra M acts.

`Generic(M)`

The full matrix algebra in which the matrix Lie algebra M is naturally embedded.

`Kernel(X)`
`Nullspace(X)`

The kernel of the homomorphism represented by the Lie matrix algebra element X.

`NullspaceOfTranspose(X)`
`RowNullSpace(X)`

The row nullspace of the homomorphism represented by the Lie matrix algebra element X.

100.14 Homomorphisms

`hom< L -> M | Q >`

Given a (structure constant) Lie algebra L of dimension n over R and either a Lie algebra M over R or a module M over R, the homomorphism from L to M specified by Q is constructed. The sequence Q may be of the form $[b_1, \ldots, b_n]$, $b_i \in B$, indicating that the i-th basis element of L is mapped to b_1 or of the form $[< a_1, b_1 >, \ldots, < a_n, b_n >]$ indicating that a_i maps to b_i, where the $a_i (1 \leq i \leq n)$ must form a basis of L.

Note that this is in general only a module homomorphism, and no check is made for it being an algebra homomorphism.

100.15 Automorphisms of Classical-type Reductive Algebras

`IdentityAutomorphism(L)`

The trivial automorphism of the Lie algebra L.

`InnerAutomorphism(L, x)`

The inner automorphism of the Lie algebra L induced by x, where x is an element of the corresponding group of Lie type.

`InnerAutomorphismGroup(L)`

The group of Lie type G corresponding to the Lie algebra L. The map $G \to \mathrm{Aut}(L)$ is returned as second value.

`DiagonalAutomorphism(L, v)`

The diagonal automorphism of the Lie algebra L induced by the vector v.

`GraphAutomorphism(L, p)`

`DiagramAutomorphism(L, p)`

`SimpleSigns` ANY *Default :* 1

The graph automorphism of the Lie algebra L induced by the permutation p. This must be either a permutation of the indices of the simple roots, or a permutation of the indices of all roots.

The optional parameter `SimpleSigns` can be used to specify the signs corresponding to each simple root. This should either be a sequence of integers ± 1, or a single integer ± 1.

Example H100E47

We construct an automorphism of order three for the simple Lie algebra of type D_4.

```
> DynkinDiagram( "D4" );
D4    3
     /
1 - 2
     \
      4
> p:= Sym(4)!(1,3,4);
> L:= LieAlgebra( "D4", Rationals() );
> f:= GraphAutomorphism( L, p );
> f(L.3);
(0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> f(L.4);
(0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> f(L.5);
(0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

100.16 Restrictable Lie Algebras

A restricted Lie algebra is a Lie algebra over a field of characteristic $p > 0$, equipped with a restriction map $x \to x^p$, satisfying the axioms given in [Jac62]. A restrictable Lie algebra is a Lie algebra which can be equipped with a restriction map. A Lie algebra is restrictable if and only if ad$L$ is closed under the pth power map. Hence restrictable Lie algebras have a standard restriction map induced by the adjoint representation. For many purposes, it suffices to know that a Lie algebra is restrictable, without needing to know a restriction.

By convention, a Lie algebra over a field of characteristic zero is always considered restrictable, and the restriction map is the identity map.

In MAGMA, we do not make a distinction between the concepts of restricted and restrictable. Note however that a Lie algebra can have a nonstandard restriction map.

`IsRestrictable(L)`
`IsRestricted(L)`
`IspLieAlgebra(L)`

Returns `true` if, and only if, the Lie algebra L is restrictable. If L is restrictable, the restriction map is returned as a second value.

`RestrictionMap(L)`
`pMap(L)`

The restriction map of the Lie algebra L. If L is not restrictable, an error is signalled.

Example H100E48

```
> L:= LieAlgebra( "A2", GF(5) );
> IsRestrictable( L );
true Mapping from: AlgLie: L to AlgLie: L given by a rule [no inverse]
> pmap:= pMap( L );
> pmap( 2*L.3 + L.4);
(0 0 0 1 0 0 0 0)
```

`RestrictedSubalgebra(Q)`
`pSubalgebra(Q)`

Given a sequence Q of elements from the Lie algebra L, the function returns the restricted subalgebra generated by the elements of Q, i.e., the smallest subalgebra containing Q which is also closed under the restriction map. If the parent of Q is not restrictable, an error is signalled.

`pClosure(L, M)`

Given Lie algebras L and M such that $L \leq M$, this function returns the closure of L under the restriction map of M. If L is not a subalgebra of M or M is not restrictable, an error is signalled.

`IsRestrictedSubalgebra(L, M)`

`IspSubalgebra(L, M)`

Return `true` if and only if the Lie algebra L is a restricted Lie subalgebra of M with the same restriction map. Note that if L is constructed using the `pClosure` intrinsic, this will always be true. However if L is constructed as a subalgebra, this may be false even if L is restrictable, since the restriction map of L will be the standard map rather than the restriction map of M.

`pQuotient(L, M)`

Given Lie algebras L and M such that $L \leq M$, this function returns the quotient of L by the p-closure of the Lie algebra M, with respect to the inherited restriction map.

`JenningsLieAlgebra(G)`

Let G be a p-group. Then the quotients of the successive terms of the Jennings series of G can be viewed as vector spaces over the field of p elements. The direct sum of these vector spaces carries the structure of a Lie algebra (coming from the commutator of G). This function returns two values. Firstly, the Lie algebra constructed from G by this process. This Lie algebra is graded. The second returned value is a sequence of sequences of two elements. The first element is the degree of a homogeneous component while the second element is its dimension. The basis elements of the Lie algebra are ordered according to increasing degree. This means that from the dimensions of the homogeneous components it is possible to derive the degree of each basis element.

Lie algebras constructed in this way are naturally restricted. Moreover, if x is a homogeneous element of degree d, then the p-th power image of x is homogeneous of degree pd.

Example H100E49

```
> G:= SmallGroup( 3^6, 196 );
> L, gr:= JenningsLieAlgebra( G );
> L;
Lie Algebra of dimension 6 with base ring GF(3)
> gr;
[
    [ 1, 3 ],
    [ 2, 1 ],
    [ 3, 2 ]
```

```
]
// So the first three basis elements are of degree 1,
// the fourth basis element is of degree 2, and so on.
> pmap:= pMap( L );
> pmap( L.1 );
(0 0 0 0 1 1)
```

100.17 Universal Enveloping Algebras

This section describes the functionality for universal enveloping algebras of Lie algebras. If a Lie algebra is semisimple and defined over a field of characteristic 0, then it is possible to write down an integral basis of the universal enveloping algebra that has nice properties. To accommodate this possibility, two constructions of a universal enveloping algebra are provided: a general construction, and one in which this integral basis is used. First we briefly describe the theoretical background behind universal enveloping algebras.

In MAGMA, universal enveloping algebras have type `AlgUE` and their elements have type `AlgUEElt`. Integral universal enveloping algebras have type `AlgIUE` and their elements have type `AlgIUEElt`. General algebras having a PBW basis (see below) have type `AlgPBW` (elements type `AlgPBWElt`) which inherit from types `Alg` and `Rng`. Consequently, the type `AlgIUE` inherits from `AlgPBW`.

100.17.1 Background

100.17.1.1 Universal Enveloping Algebras

Let L be a Lie algebra over the field F having basis $x_1, \ldots, x_n$. The universal enveloping algebra $U(L)$ of L is the associative algebra with identity, generated by n symbols which are also denoted by $x_1, \ldots, x_n$. These generators satisfy the relations

$$x_j x_i - x_i x_j = [x_j, x_i], 1 \leq i, j \leq n > .$$

Here $[x_j, x_i]$ is the product in the Lie algebra L, so it is a certain linear combination of the x_k.

The theorem of Poincaré-Birkhoff-Witt states that a basis of $U(L)$ is formed by the set of all elements

$$x_1^{k_1} \cdots x_n^{k_n},$$

where the k_i are non-negative integers. Furthermore, the product of two such basis elements may be rewritten as a linear combination of basis elements using the defining relations $x_j x_i - x_i x_j = [x_j, x_i]$ for $j > i$.

100.17.1.2 The Integral Form of a Universal Enveloping Algebra

If the Lie algebra L happens to be (split) semisimple and of characteristic 0, then the universal enveloping algebra has a nice basis described by [Kos66]. The first step in constructing this basis involves taking a Chevalley basis of L, consisting of the elements $y_1, \ldots, y_s$, $h_1, \ldots, h_r$ and $x_1, \ldots, x_s$. Here the y_i and x_i are root vectors belonging to negative roots and positive roots, respectively. The h_i are basis elements of a Cartan subalgebra. In the universal enveloping algebra we use the divided powers

$$y_i^{(n)} = \frac{y_i^n}{n!}, \quad x_i^{(n)} = \frac{x_i^n}{n!},$$

and the binomials

$$\binom{h_i}{k} = \frac{h_i(h_i-1)\cdots(h_i-k+1)}{k!}.$$

A basis of $U(L)$ is formed by the elements

$$y_1^{(m_1)} \cdots y_s^{(m_s)} \binom{h_1}{k_1} \cdots \binom{h_r}{k_r} x_1^{(n_1)} \cdots x_s^{(n_s)}.$$

This basis has the useful property that if we multiply two basis elements, the structure constants will be integers (usually of quite moderate size). So this is a basis of an integral form of the universal enveloping algebra.

100.17.2 Construction of Universal Enveloping Algebras

`UniversalEnvelopingAlgebra(L)`

This creates the universal enveloping algebra U of the Lie algebra L. Here the i-th basis element of L (i.e., `L.i`) corresponds to the i-th generator of U (i.e., `U.i`). Every product of generators is rewritten as a linear combination of Poincaré-Birkhoff-Witt monomials (cf. Section 100.17.1.1).

`IntegralUEA(L)`

`IntegralUEAlgebra(L)`

`IntegralUniversalEnvelopingAlgebra(L)`

Given a semisimple Lie algebra L of characteristic 0, create the integral universal enveloping algebra U of L. The basis described in Section 100.17.1.2 is used.

Let x, y and h denote the output of `ChevalleyBasis(L)`. Let s be the length of x, and r the length of h. Then every generator of U corresponds to an element of x, y or h. If $1 \leq i \leq s$ then the i-th generator of U (i.e., `U.i`) corresponds to the i-th element of y. It is printed as `y_i`. If $s+1 \leq i \leq s+r$, then the i-th generator of U corresponds to the k-th element of h, where $k = i - s$. It is printed as `[ h_k ; 1 ]` (i.e., `h_k` choose 1). Finally, if $s+r+1 \leq i \leq 2s+r$, then the i-th generator corresponds to the k-th element of x, where $k = i - s - r$. It is printed as `x_k`.

Using this form of the universal enveloping algebra has two advantages. Firstly, the structure constants are integers which usually remain relatively small. Secondly, multiplication of elements is, in general, much faster than is the case with universal enveloping algebras that employ PBW bases.

Example H100E50

```
> T:= [ <4,1,1,1>, <1,4,1,-1>, <4,1,3,1>, <1,4,3,-1>, <4,2,2,1>, <2,4,2,-1>,
> <4,3,1,1>, <3,4,1,-1>, <3,1,2,1>, <1,3,2,-1> ];
> L:= LieAlgebra< Rationals(), 4 | T >;
> U:= UniversalEnvelopingAlgebra(L);
> U.4*U.1;
x_1*x_4 + x_1 + x_3
> L:= LieAlgebra("F4", Rationals());
> U:= IntegralUEA(L);
> U.29*U.1;
y_1*x_1 + [ h_1 ; 1 ]
> (1/4)*U.29^2*U.1^2;
y_1^(2)*x_1^(2) + y_1*[ h_1 ; 1 ]*x_1 - 2*y_1*x_1 + [ h_1 ; 2 ]
```

In the last example we divided by 4 because `U.29^2 = 2 U.29^(2)`, and likewise for `U.1^2`.

`AssignNames(~U, Q)`

Assign the names in the sequence Q to the generators of the algebra U.

`ChangeRing(U, S)`

Given a universal enveloping algebra U with base ring R, together with a ring S, construct the algebra U' with base ring S obtained by coercing the coefficients of elements of U into S.

100.17.3 Related Structures

`CoefficientRing(U)`

`BaseRing(U)`

The ring of coefficients of the universal enveloping algebra U.

`Algebra(U)`

The Lie algebra corresponding to the universal enveloping algebra U.

100.17.4 Elements of Universal Enveloping Algebras

Most functions in this section are applicable both to universal enveloping algebras and to integral universal enveloping algebras. Therefore, they are only documented once. An exception is the function `HBinomial`, which is only applicable to integral universal enveloping algebras.

100.17.4.1 Creation of Elements

`U ! 0`

`Zero(U)`

The zero element of the universal enveloping algebra U.

`U ! 1`

`One(U)`

The identity element of the universal enveloping algebra U.

`U . i`

The i-th generator of the universal enveloping algebra U.

`U ! r`

Returns r as an element of the enveloping algebra U where r may be anything coercible into the coefficient ring of U or an element of another enveloping algebra of the same type as U whose coefficients can be coerced into the coefficient ring of U.

`HBinomial(U, i, n)`

`HBinomial(h, n)`

This function is applicable only in the case of integral universal enveloping algebras. It is used for constructing the "binomial" elements h_i choose n. In the first form U is an integral universal enveloping algebra, and i is an index between 1 and the rank of the root datum. In the second form, the element h is simply `U.(s+i)`, where s is the number of positive roots.

Example H100E51

```
> L:= LieAlgebra("E6",Rationals());
> U:= IntegralUEA(L);
> HBinomial(U, 4, 10);
[ h_4 ; 10 ]
```

100.17.4.2 Operations on Elements

`x + y` `x - y` `x * y` `c * x` `x * c` `x ^ n`

`Monomials(u)`

The sequence of the monomials that occur in the element u of a universal enveloping algebra.

`Coefficients(u)`

The sequence of coefficients of the monomials in the element u of a universal enveloping algebra. The k-th element of this sequence corresponds exactly to the k-th monomial in the sequence returned by `Monomials(u)`.

`Degree(u, i)`

Given an element u of a universal enveloping algebra U and an integer i, this function returns the degree of u in the i-th generator of U.

Example H100E52

```
> L:= LieAlgebra("G2",Rationals());
> U:= IntegralUEA(L);
> c:= U.7*U.2;c;
y_2*[ h_1 ; 1 ] + 3*y_2
> Monomials(c);
[
    y_2*[ h_1 ; 1 ],
    y_2
]
> Coefficients(c);
[ 1, 3 ]
> c:= U.10*U.7*U.2; c;
y_2*[ h_1 ; 1 ]*x_2 + 6*y_2*x_2 + [ h_1 ; 1 ]*[ h_2 ; 1 ] + 3*[ h_2 ; 1 ]
> Degree(c, 2);
1
> Degree(c, 7);
1
> Degree(c, 8);
1
```

100.18 Solvable and Nilpotent Lie Algebras Classification

This section describes functions for working with the classification of solvable Lie algebras of dimension 2, 3, and 4, and the classification of nilpotent Lie algebras having dimensions 3,4,5, and 6. The classification of solvable Lie algebras is taken from [dG05], and applies to algebras over any base field. The classification of nilpotent Lie algebras is taken from [dG07]. It lists the nilpotent Lie algebras over any base field, with the exception of fields of characteristic 2, when the dimension is 6.

The functions described here fall into two categories: functions for creating the Lie algebras of the classification, and a function for identifying a given solvable Lie algebra of dimension 2,3,4 or a given nilpotent Lie algebra of dimension 3,4,5,6 as a member of the list.

First we describe the classifications, in order to define names for the Lie algebras that occur. We then describe the functions for working with them in MAGMA.

100.18.1 The List of Solvable Lie Algebras

We denote a solvable Lie algebra of dimension n by L_n^k, where k ranges between 1 and the number of classes of solvable Lie algebras of dimension n. If the class depends on a parameter, say a, then we denote the Lie algebra by $L_n^k(a)$. In such cases we also state conditions under which $L_n^k(a)$ is isomorphic to $L_n^k(b)$ (if there are any). We list the nonzero commutators only. The field over which the Lie algebra is defined is denoted by F. Here is the list of classes of solvable Lie algebras having dimension not greater than 4:

L_2^1 Abelian.

L_2^2 $[x_2, x_1] = x_1$.

L_3^1 Abelian.

L_3^2 $[x_3, x_1] = x_1$, $[x_3, x_2] = x_2$.

$L_3^3(a)$ $[x_3, x_1] = x_2$, $[x_3, x_2] = ax_1 + x_2$.

$L_3^4(a)$ $[x_3, x_1] = x_2$, $[x_3, x_2] = ax_1$. Condition of isomorphism: $L_3^4(a) \cong L_3^4(b)$ if and only if there is an $\alpha \in F^*$ with $a = \alpha^2 b$.

L_4^1 Abelian.

L_4^2 $[x_4, x_1] = x_1$, $[x_4, x_2] = x_2$, $[x_4, x_3] = x_3$.

$L_4^3(a)$ $[x_4, x_1] = x_1$, $[x_4, x_2] = x_3$, $[x_4, x_3] = -ax_2 + (a+1)x_3$.

L_4^4 $[x_4, x_2] = x_3$, $[x_4, x_3] = x_3$.

L_4^5 $[x_4, x_2] = x_3$.

$L_4^6(a, b)$ $[x_4, x_1] = x_2$, $[x_4, x_2] = x_3$, $[x_4, x_3] = ax_1 + bx_2 + x_3$.

$L_4^7(a, b)$ $[x_4, x_1] = x_2$, $[x_4, x_2] = x_3$, $[x_4, x_3] = ax_1 + bx_2$. Isomorphism condition: $L_4^7(a, b) \cong L_4^7(c, d)$ if and only if there is an $\alpha \in F^*$ with $a = \alpha^3 c$ and $b = \alpha^2 d$.

L_4^8 $[x_1, x_2] = x_2$, $[x_3, x_4] = x_4$.

$L_4^9(a)$ $[x_4, x_1] = x_1 + ax_2$, $[x_4, x_2] = x_1$, $[x_3, x_1] = x_1$, $[x_3, x_2] = x_2$. Condition on the parameter a: $T^2 - T - a$ has no roots in F. Isomorphism condition: $L_4^9(a) \cong L_4^9(b)$ if and only if the characteristic of F is not 2 and there is an $\alpha \in F^*$ with $a + \frac{1}{4} = \alpha^2(b + \frac{1}{4})$, or the characteristic of F is 2 and $X^2 + X + a + b$ has roots in F.

$L_4^{10}(a)$ $[x_4, x_1] = x_2$, $[x_4, x_2] = ax_1$, $[x_3, x_1] = x_1$, $[x_3, x_2] = x_2$. Condition on F: the characteristic of F is 2. Condition on the parameter a: $a \notin F^2$. Isomorphism condition: $L_4^{10}(a) \cong L_4^{10}(b)$ if and only if $Y^2 + X^2 b + a$ has a solution $(X, Y) \in F \times F$ with $X \neq 0$.

$L_4^{11}(a, b)$ $[x_4, x_1] = x_1$, $[x_4, x_2] = bx_2$, $[x_4, x_3] = (1 + b)x_3$, $[x_3, x_1] = x_2$, $[x_3, x_2] = ax_1$. Condition on F: the characteristic of F is 2. Condition on the parameters a, b: $a \neq 0$, $b \neq 1$. Isomorphism condition: $L_4^{11}(a, b) \cong L_4^{11}(c, d)$ if and only if $\frac{a}{c}$ and $(\delta^2 + (b + 1)\delta + b)/c$ are squares in F, where $\delta = (b + 1)/(d + 1)$.

L_4^{12} $[x_4, x_1] = x_1$, $[x_4, x_2] = 2x_2$, $[x_4, x_3] = x_3$, $[x_3, x_1] = x_2$.

$L_4^{13}(a)$ $[x_4, x_1] = x_1 + ax_3$, $[x_4, x_2] = x_2$, $[x_4, x_3] = x_1$, $[x_3, x_1] = x_2$.

$L_4^{14}(a)$ $[x_4, x_1] = ax_3$, $[x_4, x_3] = x_1$, $[x_3, x_1] = x_2$. Condition on parameter a: $a \neq 0$. Isomorphism condition: $L_4^{14}(a) \cong L_4^{14}(b)$ if and only if there is an $\alpha \in F^*$ with $a = \alpha^2 b$.

100.18.2 Comments on the Classification over Finite Fields

Over general fields the lists are not "precise" in the sense that some classes that depend up on a parameter have an associated isomorphism condition, but not a precise parametrization of the Lie algebras in that class. However, for algebras over finite fields we are able to give a precise list, by restricting the parameter values in some cases. In this section we describe how this is done. Here F will be a finite field of size q with primitive root γ.

* If the characteristic of F is 2, then there are two algebras of type $L_3^4(a)$, namely $L_3^4(0)$ and $L_3^4(1)$. If the characteristic is not 2, then there are three algebras of this type, $L_3^4(0)$, $L_3^4(1)$, $L_3^4(\gamma)$.

* The class $L_4^7(a, b)$ splits into three classes: $L_4^7(a, a)$ $(a \in F)$, $L_4^7(a, 0)$ $(a \neq 0)$, $L_4^7(0, b)$ $(b \neq 0)$. Among the algebras of the first class there are no isomorphisms. However, for the other two classes we have the following:-

(i) $L_4^7(a, 0) \cong L_4^7(b, 0)$ if and only if there is an $\alpha \in F^*$ such that $a = \alpha^3 b$. If $q \equiv 1 \bmod 3$, then exactly a third of the elements of F^* are cubes, namely the γ^i with i divisible by 3. So in this case we get three algebras, $L_4^7(1, 0)$, $L_4^7(\gamma, 0)$, $L_4^7(\gamma^2, 0)$. If $q \not\equiv 1 \bmod 3$ then $F^3 = F$, and hence there is only one algebra, namely $L_4^7(1, 0)$.

(ii) $L_4^7(0, a) \cong L_4^7(0, b)$ if and only if there is an $\alpha \in F^*$ such that $a = \alpha^2 b$. So if q is even then we get one algebra, $L_4^7(0, 1)$. If q is odd we get two algebras, $L_4^7(0, 1)$, $L_4^7(0, \gamma)$.

* In [dG05] it is shown that there is only one Lie algebra in the class $L_4^9(a)$. We let e be the smallest positive integer such that $T^2 - T - \gamma^e$ has no roots in F. Then we take the Lie algebra $L_4^9(\gamma^e)$ as representative of the class.
* Over a finite field of characteristic 2 there are no Lie algebras of type $L_4^{10}(a)$, as $F^2 = F$ in that case.
* There is only one Lie algebra of type $L_4^{11}(a, b)$ over a field of characteristic 2, namely $L_4^{11}(1, 0)$.
* If q is even then there is only one algebra of type $L_4^{14}(a)$, namely $L_4^{14}(1)$. If q is odd, then there are two algebras, $L_4^{14}(1)$ and $L_4^{14}(\gamma)$.

100.18.3 The List of Nilpotent Lie Algebras

We denote a nilpotent Lie algebra of dimension r by N_r^k, where k ranges between 1 and the number of classes of nilpotent Lie algebras of dimension r. If the class depends on a parameter, say a, then we denote the Lie algebra by $N_r^k(a)$. The complete list of isomorphism classes of nilpotent Lie algebras having dimensions 3, 4, 5 and 6, *where in dimension 6 we exclude base fields of characteristic 2* are as follows:

N_3^1 Abelian.
N_3^2 $[x_1, x_2] = x_3$.
N_4^1 Abelian.
N_4^2 $[x_1, x_2] = x_3$.
N_4^3 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$.
N_5^1 Abelian.
N_5^2 $[x_1, x_2] = x_3$.
N_5^3 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$.
N_5^4 $[x_1, x_2] = x_5$, $[x_3, x_4] = x_5$.
N_5^5 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_2, x_4] = x_5$.
N_5^6 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_2, x_3] = x_5$.
N_5^7 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$.
N_5^8 $[x_1, x_2] = x_4$, $[x_1, x_3] = x_5$.
N_5^9 $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_2, x_3] = x_5$.

There are nine 6-dimensional nilpotent Lie algebras denoted N_6^k for $k = 1, \ldots, 9$ which are the direct sum of N_5^k and a 1-dimensional abelian ideal. Consequently, we get the following Lie algebras:-

N_6^{10} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_6$, $[x_4, x_5] = x_6$.
N_6^{11} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_6$, $[x_2, x_3] = x_6$, $[x_2, x_5] = x_6$.
N_6^{12} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_6$, $[x_2, x_5] = x_6$.
N_6^{13} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_2, x_4] = x_5$, $[x_1, x_5] = x_6$, $[x_3, x_4] = x_6$.

N_6^{14} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_2, x_3] = x_5$, $[x_2, x_5] = x_6$, $[x_3, x_4] = -x_6$.
N_6^{15} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_2, x_3] = x_5$, $[x_1, x_5] = x_6$, $[x_2, x_4] = x_6$.
N_6^{16} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_2, x_5] = x_6$, $[x_3, x_4] = -x_6$.
N_6^{17} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_1, x_5] = x_6$, $[x_2, x_3] = x_6$.
N_6^{18} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_1, x_4] = x_5$, $[x_1, x_5] = x_6$.
$N_6^{19}(a)$ $[x_1, x_2] = x_4$, $[x_1, x_3] = x_5$, $[x_2, x_4] = x_6$, $[x_3, x_5] = ax_6$.
N_6^{20} $[x_1, x_2] = x_4$, $[x_1, x_3] = x_5$, $[x_1, x_5] = x_6$, $[x_2, x_4] = x_6$.
$N_6^{21}(a)$ $[x_1, x_2] = x_3$, $[x_1, x_3] = x_4$, $[x_2, x_3] = x_5$, $[x_1, x_4] = x_6$, $[x_2, x_5] = ax_6$.
$N_6^{22}(a)$ $[x_1, x_2] = x_5$, $[x_1, x_3] = x_6$, $[x_2, x_4] = ax_6$, $[x_3, x_4] = x_5$.
N_6^{23} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_1, x_4] = x_6$, $[x_2, x_4] = x_5$.
$N_6^{24}(a)$ $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_1, x_4] = ax_6$, $[x_2, x_3] = x_6$,$[x_2, x_4] = x_5$.
N_6^{25} $[x_1, x_2] = x_3$, $[x_1, x_3] = x_5$, $[x_1, x_4] = x_6$.
N_6^{26} $[x_1, x_2] = x_4$, $[x_1, x_3] = x_5$, $[x_2, x_3] = x_6$.

Note that for all classes that depend on a parameter a, the Lie algebra with parameter a is isomorphic to the Lie algebra (from the same class) with parameter b if and only if there is an $\alpha \in F^*$ with $a = \alpha^2 b$.

100.18.4 Intrinsics for Working with the Classifications

`SolvableLieAlgebra( F, n, k :` *parameters*`)`

This function returns the solvable Lie algebra L_n^k over the field F. The multiplication table is exactly the same as that given in the classification of solvable algebras above, where the basis element x_i corresponds to the i-th basis element of the Lie algebra returned.

`pars` SeqEnum *Default :* []

If the Lie algebra L_n^k depends on one or more parameters, then the parameter `pars` specifies the parameter values corresponding to the Lie algebra which is required.

Example H100E53

```
> F<a>:= RationalFunctionField( Rationals() );
> K:= SolvableLieAlgebra( F, 3, 3 : pars:= [a] );
> K.3*K.1;
(0 1 0)
> K.3*K.2;
(a 1 0)
```

`NilpotentLieAlgebra( F, r, k : parameters)`

This function returns the nilpotent Lie algebra N_r^k over the field F. The multiplication table is exactly the same as that given in the classification of nilpotent algebras above, where the basis element x_i corresponds to the i-th basis element of the Lie algebra returned.

`pars` SeqEnum *Default :* []

If the Lie algebra N_r^k depends upon one or more parameters, then the parameter `pars` specifies the parameter values corresponding to the Lie algebra which is required.

Example H100E54

```
> F<a>:= RationalFunctionField( Rationals() );
> K:= NilpotentLieAlgebra( F, 6, 19 : pars:= [a^3] );
> K.3*K.5;
(  0   0   0   0   0 a^3)
```

`AllSolvableLieAlgebras(F, d)`

Given a finite field F and d an integer equal to 2, 3 or 4, this function returns a sequence containing all solvable Lie algebras of dimension d over the field F.

`AllNilpotentLieAlgebras(F, d)`

Given a finite field F and d an integer equal to 3, 4, 5 or 6, this function returns a sequence containing all nilpotent Lie algebras of dimension d over the field F. If the dimension is 6 then the characteristic of F may not be 2.

`IdDataSLAC(L)`

Given a solvable Lie algebra L of dimension 2, 3, or 4, this function returns data that identifies L with the isomorphic algebra in the classification of solvable Lie algebras. (SLAC stands for Solvable Lie Algebras Classification.) Three objects are returned: a string, a sequence and a map.

The string gives the name of the Lie algebra as it occurs in the classification, with information about the field and the parameters.

The sequence contains the parameters of the Lie algebra in the classification to which L is isomorphic.

The map is an isomorphism from L to the corresponding Lie algebra contained in the classification.

`IdDataNLAC(L)`

Given a nilpotent Lie algebra L of dimension 3, 4, 5 or 6 this function returns data that identifies L with the isomorphic algebra in the classification of nilpotent Lie algebras. (NLAC stands for Nilpotent Lie Algebras Classification.) Three objects are returned: a string giving the name of the algebra N in the classification, a sequence giving the parameters for N, and the isomorphism mapping L to N.

`MatrixOfIsomorphism(f)`

Given an isomorphism f as returned by either `IdDataSLAC` or `IdDataNLAC`, this function returns the matrix of that isomorphism. The row convention is used, i.e., the i-th row contains the coordinates of the image of the i-th basis element of the domain of f.

Example H100E55

We define a solvable Lie algebra of dimension 4 that depends on a parameter a. We identify this Lie algebra as a member of the classification.

```
> F<a>:= RationalFunctionField( Rationals() );
> T:= [ <1,2,2,1>, <1,2,3,a>, <1,4,4,a>, <2,1,2,-1>, <2,1,3,-a>, <4,1,4,-a> ];
> L:= LieAlgebra< F, 4 | T >;
> s,p,f:= IdDataSLAC( L );
> s;
L4_6( Univariate rational function field over Rational Field
Variables: a, 0, -a/(a^2 + 2*a + 1) )
> p;
[
    0,
    -a/(a^2 + 2*a + 1)
]
> MatrixOfIsomorphism( f );
[0   (-a - 1)/(a - 1)   (-a - 1)/(a - 1)   (a^2 + 2*a + 1)/(a^2 - a)]
[0   -1/(a - 1)   -a/(a - 1)   (a + 1)/(a - 1)]
[0   -1/(a^2 - 1)   -a/(a^2 - 1)   a/(a - 1)]
[1/(a + 1)   0   0   0]
```

So generically, the Lie algebra is isomorphic to $L_4^6(0, -a/(a^2+2a+1))$. We see that the parameters are not defined if $a = -1$. Furthermore, the isomorphism is not defined if $a = \pm 1$, or $a = 0$. We investigate those cases.

```
> a:= 1;
> T:= [ <1,2,2,1>, <1,2,3,a>, <1,4,4,a>, <2,1,2,-1>, <2,1,3,-a>, <4,1,4,-a> ];
> L:= LieAlgebra< Rationals(), 4 | T >;
> s,p,f:= IdDataSLAC( L );
> s;
L4_3( Rational Field, 0 )
> MatrixOfIsomorphism( f );
[ 0  1  1  0]
```

```
[ 0  0 -1  1]
[ 0  0  0  1]
[ 1  0  0  0]
> a:= -1;
> T:= [ <1,2,2,1>, <1,2,3,a>, <1,4,4,a>, <2,1,2,-1>, <2,1,3,-a>, <4,1,4,-a> ];
> L:= LieAlgebra< Rationals(), 4 | T >;
> s,p,f:= IdDataSLAC( L );
> s;
L4_7( Rational Field, 0, 1 )
> MatrixOfIsomorphism( f );
[   0  1/2  1/2  1/2]
[   0  1/2 -1/2 -1/2]
[   0  1/2 -1/2  1/2]
[   1    0    0    0]
> a:= 0;
> T:= [ <1,2,2,1>, <1,2,3,a>, <1,4,4,a>, <2,1,2,-1>, <2,1,3,-a>, <4,1,4,-a> ];
> L:= LieAlgebra< Rationals(), 4 | T >;
> s,p,f:= IdDataSLAC( L );
> s;
L4_4( Rational Field )
> MatrixOfIsomorphism( f );
[ 0  0  1  0]
[ 0  1  0 -1]
[ 0  1  0  0]
[ 1  0  0  0]
```

We see that for $a = 1$ the Lie algebra is isomorphic to $L_4^3(0)$, and the isomorphism is defined over any field. If $a = -1$, then the Lie algebra is isomorphic to $L_4^7(0, 1)$. However, the isomorphism is not defined if the characteristic of the field is 2. But then we are back in the case $a = 1$. Finally, for $a = 0$ the Lie algebra is isomorphic to L_4^4.

Example H100E56

The positive part of the simple Lie algebra of type G_2 is a nilpotent Lie algebra of dimension six. We identify it in the classification of nilpotent algebras, both in characteristic 0, and in characteristic 3.

```
> L:= LieAlgebra( "G2", Rationals() );
> x,y,h:= ChevalleyBasis( L );
> x;
[ (0 0 0 0 0 0 0 0 1 0 0 0 0 0), (0 0 0 0 0 0 0 0 0 0 1 0 0 0 0), (0 0 0
0 0 0 0 0 0 0 1 0 0 0), (0 0 0 0 0 0 0 0 0 0 0 1 0 0), (0 0 0 0 0 0 0
0 0 0 0 0 1 0), (0 0 0 0 0 0 0 0 0 0 0 0 0 0 1) ]
```

So we see that the positive part is spanned by basis vectors `L.i` with i=9, 10, 11, 12, 13 and 14. So

```
> K:= sub< L | [ L.i : i in [9,10,11,12,13,14] ] >;
> name,pp,f:= IdDataNLAC( K );
> name;
```

```
N6_16( Rational Field )
> MatrixOfIsomorphism( f );
[  1   0   0   0   0   0]
[  0   1   0   0   0   0]
[  0   0   1   0   0   0]
[  0   0   0 1/2   0   0]
[  0   0   0   0 1/6   0]
[  0   0   0   0   0 1/6]
> L:= LieAlgebra( "G2", GF(3) );
> K:= sub< L | [ L.i : i in [9,10,11,12,13,14] ] >;
> name,pp,f:= IdDataNLAC( K );
> name;
N6_19( Finite field of size 3, 0 )
```

We see that in characteristic 3 L is isomorphic to a Lie algebra from a different class.

100.19 Semisimple Subalgebras of Simple Lie Algebras

Here we describe the functions for working with the classification of the semisimple subalgebras of the simple Lie algebras. These subalgebras have been classified for the simple Lie algebras over the complex numbers, of ranks up to 8. They have been classified up to linear equivalence. Two subalgebras K_1, K_2 of a Lie algebra L are linearly equivalent if for every representation of L the induced representations of K_1, K_2 are equivalent. The basic function for dealing with them returns a directed graph, describing the inclusions among the subalgebras, and having the subalgebras as labels of the vertices. (We refer to [dG11] for the background details of this classification.)

`SubalgebrasInclusionGraph( t )`

Here t has to be a simple type of rank not exceeding 8. This function returns a directed graph G. The vertices of this graph are numbered from 1 to the number of semisimple subalgebras. Furthermore, the last vertex is numbered 0. A vertex has a label that is the semisimple subalgebra corresponding to it. The label of the last vertex (numbered 0), has the Lie algebra L of type t as its label. All other semisimple Lie algebras are subalgebras of this one.

The Lie algebra L (and its subalgebras) is defined over the rational numbers, or over a cyclotomic field. It is sometimes necessary to take an extension, because for some types not all subalgebras are defined over the rationals.

Moreover, in G there is an edge from the vertex with label K_1 to the vertex with label K_2 if and only if K_1 has a subalgebra that is linearly equivalent (as subalgebra of L) to K_2. We remark that it does not mean that K_2 is a subalgebra of K_1 (rather that it is linearly equivalent to a subalgebra of K_1).

Example H100E57

We consider the subalgebras of the Lie algebra of type C_3. We compute the types of its maximal subalgebras.

```
> G:= SubalgebrasInclusionGraph( "C3" );
> G;
Digraph
Vertex  Neighbours
1       ;
2       ;
3       ;
4       ;
5       ;
6       ;
7       ;
8       2 4 ;
9       5 10 ;
10      1 2 ;
11      1 2 3 ;
12      1 5 6 ;
13      3 4 6 ;
14      10 11 ;
15      9 12 14 ;
0       7 8 13 15 ;
> v:= Vertices(G);
> Label( v[10] );
Lie Algebra of dimension 6 with base ring Rational Field
> SemisimpleType( Label( v[7] ) );
A1
> SemisimpleType( Label( v[8] ) );
A2
> SemisimpleType( Label( v[13] ) );
A1 A1
> SemisimpleType( Label( v[15] ) );
A1 C2
```

`RestrictionMatrix( G, k )`

Here G is a subalgebras inclusion graph of the simple Lie algebra L, as output by the previous function, and k is a nonzero integer, corresponding to a vertex. This function returns the restriction matrix corresponding to L and the Lie algebra that is the label of the k-th vertex of G. This restriction matrix maps weights in a representation of L to weights of the subalgebra, and can be used to decompose a representation of L, as a representation of the subalgebra.

Example H100E58

We decompose the adjoint representation of the Lie algebra of type D_4, when viewed as a representation of its subalgebra of type G_2.

```
> G:= SubalgebrasInclusionGraph( "D4" );
> v:= Vertices(G);
> tt:= [ SemisimpleType( Label(a) ) : a in v ];
> Index( tt, "G2" );
17
> M:= RestrictionMatrix( G, 17 );
> R:= RootDatum( "D4" : Isogeny:= "SC" );
> S:= RootDatum( "G2" : Isogeny:= "SC" );
> D:= AdjointRepresentationDecomposition(R);
> E:= Branch( S, D, M );
> WeightsAndMultiplicities(E);
[
    (0 1),
    (1 0)
]
[ 1, 2 ]
```

100.20 Nilpotent Orbits in Simple Lie Algebras

Take a simple Lie algebra over the complex numbers, and consider the connected component of its automorphism group that contains the identity. This group acts on the Lie algebra, and there is interest in understanding the nature of its orbits. The nilpotent orbits for simple Lie algebras have been classified. We refer to the book by Collingwood and McGovern [CM93] for the details of this classification.

The main technical tools used for the classification are the weighted Dynkin diagram and the sl_2-triple. The weighted Dynkin diagram is the Dynkin diagram of the root system of the Lie algebra, with labels that can be 0, 1, 2. A nilpotent orbit is uniquely determined by its weighted Dynkin diagram. By the Jacobson-Morozov theorem a nilpotent element of a semisimple Lie algebra can be embedded (as nilpositive element) in an sl_2-triple. Now two nilpotent elements are conjugate (under the group) if and only if the corresponding sl_2-triples are conjugate. This yields a bijection between nilpotent orbits and conjugacy classes of simple subalgebras isomorphic to sl_2.

This section describes functions for working with the classification of nilpotent orbits in simple Lie algebras. One of the main invariants of a nilpotent orbit is its weighted Dynkin diagram. We represent such a diagram by a sequence of its labels; they are mapped to the nodes of the Dynkin diagram in the order determined by the Cartan matrix of the root datum. Also, an sl_2-triple is represented by a sequence $[f, h, e]$ of three elements of a Lie algebra; these satisfy the commutation relations $[h, e] = 2e$, $[h, f] = -2f$, $[e, f] = h$.

Throughout this section we consider orbits that are not the zero orbit.

`IsGenuineWeightedDynkinDiagram( L, wd )`

Given a simple Lie algebra L, and a sequence wd consisting of integers that are 0, 1, or 2, this function returns `true` if wd corresponds to a nilpotent orbit (in other words, if it is the weighted Dynkin diagram of a nilpotent orbit). If wd does corresponds to a nilpotent orbit, an sl_2-triple in L, such that the third element lies in the nilpotent orbit corresponding to the weighted Dynkin diagram is returned. If wd does not correspond to a nilpotent orbit. the second return value is a sequence consisting of three zeros of L.

Example H100E59

We can use this function to find the classification of the nilpotent orbits of a given Lie algebra. First we construct all possible weighted Dynkin diagrams, and then we remove those that do not correspond to an orbit.

```
> L:= LieAlgebra( RootDatum("D4"), Rationals() );
> [ w : i,j,k,l in [0,1,2] | IsGenuineWeightedDynkinDiagram(L, w)
>     where w := [i,j,k,l] ];
[
    [ 0, 0, 0, 2 ],
    [ 0, 0, 2, 0 ],
    [ 0, 1, 0, 0 ],
    [ 0, 2, 0, 0 ],
    [ 0, 2, 0, 2 ],
    [ 0, 2, 2, 0 ],
    [ 1, 0, 1, 1 ],
    [ 2, 0, 0, 0 ],
    [ 2, 0, 2, 2 ],
    [ 2, 2, 0, 0 ],
    [ 2, 2, 2, 2 ]
]
```

`NilpotentOrbit( L, wd )`

This returns the nilpotent orbit in the simple Lie algebra L with weighted Dynkin diagram given by the sequence wd. It is *not* checked whether the weighted Dynkin diagram really corresponds to a nilpotent orbit.

`NilpotentOrbit( L, e )`

This returns the nilpotent orbit in the simple Lie algebra L having representative e. Here e has to be a nilpotent element of the Lie algebra. This condition is not checked by the function.

Example H100E60

```
> L:= LieAlgebra( RootDatum("A2"), Rationals() );
> NilpotentOrbit( L, [2,2] );
Nilpotent orbit in Lie algebra of type A2
> NilpotentOrbit( L, L.1 );
Nilpotent orbit in Lie algebra of type A2
```

`NilpotentOrbits( L )`

Given a simple Lie algebra L, this function returns the sequence of all nilpotent orbits in the simple Lie algebra L.

Example H100E61

We compute the nilpotent orbits of the Lie algebra of type D_4, and observe that they are the same as those found in Example H100E59.

```
> L:= LieAlgebra( RootDatum("D4"), Rationals() );
> o:= NilpotentOrbits(L);
> [ WeightedDynkinDiagram(orb) : orb in o ];
[
    [ 2, 2, 2, 2 ],
    [ 2, 0, 2, 2 ],
    [ 2, 2, 0, 0 ],
    [ 0, 2, 0, 2 ],
    [ 0, 2, 2, 0 ],
    [ 0, 2, 0, 0 ],
    [ 1, 0, 1, 1 ],
    [ 2, 0, 0, 0 ],
    [ 0, 0, 0, 2 ],
    [ 0, 0, 2, 0 ],
    [ 0, 1, 0, 0 ]
]
```

`Partition( o )`

Here o is a nilpotent orbit in a simple Lie algebra of classical type (i.e., of type A_n, B_n, C_n or D_n). The nilpotent orbits for the Lie algebras of these types have been classified in terms of partitions. This function returns the partition corresponding to the orbit.

Example H100E62

```
> L:= LieAlgebra( RootDatum("D4"), Rationals() );
> orbs:= NilpotentOrbits( L );
> Partition( orbs[5] );
[ 4, 4 ]
> Partition( orbs[6] );
[ 3, 3, 1, 1 ]
```

`SL2Triple( o )`

Given a nilpotent orbit o in a simple Lie algebra L, this function returns an sl_2-triple, $[f, h, e]$ of elements of L, such that e lies in the nilpotent orbit.

`SL2Triple( L, e )`

Given a semisimple Lie algebra L of characteristic 0, and a nilpotent element e of L, this function returns an sl_2-triple $[f, h, e]$ of elements of L. It may also work for other Lie algebras, and in other characteristics, but this is not guaranteed.

`Representative( o )`

Given a nilpotent orbit o for a simple Lie algebra L, this function returns an $e \in L$ lying in the orbit.

`WeightedDynkinDiagram( o )`

Given a nilpotent orbit o for a simple Lie algebra L, this function returns its weighted Dynkin diagram.

Example H100E63

We take some nilpotent element in the Lie algebra of type E_8 and we find the weighted Dynkin diagram of the orbit it lies in.

```
> L:= LieAlgebra( RootDatum("E8"), Rationals() );
> x,_,_:= ChevalleyBasis(L);
> orb:= NilpotentOrbit( L, x[1]+x[10]-x[30]+3*x[50]-2*x[100] );
> WeightedDynkinDiagram( orb );
[ 1, 0, 0, 0, 0, 0, 0, 1 ]
```

100.21 Bibliography

[**CM93**] David H. Collingwood and William M. McGovern. *Nilpotent orbits in semisimple Lie algebras*. Van Nostrand Reinhold Mathematics Series. Van Nostrand Reinhold Co., New York, 1993.

[**CM09**] Arjeh M. Cohen and Scott H. Murray. An algorithm for Lang's Theorem. *Journal of Algebra*, 322:675–702, 2009.

[**CR09**] Arjeh M. Cohen and Dan Roozemond. Computing Chevalley bases in small characteristics. *J. Algebra*, 322(3):703–721, August 2009.

[**CSUW01**] Arjeh M. Cohen, Anja Steinbach, Rosane Ushirobira, and David Wales. Lie algebras generated by extremal elements. *J. Algebra*, 236(1):122–154, 2001.

[**dG00**] W.A. de Graaf. *Lie Algebras: Theory and Algorithms*. Number 56 in North-Holland Mathematical Library. Elsevier, 2000.

[**dG05**] W. A. de Graaf. Classification of solvable Lie algebras. *Experimental Mathematics*, 14(1):15–25, 2005.

[**dG07**] W. A. de Graaf. Classification of 6-dimensional nilpotent Lie algebras over fields of characteristic not 2. *Journal of Algebra*, 309(2):640–653, 2007.

[**dG11**] Willem A. de Graaf. Constructing semisimple subalgebras of semisimple Lie algebras. *J. Algebra*, 325:416–430, 2011.

[**Hog82**] G. M. D. Hogeweij. Almost-classical Lie algebras. I, II. *Nederl. Akad. Wetensch. Indag. Math.*, 44(4):441–452, 453–460, 1982.

[**Jac62**] N. Jacobson. *Lie algebras*. Interscience Tracts in Pure and Applied Mathematics, No. 10. Interscience Publishers New York-London, 1962.

[**Kos66**] B. Kostant. Groups over Z. In *Algebraic Groups and Discontinuous Subgroups (Proc. Sympos. Pure Math., Boulder, Colo., 1965)*, pages 90–98. Amer. Math. Soc., Providence, R.I., 1966.

[**Roo10**] D.A. Roozemond. *Algorithms for Lie algebras of algebraic groups*. PhD thesis, Technische Universiteit Eindhoven, 2010.

[**Roo11**] Dan Roozemond. On Lie algebras generated by few extremal elements. *J. Algebra*, 348:462–476, 2011.

[**SF88**] Helmut Strade and Rolf Farnsteiner. *Modular Lie algebras and their representations*, volume 116 of *Monographs and Textbooks in Pure and Applied Mathematics*. Marcel Dekker Inc., New York, 1988.

[**Str04**] Helmut Strade. *Simple Lie algebras over fields of positive characteristic. I*, volume 38 of *de Gruyter Expositions in Mathematics*. Walter de Gruyter & Co., Berlin, 2004. Structure theory.

[**ZK90**] E.I. Zel′manov and A.I. Kostrikin. A theorem on sandwich algebras. *Trudy Mat. Inst. Steklov.*, 183:106–111, 225, 1990. Translated in Proc. Steklov Inst. Math. **1991**, no. 4, 121–126, Galois theory, rings, algebraic groups and their applications (Russian).

101 KAC-MOODY LIE ALGEBRAS

Chapter 101
KAC-MOODY LIE ALGEBRAS

101.1 Introduction

Lie algebras of finite dimension are well understood, and numerous procedures for performing calculations with them are described in Chapter 100. An important class of infinite dimensional Lie algebras is that of *Kac-Moody* Lie algebras. The principal text on this subject is a book by Kac [Kac90]. Let us briefly introduce these Lie algebras.

A *generalized Cartan matrix* is an integral matrix $A = (a_{ij})_{i,j=1}^{n}$ such that $a_{ii} = 2$, $a_{ij} < 0$ for $i \neq j$, and $a_{ij} = 0$ implies $a_{ji} = 0$. (Note that in particular, a Cartan matrix in the usual sense is a generalized Cartan matrix.)

To a generalized Cartan matrix we associate a Kac-Moody Lie algebra $\mathfrak{g}(A)$. This Lie algebra is generated by $3n$ elements e_i, f_i, h_i $(i = 1, \ldots, n)$ satisfying the following defining relations:

$$[h_i, h_j] = 0, [e_i, f_i] = h_i, [e_i, f_j] = 0 \text{ if } i \neq j,$$

$$[h_i, e_j] = a_{ij} e_j, [h_i, f_j] = -a_{ij} f_j,$$

$$(\mathrm{ad} e_i)^{1-a_{ij}} e_j = 0, (\mathrm{ad} f_i)^{1-a_{ij}} f_j = 0 \text{ if } i \neq j.$$

The class of Kac-Moody Lie algebras breaks up into three subclasses:

(a) There is a vector θ of positive integers such $A\theta$ is a positive vector. In this case the Lie algebra $\mathfrak{g}(A)$ is finite-dimensional and reductive.

(b) There is a vector δ of positive integers such that $A\delta = 0$. In this case $\mathfrak{g}(A)$ is infinite-dimensional, but is of polynomial growth. These Lie algebras are called *affine Lie algebras*.

(c) There is a vector α of positive integers such that $A\alpha$ is negative. In this case $\mathfrak{g}(A)$ is infinite-dimensional and of exponential growth.

The procedures for finite-dimensional Lie algebras are described in Chapter 100. The affine Lie algebras are described in Section 101.3. The Kac-Moody Lie algebras of type (c) are not yet available.

101.2 Generalized Cartan Matrices

`IsGeneralizedCartanMatrix(C)`

Whether the square matrix C is a generalized Cartan matrix.

`KacMoodyClass(C)`

The class of the indecomposable generalized Cartan matrix C. The first return value is a string, "a", "b" or "c", corresponding to the three cases described in the introduction 101.1. The second is a positive integral column vector v such that Cv is positive, 0 or negative, respectively (so this return value corresponds to the vectors θ, δ and α in the introduction).

`KacMoodyClasses(C)`

The class of the possibly decomposable generalized Cartan matrix C. Three sequences are returned: the first is a sequence of strings "a", "b" or "c", describing the class of each component; the second is a positive integral vector v such that Cv is positive, 0 or negative, respectively (see `KacMoodyClass`).

The third sequence Q contains integral sequences Q_i such that the i-th component is formed by taking the rows and columns with index j, for $j \in Q_i$.

Example H101E1

First, we consider an indecomposable Cartan matrix.

```
> C := Matrix(Integers(), 3, 3, [2,-1,0, -5,2,-1, 0,-1,2]);
> s, v := KacMoodyClass(C);
> s;
c
> v;
[2]
[5]
[1]
> C*v;
[-1]
[-1]
[-3]
```

As a second example, we consider a decomposable Cartan matrix.

```
> C := CartanMatrix("B2 A~3");
> S, V, Q := KacMoodyClasses(C);
> S;
[ a, b ]
> Q;
[
    [ 1, 2 ],
    [ 3, 4, 5, 6 ]
]
```

```
> C1 := Submatrix(C, Q[1], Q[1]);
> KacMoodyClass(C1);
a
> C2 := Submatrix(C, Q[2], Q[2]);
> KacMoodyClass(C2);
b
```

101.3 Affine Kac-Moody Lie Algebras

For affine Lie algebras there exists a well-known explicit construction of these in terms of an underlying finite-dimensional Lie algebra and a central extension (see [Kac90, Chapters 7,8]). We briefly reiterate the construction here. Suppose A is an affine Cartan matrix, so that $\mathfrak{g}(A)$ is an affine Lie algebra; then A is of affine Cartan type $\tilde{\mathrm{X}}_n$ for X=A,B,C,D,E,F, or G, and some n. If we let $\mathfrak{g}_0$ be the finite variant (i.e., a Lie algebra of Cartan type X_n) then

$$\mathfrak{g}(A) \cong \mathfrak{g}_0 \otimes \mathbf{C}[t, t^{-1}] \oplus \mathbf{C}c \oplus \mathbf{C}d$$

for some formal basis elements c and d, where $\mathbf{C}[t, t^{-1}]$ is the ring of Laurent polynomials over $\mathbf{C}$. In MAGMA we represent affine Lie algebras and their elements using the form on the right hand side.

Multiplication is given by

$$[t^k \otimes x \oplus \lambda c \oplus \mu d, t^{k_1} \otimes y \oplus \lambda_1 c \oplus \mu_1 d] =$$

$$(t^{k+k_1} \otimes [x, y] + \mu k_1 t^{k_1} \otimes y - \mu_1 k t^k \otimes x) \oplus k\delta_{k,-k_1}(x|y)c,$$

where $(x|y)$ denotes a fixed non-degenerate invariant symmetric bilinear $\mathbf{C}$-valued form on $\mathfrak{g}_0$.

If we fix E_i, F_i to be canonical generators of g_0, then the canonical generators of $\mathfrak{g}(A)$, as described in the introduction (101.1) are given by $e_0 = t \otimes E_0$, $f_0 = t^{-1} \otimes F_0$, and $e_i = 1 \otimes E_i$, $f_i = 1 \otimes F_i$, for $i = 1, \ldots, l$, where l is the rank of the Cartan matrix.

Affine Lie algebras and their elements are of type `AlgKac` and `AlgKacElt` respectively.

101.3.1 Constructing Affine Kac-Moody Lie Algebras

`AffineLieAlgebra(N, F)`

Construct the affine Kac-Moody Lie algebra of type N over the field F. N should be a string describing an affine Cartan type (e.g. `A~3`). See Section 95.6 for more information on the conventions, syntax, and functions for creating and working with affine Cartan matrices.

`AffineLieAlgebra(C, F)`

Construct the affine Kac-Moody Lie algebra with affine Cartan matrix C over the field F.

Example H101E2

We demonstrate the construction functions.

```
> L := AffineLieAlgebra("G~2", Rationals());
> L;
Affine Kac-Moody Lie algebra over Rational Field
> C := Matrix(Integers(),3,3,[2,-1,-1,-1,2,-1,-1,-1,2]);
> CartanName(C);
A~2
> L := AffineLieAlgebra(C, Rationals());
> L;
Affine Kac-Moody Lie algebra over Rational Field
```

101.3.2 Properties of Affine Kac-Moody Lie Algebras

`CartanMatrix(L)`

The Cartan matrix of L.

`CartanName(L)`

The Cartan type of L.

`Dimension(L)`

Infinity.

`CoefficientRing(L)`

The coefficient ring of L.

`FiniteLieAlgebra(L)`

The Lie algebra $\mathfrak{g}_0$ underlying L (see the Introduction, Section 101.1).

`LaurentSeriesRing(L)`

The Laurent series ring $\mathbf{C}[t, t^{-1}]$ underlying L (see the Introduction, Section 101.1).

`StandardGenerators(L)`

The standard generators of L. These are returned as three sequences, the first containing the e_i, the second containing the f_i, and the last containing the h_i. Note that the root usually labeled "0" occurs as the last element of each of these sequences.

Example H101E3

We demonstrate some properties of affine Lie algebras.

```
> L := AffineLieAlgebra("A~2", Rationals());
> L;
Affine Kac-Moody Lie algebra over Rational Field
> Lf := FiniteLieAlgebra(L);
> Lf;
Lie Algebra of dimension 8 with base ring Rational Field
> SemisimpleType(Lf);
A2
> e,f,h := StandardGenerators(L);
> e;
[ (0 0 0 0 0 1 0 0), (0 0 0 0 0 0 1 0), (t)*(1 0 0 0 0 0 0 0) ]
> F<e1,e2,e0,f1,f2,f0> := FreeLieAlgebra(Rationals(), 6);
> phi := hom<F -> L | e cat f>;
> phi(e1);
(0 0 0 0 0 1 0 0)
> phi(e1*e0) eq phi(e1)*phi(e0);
true
```

101.3.3 Constructing Elements of Affine Kac-Moody Lie Algebras

`L . i`

The i-th basis element of the finite dimensional Lie algebra underlying L, as an element of L.

`HasAttribute(L, "c")`

`HasAttribute(L, "d")`

Return `true` and the basis element c or d of L, according to the second argument of `HasAttribute`.

`elt< L | <[<`p_1, y_1`>, ...], `λ, μ`> >`

For a 3-tuple t such that t_1 is a sequence of elements of $\mathbf{C}[t, t^{-1}] \times \mathfrak{g}_0$, and t_2 and t_3 are elements of the coefficient ring of L, construct

$$\sum_{(p,y)\in t_1} p \otimes y \oplus t_2 c \oplus t_3 d \in L.$$

See EltTup below for the converse function.

101.3.4 Properties of Elements of Affine Kac-Moody Lie Algebras

`EltTup(x)`

The element x of the affine Lie algebra L as a three-tuple t such that

$$x = \sum_{(p,y)\in t_1} p \otimes y \oplus t_2 c \oplus t_3 d.$$

The first entry, t_1, is a sequence of pairs $(p, y) \in \mathbf{C}[t, t^{-1}] \times \mathfrak{g}_0$, t_2 is the coefficient of c and t_3 is the coefficient of d.

`IsZero(x)`

Whether x is zero.

`x eq y`

Whether x and y are equal.

`x + y`
`x - y`
`x * y`

Respectively the sum, difference, and multiplication of x and y.

`-x`

The negation of x.

Example H101E4

We perform various computations with elements of an affine Lie algebra.

```
> L<t> := AffineLieAlgebra("B~3", Rationals());
> Lf := FiniteLieAlgebra(L);
> e,f,h := StandardGenerators(L);
> E,F,H := StandardBasis(Lf);
> e[1] eq L!E[1];
true
> x := e[4];
> x;
(t)*(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> EltTup(x);
<[
    <t, (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)>
], 0, 0>
> elt<L | EltTup(x) > eq x;
true
> y := elt<L | <[<t^2-t^-2, F[1]>,<-2,Lf.3>], -1/3, 1> >;
> y;
```

```
(-2)*(0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0) + (-t^-2 + t^2)*(0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0) -1/3*c + d
> z := t^3*L.2 - 1/5*h[1] + 1/7*L‘c-L‘d;
> z;
(t^3)*(0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0) + (2/5)*(0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0) (-1/5)*(0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0) + 1/7*c -1*d
> x*(y*z) + y*(z*x) + z*(x*y);
0
```

101.4 Bibliography

[**Kac90**] Victor G. Kac. *Infinite Dimensional Lie Algebras.* Cambridge University Press, 1990.

102 QUANTUM GROUPS

Chapter 102
QUANTUM GROUPS

102.1 Introduction

This chapter describes the functionality for quantum groups in MAGMA. First there are a few sections that briefly describe the theoretical background behind quantum groups (or, more precisely, quantized enveloping algebras). This fixes the notation and the terminology that we use (these vary somewhat in the literature). For this we mainly follow [Jan96]. In the remainder we describe the functions that exist in MAGMA for constructing and working with quantum groups and their representations.

In MAGMA, quantized enveloping algebras have type `AlgQUE` and their elements have type `AlgQUEElt`. These types inherit from `AlgPBW` and `AlgPBWElt` respectively, which are general types for algebras with a PBW basis and their elements and inherit from `GenMPolB`, `Alg` and `Rng` and their element types.

102.2 Background

102.2.1 Gaussian Binomials

Let v be an indeterminate over $\mathbf{Q}$. For a positive integer n we set

$$[n]_v = v^{n-1} + v^{n-3} + \cdots + v^{-n+3} + v^{-n+1}.$$

We say that $[n]_v$ is the *Gaussian integer* corresponding to n. The *Gaussian factorial* $[n]_v!$ is defined by

$$[0]_v! = 1,\ [n]_v! = [n]_v[n-1]_v \cdots [1]_v \text{ for } n > 0.$$

Finally, the *Gaussian binomial* is

$$\binom{n}{k}_v = \frac{[n]!}{[k]![n-k]!}.$$

102.2.2 Quantized Enveloping Algebras

Let L be a semisimple Lie algebra with root system Φ. By $\Delta = \{\alpha_1, \ldots, \alpha_l\}$ we denote a fixed set of simple roots of Φ. Let $C = (C_{ij})$ be the Cartan matrix of Φ (with respect to Δ, i.e., $C_{ij} = \langle \alpha_i, \alpha_j^\vee \rangle$). Let $d_1, \ldots, d_l$ be the unique sequence of positive integers with greatest common divisor 1, such that $d_i C_{ji} = d_j C_{ij}$, and set $(\alpha_i, \alpha_j) = d_j C_{ij}$. (We note that this implies that (α_i, α_i) is divisible by 2.) By P we denote the weight lattice, and we extend the form $(\ ,\ )$ to P by bilinearity.

By $W(\Phi)$ we denote the Weyl group of Φ. It is generated by the simple reflections $s_i = s_{\alpha_i}$ for $1 \leq i \leq l$ (where s_α is defined by $s_\alpha(\beta) = \beta - \langle \beta, \alpha^\vee \rangle \alpha$).

We work over the field $\mathbf{Q}(q)$. For $\alpha \in \Phi$ we set

$$q_\alpha = q^{\frac{(\alpha,\alpha)}{2}},$$

and for a non-negative integer n, $[n]_\alpha = [n]_{v=q_\alpha}$; $[n]_\alpha!$ and $\binom{n}{k}_\alpha$ are defined analogously.

The quantized enveloping algebra $U_q(L)$ is the associative algebra (with one) over $\mathbf{Q}(q)$ generated by F_α, K_α, K_α^{-1}, E_α for $\alpha \in \Delta$, subject to the following relations

$$\begin{aligned} K_\alpha K_\alpha^{-1} = K_\alpha^{-1} K_\alpha = 1,\ K_\alpha K_\beta &= K_\beta K_\alpha \\ E_\beta K_\alpha &= q^{-(\alpha,\beta)} K_\alpha E_\beta \\ K_\alpha F_\beta &= q^{-(\alpha,\beta)} F_\beta K_\alpha \\ E_\alpha F_\beta &= F_\beta E_\alpha + \delta_{\alpha,\beta} \frac{K_\alpha - K_\alpha^{-1}}{q_\alpha - q_\alpha^{-1}} \end{aligned}$$

together with, for $\alpha \neq \beta \in \Delta$,

$$\sum_{k=0}^{1-\langle \beta, \alpha^\vee \rangle} (-1)^k \binom{1 - \langle \beta, \alpha^\vee \rangle}{k}_\alpha E_\alpha^{1-\langle \beta, \alpha^\vee \rangle - k} E_\beta E_\alpha^k = 0$$

$$\sum_{k=0}^{1-\langle \beta, \alpha^\vee \rangle} (-1)^k \binom{1 - \langle \beta, \alpha^\vee \rangle}{k}_\alpha F_\alpha^{1-\langle \beta, \alpha^\vee \rangle - k} F_\beta F_\alpha^k = 0.$$

The quantized enveloping algebra has an automorphism ω defined by $\omega(F_\alpha) = E_\alpha$, $\omega(E_\alpha) = F_\alpha$ and $\omega(K_\alpha) = K_\alpha^{-1}$. Also there is an anti-automorphism τ defined by $\tau(F_\alpha) = F_\alpha$, $\tau(E_\alpha) = E_\alpha$ and $\tau(K_\alpha) = K_\alpha^{-1}$. We have $\omega^2 = 1$ and $\tau^2 = 1$.

If the Dynkin diagram of Φ admits a diagram automorphism π, then π induces an automorphism of $U_q(L)$ in the obvious way (π is a permutation of the simple roots; we permute the F_α, E_α, $K_\alpha^{\pm 1}$ accordingly).

Now we view $U_q(L)$ as an algebra over $\mathbf{Q}$, and we let $\overline{\ \ } : U_q(L) \to U_q(L)$ be the automorphism defined by $\overline{F_\alpha} = F_\alpha$, $\overline{K_\alpha} = K_\alpha^{-1}$, $\overline{E_\alpha} = E_\alpha$, $\overline{q} = q^{-1}$. This map is called the *bar-automorphism.*

102.2.3 Representations of $U_q(L)$

Let $\lambda \in P$ be a dominant weight. Then there is a unique irreducible highest-weight module over $U_q(L)$ with highest weight λ. We denote it by $V(\lambda)$. It has the same character as the irreducible highest-weight module over L with highest weight λ. Furthermore, every finite-dimensional $U_q(L)$-module is a direct sum of irreducible highest-weight modules. In [Gra04] a few algorithms for constructing $V(\lambda)$ are given. In the MAGMA implementation the algorithm based on Gröbner bases is used.

It is well-known that $U_q(L)$ is a Hopf algebra. The comultiplication $\Delta : U_q(L) \to U_q(L) \otimes U_q(L)$ is defined by

$$\begin{aligned}\Delta(E_\alpha) &= E_\alpha \otimes 1 + K_\alpha \otimes E_\alpha \\ \Delta(F_\alpha) &= F_\alpha \otimes K_\alpha^{-1} + 1 \otimes F_\alpha \\ \Delta(K_\alpha) &= K_\alpha \otimes K_\alpha.\end{aligned}$$

(Note that we use the same symbol (Δ) to denote a set of simple roots of Φ; of course this does not cause confusion.) The counit $\varepsilon : U_q(L) \to \mathbf{Q}(q)$ is a homomorphism defined by $\varepsilon(E_\alpha) = \varepsilon(F_\alpha) = 0$, $\varepsilon(K_\alpha) = 1$. Finally, the antipode $S : U_q(L) \to U_q(L)$ is an anti-automorphism given by $S(E_\alpha) = -K_\alpha^{-1}E_\alpha$, $S(F_\alpha) = -F_\alpha K_\alpha$, $S(K_\alpha) = K_\alpha^{-1}$.

Using Δ we can make the tensor product $V \otimes W$ of two $U_q(L)$-modules V, W into a $U_q(L)$-module. The counit ε yields a trivial 1-dimensional $U_q(L)$-module. And with S we can define a $U_q(L)$-module structure on the dual V^* of a $U_q(L)$-module V, by $(u \cdot f)(v) = f(S(u) \cdot v)$.

The Hopf algebra structure given above is not the only one possible. For example, we can twist Δ, ε, S by an automorphism, or an anti-automorphism f. The twisted comultiplication is given by

$$\Delta^f = f \otimes f \circ \Delta \circ f^{-1},$$

the twisted antipode by

$$S^f = f \circ S \circ f^{-1},$$

if f is an automorphism, and

$$S^f = f \circ S^{-1} \circ f^{-1},$$

if f is an anti-automorphism. The twisted counit is given by $\varepsilon^f = \varepsilon \circ f^{-1}$.

102.2.4 PBW-type Bases

The first problem one has to deal with when working with $U_q(L)$ is finding a basis of it, along with an algorithm for expressing the product of two basis elements as a linear combination of basis elements. First of all we have that $U_q(L) \cong U^- \otimes U^0 \otimes U^+$ (as vector spaces), where U^- is the subalgebra generated by the F_α, U^0 is the subalgebra generated by the K_α, and U^+ is generated by the E_α. So a basis of $U_q(L)$ is formed by all elements FKE, where F, K, E run through bases of U^-, U^0, U^+ respectively.

Finding a basis of U^0 is easy: it is spanned by all $K_{\alpha_1}^{r_1} \cdots K_{\alpha_l}^{r_l}$, where $r_i \in \mathbf{Z}$. For U^- and U^+ we use the so-called *PBW-type* bases. They are defined as follows. For $\alpha, \beta \in \Delta$

we set $r_{\beta,\alpha} = -\langle \beta, \alpha^\vee \rangle$. Then for $\alpha \in \Delta$ we have the automorphism $T_\alpha : U_q(L) \to U_q(L)$ defined by

$$\begin{aligned} T_\alpha(E_\alpha) &= -F_\alpha K_\alpha \\ T_\alpha(E_\beta) &= \sum_{i=0}^{r_{\beta,\alpha}} (-1)^i q_\alpha^{-i} E_\alpha^{(r_{\beta,\alpha}-i)} E_\beta E_\alpha^{(i)}, \ (\alpha \neq \beta) \\ T_\alpha(K_\beta) &= K_\beta K_\alpha^{r_{\beta,\alpha}} \\ T_\alpha(F_\alpha) &= -K_\alpha^{-1} E_\alpha \\ T_\alpha(F_\beta) &= \sum_{i=0}^{r_{\beta,\alpha}} (-1)^i q_\alpha^{i} F_\alpha^{(i)} F_\beta F_\alpha^{(r_{\beta,\alpha}-i)}, \ (\alpha \neq \beta) \end{aligned}$$

(where $E_\alpha^{(k)} = E_\alpha^k/[k]_\alpha!$, and likewise for $F_\alpha^{(k)}$).

Let $w_0 = s_{i_1} \cdots s_{i_t}$ be a reduced expression for the longest element in the Weyl group $W(\Phi)$. For $1 \leq k \leq t$ set $F_k = T_{\alpha_{i_1}} \cdots T_{\alpha_{i_{k-1}}}(F_{\alpha_{i_k}})$, and $E_k = T_{\alpha_{i_1}} \cdots T_{\alpha_{i_{k-1}}}(E_{\alpha_{i_k}})$. Then $F_k \in U^-$, and $E_k \in U^+$. Furthermore, the elements $F_1^{m_1} \cdots F_t^{m_t}$, $E_1^{n_1} \cdots E_t^{n_t}$ (where the m_i, n_i are non-negative integers) form bases of U^- and U^+ respectively.

The elements F_α and E_α are said to have weight $-\alpha$ and α respectively, where α is a simple root. Furthermore, the weight of a product ab is the sum of the weights of a and b. Now elements of U^-, U^+ that are linear combinations of elements of the same weight are said to be homogeneous. It can be shown that the elements F_k, and E_k are homogeneous of weight $-\beta$ and β respectively, where $\beta = s_{i_1} \cdots s_{i_{k-1}}(\alpha_{i_k})$.

In the following we use the notation $F_k^{(m)} = F_k^m/[m]_{\alpha_{i_k}}!$, and $E_k^{(n)} = E_k^n/[n]_{\alpha_{i_k}}!$.

We refer to [Gra01] for an account of algorithms for expressing the product of two elements of a PBW-type basis as a linear combination of such elements. These algorithms are implemented in Magma.

102.2.5 The Z-form of $U_q(L)$

For $\alpha \in \Delta$ set

$$\binom{K_\alpha}{n} = \prod_{i=1}^{n} \frac{q_\alpha^{-i+1} K_\alpha - q_\alpha^{i-1} K_\alpha^{-1}}{q_\alpha^i - q_\alpha^{-i}}.$$

Then according to [Lus90], Theorem 6.7 the elements

$$F_1^{(k_1)} \cdots F_t^{(k_t)} K_{\alpha_1}^{\delta_1} \binom{K_{\alpha_1}}{m_1} \cdots K_{\alpha_l}^{\delta_l} \binom{K_{\alpha_l}}{m_l} E_1^{(n_1)} \cdots E_t^{(n_t)},$$

(where $k_i, m_i, n_i \geq 0$, $\delta_i = 0, 1$) form a basis of $U_q(L)$, such that the product of any two basis elements is a linear combination of basis elements with coefficients in $\mathbf{Z}[q, q^{-1}]$. The quantized enveloping algebra over $\mathbf{Z}[q, q^{-1}]$ with this basis is called the $\mathbf{Z}$-form of $U_q(L)$, and denoted by $U_{\mathbf{Z}}$. Since $U_{\mathbf{Z}}$ is defined over $\mathbf{Z}[q, q^{-1}]$ we can specialize q to any nonzero element ϵ of a field F, and obtain an algebra U_ϵ over F. In particular, if we take $\epsilon = 1$, then we obtain an algebra U_1 over $\mathbf{Q}$. Let I be the ideal of U_1 generated by $K_{\alpha_1} - 1, \ldots, K_{\alpha_l} - 1$. Then U_1/I is isomorphic to the universal enveloping algebra $U(L)$

of L. Also, the homomorphism $U_q(L) \to U(L)$ maps the basis above onto an integral basis of $U(L)$ ([Lus90]).

We call $q \in \mathbf{Q}(q)$, and $\epsilon \in F$ the quantum parameter of $U_q(L)$ and U_ϵ respectively.

102.2.6 The Canonical Basis

As in Section 102.2.4 we let U^- be the subalgebra of $U_q(L)$ generated by the F_α for $\alpha \in \Delta$. Kashiwara and Lusztig have (independently) given constructions of a basis of U^- with very nice properties, called the *canonical basis*.

Let $w_0 = s_{i_1} \cdots s_{i_t}$, and the elements F_k be as in Section 102.2.4. Then, in order to stress the dependency of the monomial

$$F_1^{(n_1)} \cdots F_t^{(n_t)}$$

on the choice of reduced expression for the longest element in $W(\Phi)$ we say that it is a w_0-monomial. The integer n_1 is called its first exponent.

Now we let $^-$ be the automorphism of U^- defined in Section 102.2.2. Elements that are invariant under $^-$ are said to be bar-invariant.

By results of Lusztig ([Lus93], Theorem 42.1.10, [Lus96], Proposition 8.2), there is a unique basis $\mathbf{B}$ of U^- with the following properties. Firstly, all elements of $\mathbf{B}$ are bar-invariant. Secondly, for any choice of reduced expression w_0 for the longest element in the Weyl group, and any element $X \in \mathbf{B}$ we have that $X = x + \sum_i \zeta_i x_i$, where x, x_i are w_0-monomials, $x \neq x_i$ for all i, and $\zeta_i \in q\mathbf{Z}[q]$. The basis $\mathbf{B}$ is called the canonical basis. If we work with a fixed reduced expression for the longest element in $W(\Phi)$, and write $X \in \mathbf{B}$ as above, then we say that x is the *principal monomial* of X.

Let $\mathcal{L}$ be the $\mathbf{Z}[q]$-lattice in U^- spanned by $\mathbf{B}$. Then $\mathcal{L}$ is also spanned by all w_0-monomials (where w_0 is a fixed reduced expression for the longest element in $W(\Phi)$).

Now let $\widetilde{w}_0$ be a second reduced expression for the longest element in $W(\Phi)$. Let x be a w_0-monomial, and let X be the element of $\mathbf{B}$ with principal monomial x. Write X as a linear combination of $\widetilde{w}_0$-monomials, and let $\widetilde{x}$ be the principal monomial of that expression. Then we write $\widetilde{x} = R_{w_0}^{\widetilde{w}_0}(x)$. Note that $x = \widetilde{x} \bmod q\mathcal{L}$.

Now let $\mathcal{B}$ be the set of all $x \bmod q\mathcal{L}$, where x runs through the set of w_0-monomials. Then $\mathcal{B}$ is a basis of the $\mathbf{Z}$-module $\mathcal{L}/q\mathcal{L}$. Moreover, $\mathcal{B}$ is independent of the choice of w_0. Let $\alpha \in \Delta$, and let $\widetilde{w}_0$ be a reduced expression for the longest element in $W(\Phi)$, starting with s_α. The Kashiwara operators $\widetilde{F}_\alpha : \mathcal{B} \to \mathcal{B}$ and $\widetilde{E}_\alpha : \mathcal{B} \to \mathcal{B} \cup \{0\}$ are defined as follows. Let $b \in \mathcal{B}$ and let x be the w_0-monomial such that $b = x \bmod q\mathcal{L}$. Set $\widetilde{x} = R_{w_0}^{\widetilde{w}_0}(x)$. Let $\widetilde{x}'$ be the $\widetilde{w}_0$-monomial constructed from $\widetilde{x}$ by increasing its first exponent by 1. Then $\widetilde{F}_\alpha(b) = R_{\widetilde{w}_0}^{w_0}(\widetilde{x}') \bmod q\mathcal{L}$. For $\widetilde{E}_\alpha$ we let $\widetilde{x}'$ be the $\widetilde{w}_0$-monomial constructed from $\widetilde{x}$ by decreasing its first exponent by 1, if this exponent is ≥ 1. Then $\widetilde{E}_\alpha(b) = R_{\widetilde{w}_0}^{w_0}(\widetilde{x}') \bmod q\mathcal{L}$. Furthermore, $\widetilde{E}_\alpha(b) = 0$ if the first exponent of $\widetilde{x}$ is 0. It can be shown that this definition does not depend on the choice of w_0, $\widetilde{w}_0$. Furthermore we have $\widetilde{F}_\alpha \widetilde{E}_\alpha(b) = b$, if $\widetilde{E}_\alpha(b) \neq 0$, and $\widetilde{E}_\alpha \widetilde{F}_\alpha(b) = b$ for all $b \in \mathcal{B}$.

Now let $V(\lambda)$ be a highest-weight module over $U_q(L)$, with highest weight λ. Let v_λ be a fixed highest weight vector. Then $\mathbf{B}_\lambda = \{X \cdot v_\lambda \mid X \in \mathbf{B}\} \setminus \{0\}$ is a basis of $V(\lambda)$,

called the *canonical basis* of $V(\lambda)$. Let $\mathcal{L}(\lambda)$ be the $\mathbf{Z}[q]$-lattice in $V(\lambda)$ spanned by $\mathbf{B}_\lambda$. We let $\mathcal{B}(\lambda)$ be the set of all $x \cdot v_\lambda \bmod q\mathcal{L}(\lambda)$, where x runs through all w_0-monomials, such that $X \cdot v_\lambda \neq 0$, where $X \in \mathbf{B}$ is the element with principal monomial x. Then the Kashiwara operators are also viewed as maps $\mathcal{B}(\lambda) \to \mathcal{B}(\lambda) \cup \{0\}$, in the following way. Let $b = x \cdot v_\lambda \bmod q\mathcal{L}(\lambda)$ be an element of $\mathcal{B}(\lambda)$, and let $b' = x \bmod q\mathcal{L}$ be the corresponding element of $\mathcal{B}$. Let y be the w_0-monomial such that $\widetilde{F}_\alpha(b') = y \bmod q\mathcal{L}$. Then $\widetilde{F}_\alpha(b) = y \cdot v_\lambda \bmod q\mathcal{L}(\lambda)$. The description of $\widetilde{E}_\alpha$ is analogous. (In [Jan96], Chapter 9 a different definition is given; however, by [Jan96], Proposition 10.9, Lemma 10.13, the two definitions agree).

The set $\mathcal{B}(\lambda)$ has $\dim V(\lambda)$ elements. We let Γ be the coloured directed graph defined as follows. The points of Γ are the elements of $\mathcal{B}(\lambda)$, and there is an arrow with colour $\alpha \in \Delta$ connecting $b, b' \in \mathcal{B}$, if $\widetilde{F}_\alpha(b) = b'$. The graph Γ is called the *crystal graph* of $V(\lambda)$.

In [Gra02] algorithms are given for computing the action of the Kashiwara operators on $\mathcal{B}$ (without computing $\mathbf{B}$ first), and for computing elements of $\mathbf{B}$.

102.2.7 The Path Model

In this section we recall some basic facts on Littelmann's path model.

From Section 102.2.2 we recall that P denotes the weight lattice. Let P_R be the vector space over R spanned by P. Let Π be the set of all piecewise linear paths $\xi : [0,1] \to P_R$, such that $\xi(0) = 0$. For $\alpha \in \Delta$ Littelmann defined *path operators* $f_\alpha, e_\alpha : \Pi \to \Pi \cup \{0\}$. Let λ be a dominant weight and let ξ_λ be the path joining λ and the origin by a straight line. Let Π_λ be the set of all nonzero $f_{\alpha_{i_1}} \cdots f_{\alpha_{i_m}}(\xi_\lambda)$ for $m \geq 0$. Then $\xi(1) \in P$ for all $\xi \in \Pi_\lambda$. Let $\mu \in P$ be a weight, and let $V(\lambda)$ be the highest-weight module over $U_q(L)$ of highest weight λ. A theorem of Littelmann states that the number of paths $\xi \in \Pi_\lambda$ such that $\xi(1) = \mu$ is equal to the dimension of the weight space of weight μ in $V(\lambda)$ ([Lit95], Theorem 9.1).

All paths appearing in Π_λ are so-called Lakshmibai–Seshadri paths (LS-paths for short). They are defined as follows. Let $\leq$ denote the Bruhat order on $W(\Phi)$. For $\mu, \nu \in W(\Phi) \cdot \lambda$ (the orbit of λ under the action of $W(\Phi)$), write $\mu \leq \nu$ if $\tau \leq \sigma$, where $\tau, \sigma \in W(\Phi)$ are the unique elements of minimal length such that $\tau(\lambda) = \mu$, $\sigma(\lambda) = \nu$. Now a rational path of shape λ is a pair $\pi = (\nu, a)$, where $\nu = (\nu_1, \ldots, \nu_s)$ is a sequence of elements of $W(\Phi) \cdot \lambda$, such that $\nu_i > \nu_{i+1}$ and $a = (a_0 = 0, a_1, \cdots, a_s = 1)$ is a sequence of rationals such that $a_i < a_{i+1}$. The path π corresponding to these sequences is given by

$$\pi(t) = \sum_{j=1}^{r-1} (a_j - a_{j-1})\nu_j + \nu_r(t - a_{r-1})$$

for $a_{r-1} \leq t \leq a_r$. Now an LS-path of shape λ is a rational path satisfying a certain integrality condition (see [Lit94], [Lit95]). We note that the path $\xi_\lambda = ((\lambda), (0,1))$ joining the origin and λ by a straight line is an LS-path of shape λ. Furthermore, all paths obtained from ξ_λ by applying the path operators are LS-paths of shape λ.

From [Lit94], [Lit95]) we transcribe the following:

(a) Let π be an LS-path. Then $f_\alpha \pi$ is an LS-path or 0; and the same holds for $e_\alpha \pi$.

(b) The action of f_α, e_α can easily be described combinatorially (see [Lit94]).

(c) The endpoint of an LS-path is an integral weight.

(d) Let $\pi = (\nu, a)$ be an LS-path. Then by $\phi(\pi)$ we denote the unique element σ of $W(\Phi)$ of shortest length such that $\sigma(\lambda) = \nu_1$.

Let λ be a dominant weight. Then we define a labeled directed graph Γ as follows. The points of Γ are the paths in Π_λ. There is an edge with label $\alpha \in \Delta$ from π_1 to π_2 if $f_\alpha \pi_1 = \pi_2$. Now by [Kas96] this graph Γ is isomorphic to the crystal graph of the highest-weight module with highest weight λ. So the path model provides an efficient way of computing the crystal graph of a highest-weight module, without constructing the module first. Also we see that $f_{\alpha_{i_1}} \cdots f_{\alpha_{i_r}} \xi_\lambda = 0$ is equivalent to $\widetilde{F}_{\alpha_{i_1}} \cdots \widetilde{F}_{\alpha_{i_r}} v_\lambda = 0$, where $v_\lambda \in V(\lambda)$ is a highest weight vector (or rather the image of it in $\mathcal{L}(\lambda)/q\mathcal{L}(\lambda)$), and the $\widetilde{F}_{\alpha_k}$ are the Kashiwara operators on $\mathcal{B}(\lambda)$ (see Section 102.2.6).

102.3 Gauss Numbers

`GaussNumber(n, v)`

Given an integer n and a ring element v, this function returns the Gauss number corresponding to n with parameter v, i.e., the element $[n]_v = v^{n-1} + v^{n-3} + \cdots + v^{-n+3} + v^{-n+1}$.

`GaussianFactorial(n, v)`

Given an integer n and a ring element v, this function returns the Gaussian factorial corresponding to n with parameter v, i.e., the element $[n]_v! = [n]_v[n-1]_v \cdots [1]_v$.

`GaussianBinomial(n, k, v)`

Given an integer n, an integer k and a ring element v, this function returns the Gaussian binomial n choose k with parameter v, i.e., the element $[n]_v!/([k]_v![n-k]_v!)$.

Example H102E1

```
> F<q>:= RationalFunctionField(Rationals());
> GaussianBinomial(5, 3, q^2);
(q^24 + q^20 + 2*q^16 + 2*q^12 + 2*q^8 + q^4 + 1)/q^12
```

102.4 Construction

`QuantizedUEA(R)`

`QuantizedUEAlgebra(R)`

`QuantizedUniversalEnvelopingAlgebra(R)`

This creates the quantized enveloping algebra U corresponding to the root datum R. The algebra U will be defined over the rational function field in one variable, q, over the rational numbers.

Let n and r respectively be the number of positive roots, and the rank of R. Then U has $2n+r$ generators, accessible as `U.1`, `U.2` and so on. The first n of these are printed as `F_1`, ... , `F_n`. They generate a PBW-type basis of the subalgebra U^- (cf. Section 102.2.4). The next r generators are printed as `K_1`, ... , `K_r`; together with their inverses they generate the algebra U^0. The final n generators are printed as `E_1`, ... , `E_n`. They generate a PBW-type basis of U^+.

In U we use a basis of the integral form of U (Section 102.2.5). This means that instead of F_k^s and E_k^s we use the divided powers $F_k^{(s)}$ and $E_k^{(s)}$. Furthermore, a general basis element of U^0 is a product of elements which are of the form $[K_i;t]$, or $K_i[K_i;t]$. Here $[K_i;t]$ represents the "binomial" K_i choose t as described in Section 102.2.5.

`w0` SeqEnum *Default :*

It is also possible to give a reduced expression for the longest element in the Weyl group, by setting the optional parameter `w0` equal to a sequence of indices lying between 1 and the rank of R. If we replace each index by the corresponding simple reflection, then a reduced expression for the longest element in the Weyl group has to be obtained. In that case the PBW-basis relative to that sequence will be created (and used in subsequent computations). If this parameter is not given, then the lexicographically smallest reduced expression will be used.

Example H102E2

We construct the quantum group corresponding to the root datum of type C_3.

```
> R:= RootDatum("C3");
> U:= QuantizedUEA(R);
> U.9; U.10; U.15;
F_9
K_1
E_3
> U.21*U.14*U.10*U.9*U.1;
1/q*F_1*F_9*K_1*E_2*E_9 - 1/q*F_1*F_9*K_1*E_6 + 1/q^3*F_1*K_1*[ K_3 ; 1 ]*E_2 -
    F_9*E_3*E_9 + F_9*E_8 - 1/q^2*[ K_3 ; 1 ]*E_3
```

Now we construct the same algebra, but use the PBW-basis relative to a different reduced expression of the longest element in the Weyl group.

```
> U:= QuantizedUEA(R : w0:= [2,3,1,2,3,1,2,3,1]);
```

```
> U.21*U.14*U.10*U.9*U.1;
q^2*F_1*F_9*K_1*E_2*E_9 + (q^2 - 1)/q^3*F_9*K_1*[ K_2 ; 1 ]*E_6*E_9 -
    1/q^2*F_9*K_1*K_2*E_6*E_9 - q^2*F_1*F_9*K_1*E_4 + q^2*F_1*K_1[ K_1 ; 1 ]*E_2 +
    q*F_3*K_1*E_2*E_9 + (-q^2 + 1)/q^3*F_9*K_1*[ K_2 ; 1 ]*E_7 +
    1/q^2*F_9*K_1*K_2*E_7 + (q^2 - 1)/q*K_1[ K_1 ; 1 ]*[ K_2 ; 1 ]*E_6
    - K_1[ K_1 ; 1 ]*K_2*E_6 - q*F_3*K_1*E_4 + q*F_1*E_2
```

`AssignNames(U, S)`

Assign the names in the sequence S to the generators of the algebra U.

`ChangeRing(U, R)`

Return the algebra identical to the algebra U but having coefficient ring R.

102.5 Related Structures

`CoefficientRing(U)`

This returns the ring of coefficients of the quantized enveloping algebra U.

`RootDatum(U)`

This returns the root datum corresponding to the quantized enveloping algebra U.

`PositiveRootsPerm(U)`

Given a quantized universal enveloping algebra U with root datum R returns a sequence consisting of the integers between 1 and the number of positive roots of R. If the k-th element of this sequence is m, then the generator F_k of U is of weight $-\beta_m$, where β_m is the m-th positive root of R (as returned by `PositiveRoots(R)`). (For the definition of weight of an element of U see Section 102.2.4.) Furthermore, the generator E_k is of weight β_m.

Example H102E3

```
> R:= RootDatum("D4");
> U:= QuantizedUEA(R);
> CoefficientRing(U);
Univariate rational function field over Rational Field
Variables: q
> RootDatum(U);
Adjoint root datum of type D4
> PositiveRootsPerm(U);
[ 1, 5, 2, 8, 6, 3, 12, 11, 9, 10, 7, 4 ]
```

So for instance this means that F_6 is of weight $-\beta_3$.

102.6 Operations on Elements

The generators of a quantized enveloping algebra U can be constructed by using the dot operator, e.g., `U.5`. More general elements can then be constructed using the operations of scalar multiplication, addition, and multiplication.

Note that for the generators denoted F_k and E_k we use divided powers instead of normal powers. This means for instance that $F_k^s = [s]!F_k^{(s)}$, i.e., exponentiation causes multiplication by a scalar factor.

`x + y` `x - y` `x * y` `c * x` `x * c` `x ^ n`

`U ! 0`
`Zero(U)`

The zero element of the quantized enveloping algebra U.

`U ! 1`
`One(U)`

The identity element of the quantized enveloping algebra U.

`U . i`

The i-th generator of the quantized enveloping algebra U. Let the root datum have s positive roots and rank r. If $1 \leq i \leq s$ then `U.i` is F_i. If $s+1 \leq i \leq s+r$, then `U.i` is K_j where $j = i - s$. If $s + r + 1 \leq i \leq 2s + r$ then `U.i` is E_j, where $j = i - s - r$.

`U ! r`

Returns r as an element of the quantized universal enveloping algebra U where r may be anything coercible into the coefficient ring of U or an element of another quantized enveloping algebra whose coefficients may be coerced into the coefficient ring of U.

`KBinomial(U, i, s)`
`KBinomial(K, s)`

Given a quantized enveloping algebra U corresponding to a root datum of rank r, an integer i between 1 and r, and a positive integer s, return the element $[K_i; s]$. This can be used to construct general elements in the subalgebra U^0 (cf. Section 102.2.4).

Or given an element $K = K_i$, i.e., equal to `U.(n+i)`, where n is the number of positive roots of the root datum, return $[K; s]$.

`Monomials(u)`

Given an element u of a quantized enveloping algebra, returns the sequence consisting of the monomials of u. This sequence corresponds exactly to the one returned by `Coefficients(u)`.

`Coefficients(u)`

Given an element u of a quantized enveloping algebra, returns the sequence consisting of the coefficients of the monomials that occur in u. This sequence corresponds exactly to the one returned by `Monomials(u)`.

`K ^ -1`

Given a generator K of a quantized enveloping algebra U of the form K_i, i.e., it is equal to `U.k`, for some $n+1 \leq k \leq n+r$ where U corresponds to a root datum of rank r with n positive roots, return the inverse of K.

`Degree(u, i)`

Given an element u of a quantized enveloping algebra U and an integer $1 \leq i \leq n$ or $n+r+1 \leq i \leq 2n+r$, where the root datum corresponding to U has n positive roots and rank r (i.e., `U.i` is equal to F_i or to E_k, where $k = i-n-r$), return the degree of u in the generator F_i if $1 \leq i \leq n$, otherwise return the degree of u in the generator E_k, where $k = i-n-r$.

`KDegree(m, i)`

Given a single monomial m in a quantized enveloping algebra and an integer $1 \leq i \leq r$, where r is the rank of the corresponding root datum return a tuple of 2 integers, where the first is 0 or 1, and the second is non-negative. Denote this tuple by $\langle d, k \rangle$. If $d = 0$ then the factor $[K_i; k]$ occurs in the monomial m. If $d = 1$, then the factor $K_i[K_i; k]$ occurs in the monomial m.

Example H102E4

```
> R:= RootDatum("G2");
> U:= QuantizedUEA(R);
> u:= U.10*U.7^3*U.1;
> m:= Monomials(u); m;
[
    F_1*K_1[ K_1 ; 2 ]*E_2,
    F_1*[ K_1 ; 1 ]*E_2,
    F_1*K_1*E_2,
    K_1[ K_1 ; 1 ]*E_3,
    E_3
]
> Coefficients(u);
[
    (q^6 - q^4 - q^2 + 1)/q^17,
    (q^2 - 1)/q^14,
    1/q^15,
    (-q^2 + 1)/q^9,
    -1/q^8
]
> Degree(m[1], 1);
```

```
1
> Degree(m[1], 9);
0
> Degree(m[1], 10);
1
> KDegree(m[1], 1);
<1, 2>
> U.7^-1;
(-q^2 + 1)/q*[ K_1 ; 1 ] + K_1
> U.7*U.7^-1;
1
```

102.7 Representations

In this section we describe some functions for working with left-modules over quantized enveloping algebras. For a general introduction into algebra modules in MAGMA we refer to Chapter 89.

`HighestWeightRepresentation(U, w)`

Given a quantized enveloping algebra U corresponding to a root datum of rank r and a sequence w of non-negative integers of length r, returns the irreducible representation of U with highest weight w. The object returned is a function which given an element of U computes its matrix.

`HighestWeightModule(U, w)`

Given a quantized enveloping algebra U corresponding to a root datum of rank r and a sequence w of non-negative integers of length r, returns the irreducible U-module with highest weight w. The object returned is a left module over U.

Example H102E5

```
> R:= RootDatum("G2");
> U:= QuantizedUEA(R);
> f:= HighestWeightRepresentation(U, [1,1]);
> f(U.6);
[ 0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0]
[ 0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0]
[ 0  0  0  0 -q  0  0]
[ 0  0  0  0  0  0  0]
> M:= HighestWeightModule(U, [1,0]);
> U.6^M.5;
(0  0  0  0  0 -q  0)
```

`WeightsAndVectors(V)`

For a module V over a quantized universal enveloping algebra this returns two sequences. The first sequence consists of the weights that occur in V. The second sequence is a sequence of sequences of elements of V, in bijection with the first sequence. The i-th element of the second sequence consists of a basis of the weight space of weight equal to the i-th weight of the first sequence.

`HighestWeightsAndVectors(V)`

This function is analogous to the previous one. Except in this case the first sequence consists of highest weights, i.e., those weights which occur as highest weights of an irreducible constituent of V. The second sequence consists of sequences that contain the corresponding highest weight vectors. So the submodules generated by the vectors in the second sequence form a direct sum decomposition of V.

`CanonicalBasis(V)`

Given a (left-) module V over a quantized universal enveloping algebra, returns the canonical basis of V. If V is not irreducible, then the union of the canonical bases of the irreducible components of V is returned.

Example H102E6

```
> U:= QuantizedUEA(RootDatum("B2"));
> V:= HighestWeightModule(U, [1,0]);
> C:= CanonicalBasis(V); C;
[
    V: (1 0 0 0 0),
    V: (0 1 0 0 0),
    V: (0 0 1 0 0),
    V: (0 0 0 1 0),
    V: (    0      0      0      0 -1/q^2)
]
```

We can compute the action of elements of U with respect to the canonical basis by using `ModuleWithBasis`:

```
> M:= ModuleWithBasis(C);
> ActionMatrix(M, U.1);
[0 0 0 0 0]
[1 0 0 0 0]
[0 0 0 0 0]
[0 0 0 0 0]
[0 0 0 1 0]
```

`TensorProduct(Q)`

Given a sequence Q of left-modules over a quantized universal enveloping algebra, returns the module M that is the tensor product of the elements of Q. It also returns a map from the Cartesian product of the elements of Q to M. This maps a tuple t to the element of M that is formed by tensoring the elements of t.

Example H102E7

```
> U:= QuantizedUEA(RootDatum("B2"));
> v1:= HighestWeightModule(U, [1,0]);
> V1:= HighestWeightModule(U, [1,0]);
> V2:= HighestWeightModule(U, [0,1]);
> W, f:= TensorProduct([V1,V2]);
> Dimension(W);
20
> HighestWeightsAndVectors(W);
[
    (1 1),
    (0 1)
]
[
    [
        W: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        W: (0   0   1   0   0   q^4   0   0   -q^7/(q^2 + 1)   0   0   0   0   0
            0   0   0   0   0   0)
    ]
]
> f(<V1.2,V2.4>);
W: (0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0)
```

So in particular we see that $V1 \otimes V2$ is the direct sum of two irreducible modules, one with highest weight $(1, 1)$, the other with highest weight $(0, 1)$. The corresponding highest-weight vectors are also given.

102.8 Hopf Algebra Structure

In this section we describe the functions for working with the Hopf algebra structure of a quantized universal enveloping algebra (cf. Section 102.2.3).

`UseTwistedHopfStructure(U, f, g)`

The Hopf algebra structure that is used by default is the one described in Section 102.2.3. As explained in that same section, it is possible to twist this by an automorphism, or an antiautomorphism.

Given a quantized universal enveloping algebra U and (anti-) automorphisms f and g of U where g is the inverse of f (this is not checked by MAGMA) set U to use the corresponding twisted Hopf algebra structure.

This command has to be given before using the Hopf algebra structure, otherwise the default structure will be used. This includes creating a tensor product.

For some (anti-) automorphisms we refer to Section 102.9.

`HasTwistedHopfStructure(U)`

This function checks whether the quantized enveloping algebra U has been set to use a twisted Hopf structure. If the first value returned by this function is `true`, then the (anti-) automorphism and its inverse are also returned.

`Counit(U)`

Returns the counit of the quantized enveloping algebra U. It is a map from U into the ground field of U.

`Antipode(U)`

Returns the antipode of the quantized enveloping algebra U. It is an antiautomorphism of U.

`Comultiplication(U, d)`

Returns the comultiplication of degree d of the quantized enveloping algebra U. This is a map from U into the d-fold tensor power of U. The comultiplication given in Section 102.2.3 is of degree 2. The comultiplications of higher degree are obtained by repeating this map. So in particular, d has to be at least 2.

An element of the d-fold tensor power of U is represented (rather primitively) by a list of d-tuples, each followed by a coefficient. The d-tuples are d-tuples of basis elements of U. The element represented by this list is the sum of the elements obtained by multiplying the i-th coefficient and the tensor product of the elements in the i-th d-tuple.

Example H102E8

```
> U:= QuantizedUEA(RootDatum("A3"));
> d:= Comultiplication(U, 2);
> d(U.1);
[*
<1, F_1>,
1,
<F_1, K_1>,
1,
<F_1, [ K_1 ; 1 ]>,
(-q^2 + 1)/q
*]
```

102.9 Automorphisms

`BarAutomorphism(U)`

For a quantized enveloping algebra U this returns the bar-automorphism of U (Section 102.2.2). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

`AutomorphismOmega(U)`

For a quantized enveloping algebra U this returns the automorphism of U that is denoted by ω (Section 102.2.2). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

`AntiAutomorphismTau(U)`

For a quantized enveloping algebra U this returns the anti-automorphism of U that is denoted by τ (Section 102.2.2). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

`AutomorphismTalpha(U, k)`

Let U be a quantized enveloping algebra, and let k be an integer between 1 and the rank of the root datum. Then this function returns the automorphism T_{α_k} of U, corresponding to the k-th simple root (Section 102.2.4). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

`DiagramAutomorphism(U, p)`

`GraphAutomorphism(U, p)`

Let U be a quantized enveloping algebra, and let p be a permutation of $\{1, \ldots, r\}$, where r is the rank of the root datum. Here p must represent a diagram automorphism of the root datum (i.e., it leaves the Dynkin diagram invariant). Then this function returns the corresponding automorphism of U (see Section 102.2.2). The map returned by this function has its inverse stored, which can be retrieved using `Inverse`.

Example H102E9

```
> R:= RootDatum("G2");
> U:= QuantizedUEA(R);
> b:= BarAutomorphism(U);
> b(U.3);
(q^10 - q^6 - q^4 + 1)/q^4*F_1^(2)*F_6 + (q^4 - 1)/q^2*F_1*F_5 + F_3
```

A known result states that $T_{\alpha_r}^{-1} = \tau \circ T_{\alpha_r} \circ \tau$. We check that for the quantum group of type C_3, and the third simple root.

```
> U:= QuantizedUEA(RootDatum("C3"));
> t:= AntiAutomorphismTau(U);
> T:= AutomorphismTalpha(U, 3);
> Ti:= Inverse(T);
> f:= t*T*t;
> &and[ Ti(U.i) eq f(U.i) : i in [1..21] ];
true
```

A diagram automorphism maps the canonical basis into itself. We check that for the set of elements of the canonical basis of the quantized enveloping algebra of type D_4 of weight $\alpha_1 + 3\alpha_2 + 2\alpha_3 + 2\alpha_4$. (Here α_i is the i-th simple root.) The chosen diagram automorphism maps this weight to $2\alpha_1 + 3\alpha_2 + \alpha_3 + 2\alpha_4$. Therefore we also compute the elements of the canonical basis of that weight.

```
> U:= QuantizedUEA(RootDatum("D4"));
> p:= SymmetricGroup(4)!(1,3,4);
> d:= DiagramAutomorphism(U, p);
> e1:= CanonicalElements(U, [1,3,2,2]);
> e2:= CanonicalElements(U, [2,3,1,2]);
> &and[ d(x) in e2 : x in e1 ];
true
```

102.10 Kashiwara Operators

`Falpha(m, i)`

Given a monomial m in U^- for some quantized enveloping algebra U, i.e., m must be a monomial in the first n generators of U, where n is the number of positive roots of the corresponding root datum, returns another monomial in the negative part of U that is obtained by applying the i-th Kashiwara operator $\tilde{F}_i$ to m (see Section 102.2.6). Here i must lie between 1 and the rank of the root datum.

`Ealpha(m, i)`

Given a monomial m in U^- for some quantized enveloping algebra U, i.e., m must be a monomial in the first n generators of U, where n is the number of positive roots of the corresponding root datum, return $\tilde{E}_i(m)$ (see Section 102.2.6) if the i-th Kashiwara operator $\tilde{E}_i$ is applicable to m. Otherwise the zero element of U is returned. Here i must lie between 1 and the rank of the root datum.

Example H102E10

```
> R:= RootDatum("F4");
> U:= QuantizedUEA(R);
> m:= U.1*U.5*U.10*U.18*U.24;
> m;
F_1*F_5*F_10*F_18*F_24
> Falpha(m, 3);
F_1*F_6*F_7*F_10*F_18*F_24
> Ealpha(m, 4);
F_1*F_4*F_5*F_7*F_9*F_18*F_24
> Ealpha(m, 2);
0
```

102.11 The Path Model

In this section we describe functions for working with Littelmann's path model (cf. Section 102.2.7). A special role is played by the zero path. The path operators cannot be applied to the zero path. However, on some occasions they do produce the zero path.

`DominantLSPath(R, hw)`

Given a root datum R and a sequence hw of non-negative integers returns the path that is the straight line from the origin to hw.

`Falpha(p, i)`

Given a (non-zero) path p and an integer i between 1 and the rank of the root datum returns the result of applying the path operator f_{α_i} to p (where α_i is the i-th simple root).

`Ealpha(p, i)`

Given a (non-zero) path p and an integer i between 1 and the rank of the root datum returns the result of applying the path operator e_{α_i} to p (where α_i is the i-th simple root).

`WeightSequence(p)`

For a path p this returns the sequence of weights that, along with the sequence of rational numbers, defines the path (cf. Section 102.2.7).

`RationalSequence(p)`

For a path p this returns the sequence of rational numbers that, along with the sequence of weights, defines the path (cf. Section 102.2.7).

`EndpointWeight(p)`

Returns the weight which is the end point of the path p.

`Shape(p)`

Returns the weight which is the shape of the path p.

`WeylWord(p)`

Returns a reduced expression for the element σ of the Weyl group, of shortest length such that $\sigma(\lambda) = \nu_1$, where λ is the shape of the path p, and ν_1 is the first weight in the sequence `WeightSequence(p)`. The reduced expression is represented as a sequence of integers between 1 and the rank of the root datum. In this sequence the index i represents the i-th simple reflection.

`IsZero(p)`

Returns `true` if the path p is the zero path, `false` otherwise.

`p1 eq p2`

Returns `true` if the paths $p1$ and $p2$ are equal, `false` otherwise.

Example H102E11

```
> R:= RootDatum("B2");
> p:= DominantLSPath(R, [ 2, 3 ]);
> p;
LS-path of shape (2 3) ending in (2 3)
> Falpha(p, 1);
LS-path of shape (2 3) ending in (0 5)
> Ealpha(Falpha(p, 1), 1);
LS-path of shape (2 3) ending in (2 3)
> p1:= Falpha(Falpha(Falpha(p, 1), 2), 1);
> p1;
LS-path of shape (2 3) ending in (-1  5)
> WeightSequence(p1);
[
    (5 -7),
    (-2  7)
]
> RationalSequence(p1);
[ 0, 1/7, 1 ]
> WeylWord(p1);
[ 2, 1 ]
```

So $s_2 s_1(2,3) = (5,-7)$.

`CrystalGraph(R, hw)`

For a root datum R and a sequence of non-negative integers hw (of length equal to the rank of the root datum), this function returns the corresponding crystal graph G, along with a sequence of paths. The graph G is a directed labelled graph. The labels on the edges are integers between 1 and the rank of the root system. If there is an edge from i to j with label s, then $f_{\alpha_s}(p_i) = p_j$, where p_i, p_j are the i-th and j-th elements of the sequence of paths returned by this function (and f_{α_s} is the root operator corresponding to the s-th simple root). In other words, the i-th path is the i-th point of the graph G.

Example H102E12

```
> R:= RootDatum("G2");
> G, pp:= CrystalGraph(R, [0,1]);
> G;
Digraph
Vertex  Neighbours
1       2 ;
2       3 ;
3       4 ;
4       5 6 ;
```

```
5       7 ;
6       8 ;
7       9 ;
8       10 ;
9       11 ;
10      11 ;
11      12 ;
12      13 ;
13      14 ;
14      ;
> e:= Edges(G);
> e[10];
[9, 11]
> Label(e[10]);
1
> Falpha(pp[9], 1) eq pp[11];
true
```

102.12 Elements of the Canonical Basis

`CanonicalElements(U, w)`

Given a quantized enveloping algebra U, corresponding to a root datum R of rank r and a sequence w of non-negative integers of length r returns the sequence consisting of the elements of the canonical basis of the negative part of U that are of weight ν, where ν denotes the linear combination of the simple roots of R defined by w (i.e., $\nu = w[1]\alpha_1 + \cdots + w[r]\alpha_r$ and $\alpha_1, \ldots, \alpha_r$ denote the simple roots of R).

Example H102E13

```
> R:= RootDatum("F4");
> U:= QuantizedUEA(R);
> c:= CanonicalElements(U, [1,2,1,1]); c;
[
    F_1*F_3^(2)*F_9*F_24,
    q*F_1*F_3^(2)*F_9*F_24 + F_1*F_3^(2)*F_23,
    q^4*F_1*F_3^(2)*F_9*F_24 + F_1*F_3*F_7*F_24,
    q^5*F_1*F_3^(2)*F_9*F_24 + q^4*F_1*F_3^(2)*F_23 + q*F_1*F_3*F_7*F_24 +
        F_1*F_3*F_21,
    q^4*F_1*F_3^(2)*F_9*F_24 + F_2*F_3*F_9*F_24,
    q^5*F_1*F_3^(2)*F_9*F_24 + q^4*F_1*F_3^(2)*F_23 + q*F_2*F_3*F_9*F_24 +
        F_2*F_3*F_23,
    (q^6 + q^2)*F_1*F_3^(2)*F_9*F_24 + q^2*F_1*F_3*F_7*F_24 +
        q^2*F_2*F_3*F_9*F_24 + F_2*F_7*F_24,
    (q^7 + q^3)*F_1*F_3^(2)*F_9*F_24 + (q^6 + q^2)*F_1*F_3^(2)*F_23 +
        q^3*F_1*F_3*F_7*F_24 + q^3*F_2*F_3*F_9*F_24 + q^2*F_1*F_3*F_21 +
```

```
        q^2*F_2*F_3*F_23 + q*F_2*F_7*F_24 + F_2*F_21,
    q^8*F_1*F_3^(2)*F_9*F_24 + q^4*F_1*F_3*F_7*F_24 + q^4*F_2*F_3*F_9*F_24 +
        q^2*F_2*F_7*F_24 + F_3*F_4*F_24,
    q^9*F_1*F_3^(2)*F_9*F_24 + q^8*F_1*F_3^(2)*F_23 + q^5*F_1*F_3*F_7*F_24 +
        q^5*F_2*F_3*F_9*F_24 + q^4*F_1*F_3*F_21 + q^4*F_2*F_3*F_23 +
        q^3*F_2*F_7*F_24 + q*F_3*F_4*F_24 + q^2*F_2*F_21 + F_3*F_18
]
> b:= BarAutomorphism(U);
> [ b(u) eq u : u in c ];
[ true, true, true, true, true, true, true, true, true, true ]
```

All elements of the canonical basis are invariant under the bar-automorphism.

Example H102E14

In the next example we show how to use the crystal graph to determine whether an element of the canonical basis acting on the highest weight vector of an irreducible module gives zero or not.

```
> U:= QuantizedUEA(RootDatum("A2"));
> G, p:= CrystalGraph(RootDatum(U), [1,1]);
> e:= Edges(G);
> for edge in e do
> print edge, Label(edge);
> end for;
[1, 2] 1
[1, 3] 2
[2, 4] 2
[3, 5] 1
[4, 6] 2
[5, 7] 1
[6, 8] 1
[7, 8] 2
```

We see that $f_{\alpha_1} f_{\alpha_2} f_{\alpha_2} f_{\alpha_1}(p_1) = p_8$ (where p_i is the i-th path in p). We apply the same sequence of Kashiwara operators to the identity element of U.

```
> Falpha(Falpha(Falpha(Falpha(One(U), 1), 2), 2), 1);
F_1*F_2*F_3
```

Now the element of the canonical basis with this principal monomial (see Section 102.2.6) acting on the highest weight vector of the irreducible module with highest weight `[1,1]` gives a non-zero result. The weight of this monomial is $2\alpha_1 + 2\alpha_2$. All other elements of the canonical basis of this weight give zero, as there is only one point of the crystal graph that gives a monomial of this weight.

```
> V:= HighestWeightModule(U, [1,1]);
> ce:= CanonicalElements(U, [2,2]);
> ce;
[
    F_1^(2)*F_3^(2),
    (q^3 + q)*F_1^(2)*F_3^(2) + F_1*F_2*F_3,
    q^4*F_1^(2)*F_3^(2) + q*F_1*F_2*F_3 + F_2^(2)
```

```
]
> v0:= V.1;
> ce[2]^v0;
V: (  0    0    0    0    0    0    0 -1/q)
> ce[1]^v0;
V: (0 0 0 0 0 0 0 0)
> ce[3]^v0;
V: (0 0 0 0 0 0 0 0)
```

102.13 Homomorphisms to the Universal Enveloping Algebra

`QUAToIntegralUEAMap(U)`

Given a quantized enveloping algebra U returns the map from U onto the integral form of the universal enveloping algebra of the corresponding Lie algebra (cf. Section 102.2.5). We refer to Section 100.17 for an account of universal enveloping algebras in MAGMA.

Example H102E15

```
> U:= QuantizedUEA(RootDatum("C3"));
> f:= QUAToIntegralUEAMap(U);
> p:= CanonicalElements(U, [1,2,1]);
> [ f(u) : u in p ];
[
    y_1*y_2^(2)*y_3,
    2*y_1*y_2^(2)*y_3 + y_1*y_2*y_5,
    y_1*y_2^(2)*y_3 + y_1*y_2*y_5 + y_1*y_7,
    y_1*y_2^(2)*y_3 + y_2*y_3*y_4 - y_2*y_6,
    2*y_1*y_2^(2)*y_3 + y_1*y_2*y_5 + y_2*y_3*y_4 - y_2*y_6 + y_4*y_5,
    2*y_1*y_2^(2)*y_3 + y_1*y_2*y_5 + 2*y_2*y_3*y_4 - y_2*y_6 + y_4*y_5,
    y_1*y_2^(2)*y_3 + y_1*y_2*y_5 + y_2*y_3*y_4 + y_1*y_7 + y_4*y_5 + y_8
]
```

So this allows one to construct elements of the canonical basis of a universal enveloping algebra (of a semisimple Lie algebra).

102.14 Bibliography

[**Gra04**] W. A. de Graaf. Five constructions of representations of quantum groups. *Note di Matematica*, 22(1):27–48, 2003/04.

[**Gra01**] W. A. de Graaf. Computing with quantized enveloping algebras: PBW-type bases, highest-weight modules, R-matrices. *J. Symbolic Comput.*, 32(5):475–490, 2001.

[**Gra02**] W. A. de Graaf. Constructing canonical bases of quantized enveloping algebras. *Experimental Mathematics*, 11(2):161–170, 2002.

[**Jan96**] J. C. Jantzen. *Lectures on Quantum Groups*, volume 6 of *Graduate Studies in Mathematics*. American Mathematical Society, 1996.

[**Kas96**] M. Kashiwara. Similarity of crystal bases. In *Lie algebras and their representations (Seoul, 1995)*, pages 177–186. Amer. Math. Soc., Providence, RI, 1996.

[**Lit94**] P. Littelmann. A Littlewood-Richardson rule for symmetrizable Kac-Moody algebras. *Invent. Math.*, 116(1-3):329–346, 1994.

[**Lit95**] P. Littelmann. Paths and root operators in representation theory. *Ann. of Math. (2)*, 142(3):499–525, 1995.

[**Lus90**] G. Lusztig. Quantum groups at roots of 1. *Geom. Dedicata*, 35(1-3):89–113, 1990.

[**Lus93**] G. Lusztig. *Introduction to quantum groups*. Birkhäuser Boston Inc., Boston, MA, 1993.

[**Lus96**] G. Lusztig. Braid group action and canonical bases. *Adv. Math.*, 122(2):237–261, 1996.

103 GROUPS OF LIE TYPE

Chapter 103
GROUPS OF LIE TYPE

103.1 Introduction

This chapter describes MAGMA functions for computing with groups of Lie type. These functions are based on [CMT04] for split types, and [Hal05] for twisted types.

Given an extended root datum and ring with a Γ-action, a group of Lie type can be constructed in MAGMA. Such groups include reductive Lie groups (when the ring is $\mathbf{R}$ or $\mathbf{C}$), reductive algebraic groups (when the ring is an algebraically closed field), and finite groups of Lie type (when the ring is a finite field).

103.1.1 The Steinberg Presentation

The approach to computation in split groups of Lie type described here is based on the Steinberg presentation[Ste62] Let G be a split group of Lie type with root datum R over the ring k. Suppose the roots of R are $\alpha_1, \ldots, \alpha_{2N}$ ordered as in Section 97.5 and n is the rank of R. Then G contains *root elements* $x_r(t) = x_{\alpha_r}(t)$ for t in k. If R is semisimple, the root elements generate G. In the general case, it is necessary to introduce extra torus elements. Let $Y = \mathbf{Z}^d$ be the coroot space of the root datum. The torus is taken to be the abelian group $Y \otimes k^\times$, represented as the set of vectors in k^d with each component invertible, and multiplication is performed componentwise. The Weyl group of G is just the Coxeter group of the root datum RD. Redundant generators n_r are also included, corresponding to the generators s_r of the Weyl group.

Since the generating set is parametrised by field elements it is generally not possible to define G within the category of finitely presented groups `GrpFP`, so groups of Lie type form their own category, `GrpLie`.

Note that groups of Lie type in MAGMA are designed primarily for fields whose elements are exact. While it is possible to define these groups over real and complex fields (Chapter 25), no attempt has been made to control rounding error in this case.

103.1.2 Bruhat Normalisation

The Bruhat decomposition [Car93, Chapter 2] gives us a useful normal form for elements of a split group of Lie type defined over a field k. Every $g \in G$ can be written in the form $uh\dot{w}u'$ where

1. u is a unipotent element written in the form $\prod_{r=1}^{N} x_r(t_r)$;
2. h is a torus element represented as an element of R^d with each entry invertible;
3. $\dot{w} = \dot{s}_{r_1} \cdots \dot{s}_{r_k}$ where $s_{r_1} \cdots s_{r_k}$ is a reduced word for w in the Weyl group.
4. $u' = \prod_{r \in \Phi_w^+} x_r(t'_r)$ where $\Phi_w^+ = \{r \mid \alpha_r \text{ and } \alpha_r w \text{ are positive}\}$ and the terms are in the usual order.

103.1.3 Twisted Groups of Lie type

Let G be a connected reductive linear algebraic group defined over the field k. We say that H is a *form* of G if it there is a $\bar{k}$-isomorphism between then, for $\bar{k}$ the algebraic closure of k. If some maximal torus of $G(\bar{k})$ is a k-split torus, we say that G is *split*, otherwise G is *twisted*. If some maximal torus of $G(\bar{k})$ is defined over k, we say that G is *quasisplit*. There is a unique split form of every reductive linear algebraic group.

The group $\Gamma := \mathrm{Gal}(\bar{k} : k)$ acts on G in the usual way and G is a Γ-group in the sense of the Section 68.10. The group $\mathrm{Aut}(G)$ of algebraic automorphisms of G is also a Γ-group. The twisted forms of G are in one-to-one correspondence with the 1-cocycles of Γ on $\mathrm{Aut}(G)$ and the forms are conjugate if and only if the cocycles are cohomologous. For practical purposes it is sufficient to compute the cohomology of $\Gamma = \mathrm{Gal}(K : k)$ on $\mathrm{Aut}_K(G)$ for some finite Galois extension K of k, where $\mathrm{Aut}_K(G)$ is the group of K-algebraic automorphisms of G.

The action of Γ on G induces an action on the root datum of G, and so we get an extended root datum. If G is quasisplit, then it is determined by the extended root datum and the action of Γ on K. In general, a cocycle is required to fully determine G.

103.2 Constructing Groups of Lie Type

103.2.1 Split Groups

The following optional parameters are common to most of the intrinsics described in this section:

`Normalising` BoolElt *Default :* `true`

The flag `Normalising` determines whether elements will be automatically converted to Bruhat form. This flag is automatically set to `false`if the group is defined over a nonfield.

`Isogeny` BoolElt *Default :* "*Ad*"

`Signs` Any *Default :* 1

The optional parameters `Isogeny` and `Signs` can take the values described in Section 97.2.

`Method` MonStgElt *Default :* "*Default*"

The method to be used for operations with unipotent elements. See [CHM08] for more details on the algorithms. Possible values are

- `"CollectionFromLeft"` uses collection from left.
- `"CollectionFromOutside"` uses collection from outside.
- `"Classical"` uses formulas for classical types [CHM08]. This is only available for groups defined over a sparse (classical) root datum.
- `"Collection"` will choose the best of the above methods automatically.
- `"SymbolicFromLeft"` uses Hall polynomials, which are computed using collection from left.

- `"SymbolicFromOutside"` uses Hall polynomials, which are computed using collection from outside.
- `"SymbolicClassical"` uses Hall polynomials, which are computed by formulas. This is only available for groups defined over a sparse (classical) root datum.
- `"Symbolic"` will choose the best symbolic method automatically.
- `"Default"` will choose the best of all above methods automatically.

`GroupOfLieType(N, k)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Cartan name given by the string N (see Section 95.6) over the ring k.

`GroupOfLieType(N, q)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Cartan name given by the string N (see Section 95.6) over the finite field of order q.

`GroupOfLieType(W, k)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Weyl group W over the ring k. The group W must be a finite Coxeter group, given either as a permutation group or as a reflection group.

`GroupOfLieType(W, q)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Weyl group W over the finite field of order q. The group W must be a finite Coxeter group, given either as a permutation group or as a reflection group.

`GroupOfLieType(R, k)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with root datum R over the ring k.

`GroupOfLieType(R, q)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with root datum R over the finite field of order q.

`GroupOfLieType(C, k)`

`GroupOfLieType(D, k)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Cartan matrix C or Dynkin digraph D, over the ring k.

`GroupOfLieType(C, q)`

`GroupOfLieType(D, q)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the group of Lie type with Cartan matrix C or Dynkin digraph D, over the finite field of order q.

`SimpleGroupOfLieType(X, n, k)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the simple group of Lie type with Cartan name X_n over the ring k, where the Cartan name is given by the string X and integer n (see also Section 95.6).

`SimpleGroupOfLieType(X, n, q)`

`Isogeny`	BoolElt	*Default :* "*Ad*"
`Signs`	Any	*Default :* 1
`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

Construct the simple group of Lie type with name X_n over the finite field of order q, where the Cartan name is given by the string X and integer n (see also Section 95.6).

`GroupOfLieType(L)`

The group of Lie type corresponding to the Lie algebra L. The Lie algebra must be the algebraic (i.e., it must correspond to some group), and Magma must be able to determine that it is algebraic.

`IsNormalising(G)`

Returns the value of the flag `Normalising` of the group of Lie type G.

Example H103E1

```
> G := GroupOfLieType("E8", 2);
> G;
G: Group of Lie type E8 over Finite field of size 2
```

103.2.2 Galois Cohomology

If G is a linear algebraic group defined over the field k and L is the algebraic closure of k, then the group $\Gamma := \mathrm{Gal}(L : k)$ acts on G in the usual way and G becomes a Γ-group in the sense of the Section 68.10 and $\mathrm{Aut}(G)$, the group of algebraic automorphisms of G also becomes a Γ-group.

Now the twisted forms of G are in one-to-one correspondence to the 1-cocycles of Γ on $\mathrm{Aut}(G)$ and the forms are conjugate if and only if the cocycles are cohomologous.

For practical purposes it is sufficient to compute the cohomology of $\mathrm{Gal}(K : k)$ on $\mathrm{Aut}_K(G)$ for some finite Galois field extension of k, where $\mathrm{Aut}_K(G)$ is the group of K-algebraic automorphisms of G.

These functions are based on [Hal05].

`GammaGroup(k, G)`

Returns the group of Lie type G as a Γ-group with $\Gamma = \mathrm{Gal}(K : k)$, where K is the base field of G. The field k must be a subfield of K.

`GammaGroup(k, A)`

Returns the group $A = \mathrm{Aut}_K(G)$ of automorphisms of the group of Lie type G as a Γ-group with $\Gamma = \mathrm{Gal}(K : k)$, where K is the base field of G. The field k must be a subfield of K.

`ActingGroup(G)`

`ActingGroup(A)`

Given the group of Lie type G or the group A of its automorphisms as a Γ-group, return $\Gamma = \mathrm{Gal}(K : k)$ together with the map m from the abstract Galois group Γ into the set of field automorphisms, such that $m(\gamma)$ is the actual field automorphism for every $\gamma \in \Gamma$.

`ExtendGaloisCocycle(c)`

`GBAl`	MonStgElt	*Default :* "*Walk*"
`Printeqs`	BoolElt	*Default :* `false`

The analogue to `ExtendCocycle`. Given a cocycle c in $H^1(\Gamma, A/A_0)$, where $A = \mathrm{Aut}_K(G)$ and $\Gamma = \mathrm{Gal}(K : k)$, extend the cocycle to a cocycle in $H^1(\Gamma, A)$. The optional parameter `GBAl` can be used to set the algorithm used for computing the Gröbner bases. The parameter `Printeqs` may be used to print out the polynomials whose Gröbner bases are computed. The current implementation only works for finite fields.

`GaloisCohomology(A)`

`GBAl`	MonStgElt	*Default :* "*Walk*"
`Printeqs`	BoolElt	*Default :* `false`
`Recompute`	BoolElt	*Default :* `false`

Computes the Galois cohomology $H^1(\Gamma, \mathrm{Aut}_K(G))$, where A is the automorphism group of G as a Γ-group returned by `GammaGroup` and $\Gamma = \mathrm{Gal}(K : k)$. The optional parameter `GBAl` can be used to set the algorithm used for computing the Gröbner bases. The parameter `Printeqs` may be used to print out the polynomials whose Gröbner bases are computed. And `Recompute` may be used to recompute the Galois cohomology. The current implementation only works for finite fields.

`IsInTwistedForm(x, c)`

Returns `true` if and only if the element x of a group of Lie type is contained in the twisted form of its parent defined by the cocycle c.

Example H103E2

Compute the Galois cohomology of $A_3(5^2)$:

```
> q := 5;
> k := GF(q);
> K := GF(q^2);
>
> G := GroupOfLieType( "A3", K : Isogeny:="SC" );
> A := AutomorphismGroup(G);
>
> AGRP := GammaGroup( k, A );
> Gamma,m := ActingGroup(AGRP);
> Gamma;
Symmetric group Gamma acting on a set of cardinality 2
Order = 2
    (1, 2)
> m;
Mapping from: GrpPerm: Gamma to Set of all maps from GF(5^2) to GF(5^2)
given by a rule [no inverse]
> action  := GammaAction(AGRP);
>
> time GaloisCohomology(AGRP);
[
    [
        One-Cocycle
        defined by [
        Automorphism of $: Group of Lie type A3 over Finite field of size 5^2
        given by: Mapping from: $: Group of Lie type  to $: Group of Lie type
        Composition of Mapping from: $: Group of Lie type  to $: Group of
        Lie type  given by a rule and
        Mapping from: $: Group of Lie type  to $: Group of Lie type
        given by a rule
        Decomposition:
          Mapping from: GF(5^2) to GF(5^2)
        Composition of Mapping from: GF(5^2) to GF(5^2) given by a rule and
        Mapping from: GF(5^2) to GF(5^2) given by a rule,
          Id($),
          1
        ]
    ],
    [
        One-Cocycle
        defined by [
        Automorphism of $: Group of Lie type A3 over Finite field of size 5^2
        given by: Mapping from: $: Group of Lie type  to $: Group of Lie type
        Composition of Mapping from: $: Group of Lie type  to $: Group of
        Lie type  given by a rule and
        Mapping from: $: Group of Lie type  to $: Group of Lie type
```

```
        given by a rule
        Decomposition:
          Mapping from: GF(5^2) to GF(5^2)
        Composition of Mapping from: GF(5^2) to GF(5^2) given by a rule and
        Mapping from: GF(5^2) to GF(5^2) given by a rule,
          (1, 3),
          1
        ]
    ]
]
Time: 0.470
```

Now create the trivial cocycle:

```
> TrivialOneCocycle( AGRP );
One-Cocycle
defined by [
Automorphism of $: Group of Lie type A3 over Finite field of size 5^2
given by: Mapping from: $: Group of Lie type  to $: Group of Lie type
given by a rule
Decomposition:
  Mapping from: GF(5^2) to GF(5^2) given by a rule,
  Id($),
  1
]
>
```

And now the cocycle defining the group ${}^2A_3(5)$ and check for two elements if they are contained in ${}^2A_3(5)$:

```
> c := OneCocycle( AGRP, [GraphAutomorphism(G, Sym(3)!(1,3))] );
>
> x := Random(G);
> IsInTwistedForm( x, c );
false
>
> x := elt< G | <1,y>, <3,y @ m(Gamma.1)> > where y is Random(K);
> IsInTwistedForm( x, c );
true
>
```

103.2.3 Twisted Groups

The description of the twisted groups of Lie type is based on the extended root data, as described in the Section 97.1.7. These functions are mainly based on [Hal05].

`TwistedGroupOfLieType(c)`

Given the cocycle c on the group of automorphisms of a split group of Lie type G, return the twisted form of G, defined by that cocycle.

`TwistedGroupOfLieType(R, k, K)`

`Normalising`	BoolElt	*Default :* `true`
`Method`	MonStgElt	*Default :* "*Default*"

The twisted group of Lie type defined over the field k with coefficients in the field K corresponding to the twisted root datum R.

`BaseRing(G)`

`CoefficientRing(G)`

The coefficient ring of the (twisted) group of Lie type G, that is the base ring of the untwisted overgroup of G.

`DefRing(G)`

The ring over which the (twisted) group of Lie type G is defined. If G is split, this is the same as the base ring of G.

`UntwistedOvergroup(G)`

The untwisted overgroup, inside which the twisted group of Lie type G was constructed.

Example H103E3

The twisted group ${}^2A_3(5)$ as a subgroup of $A_3(5^2)$.

```
> R := RootDatum("A3" : Twist := 2);
> G := TwistedGroupOfLieType(R,5,25);
> G;
G: Twisted group of Lie type 2A3,2 over GF(5) with entries over GF(5^2)
> BaseRing(G);
Finite field of size 5^2
> DefRing(G);
Finite field of size 5
> UntwistedOvergroup(G);
Group of Lie type A3 over GF(5^2)
```

`RelativeRootElement(G,delta,t)`

The relative root element corresponding to the relative root δ of the twisted group of Lie type G and the field elements given by the sequence t. This is the element $u_\delta(t)$ in [Hal05, (4.5)].

Example H103E4

Here we create the same group as in the previous example, but using a cocycle.

```
> q := 5; k := GF(q); K := GF(q^2);
>
> G := GroupOfLieType( "A3", K );
> A := AutomorphismGroup(G);
>
> AGRP := GammaGroup( k, A );
> c := OneCocycle( AGRP, [GraphAutomorphism(G, Sym(3)!(1,3))] );
>
> T := TwistedGroupOfLieType(c);
> T eq TwistedGroupOfLieType(RootDatum("A3":Twist:=2),k,K);
true
> G eq UntwistedOvergroup(T);
true
>
> x := Random(G); x in T;
false
>
> x := RelativeRootElement(T,2,[Random(K)]); x;
x1($.1^22) x3($.1^14)
> x in T;
true
```

103.3 Operations on Groups of Lie Type

Many of the basic operations for Coxeter groups are shortcuts for obtaining information about the underlying root datum (Chapter 97). Such functions are listed here; see Sections 97.3, 97.4, 97.5, and 98.4 for more details and examples of their use.

`G eq H`

Returns `true` iff the groups of Lie type G and H are equal.

`G subset H`

Returns `true` iff the group of Lie type G is a subset of H.

`IsAlgebraicallyIsomorphic(G, H)`

Returns `true` if the semisimple groups G and H are isomorphic as algebraic groups (i.e. they have the same base rings and isomorphic root data). If `true`, then the second value returned is an isomorphism.

`IsIsogenous(G, H)`

Returns `true` if G and H are isogenous. The groups must be semisimple and defined over the same field. If `true`, the subsequent values returned are: the corresponding adjoint group G_{ad}, the homomorphisms $G_{ad} \to G$ and $G_{ad} \to H$, the corresponding simply connected root datum G_{sc}, and the homomorphisms $G \to G_{sc}$ and $H \to G_{sc}$.

`IsCartanEquivalent(G, H)`

Returns `true` if, and only if, the groups of Lie type G and H are Cartan equivalent, i.e. they have isomorphic Dynkin diagrams and defined over the same ring.

`BaseRing(G)`

`CoefficientRing(G)`

The base ring k of the group of Lie type G.

`BaseExtend(G, K)`

Given a group of Lie type G with base ring k and a larger ring K, return the group $G(K)$ gotten by extending the base ring and the injection $G \to G(K)$.

`ChangeRing(G, K)`

Given a group of Lie type G and a ring K, return the group with the same root datum, but defined over a different ring.

`Generators(G)`

Generators for the group of Lie type G as an abstract group. This is currently only implemented when the base ring is a finite field.

`NumberOfGenerators(G)`

`Ngens(G)`

The number of generators for the group of Lie type G as an abstract group. This is currently only implemented when the base ring is a finite field.

`AlgebraicGenerators(G)`

A set of generators for the group of Lie type G as an algebraic group.

`NumberOfAlgebraicGenerators(G)`

`Nalggens(G)`

The number of generators for the group of Lie type G as an algebraic group.

Example H103E5

```
> k<z> := GF(4);
> G := GroupOfLieType("A2", k : Normalising:=false);
> Generators(G);
[ x1(1) , x4(1) , x1(z) , x4(z) , x2(1) , x5(1) , x2(z) , x5(z) , ( z   1) ,
(1   z)  ]
> AlgebraicGenerators(G);
[ x1(1) , x2(1) , x4(1) , x5(1) , ( z   1) , ( 1   z)  ]
```

`Order(G)`
`#G`

The order of the group of Lie type G.

`FactoredOrder(G)`

The factored order of the group of Lie type G.

`Dimension(G)`

The dimension of the group of Lie type G, considered as an algebraic variety.

Example H103E6

```
> G := GroupOfLieType("G2", 3);
> Order(G);
4245696
> FactoredOrder(G);
[ <2, 8>, <13, 1>, <3, 6>, <7, 1> ]
> G := GroupOfLieType("G2", Rationals());
> Order(G);
Infinity
> Dimension(G);
14
```

`CartanName(G)`

The Cartan name of the group of Lie type G.

`RootDatum(G)`

The root datum of the group of Lie type G.

`DynkinDiagram(G)`

Print the Dynkin diagram of the group of Lie type G.

`CoxeterDiagram(G)`

Print the Coxeter diagram of the group of Lie type G.

`CoxeterMatrix(G)`

The Coxeter matrix of the group of Lie type G.

`CoxeterGraph(G)`

The Coxeter graph of the group of Lie type G.

`CartanMatrix(G)`

The Cartan matrix of the group of Lie type G.

`DynkinDigraph(G)`

The Dynkin digraph of the group of Lie type G.

`Rank(G)`
`ReductiveRank(G)`

The reductive rank of the group of Lie type G, i.e. the dimension of the underlying root datum.

`SemisimpleRank(G)`

The semisimple rank of the group of Lie type G, i.e. the rank of the underlying root datum.

`CoxeterNumber(G)`

The Coxeter number of the group of Lie type G, i.e. the order of the Coxeter element in the Weyl group of G.

`WeylGroup(G)`
`WeylGroup(GrpPermCox, G)`

The Weyl group of the group of Lie type G as a permutation Coxeter group. This is a crystallographic Coxeter group, see Chapter 98.

`WeylGroup(GrpFPCox, G)`

The Weyl group of the group of Lie type G as a finitely presented Coxeter group. This is a crystallographic Coxeter group, see Chapter 98.

`WeylGroup(GrpMat, G)`

The Weyl group of the group of Lie type G as a reflection group. This is a crystallographic Coxeter group, see Chapter 99.

`FundamentalGroup(G)`

The fundamental group of the group of Lie type G, together with the projection of the weight lattice onto the fundamental group.

`IsogenyGroup(G)`

The isogeny group of the group of Lie type G, together with its injection into the fundamental group.

`CoisogenyGroup(G)`

The coisogeny group of the group of Lie type G, together with its projection onto the fundamental group.

103.4 Properties of Groups of Lie Type

`IsFinite(G)`

Return `true` if and only if the group of Lie type G is finite.

`IsAbelian(G)`

Returns `true` if the group of Lie type G is abelian.

`IsSimple(G)`

Returns `true` if the group of Lie type G is a simple group as an algebraic group, ie, G has no proper *connected* normal subgroups. This is true if, and only if, the underlying root datum is irreducible. Note that this does not usually mean that G is simple as an abstract group. In previous releases of Magma this function was incorrectly called `IsIrreducible`.

`IsSimplyLaced(G)`

Returns `true` if the group of Lie type G is simply laced, i.e. its Dynkin diagram contains no multiple bonds.

`IsSemisimple(G)`

Returns `true` if the group of Lie type G is semisimple.

`IsAdjoint(G)`

Returns `true` if, and only if, the group of Lie type G is adjoint (i.e. the isogeny group is trivial).

`IsWeaklyAdjoint(G)`

Returns `true` if, and only if, the group of Lie type G is weakly adjoint, i.e. its isogeny group is isomorphic to $\mathbf{Z}^n$, where n is the difference between the rank and the semisimple rank of G. Note that if G is semisimple then this function is identical to `IsAdjoint`.

`IsSimplyConnected(G)`

Returns `true` if, and only if, the group of Lie type G is simply connected (i.e. the isogeny group is equal to the fundamental group, i.e. the coisogeny group is trivial).

`IsWeaklySimplyConnected(G)`

Returns `true` if, and only if, the group of Lie type G is weakly simply connected, i.e. its coisogeny group is isomorphic to $\mathbf{Z}^n$, where n is the difference between the rank and the semisimple rank of G. Note that if G is semisimple then this function is identical to IsSimplyConnected.

`IsSplit(G)`

Returns `true` if and only if the group of Lie type G is split.

`IsTwisted(G)`

Returns `true` if and only if the group of Lie type G is twisted.

103.5 Constructing Elements

`elt< G | L >`

Given a group of Lie type G over the ring R and a list L of appropriate objects, construct an element of G. Suppose the underlying root datum has dimension d, rank n, and roots $\alpha_1, \ldots, \alpha_{2N}$. Each entry in the list can be one of the following:

1. A tuple $< r, t >$ where $r = 1, \ldots, 2N$ and $t \in R$. This corresponds to the unipotent term $x_r(t)$.
2. A sequence of tuples as in item (1).
3. A sequence $[t_1, \ldots, t_N]$ of elements of R. This corresponds to the unipotent element $x_1(t_1) \cdots x_N(t_N)$.
4. An integer $r = 1, \ldots, 2N$. This corresponds to the Weyl group representative n_r.
5. A Weyl group element w, either as a word or as a permutation. This corresponds to the Weyl group representative $\dot{w}$.
5. A vector $v \in R^d$ with each entry invertible. This corresponds to an element of the torus.
6. An element of G.

`Identity(G)`
`Id(G)`
`G ! 1`
`elt< G | >`

The identity element of the group of Lie type G.

Example H103E7

```
> G := GroupOfLieType("A5", Rationals() : Normalising := false);
> V := VectorSpace(Rationals(), 5);
> NumPosRoots(G);
15
> elt< G | <5,1/2>, 1,3,2, [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15],
>          V![6,1/3,-1,3,2/3] >;
x5(1/2) n1 n3 n2 x1(1) x2(2) x3(3) x4(4) x5(5) x6(6) x7(7) x8(8) x9(9)
x10(10) x11(11) x12(12) x13(13) x14(14) x15(15) ( 6 1/3  -1   3 2/3)
```

`TorusTerm(G, r, t)`

The torus term $h_r(t) = \alpha_r^\star \otimes t$ in the group of Lie type G, where r is the index of the coroot $\alpha_r^\star$ and t an element of the base ring of G.

`CoxeterElement(G)`

The Coxeter element of the group of Lie type G, i.e. the representative of the Coxeter element in the Weyl group of G.

`Random(G)`

`Uniform` BoolElt *Default :* `true`

An element of the finite group of Lie type G chosen at random. The base ring of G must be finite. If the optional parameter `Uniform` is set to `true`, the random elements to be distributed uniformly. If the optional parameter `Uniform` is set to `false`, this function is much faster but the random elements are not distributed uniformly. Instead each double coset of the Borel subgroup occurs with equal frequency, and the elements are uniformly distributed within each double coset.

`Eltlist(g)`

The list corresponding to the element g of a group of Lie type.

`CentrePolynomials(G)`

`CenterPolynomials(G)`

A set of polynomials which are satisfied by the coordinates of a torus element h of the group of Lie type G if, and only if, h is in the centre of G.

Example H103E8

The centre of a semisimple group is finite, so the centre polynomials can be used to find all central elements.

```
> G := GroupOfLieType("B3", Rationals() : Isogeny:="SC");
> pols := CentrePolynomials(G);
> pols;
{
    -h[2] + h[3]^2,
    h[1]^2 - h[2],
    -h[1]*h[3]^2 + h[2]^2
}
> S := Scheme(AffineSpace(Rationals(), 3), Setseq(pols));
> pnts := RationalPoints(S);
> pnts;
{@ (0, 0, 0), (1, 1, -1), (1, 1, 1) @}
```

The rational points of S can be converted into elements of G, taking care to eliminate any point which has a coordinate equal to zero:

```
> V := VectorSpace(Rationals(), 3);
> [ elt< G | V!Eltseq(pnt) > : pnt in pnts | &*Eltseq(pnt) ne 0 ];
[ (1  1 -1) , 1 ]
```

103.6 Operations on Elements

103.6.1 Basic Operations

`g * h`

The product of two elements of a group of Lie type. If the Normalising flag is set for the group, then the product is normalised using the algorithms of [CMT04, CHM08]. Otherwise, the words are just concatenated.

Example H103E9

If the `Normalising` flag is set, the product is normalised, otherwise multiplication is just concatenation.

```
> G := GroupOfLieType("G2", GF(3) : Normalising:=false );
> V := VectorSpace(GF(3),2);
> g := elt< G | 1,2,1,2, V![2,2], <1,2>,<5,1> >;
> h := elt< G | <3,2>, V![1,2], 1 >;
> g*h;
n1 n2 n1 n2 (2 2) x1(2) x5(1) x3(2) (1 2) n1
> H := GroupOfLieType("G2", GF(3) : Normalising:=true  );
> g := elt< H | 1,2,1,2, V![2,2], <1,2>,<5,1> >;
> h := elt< H | <3,2>, V![1,2], 1 >;
```

```
> g*h;
x2(1) x3(1) (1 2) n1 n2 n1 n2 n1 x4(1)
```

`g ^ -1`

`Inverse(G)`

The inverse of the element g of a group of Lie type.

`g ^ n`

The nth power of the element g of a group of Lie type.

`g ^ h`

The conjugate $h^{-1}gh$, where g and h are elements of a group of Lie type.

`(g, h)`

`Commutator(g, h)`

The commutator $g^{-1}h^{-1}gh$ of g and h, where g and h are elements of a group of Lie type.

`Normalise(~g)`

`Normalize(~g)`

`Normalise(g)`

`Normalize(g)`

Normalise the element g of a group of Lie type G. The procedural form is slightly more efficient than the functional form. If the `Normalise` flag is set for G, this operation has no effect. This uses the algorithms of [CMT04, CHM08].

Example H103E10

Arithmetic in groups of Lie type.

```
> k<z> := GF(4);
> G := GroupOfLieType("C3", k);
> V := VectorSpace(k, 3);
> g := elt< G | 1,2,3, <3,z>,<4,z^2>, V![1,z^2,1] >;
> g;
n1 n2 n3 x3(z) x4(z^2) ( 1 z^2   1)
> h := elt< G | [0,1,z,1,0,z^2,1,1,z] >;
> h;
x2(1) x3(z) x4(1) x6(z^2) x7(1) x8(1) x9(z)
> g * h^-1;
x3(1) x5(z) x6(z^2) x8(1) (z^2 z^2   z) n1 n2 n3 x3(z^2) x5(z^2)
> g^3;
x3(z) x5(1) x7(z^2) x8(z^2) ( 1   1   z) n1 n2 n3 n1 n2 n3 n1 n2 n3 x1(1)
```

```
x2(z^2) x3(1) x4(z) x7(z) x9(z)
```

103.6.2 Decompositions

`Bruhat(g)`

Given an element g of a group of Lie type the Bruhat decomposition of g is returned. The function returns elements u, h, $\dot{w}$, u' with the properties described in Subsection 103.1.3 and so that $g = uh\dot{w}u'$.

Example H103E11

```
> k<z> := GF(4);
> G := GroupOfLieType("C3", k);
> V := VectorSpace(k, 3);
> g := elt< G | 1,2,3, <3,z>,<4,z^2>, V![1,z^2,1] >;
> Normalise(g);
x7(z^2) x8(z^2) (z^2 z^2   z) n1 n2 n3 x3(1) x6(z)
> u, h, w, up := Bruhat(g);
> u; h; w; up;
x7(z^2) x8(z^2)
(z^2 z^2   z)
n1 n2 n3
x3(1) x6(z)
```

`MultiplicativeJordanDecomposition(x)`

The multiplicative Jordan decomposition of the element x of the group of Lie type.

103.6.3 Conjugacy and Cohomology

`ConjugateIntoTorus(g)`

Given an semisimple element g in a finite group of Lie type, return a torus element t and conjugator x such that $t = xgx^{-1}$. The elements returned may be defined over a larger field that the input element.

`ConjugateIntoBorel(g)`

Given a semisimple element g in a finite group of Lie type, return a Borel element b and conjugator x such that $b = xgx^{-1}$. The elements returned may be defined over a larger field that the input element. Although any element of a group of Lie type can be conjugated into the Borel subgroup, this function is currently only implemented for semisimple elements.

`Lang(c, q)`

Given an element c in a finite group of Lie type and q a power of the characteristic, return a solution a of the Lang equation $c = a^- Fa$. Here F is the Frobenius automorphism gotten by q powers in the field.

103.7 Properties of Elements

`IsSemisimple(x)`

Return `true` if, and only if, the element x of the group of Lie type is semisimple.

`IsUnipotent(x)`

Return `true` if, and only if, the element x of the group of Lie type is unipotent.

`IsCentral(x)`

Return `true` if, and only if, the element x of the group of Lie type is in the centre of its parent group.

103.8 Roots, Coroots and Weights

The roots are stored as an indexed set

$$\{@ \alpha_1, \ldots, \alpha_N, \alpha_{N+1}, \ldots, \alpha_{2N} @\},$$

where $\alpha_1, \ldots, \alpha_N$ are the positive roots in an order compatible with height; and $\alpha_{N+1}, \ldots, \alpha_{2N}$ are the corresponding negative roots (i.e. $\alpha_{i+N} = -\alpha_i$). The simple roots are $\alpha_1, \ldots, \alpha_n$ where n is the rank.

Many of these functions have an optional argument `Basis` which may take one of the following values

1. `"Standard"`: the standard basis for the (co)root space. This is the default.
2. `"Root"`: the basis of simple (co)roots.
3. `"Weight"`: the basis of fundamental (co)weights (see Subsection 99.8.3 below).

103.8.1 Accessing Roots and Coroots

`RootSpace(G)`
`CorootSpace(G)`

The lattice containing the (co)roots of the group of Lie type G.

`SimpleRoots(G)`
`SimpleCoroots(G)`

The simple (co)roots of the group of Lie type G as the rows of a matrix.

`NumberOfPositiveRoots(G)`
`NumPosRoots(G)`

The number of positive roots of the group of Lie type G.

`Roots(G)`
`Coroots(G)`

`Basis` MonStgElt *Default :* "*Standard*"

An indexed set containing the (co)roots of the group of Lie type G.

`PositiveRoots(G)`
`PositiveCoroots(G)`

`Basis` MonStgElt *Default :* "*Standard*"

An indexed set containing the positive (co)roots of the group of Lie type G.

`Root(G, r)`
`Coroot(G, r)`

`Basis` MonStgElt *Default :* "*Standard*"

The rth (co)root of the group of Lie type G.

`RootPosition(G, v)`
`CorootPosition(G, v)`

`Basis` MonStgElt *Default :* "*Standard*"

If v is a (co)root of the group of Lie type G, this returns its position; otherwise it returns 0.

Example H103E12

```
> G := GroupOfLieType("A3", 25 : Isogeny := 2);
> Roots(G);
{@
    (1 0 0),
    (0 1 0),
    (1 0 2),
    (1 1 0),
    (1 1 2),
    (2 1 2),
    (-1  0  0),
    (0 -1  0),
    (-1  0 -2),
    (-1 -1  0),
    (-1 -1 -2),
    (-2 -1 -2)
@}
> PositiveCoroots(G);
{@
    (2 -1 -1),
    (-1  2  0),
    (0 -1  1),
    (1  1 -1),
    (-1  1  1),
    (1 0 0)
@}
> #Roots(G) eq 2*NumPosRoots(G);
true
> Coroot(G, 4);
(1  1 -1)
> Coroot(G, 4 : Basis := "Root");
(1 1 0)
> CorootPosition(G, [1,1,-1]);
4
> CorootPosition(G, [1,1,0] : Basis := "Root");
4
```

`HighestRoot(G)`

`HighestLongRoot(G)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique (long) root of greatest height in the root datum of the group of Lie type G.

`HighestShortRoot(G)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique short root of greatest height in the root datum of the group of Lie type G.

Example H103E13

```
> G := GroupOfLieType("G2", RealField());
> HighestRoot(G);
(3 2)
> HighestLongRoot(G);
(3 2)
> HighestShortRoot(G);
(2 1)
```

103.8.2 Reflections

The reflections in the Weyl group have representatives in the group of Lie type.

`Reflections(G)`

The sequence of representatives of reflections in the group of Lie type G.

`Reflection(G, r)`

The representative of the reflections in the rth root in the group of Lie type G.

Example H103E14

```
> G := GroupOfLieType("A2", Rationals());
> Reflections(G);
[ n1 , n2 , n1 n2 n1  ]
```

103.8.3 Operations and Properties for Root and Coroot Indices

`RootHeight(G, r)`

`CorootHeight(G, r)`

The height of the rth (co)root of the group of Lie type G, i.e. the sum of the coefficients of α_r (resp. $\alpha_r^\star$) with respect to the simple (co)roots.

`RootNorms(G)`

`CorootNorms(G)`

The sequence of squares of the lengths of the (co)roots of the group of Lie type G.

`RootNorm(G, r)`

`CorootNorm(G, r)`

The square of the length of the rth (co)root of the group of Lie type G.

`IsLongRoot(G, r)`

Returns `true` if, and only if, the rth root of the group of Lie type G is long, i.e. the rth coroot is short.

`IsShortRoot(G, r)`

Returns `true` if, and only if, the rth root of the group of Lie type G is short, i.e. the rth coroot is long.

`AdditiveOrder(G)`

An additive order on the positive roots of the group of Lie type G, i.e. a sequence containing the numbers $1, \ldots, N$ in some order so that $\alpha_r + \alpha_s = \alpha_t$ implies t is between r and s. This is computed using the techniques of [Pap94]

Example H103E15

```
> G := GroupOfLieType("A5", GF(3));
> a := AdditiveOrder(G);
> Position(a, 2);
6
> Position(a, 3);
10
```

103.8.4 Weights

`WeightLattice(G)`

`CoweightLattice(G)`

The (co)weight lattice of the group of Lie type G.

`FundamentalWeights(G)`

`FundamentalCoweights(G)`

`Basis` MonStgElt *Default :* "*Standard*"

The fundamental (co)weights of the group of Lie type G as the rows of a matrix.

`DominantWeight(G, v)`

`Basis` MonStgElt *Default :* "*Standard*"

The unique dominant weight in the same W-orbit as v, where W is the Weyl group of G and v is a weight given as a vector or a sequence representing a vector. The second value returned is a Weyl group element taking v to the dominant weight.

103.9 Building Groups of Lie Type

Currently the only subgroups of a group of Lie type that can be constructed are subsystem subgroups.

`SubsystemSubgroup(G, a)`

The subsystem subgroup of the group of Lie type G generated by the standard maximal torus and the root subgroups with roots $\alpha_{a_1}, \ldots, \alpha_{a_k}$ where $a = \{a_1, \ldots, a_k\}$ is a set of integers.

`SubsystemSubgroup(G, s)`

The subsystem subgroup of the group of Lie type G generated by the standard maximal torus and the root subgroups with roots $\alpha_{s_1}, \ldots, \alpha_{s_k}$ where $s = [s_1, \ldots, s_k]$ is a *sequence* of integers. In this version the roots must be simple in the root subdatum (i.e. none of them may be a summand of another) otherwise an error is signalled. The simple roots will appear in the subdatum in the given order.

Example H103E16

```
> G := GroupOfLieType("A4",Rationals());
> PositiveRoots(G);
{@
    (1 0 0 0),
    (0 1 0 0),
    (0 0 1 0),
    (0 0 0 1),
    (1 1 0 0),
    (0 1 1 0),
    (0 0 1 1),
    (1 1 1 0),
    (0 1 1 1),
    (1 1 1 1)
@}
> H := SubsystemSubgroup(G, [6,1,4]);
> H;
H: Group of Lie type A3 over Rational Field
> PositiveRoots(H);
{@
    (0 1 1 0),
    (1 0 0 0),
    (0 0 0 1),
    (1 1 1 0),
    (0 1 1 1),
    (1 1 1 1)
@}
> h := elt<H|<2,2>,1>;
> h;  G!h;
x2(2) n1
x1(2) ( 1 -1  1 -1) n2 n3 n2
```

`DirectProduct(G1, G2)`

The direct product of the groups G_1 and G_2. The two groups must have the same base ring.

`Dual(G)`

The dual of the group of Lie type G, obtained by swapping the roots and coroots.

`SolubleRadical(G)`

The soluble radical of the group of Lie type G.

`StandardMaximalTorus(G)`

The standard maximal torus of the group of Lie type G.

Example H103E17

```
> G1 := GroupOfLieType( "A5", GF(7) );
> G2 := GroupOfLieType( "B4", GF(7) );
> DirectProduct(G1, Dual(G2));
$: Group of Lie type A5 C4 over Finite field of size 7
>
> G := GroupOfLieType(StandardRootDatum("A",3), GF(17));
> SolubleRadical(G);
$: Torus group of Dimension 1 over Finite field of size 17
```

103.10 Automorphisms

The following functions construct the standard automorphisms of a group of Lie type, as described in [Car72] (except for the graph automorphism of G_2). In many cases, including the finite groups, every automorphism is a product of these standard automorphisms.

103.10.1 Basic Functionality

`AutomorphismGroup(G)`

Automorphism group of a group of Lie type G.

`IdentityAutomorphism(G)`

`One(A)`

`Id(A)`

The identity automorphism of the group of Lie type G.

`Mapping(a)`

The map object associated with the automorphism a.

`Automorphism(m)`

Given a map object m from G to G, which is an isomorphism, returns the associated automorphism as an automorphism of a group of Lie type.

`h * g`

The composition of the group of Lie type automorphisms h and g.

`h ^ n`

The nth power of the group of Lie type automorphism h.

`g ^ h`

The conjugate $h^{-1}gh$, where g and h are group of Lie type automorphisms g and h

`Domain(A)`
`Codomain(A)`
`Domain(h)`
`Codomain(h)`

Domain or codomain of an automorphism of a group of Lie type or of the group of automorphisms.

103.10.2 Constructing Special Automorphisms

`InnerAutomorphism(G, x)`

The inner automorphism taking $g \in G$ to g^x, where x is an element of the group of Lie type G.

`DiagonalAutomorphism(G, v)`

The diagonal automorphism of the semisimple group of Lie type G given by the vector v. Let n be the semisimple rank of G and let k be its base field. Then v must be a vector in k^n with every component nonzero. The function returns the automorphism given by the character χ defined by $\chi(\alpha_i) = v_i$, where α_i is the ith simple root. Since our groups are algebraic, a diagonal automorphism is just a special case of an inner automorphism.

`GraphAutomorphism(G, p)`
`DiagramAutomorphism(G, p)`

`SimpleSigns` ANY *Default :* 1

The graph automorphism of the group of Lie type G given by the permutation p. The permutation must act on the indices of simple roots of G or the indices of all roots of G. The graph automorphism of the group of type G_2 has not been implemented yet.

The optional parameter SimpleSigns can be used to specify the signs corresponding to each simple root. This should either be a sequence of integers ± 1, or a single integer ± 1.

`FieldAutomorphism(G, sigma)`

The field automorphism of the group of Lie type G induced by σ, an element of the automorphism group of the base field of G

`RandomAutomorphism(G)`
`Random(A)`

A random element in A, the automorphism group of the group of Lie type G.

`DualityAutomorphism(G)`

The duality automorphism of G. This is an automorphism that takes every unipotent term $x_r(t)$ to $x_s(\pm t)$, where $s =$`Negative(RootDatum(G),r))`.

`FrobeniusMap(G,q)`

The Frobenius automorphism of the finite group of Lie type G gotten by qth powers in the base field. The integer q must be a power of the characteristic of the base field of G.

103.10.3 Operations and Properties of Automorphisms

`DecomposeAutomorphism(h)`

Given a group of Lie type automorphism h, this returns a field automorphism f, a graph automorphism g and an inner automorphism i such that $h = fgi$. This only works for groups defined over finite fields. The algorithm is due to Scott Murray and Sergei Haller.

`IsAlgebraic(h)`

Returns `true` if and only if the automorphism h is algebraic.

Example H103E18

Some automorphisms of $B_2(4)$

```
> G := GroupOfLieType("B2", GF(4));
> A := AutomorphismGroup(G);
> A!1 eq IdentityAutomorphism(G);
true
> g := GraphAutomorphism(G, Sym(2)!(1,2));
> g;
Automorphism of Group of Lie type B2 over Finite field of size 2^2
given by: Mapping from: Group of Lie type  to Group of Lie type
given by a rule
Decomposition:
  Mapping from: GF(2^2) to GF(2^2) given by a rule,
  (1, 2),
  1
```

The automorphism of $B_2(4)$ whose stabiliser is ${}^2B_2(4)$ is constructed by the following code.

```
> sigma := iso< GF(4) -> GF(4) | x :-> x^2, x :-> x^2 >;
> h := FieldAutomorphism(G, sigma) * g;
> h in A;
true
> f,g,i := DecomposeAutomorphism(h);
> assert f*g*i eq h;
```

103.11 Algebraic Homomorphisms

`GroupOfLieTypeHomomorphism(phi, k)`

The algebraic homomorphism of groups of Lie type over the ring k corresponding to the root datum morphism ϕ. See Chapter 103.

Example H103E19

This example constructs the algebraic projection $GL_4(\mathbf{Q}) \to PGL_4(\mathbf{Q})$.

```
> RGL := StandardRootDatum( "A", 3 );
> RPGL := RootDatum( "A3" );
> A := VerticalJoin( SimpleRoots(RGL), Vector([Rationals()|1,1,1,1]) )^-1 *
>   VerticalJoin( SimpleRoots(RPGL), Vector([Rationals()|0,0,0]) );
> B := VerticalJoin( SimpleCoroots(RGL), Vector([Rationals()|1,1,1,1]) )^-1 *
>   VerticalJoin( SimpleCoroots(RPGL), Vector([Rationals()|0,0,0]) );
> phi := GroupOfLieTypeHomomorphism( hom< RGL -> RPGL | A, B >, Rationals() );
> GL := Domain( phi );
> phi( elt<GL|<1,2>, Vector([Rationals()| 7,1,11,1])> );
x1(2) (   7 1/11   11)
```

103.12 Twisted Tori

The functionality presented here deals with the computation of the twisted tori of a finite group of Lie type.

Note that for a given group $G(k)$, the twisted tori are returned as subgroups of the standard torus of $G(K)$ for the smallest field extension K of k, where this is possible.

For finite fields and an untwisted group of Lie type $G(k)$, a twisted torus $T_w(k)$ of $G(k)$ has the form

$$T_w(k) = \{t \in T(K) | t^{\sigma w} = t\},$$

where $T(K)$ is the standard K-split torus of $G(K)$, σ is the generator of the Galois group $\mathrm{Gal}(K : k)$ and w is an element of the Weyl group of $G(k)$.

`TwistedTorusOrder(R, w)`

Given the root datum R and a Weyl group element w, computes the orders of the cyclic components of the twisted torus $T_w(k) \subset G(R, k)$ as sequence of polynomials in q, the order of the field k.

`TwistedToriOrders(G)`

`TwistedToriOrders(R)`

Given a group of Lie type G or a root datum R, takes for every concjugacy class of the Weyl group of G a representative w, and computes `TwistedTorusOrder(R, w)`. Returns the sequence of the lists consisting of `TwistedTorusOrder(R, w)` and w for every concjugacy class.

`TwistedTorus(G, w)`

Computes the twisted torus $T_w(k)$ of the group of Lie type G for the given element w of the Weyl group of G. Returned is the list consisting of three elements, first of them being the sequence of orders of cyclic parts of the torus, the second being the sequence of generators of the respective orders and the third being w. See [Hal05] for the algorithm used.

`TwistedTori(G)`

Computes one twisted torus $T_w(k)$ of the group of Lie type G for each conjugacy class w^W of the Weyl group W of G. A sequence of them is returned. See [Hal05] for the algorithm used.

Example H103E20

We compute all twisted tori of $A_1(5)$:

```
> G := GroupOfLieType("A1", 5);
> TwistedToriOrders(G);
[ [*
    [
        q - 1
    ],
    Id($)
*], [*
    [
        q + 1
    ],
    (1, 2)
*] ]
> TwistedTori(G);
[ [*
    [ 4 ],
    [ (2)  ],
    Id($)
*], [*
    [ 6 ],
    [ ( k.1^4)  ],
    (1, 2)
*] ]
```

As we may notice, the second one is contained in the group over the quadratic field extension:

```
> Universe($1[2][2]);
$: Group of Lie type A1 over Finite field of size 5^2
```

Example H103E21

These are the orders of the decompositions of all (up to conjugacy) maximal tori of the group $G_2(q)$ as polynomials in q:

```
> R := RootDatum("G2");
> [ t[1] : t in TwistedToriOrders(R) ];
[
    [  q - 1,  q - 1  ],
    [  q + 1,  q + 1  ],
    [  q^2 - 1        ],
    [  q^2 - 1        ],
    [  q^2 + q + 1    ],
    [  q^2 - q + 1    ]
]
```

103.13 Sylow Subgroups

We present here the functionality which allows to compute the Sylow subgroups of finite groups of Lie type.

`PrintSylowSubgroupStructure(G)`

This procedure prints out a list of all primes p dividing the order of the group of Lie type G along with the "goodness" of p, the exponent of p in the factorisation of $|G|$ and a sequence of integers. The positive integers give the orders of the decomposition of a torus T_w into cyclic groups such that the Sylow subgroup is contained in $\langle T_w, C_W(w)\rangle$. The negative number indicates the p-part coming from $C_W(w)$. If more than one such torus exists, then one line is printed for each of them.

A prime is said to be "GOOD" if it is equal to the characteristic of the base field k of G, "good" if the Sylow subgroup is abelian, thus contained in a torus, and "bad" if it is not abelian and thus not contained in a torus. See [Hal05] for the algorithm used.

`SylowSubgroup(G, p)`

Compute a p-Sylow subgroup S of the group of Lie type G. Returned is a list of a two sequences. The second sequence contains generators of S. The first one is a sequence of integers giving the orders of the respective generator if the generator is a torus element and the negative of the order of $\langle g\rangle/(\langle g\rangle \cap T_w)$ in case the generator g is not a torus element. See [Hal05] for the algorithm used.

Example H103E22

Compute

```
> G := GroupOfLieType("G2", 5);
> PrintSylowSubgroupStructure(G);
G: Group of Lie type G2 over Finite field of size 5
Order(G) is  2^6 * 3^3 * 5^6 * 7^1 * 31^1
Order(W) is  2^2 * 3^1
...compute tori...
...compute sylows...
   2 (bad)  : 6 [ 4, 4, -4 ]
   3 (bad)  : 3 [ 6, 6, -3 ]
   5 (GOOD) : The unipotent subgroup of G
   7 (good) : 1 [ 21 ]
   31 (good) : 1 [ 31 ]
> SylowSubgroup(G,2);
[*
    [ 4, 4, -2, -2 ],
    [ (2 1) , (1 2) , n2 , n1 n2 n1 n2 n1 n2  ]
*]
```

note that the orders of the non-toral elements is not necessarily the corresponding integer in the first sequence:

```
> gens := $1[2];
> [ Order(g) : g in gens ];
[ 4, 4, 4, 4 ]
```

but, in this example, their squares are contained in the torus:

```
> gens[3]^2 eq gens[2]^2, gens[4]^2 eq gens[2]^2;
true true
```

103.14 Representations

This section describes basic functionality for Lie algebra representations: see Chapter 104 for more functions for highest weight representations and decompositions.

`StandardRepresentation(G)`

The standard (projective) representation of the semisimple group of Lie type G over an extension its base ring. In other words, the smallest dimension highest-weight representation. For the classical groups, this is the natural representation. If this is a projective representation rather than a linear representation, a warning is given. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

`AdjointRepresentation(G)`

The adjoint (projective) representation of the group of Lie type G over an extension of its base ring, i.e. the representation given by the action of G on its Lie algebra. The Lie algebra itself is the second returned value. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

`LieAlgebra(G)`

The Lie algebra of the group of Lie type G, together with the adjoint representation. If this is a projective representation rather than a linear representation, a warning is given.

`HighestWeightRepresentation(G, v)`

The highest weight (projective) representation with highest weight v of the group of Lie type G over an extension of its base ring. If this is a projective representation rather than a linear representation, a warning is given. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

Example H103E23

```
> G := GroupOfLieType("A2", Rationals() : Isogeny := "SC");
> rho := StandardRepresentation(G);
> rho(elt< G | 1 >);
[ 0 -1  0]
[ 1  0  0]
[ 0  0  1]
> rho(elt<G | <2,1/2> >);
[  1   0   0]
[  0   1   0]
[  0 1/2   1]
> rho(elt< G | VectorSpace(Rationals(),2)![3,5] >);
[  3   0   0]
[  0 5/3   0]
[  0   0 1/5]
>
> G := GroupOfLieType("A2", Rationals());
> Invariants(CoisogenyGroup(G));
[ 3 ]
> rho := StandardRepresentation(G);
Warning: Projective representation
> BaseRing(Codomain(rho));
Algebraically closed field with no variables
> rho(elt< G | VectorSpace(Rationals(),2)![3,1] >);
[r1  0  0]
[ 0 r2  0]
[ 0  0 r2]
> rho(elt< G | VectorSpace(Rationals(),2)![3,1] >)^3;
```

```
[  9   0   0]
[  0 1/3   0]
[  0   0 1/3]
```

`GeneralisedRowReduction(`ρ`)`

`RowReductionHomomorphism(`ρ`)`

`Inverse(`ρ`)`

Given a projective matrix representation $\rho : G \to \mathrm{GL}_m(k)$, return its inverse.

103.15 Bibliography

[**Car72**] Roger W. Carter. *Simple groups of Lie type*. John Wiley & Sons, London-New York-Sydney, 1972. Pure and Applied Mathematics, Vol. 28.

[**Car93**] Roger W. Carter. *Finite groups of Lie type*. John Wiley & Sons, Chichester, 1993. Conjugacy classes and complex characters, Reprint of the 1985 original, A Wiley-Interscience Publication.

[**CHM08**] Arjeh M. Cohen, Sergei Haller, and Scott H. Murray. Computing in unipotent and reductive algebraic groups. *LMS J. Comput. Math.*, 11:343–366, 2008.

[**CMT04**] Arjeh M. Cohen, Scott H. Murray, and D. E. Taylor. Computing in groups of Lie type. *Math. Comp.*, 73(247):1477–1498, 2004.

[**Hal05**] Sergei Haller. *Computing Galois Cohomology and Forms of Linear Algebraic Groups*. Phd thesis, Technical University of Eindhoven, 2005.

[**Pap94**] Paolo Papi. A characterization of a special ordering in a root system. *Proc. Amer. Math. Soc.*, 120(3):661–665, 1994.

[**Ste62**] Robert Steinberg. Générateurs, relations et revˆetements de groupes algébriques. In *Colloq. Théorie des Groupes Algébriques (Bruxelles, 1962)*, pages 113–127. Librairie Universitaire, Louvain, 1962.

104 REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS

Chapter 104
REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS

104.1 Introduction

This chapter gives functionality for direct sums of *highest weight representations* (or modules). This is an important class of representations of (almost) semisimple Lie algebras (Chapter 100) and connected reductive algebraic groups (Chapter 103). This class includes all finite dimensional representations if the base field is the complex field.

The representations we are considering are in bijection with sets of dominant weights with multiplicities. Such sets are called *decomposition multisets*. Many interesting computations in representation theory can be done combinatorially with weight multisets, without the need to construct the module itself. Examples of the things we can compute include: module dimension, the multiset of all the weights, and decomposition multisets for symmetric powers, alternating powers, and tensor products. We can also restrict a decomposition multiset to a subgroup or induce it to a supergroup.

The code for such combinatorial computations is based on the LiE software package [vLCL92]. The algorithms for computing the actual representations are from [dG01] in the Lie algebra case, and from [CMT04] in the group case.

104.1.1 Highest Weight Modules

This introduction is inspired by the LiE manual [vLCL92].

First consider connected reductive Lie groups over the complex field. If G is a connected reductive complex Lie group, then it is a homomorphic image $G = \xi(G')$, where ξ is a Lie-group homomorphism with finite kernel and G' is the direct product of a simply connected group and a torus. Recall that a simply connected group is a direct product of *simple* simply connected groups. In particular, such groups are determined by their Cartan name and the dimension of the torus. For example, we denote the direct product of the group of type $A_4C_3B_2$ with a two dimensional torus by $A_4C_3B_2T_2$. Most of the code ported from LiE works only for groups of this form. Similar terminology is used for the root datum corresponding to a group.

Connected reductive complex Lie groups have a very pleasing representation theory:

– Every module decomposes as a direct sum of irreducible representations.

– The (finite dimensional) irreducible representations correspond to dominant weights.

It follows that representations correspond to finite sets of dominant weights with multiplicity. These multisets are called *decomposition multisets*. We can use this classification to do useful computations about representations, without having to explicitly construct them.

Multisets of weights can be used for other purposes as well: The multiset of all weights occurring in a module M is called the *character multiset*. Since the Weyl group permutes the weights occurring in the character of M, it suffices to consider only the dominant weights with their multiplicities. This is called the *dominant character multiset*. In the LiE system, multisets of weights are represented by polynomials: for example, the decomposition multiset is called a decomposition character.

When using the functions in this section, it is important to keep track of which kind of multiset you are using. For example, if you input a decomposition multiset to a function that expects a dominant character multiset, the output is meaningless.

We often abbreviate decomposition multiset to *decomposition*, and similarly for character multisets. Write R_D for the root datum of the group of the decomposition D. Denote the irreducible module for the group with root datum R with highest weight v by V_v^R, or to V_v if R is clear from the context.

It is often useful to define consider *virtual* multisets, which allow weights to have negative multiplicities. We call a virtual multiset *proper* if its weights all have nonnegative multiplicities. A decomposition corresponds to an actual module if and only if it is proper.

104.1.2 Toral Elements

Many functions use a special syntax for finite-order elements of the torus of a Lie group G (we are rarely interested in infinite-order elements). Recall that a weight is in fact a mapping from the torus T to $C^\star$, and thus a weight λ can be evaluated at an element $t \in T$. The resulting element is written t^λ. A set of fundamental weights $\omega_1, \ldots, \omega_r$ has the property that any element $t \in T$ is uniquely determined by the values $t^{\omega_1}, \ldots, t^{\omega_r}$. Therefore, we may represent t as a vector $(a_1, \ldots, a_r, n)$, with the property that $t^{\omega_i} = e^{2\pi i a_i/n} = \zeta_n^{a_i}$, where $\zeta_n = e^{2\pi i/n}$ is the canonical n-th root of unity. An example of a function which uses this syntax for toral elements is `Spectrum`. This function also provides a means to convert toral elements into a more natural form: see Example H104E10.

104.1.3 Other Highest Weight Representations

MAGMA can also construct highest weight representations for:

– (Almost) reductive Lie algebras (Chapter 100); and

– Split groups of Lie type (Chapter 103).

If the base field ha positive characteristic, highest weight representations are indecomposable, but not necessarily irreducible. In some cases there are irreducible representations which are not highest weight representations.

For groups of Lie type, we consider projective representations (i.e., homomorphisms to a projective general linear group). Suppose G is a split group of Lie type defined over the field k and r is the least common multiple of the nonzero abelian-group invariants of the coisogeny group of G (see Section 97.1.6). Let K be an extension of k containing at least one rth root of each element of k (i.e., K contains a Kummer extension). Then highest weight representations are projective representations defined over K, and are constructed using polynomial functions and rth roots.

If k already contains all rth roots, then no extension is needed and the representation will be linear rather than projective. This happens when $r = 1$, i.e., the coisogeny group is torsion free. This includes direct products of a simply connected group and a torus. The general linear group also has this property. It also happens when k is the complex field or field of algebraic numbers, when k is the real field and r is odd, and when k is finite and $|k| - 1$ is coprime to r.

The functions give a warning when the representation is not linear, but this can be avoided using the optional parameter `NoWarning`. Note that an appropriate extension K can be constructed for all fields other than rational function fields, fields of Laurent series, and local fields. In these cases, as well as for nonfields, the representations can only be computed when $r = 1$.

104.2 Constructing Weight Multisets

In this section, we describe how to construct weight multisets.

`TrivialLieRepresentationDecomposition(R)`

`LieRepresentationDecomposition(R)`

The decomposition multiset of the trivial representation. The root datum R must be weakly simply connected.

`LieRepresentationDecomposition(R, v)`

The decomposition multiset of the highest weight representation with weight v, i.e., the singleton multiset. The root datum R must be weakly simply connected. The weight v must be a sequence of length d or an element of $\mathbf{Z}^d$, where d is the dimension of the root datum R.

`LieRepresentationDecomposition(R, Wt, Mp)`

The decomposition multiset with weights given by the sequence Wt and multiplicities given by of the sequence Mp. The root datum R must be weakly simply connected. The weights must be a sequences of length d or elements of $\mathbf{Z}^d$, where d is the dimension of the root datum R.

`AdjointRepresentationDecomposition(R)`

The decomposition multiset of the adjoint representation. This has the highest root of R as its highest weight with multiplicity one. The root datum R must be weakly simply connected.

Example H104E1

The adjoint representation:

```
> R := RootDatum("D4" : Isogeny := "SC");
> D := AdjointRepresentationDecomposition(R);
> D:Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Dimension of weight space:4
     Weights:
          [
               (0 1 0 0)
          ]
     Multiplicities:
          [ 1 ]
> HighestRoot(R : Basis := "Weight");
(0 1 0 0)
```

104.3 Constructing Representations

104.3.1 Lie Algebras

The functions described in this section are applicable only to almost reductive structure constant Lie algebras.

If L has of large dimension, the step that calculates the information needed to compute preimages for these representations can be quite time consuming. So if there is no requirement for preimages, this step may be skipped by setting the optional argument `ComputePreImage` to `false`.

`TrivialRepresentation(L)`

The one-dimensional trivial representation of the Lie algebra L over its base ring.

`AdjointRepresentation(L)`

`ComputePreImage` BoolElt *Default :* `true`

The adjoint representation of the Lie algebra L acting on itself.

`StandardRepresentation(L)`

`ComputePreImage` BoolElt *Default :* `true`

The standard representation of the semisimple Lie algebra L over its base ring. This is the smallest dimensional faithful representation of G (with a few small exceptions). The Killing form of L must be nondegenerate.

Example H104E2

```
> R := RootDatum("A2");
> #CoisogenyGroup(R);
3
> L := LieAlgebra(R, GF(2));
> h := StandardRepresentation(L);
> h(L.1);
[0 0 1]
[0 0 0]
[0 0 0]
> L := LieAlgebra(R, GF(3));
> h := StandardRepresentation(L);
>> h := StandardRepresentation(L);
                               ^
Runtime error in 'StandardRepresentation': Cannot compute the standard
representation in characteristic  3
```

The coisogeny group of a simply connected root datum always has order one, so we can compute the standard representation in this case.

```
> R := RootDatum("A2" : Isogeny:="SC");
> L := LieAlgebra(R, GF(3));
> h := StandardRepresentation(L);
```

`HighestWeightRepresentation(L, w)`

The representation of the Lie algebra L with highest weight w (given either as a vector or as a sequence representing a vector). The result is a function, which for an element of L gives the corresponding matrix. The algorithm used is described in [dG01].

Example H104E3

```
> L:= LieAlgebra("G2", RationalField());
> DimensionOfHighestWeightModule(RootDatum(L), [1,0]);
7
> rho:= HighestWeightRepresentation(L, [1,0]);
> e, f, h := ChevalleyBasis(L);
> rho(e[1]+f[1]);
[ 0  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
[ 0  0  0 -2  0  0  0]
[ 0  0 -1  0 -1  0  0]
[ 0  0  0 -2  0  0  0]
[ 0  0  0  0  0  0  1]
[ 0  0  0  0  0  1  0]
```

```
> Codomain(rho);
Full Matrix Lie Algebra of degree 7 over Rational Field
> N := sub<Codomain(rho) | [ rho(x) : x in e ]>;
> Dimension(N);
6
> IsSolvable(N);
true
```

`HighestWeightModule(L, w)`

Given a semisimple Lie algebra L corresponding to a root datum of rank r and a sequence w of non-negative integers of length r, this returns the irreducible L-module with highest weight w. The object returned is a left module over L. The algorithm used is described in [dG01].

`TensorProduct(Q)`

Given a sequence Q of left-modules over a Lie algebra, this function returns the module M that is the tensor product of the elements of Q. It also returns a map ϕ from the Cartesian product P of the modules in Q to M as the second return value. If t is an tuple whose i-th component is an element from the i-th module in Q then ϕ maps t to the element of M that corresponds to the tensor product of the elements of t.

`SymmetricPower(V, n)`

Given a left-module V over a Lie algebra, and an integer $n \geq 2$, this function returns the module M that is the n-th symmetric power of V. It also returns a map f from the n-fold Cartesian product of V to M. This map is multilinear and symmetric, i.e., if two of its arguments are interchanged then the image remains the same. Furthermore, f has the universal property, i.e., any multilinear symmetric map from the n-fold Cartesian product into a vector space W can be written as the composition of f with a map from M into W.

`ExteriorPower(V, n)`

Given a left-module V over a Lie algebra, and an integer $2 \leq n \leq \dim(V)$, this function returns the module M that is the n-th exterior power of V. It also returns a map f from the n-fold Cartesian product of V to M. This map is multilinear and antisymmetric, i.e., if two of its arguments are interchanged then the image is multiplied by -1. Furthermore, f has the universal property, i.e., any multilinear antisymmetric map from the n-fold Cartesian product into a vector space W can be written as the composition of f with a map from M into W.

Example H104E4

```
> L:= LieAlgebra("G2", Rationals());
> V1:= HighestWeightModule(L, [1,0]);
> V2:= HighestWeightModule(L, [0,1]);
> T,f:= TensorProduct([V1,V2]);
> HighestWeightsAndVectors(T);
[
    (1 1),
    (2 0),
    (1 0)
]
[
    [
        T: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        T: (0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 -3 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        T: (0 0 0 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 -1 0 0
            0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 -3 0 0 0 0 0 0 0 0 0 0
            0 0 -3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ]
]
> DecomposeTensorProduct(RootDatum(L), [1,0], [0,1]);
[
    (1 1),
    (2 0),
    (1 0)
]
[ 1, 1, 1 ]
> f(<V1.2,V2.3>);
T: (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

So we see that the tensor product T decomposes as a direct sum of three submodules. This information can also be computed by using `DecomposeTensorProduct`. However, in the former case, the corresponding highest-weight vectors are also given.

```
> E,h:= ExteriorPower(V1, 3);
> h(<V1.1,V1.3,V1.4>);
E: (0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> h(<V1.1,V1.4,V1.3>);
```

```
E: (0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> DecomposeExteriorPower( RootDatum(L), 3, [1,0] );
[
    (2 0),
    (1 0),
    (0 0)
]
[ 1, 1, 1 ]
> HighestWeightsAndVectors(E);
[
    (2 0),
    (1 0),
    (0 0)
]
[
    [
        E: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0)
    ],
    [
        E: (0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0)
    ],
    [
        E: (0 0 0 0 0 0 0 0 0 0 0 1 2 0 0 0 0 0 2 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
        0)
    ]
]
```

104.3.2 Groups of Lie Type

`TrivialRepresentation(G)`

The one-dimensional trivial representation of the group of Lie type G over its base ring.

`StandardRepresentation(G)`

The standard (projective) representation of the semisimple group of Lie type G over an extension its base ring. In other words, the smallest dimension highest-weight representation. For the classical groups, this is the natural representation. If this is a projective representation rather than a linear representation, a warning is given. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

`AdjointRepresentation(G)`

The adjoint (projective) representation of the group of Lie type G over an extension of its base ring, ie. the representation given by the action of G on its Lie algebra. The Lie algebra itself is the second returned value. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

`LieAlgebra(G)`

The Lie algebra of the group of Lie type G, together with the adjoint representation. If this is a projective representation rather than a linear representation, a warning is given.

`HighestWeightRepresentation(G, v)`

The highest weight (projective) representation with highest weight v of the group of Lie type G over an extension of its base ring. If this is a projective representation rather than a linear representation, a warning is given. This is constructed from the corresponding Lie algebra representation, using the algorithm in [CMT04].

Example H104E5

```
> G := GroupOfLieType("A2", Rationals() : Isogeny := "SC");
> rho := StandardRepresentation(G);
> rho(elt< G | 1 >);
[ 0 -1  0]
[ 1  0  0]
[ 0  0  1]
> rho(elt<G | <2,1/2> >);
[  1   0   0]
[  0   1   0]
[  0 1/2   1]
> rho(elt< G | VectorSpace(Rationals(),2)![3,5] >);
[  3   0   0]
[  0 5/3   0]
[  0   0 1/5]
>
> G := GroupOfLieType("A2", Rationals());
> Invariants(CoisogenyGroup(G));
[ 3 ]
> rho := StandardRepresentation(G);
Warning: Projective representation
> BaseRing(Codomain(rho));
Algebraically closed field with no variables
> rho(elt< G | VectorSpace(Rationals(),2)![3,1] >);
[r1  0  0]
[ 0 r2  0]
[ 0  0 r2]
> rho(elt< G | VectorSpace(Rationals(),2)![3,1] >)^3;
```

```
[  9   0   0]
[  0 1/3   0]
[  0   0 1/3]
```

104.4 Operations on Weight Multisets

104.4.1 Basic Operations

In this section, basic access and arithmetic operations for weight multisets are described. Addition generally corresponds to direct sum of representations. The other arithmetic operations do not necessarily correspond to meaningful operations on the corresponding representation.

`RootDatum(D)`

The Root datum over which the weight multiset D is defined.

`Weights(D)`

`WeightsAndMultiplicities(D)`

The sequences of weights and multiplicities in the weight multiset D.

`Multiset(D)`

The weights and multiplicities of the weight multiset D as a normal multiset consisting of vectors.

`Multiplicity(D, v)`

The multiplicity of the weight v in the weight multiset D.

`D eq E`

Returns `true` if, and only if, the weight multisets D and E are identical, i.e. they are defined over identical root data, with equal weights and multiplicities.

`D + E`

The sum (union) of weight multisets D and E, i.e. this corresponds to the direct sum of the two decomposition multisets. The underlying root data must be the same.
Add V_v to D. The length of v must be equal to $\dim(R_D)$.

`D +:= E`

Add the weight multiset E to D. R_D must be equal to R_E.

`AddRepresentation(~D, E, c)`

`AddRepresentation(~D, E)`

Add c times the weight multiset E to D. The integer c may be omitted, in which case it is assumed to be equal to 1. The root data of D and E must be identical.

`D + v`

Add the weight v to the multiset D. The weight v must be a sequence of length d or an element of $\mathbf{Z}^d$, where d is the dimension of the root datum R.

`AddRepresentation(~D, v, c)`

`AddRepresentation(~D, v)`

Add c times the weight v to the multiset D. The integer c may be omitted, in which case it is assumed to be equal to 1. The length of v must be equal to the dimension of the root datum of D.

`D +:= v`

Add the weight v to the multiset D. The length of v must be equal to the dimension of the root datum of D.

`D * c`

The multiset whose weights are equal to those of D, and whose multiplicities are c times the multiplicities of D.

`D / c`

The multiset whose weights are equal to those of D, and whose multiplicities are the multiplicities of D divided by c. An error is flagged if any of the multiplicities of D is not divisible by c.

`D *:= c`

Multiply all multiplicities of the weight multiset D by c.

`D /:= c`

Divide all multiplicities of the weight multiset D by c. An error is flagged if a multiplicity of D is not divisible by c.

`D * E`

`ProductRepresentation(D, E)`

The product of the two weight multisets D and E, viewed as polynomials as in the LiE package [vLCL92]. The root datum of the resulting decomposition is the direct sum of the root data of D and E. Note that this is does not correspond to the direct sum or tensor product of representations.

`ProductRepresentation(D, E, R)`

The product of the two weight multisets D and E, viewed as polynomials as in the LiE package [vLCL92]. The product is interpreted as a weight multiset over the root datum R. An error is flagged if the dimension of R is not the sum of the dimensions of the root data of D and E.

`SubWeights(D, Q, S)`

Let k be the length of the sequence Q. The resulting decomposition E has Root datum S, and to each highest weight of D corresponds a highest weight w' of E, with $w'_i = w_{Q[i]}$, where $i = 1, \ldots, k$. The multiplicities of E are equal to the multiplicities of D, but one should note that E might in fact have fewer unique highest weights than D, especially if $k < \dim(R_D)$. The dimension of the root datum S must be equal to k.

`PermuteWeights(D, pi, S)`

Permute the components of the weights in the multiset D by the permutation π and interpret the result as a weight multiset over the root datum S. If the underlying root datum of D has dimension d, then S must also have dimension d and π must be an element of $\mathrm{Sym}(d)$.

Example H104E6

Arithmetic with decompositions:

```
> R := RootDatum("A2" : Isogeny := "SC");
> D := LieRepresentationDecomposition(R, [[2,3],[4,3]], [1,3]);
> D:Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 2 of type A2
     Dimension of weight space:2
     Weights:
          [
               (2 3),
               (4 3)
          ]
     Multiplicities:
          [ 1, 3 ]
> E := D + [5,2];
> E:Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 2 of type A2
     Dimension of weight space:2
     Weights:
          [
               (2 3),
               (4 3),
               (5 2)
```

```
          ]
     Multiplicities:
          [ 1, 3, 1 ]
> PermuteWeights(E, Sym(2)!(1,2), R):Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 2 of type A2
     Dimension of weight space:2
     Weights:
          [
               (3 2),
               (3 4),
               (2 5)
          ]
     Multiplicities:
          [ 1, 3, 1 ]
> S := RootDatum("A1" : Isogeny := "SC");
> SubWeights(E, [2], S):Maximal;
Highest weight decomposition of representation of:
     S: Simply connected root datum of dimension 1 of type A1
     Dimension of weight space:1
     Weights:
          [
               (3),
               (2)
          ]
     Multiplicities:
          [ 4, 1 ]
```

104.4.2 Conversion Functions

Functions for converting between different kinds of weight multiset (decomposition, character, and dominant character multisets). Note that it is the users responsibility to keep track of what kind of multiset they are using. If a function that expects one kind of set receives another, the output is likely to be meaningless.

`VirtualDecomposition(C)`

`VirtualDecomposition(R, v)`

The virtual decomposition multiset of the virtual module with dominant character multiset C. The second version is provided for convenience, and equivalent to `VirtualDecomposition(LieRepresentationDecomposition(R,v))`.

`DecomposeCharacter(C)`

The decomposition multiset of the module with dominant character multiset C. An error is flagged if D is virtual, i.e. if dominant weights occur with negative multiplicities.

`DominantCharacter(D)`

Returns the dominant character multiset with decomposition D.

104.4.3 Calculating with Representations

As described earlier, many operations on representations carry over naturally to operations on their decompositions. This section describes the various functions for this purpose that were ported from LiE.

Note that many functions in this sections have two variants: one that takes decompositions as an argument and one that takes a root datum and a highest weight.

`RepresentationDimension(D)`

The dimension of the module with decomposition polynomial D. The algorithm used is described in [vLCL92].

`RepresentationDimension(R, v)`

The dimension of the module with highest weight v over the root datum R. The algorithm used is described in [vLCL92].

`CasimirValue(R, w)`

The value of the quadratic Casimir on representation with highest weight w, normalised to take the value 2 on the highest weight of the adjoint representation. This function is due to Dr. Bruce Westbury, University of Warwick

`QuantumDimension(R, w)`

Two Multisets of positive integers, Num and Den, which should be read as follows. Take the product of the integers in Num and divide by the product of the integers in Den to get the ordinary dimension. Replacing each integer by the quantum integer will give the quantum dimension. This function is due to Dr. Bruce Westbury, University of Warwick

Example H104E7

Dimensions:

```
> R := RootDatum("D4" : Isogeny := "SC");
> D := AdjointRepresentationDecomposition(R);
> RepresentationDimension(D);
28
> wts, mps := WeightsAndMultiplicities(D); wts,mps;
[
    (0 1 0 0)
]
[ 1 ]
> num,den := QuantumDimension(R, wts[1]); num,den;
{* 4^^2, 7 *}
{* 1, 2^^2 *}
```

```
> &*num/&*den;
28
```

`Branch(FromGrp, ToGrp, v, M)`

`Virtual` BoolElt *Default :* `false`

The decomposition polynomial of the restriction to ToGrp of the irreducible module V_v with respect to the restriction matrix M. M must be a matrix with dim(FromGrp) rows and Dim(ToGrp) columns.

The matrix M is used in such a way that any weight v' (expressed on the basis of fundamental weights for g), when restricted to a torus of ToGrp, becomes the weight $v'M$ (expressed on the basis of fundamental weights for ToGrp). A suitable restriction matrix can often be obtained by use of `RestrictionMatrix`. The algorithm used is described in [vLCL92].

The optional argument `Virtual` may be set to `true` to allow occurrence of virtual weights.

`Branch(ToGrp, D, M)`

`Virtual` BoolElt *Default :* `false`

As `Branch(FromGrp, ToGrp, v, M)` but with the irreducible module v replaced by the module with decomposition D.

`Collect(R, D, M)`

This function attempts to perform the inverse operation of `Branch`, namely to reconstruct an R-module from its restriction to R_D.

Please note that in LiE one must supply the inverse of the matrix used in Branch. Magma, however, is able to compute inverses itself, so one needs to provide the matrix used in Branch, and not its inverse.

M must be a square matrix whose dimension is equal to the dimension of R_D. The dimension of R must be equal to the dimension of R_D as well. The algorithm used is described in [vLCL92].

Example H104E8

Branch and Collect:

```
> R := RootDatum("D4" : Isogeny := "SC");
> S := RootDatum("A3T1" : Isogeny := "SC");
> M := RestrictionMatrix(R, S);
> br := Branch(R, S, [1,0,0,0], M);
> br;
Highest weight decomposition of representation of:
     S: Simply connected root datum of dimension 4 of type A3
     Number of terms: 2
> cl := Collect(R, br, M);
```

```
> cl:Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Dimension of weight space:4
     Weights:
          [
               (1 0 0 0)
          ]
     Multiplicities:
          [ 1 ]
```

`TensorProduct(R, v, w)`

`Goal` ANY *Default :*

The decomposition multiset of the tensor product of the representations with highest weights v and w over the root datum R.

If the optional parameter `Goal` is set, only the multiplicity of the irreducible module with highest weight `Goal` is returned. This does not greatly speed up the process, as the same computational steps need to be made, but it will significantly reduce memory consumption. The algorithm used is described in [vLCL92].

`TensorProduct(D, E)`

`Goal` ANY *Default :*

The decomposition multiset of the tensor product of the representations with decomposition multisets D and E.

If the optional parameter `Goal` is set, only the multiplicity of the irreducible module with highest weight `Goal` is returned. This does not greatly speed up the process, as the same computational steps need to be made, but it will significantly reduce memory consumption. The algorithm used is described in [vLCL92].

`TensorProduct(Q)`

`Goal` ANY *Default :*

The decomposition multiset of the tensor product of the representations with decomposition multisets in the sequence Q.

If the optional parameter `Goal` is set, only the multiplicity of the irreducible module with highest weight `Goal` is returned. This does not greatly speed up the process, as the same computational steps need to be made, but it will significantly reduce memory consumption. The algorithm used is described in [vLCL92].

`TensorPower(R, n, v)`

`TensorPower(D, n)`

The decomposition of the n-th tensor power of V_v^R or D.

Example H104E9

Taking tensor powers nicely shows how rapidly the complexity of representations increases, especially if we have a reasonably high weight as highest weight:

```
> R := RootDatum("D4" : Isogeny := "SC");
> DAd := AdjointRepresentationDecomposition(R);
> pwrs := function(D, n)
>   Q := [D];
>   for i in [2..n] do
>     Q[i] := Tensor(Q[1], Q[i-1]);
>   end for;
>   return Q;
> end function;
> time Q := pwrs(DAd, 7);
Time: 4.900
> [ #q : q in Q ];
[ 1, 7, 15, 30, 54, 91, 143 ]
> DH := LieRepresentationDecomposition(R, [2,2,0,0]);
> time Q := pwrs(DH, 4); [ #q : q in Q ];
Time: 99.070
[ 1, 105, 390, 1017 ]
```

`AdamsOperator(R, n, v)`

`AdamsOperator(D, n)`

The decomposition polynomial of the virtual module obtained by applying the n-th Adams operator to V_v^R or D. The algorithm used is described in [vLCL92].

`SymmetricPower(R, n, v)`

`SymmetricPower(D, n)`

The decomposition polynomial of $S^n(V_v^R)$, the n-th symmetric tensor power of V_v^R.

In the second form the irreducible module V_v^R is replaced by the module with decomposition D. The algorithm used is described in [vLCL92].

`AlternatingPower(R, n, v)`

`AlternatingPower(D, n)`

The decomposition polynomial of $\mathrm{Alt}^n(V_v^R)$, the n-th alternating tensor power of V_v^R.

In the second form the irreducible module V_v^R is replaced by the module with decomposition D. The algorithm used is described in [vLCL92].

`Plethysm(R, lambda, v)`

`Plethysm(D, lambda)`

The decomposition multiset of the R_D-module of the plethysm of V_v^R corresponding to the partition λ. Here λ should be a partition of $d = \dim V_v^R$, i.e., a non-increasing sequence consisting of positive integers with sum d. The value returned is the decomposition multiset of the representation of R_D that is obtained by composing the representation of R_D afforded by V_v^R, with the representation of $\mathrm{GL}(V_v^R)$ corresponding to the partition λ. The classical Frobenius formula is used (see [And77] and [JK81]).

In the second form the irreducible module V_v^R is replaced by the module with decomposition D.

`Spectrum(R, v, t)`

`Spectrum(D, t)`

Let n be the last entry of the sequence t; the toral element $t \in T$ will act in any representation of R as a diagonalisable transformation, all of whose eigenvalues are n-th roots of unity. This function returns a sequence in which the i-th entry is the multiplicity of the eigenvalue ζ^i in the action of the toral element t on the irreducible module V_v^R (or the module with decomposition D, in the second case). Here ζ is the complex number $e^{2\pi i/n}$.

See Section 104.1.2 for a description of the format of t.

Example H104E10

Spectrum provides a means to recognise toral elements in a more natural form. [vLCL92, Section 5.7.3].

```
> R := RootDatum("A4" : Isogeny := "SC");
> stdrep := [1,0,0,0];
> t := [1,0,0,0,2];
> stdrep := [1,0,0,0];
> Spectrum(R, stdrep, t);
[ 3, 2 ]
/* Showing that t has 3 eigenvalues 1 (1st root of unity),
   and 2 eigenvalues -1 (2nd root of unity) */
/* We may use the following function for constructing
   toral elements of A_n in the LiE format: */
> mktoral := function(b, d)
>     r := [ (i eq 1)
>                 select b[i]
>                 else b[i-1]+b[i] mod d
>              : i in [1..(#b-1)]
>          ];
>     r[#b] := d;
>     return r;
```

```
> end function;
> t2 := mktoral([0,0,0,1,1], 2); t2;
[ 0, 0, 0, 1, 2 ]
/* We restrict to a one parameter subgroup */
> RM := Transpose(Matrix([[0,0,0,1]]));
> T1 := RootDatum("T1" : Isogeny := "SC");
> Branch(R, T1, stdrep, RM):Maximal;
Highest weight decomposition of representation of:
     T1: Toral root datum of dimension 1
     Dimension of weight space:1
     Weights:
          [
               (1),
               (0),
               (-1)
          ]
     Multiplicities:
          [ 1, 3, 1 ]
/* Indicating that the element of that one parameter
   subgroup parametrised by some complex number z has
   one eigenvalue z^-1, three eigenvalues 1, and one
   eigenvalue z in the standard representation. */
```

`Demazure(R, v, w)`

`Demazure(D, w)`

Starting with the highest weight v of R, or the decomposition D, repeatedly apply the Demazure operator M_{α_i}, taking for i the successive entries of the Weyl word w (viewed as product of simple reflections).

`Demazure(R, v)`

`Demazure(D)`

Equivalent to `Demazure(R, v, w)` or `Demazure(D, w)` where w is the longest word of the Coxeter group of R or R_D.

If D is a decomposition polynomial, then the result E is the character polynomial of this decomposition. This is not the most efficient way to compute characters, but it can be very useful in checking other algorithms, since only the most elementary manipulations are involved.

Example H104E11

The Demazure operator:

```
> R := RootDatum("D4" : Isogeny := "SC");
> DAd := AdjointRepresentationDecomposition(R);
> DAdCp := Demazure(DAd); DAdCp;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Number of terms: 25
> DAd2 := AlternatingDominant(DAdCp); DAd2;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Number of terms: 1
> DAd2 eq DAd;
true
```

`LittlewoodRichardsonTensor(p, q)`

`LittlewoodRichardsonTensor(P, M, Q, N)`

In the first form, p and q are interpreted as dominant weights for the group SL_n (of type A_{n-1}) expressed in partition coordinates. Here n is the number of elements of p (which must be equal to the number of elements of q).

The tensor product of the corresponding highest weight modules is computed using the Littlewood-Richardson rule, and the result is expressed again in partition coordinates. To be precise, two sequences P, M, are returned, meaning that the highest weight module with partition coordinates P[i] occurs in the tensor product with multiplicity M[i].

In the second form, instead of two irreducible modules, the tensor product of the module having partition coordinates P[i] with multiplicity M[i] and the module having partition coordinates Q[j] with multiplicity N[j] is computed.

`LittlewoodRichardsonTensor(R, v, w)`

`LittlewoodRichardsonTensor(D, E)`

In the first form, compute the tensor product of the irreducible A_n representations with highest weights v and w using the Littlewood-Richardson rule. In the second form, compute the tensor product of the representations with decompositions D and E.

This procedure converts the weights to partitions, computes the tensor product using the Littlewood-Richardson rule (as described above, see LittlewoodRichardsonTensor), and converts the result back to a weight multiset.

Example H104E12

We compare the Littlewood-Richardson tensor and the normal tensor:

```
> R := RootDatum("A2" : Isogeny := "SC");
> v := [1,2];
> w := [1,1];
> D1 := Tensor(R, v, w);
> D1;
Highest weight decomposition of representation of:
    R: Simply connected root datum of dimension 2 of type A2
    Weights:
        [
            (0 1),
            (2 0),
            (0 4),
            (3 1),
            (2 3),
            (1 2)
        ]
    Multiplicities:
        [ 1, 1, 1, 1, 1, 2 ]
> D2 := LittlewoodRichardsonTensor(R, v, w);
> D2;
Highest weight decomposition of representation of:
    R: Simply connected root datum of dimension 2 of type A2
    Weights:
        [
            (1 2),
            (2 3),
            (2 0),
            (0 4),
            (0 1),
            (3 1)
        ]
    Multiplicities:
        [ 2, 1, 1, 1, 1, 1 ]
> D1 eq D2;
true
```

So the results are identical, as they should be. We could also convert the weights to partitions by hand, directly compute the Littlewood- Richardson tensor, and compare that to the previous result:

```
> vp := WeightToPartition(v); wp := WeightToPartition(w);
> vp, wp;
[ 3, 2, 0 ]
[ 2, 1, 0 ]
> parts, mps := LittlewoodRichardsonTensor(vp, wp);
> parts, mps;
```

```
[
    (4 4 0),
    (4 3 1),
    (3 3 2),
    (5 3 0),
    (5 2 1),
    (4 2 2)
]
[ 1, 2, 1, 1, 1, 1 ]
> [ PartitionToWeight(p) : p in parts ];
[
    (0 4),
    (1 2),
    (0 1),
    (2 3),
    (3 1),
    (2 0)
]
```

So that again gives the same representation. Finally, note that in some cases computing tensor products using the Littlewood-Richardson rule may be faster than computing them in the normal way:

```
> R := RootDatum("A8" : Isogeny := "SC");
> v := [0,0,2,0,1,0,1,2];
> w := [0,2,1,2,0,0,1,0];
> time _ := Tensor(R, v, w);
Time: 2.630
> time _ := LittlewoodRichardsonTensor(R, v, w);
Time: 0.210
```

`AlternatingDominant(D, w)`

`AlternatingDominant(R, wt, w)`

Alternating Dominant of the representation with decomposition D or the irreducible representation V_{wt}, with respect to Weyl group element w. Starting with D, the following operation is repeatedly applied, taking for i the successive entries of w (viewed as reflection). For any (weight, multiplicity) pair (v, c) of D let $v_i = \langle v, \alpha_i \rangle$ be its coefficient of w_i; the term is

- unaltered if $v_i \geq 0$,
- removed if $v_i = -1$, and
- replaced by $((v + w_i)r_i - w_i, -c)$ if $v_i = -2$. As a result of the operation for i, the coefficient v_i is made non-negative without affecting the image $M_{\alpha_i}(D)$ under the Demazure operator, and hence also without changing the value of its alternating Weyl sum `AlternatingWeylSum`.

`AlternatingDominant(D)`

`AlternatingDominant(R, wt)`

Equivalent to (but somewhat faster than) the previous `AlternatingDominant`(D, w) and `AlternatingDominant`(R, wt, w), with w the longest element of the corresponding Weyl group. If D is interpreted as dominant weights with multiplicities, then the result E contains highest weights and multiplicities.

Example H104E13

Example of the alternating dominant:

```
> R := RootDatum("D4" : Isogeny := "SC");
> v := [1,5,2,1];
> Dec1 := LieRepresentationDecomposition(R, v);
> // First, we construct the character polynomial for the
> // module with highest weight lambda
> Dom := DominantCharacter(Dec1 : InBasis := "Weight"); Dom;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Number of terms: 176
> W := CoxeterGroup(R); #W; act := RootAction(W);
192
> domwts, dommps := WeightsAndMultiplicities(Dom);
> CP := LieRepresentationDecomposition(R);
> for i in [1..#domwts] do
>   wt := domwts[i]; mp := dommps[i];
>   wtor := WeightOrbit(W, wt : Basis := "Weight");
>   for wti in wtor do
>     AddRepresentation(~CP, wti, mp);
>   end for;
> end for;
> CP;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Number of terms: 17712
> time ad := AlternatingDominant(CP); ad:Maximal;
Time: 54.200
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 4 of type D4
     Dimension of weight space:4
     Weights:
          [
               (1 5 2 1)
          ]
     Multiplicities:
          [ 1 ]
> time adalt := AlternatingDominant(CP, LongestElement(W));
```

```
Time: 8.330
> ad eq adalt;
true
```

`AlternatingWeylSum(R, v)`

`AlternatingWeylSum(D)`

The alternating Weyl sum of V_v^R or D. Useful for demonstration purposes, but the fact that the number of terms in the result is a multiple of the order of the CoxeterGroup of R makes it impractical for most groups.

104.5 Operations on Representations

104.5.1 Lie Algebras

The functions described in this section are applicable only to modules of almost reductive structure constant Lie algebras.

`CharacterMultiset(V)`

`CharacterMultiset(ρ)`

The character multiset of the Lie-algebra module V or representation ρ.

`Weights(V)`

`WeightsAndVectors(V)`

For a module V over a semisimple Lie algebra this returns two sequences. The first sequence consists of the weights that occur in V. The second sequence is a sequence of sequences of elements of V, in bijection with the first sequence. The i-th element of the second sequence consists of a basis of the weight space of weight equal to the i-th weight of the first sequence.

`Weights(ρ)`

`WeightsAndVectors(ρ)`

For a representation ρ of a semisimple Lie algebra this returns two sequences. The first sequence consists of the weights of ρ. The second sequence is a sequence of sequences of elements of the underlying vector space, in bijection with the first sequence. The i-th element of the second sequence consists of a basis of the weight space of weight equal to the i-th weight of the first sequence.

`DecompositionMultiset(V)`

`DecompositionMultiset(ρ)`

The decomposition multiset of the Lie-algebra module V or representation ρ.

`HighestWeightsAndVectors(V)`

This function is analogous to the previous one. Except in this case the first sequence consists of highest weights, i.e., those weights which occur as highest weights of an irreducible constituent of V. The second sequence consists of sequences that contain the corresponding highest weight vectors. So the submodules generated by the vectors in the second sequence form a direct sum decomposition of V.

`DirectSum(U, V)`

The direct sum of the Lie algebra modules U and V.

`DirectSumDecomposition(V)`

`IndecomposableSummands(V)`

Given a Lie algebra module V, return the direct sum decomposition of V as a sequence of submodules whose sum is V and each of which cannot be further decomposed into a direct sum. If the Lie algebra is semisimple over a field of characteristic zero, the summands are known to be irreducible highest weight modules.

`DirectSum(`ρ`, `τ`)`

The direct sum of the Lie algebra representations ρ and τ.

`DirectSumDecomposition(`ρ`)`

`IndecomposableSummands(`ρ`)`

Given a Lie algebra representation ρ, return the direct sum decomposition of ρ as a sequence of indecomposable subrepresentation. If the Lie algebra is semisimple over a field of characteristic zero, the summands are known to be irreducible highest weight representations.

`TensorProduct(Q)`

Given a sequence Q of left-modules over a Lie algebra, this function returns the module M that is the tensor product of the elements of Q. Secondly it returns a map from the Cartesian product of the elements of Q to M. This maps a tuple t to the element of M that is formed by tensoring the elements of t.

`SymmetricPower(V, n)`

Given a left-module V over a Lie algebra, and an integer $n \geq 2$, this function returns the module M that is the n-th symmetric power of V. It also returns a map f from the n-fold Cartesian product of V to M. This map is multilinear and symmetric, i.e., if two of its arguments are interchanged then the image remains the same. Furthermore, f has the universal property, i.e., any multilinear symmetric map from the n-fold Cartesian product into a vector space W can be written as the composition of f with a map from M into W.

```
ExteriorPower(V, n)
```

Given a left-module V over a Lie algebra, and an integer $2 \le n \le \dim(V)$, this function returns the module M that is the n-th exterior power of V. It also returns a map f from the n-fold Cartesian product of V to M. This map is multilinear and antisymmetric, i.e., if two of its arguments are interchanged then the image is multiplied by -1. Furthermore, f has the universal property, i.e., any multilinear antisymmetric map from the n-fold Cartesian product into a vector space W can be written as the composition of f with a map from M into W.

Example H104E14

```
> L:= LieAlgebra("G2", Rationals());
> V1:= HighestWeightModule(L, [1,0]);
> V2:= HighestWeightModule(L, [0,1]);
> T,f:= TensorProduct([V1,V2]);
> HighestWeightsAndVectors(T);
[
    (1 1),
    (2 0),
    (1 0)
]
[
    [
        T: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        T: (0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 -3 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ],
    [
        T: (0 0 0 0 0 0 1 2 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 -1 0 0
            0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 -3 0 0 0 0 0 0 0 0 0 0
            0 0 -3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
    ]
]
> DecomposeTensorProduct(RootDatum(L), [1,0], [0,1]);
[
    (1 1),
    (2 0),
    (1 0)
]
[ 1, 1, 1 ]
> f(<V1.2,V2.3>);
T: (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
```

So we see that the tensor product T decomposes as a direct sum of three submodules. This information can also be computed by using `DecomposeTensorProduct`. However, in the former case, the corresponding highest-weight vectors are also given.

```
> E,h:= ExteriorPower(V1, 3);
> h(<V1.1,V1.3,V1.4>);
E: (0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> h(<V1.1,V1.4,V1.3>);
E: (0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
> DecomposeExteriorPower( RootDatum(L), 3, [1,0] );
[
    (2 0),
    (1 0),
    (0 0)
]
[ 1, 1, 1 ]
> HighestWeightsAndVectors(E);
[
    (2 0),
    (1 0),
    (0 0)
]
[
    [
        E: (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0)
    ],
    [
        E: (0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
        0)
    ],
    [
        E: (0 0 0 0 0 0 0 0 0 0 0 1 2 0 0 0 0 0 2 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0
        0)
    ]
]
```

104.5.2 Groups of Lie Type

These functions apply to projective representations of groups of Lie type. Note that modules have not yet been implemented for these groups.

`DirectSum(ρ, τ)`

The direct sum of the group of Lie type representations ρ and τ.

`DirectSumDecomposition(ρ)`
`IndecomposableSummands(ρ)`

Given a group of Lie type representation ρ, return the direct sum decomposition of ρ as a sequence of indecomposable subrepresentation. If the base field has characteristic zero, the summands are known to be irreducible highest weight representations.

`CharacterMultiset(V)`
`CharacterMultiset(ρ)`

The character weight multiset of the group of Lie type representation ρ.

`Weights(ρ)`
`WeightsAndVectors(ρ)`

The weights of the representation ρ, together with the corresponding weight vectors.

`WeightVectors(ρ)`

A basis of weight vectors of the representation ρ.

`Weight(ρ, v)`

The weight corresponding to the weight vector v of the representation ρ.

`DecompositionMultiset(V)`
`DecompositionMultiset(ρ)`

The decomposition multiset of the group of Lie type representation ρ.

`HighestWeights(ρ)`

The highest weights of the representation ρ, together with the corresponding highest weight vectors. This function may fail for small finite fields.

`HighestWeightVectors(ρ)`

The highest weight vectors of the representation ρ.

`GeneralisedRowReduction(ρ)`

Given a projective matrix representation $\rho : G \to \mathrm{GL}_m(k)$, return its inverse. This algorithm is based on [CMT04].

104.6 Other Functions for Representation Decompositions

In this section, we describe more complicated functions for dealing with representation decompositions.

`FundamentalClosure(R, S)`

A set of fundamental roots in the minimal subsystem (not necessarily closed!) of the root system R that contains all the roots in S. This function is equivalent to the `fundam` function in LiE.

The set S should contain either roots or root indices; the returned set will then contain objects of the same type.

`Closure(R, S)`

A set of fundamental roots in the minimal subsystem (not necessarily closed!) of the root system R that contains all the roots in S. This function is equivalent to the `closure` function in LiE.

The set S should contain either roots or root indices; the returned set will then contain objects of the same type.

`RestrictionMatrix(R, Q)`

For a simply connected root datum R and a sequence of roots Q forming a fundamental basis for a closed subdatum S of R, this function computes a restriction matrix for the fundamental Lie subgroup of type S of the Lie group corresponding to R.

The sequence Q may contain either integers (where i corresponds to the i-th root of R) or vectors (interpreted as root vectors written in the root basis of R).

Note that the result is not unique. Moreover, if the result is to be used by Branch or Collect the roots in Q must be positive roots, and their mutual inner products must be non-positive.

`RestrictionMatrix(R, S)`

Let S be a sub root datum of R, constructed for example (but not necessarily) using a call to `sub<...>`. Then the matrix M returned by this function maps the fundamental weights of R to those of S. Note that, if the rank of S is smaller than the rank of R, there will be more than one such matrix.

Example H104E15

Constructing a restriction matrix

```
> R := RootDatum("D4":  Isogeny := "SC");
> sub<R | [1,3,4]>;
Root datum of dimension 4 of type A1 A1 A1
[ 1, 3, 4, 13, 15, 16 ]
> S := RootDatum("A1A1A1T1" : Isogeny := "SC");
> M := RestrictionMatrix(R, S); M;
```

```
[ 1  0  0 -1]
[ 0  1  0 -2]
[ 0  0  1 -1]
[ 0  0  0  4]
> imgR := FundamentalWeights(R)*M; imgR;
[ 1  0  0 -1]
[ 0  1  0 -2]
[ 0  0  1 -1]
[ 0  0  0  4]
> FundamentalWeights(S);
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
/* M is of the required form, since: */
> [ BasisChange(S, BasisChange(S, imgR[i]
>     : InBasis := "Standard", OutBasis := "Weight")
>     : InBasis := "Weight", OutBasis := "Standard")
>   : i in [1..4]
> ];
[
    (1 0 0 0),
    (0 1 0 0),
    (0 0 1 0),
    (0 0 0 0)
]
```

`KLPolynomial(x, y)`

`Ring` RngUPol *Default :* **Z**[X]

The Kazhdan-Lusztig polynomial $P_{x,y}$. We use the recursion given originally by Kazhdan and Lusztig [KL79].

`RPolynomial(x, y)`

`Ring` RngUPol *Default :* **Z**[X]

The R-polynomial $R_{x,y}$.

Example H104E16

There is a relation between Kazhdan-Lusztig polynomials and R-polynomials. We should have, for any $x, w \in W$:

$$X^{l(w)-l(x)}\overline{P_{x,w}} - P_{x,w} = \sum_{x<y\leq w} R_{x,w}P_{y,w},$$

where the bar indicates a sign change of all the exponents. We need some fiddling around in order to implement this sign change, since MAGMA doesn't support negative exponents at the moment, but we can make it work:

```
> signchange := function(pol, pwr)
>     //returns X^pwr * bar(pol)
>     deg := Degree(pol);
>     P := Parent(pol);
>     if (deg gt pwr) then return "Failed: Can't do sign change"; end if;
>     return (P.1)^(pwr-deg)*P!Reverse(Eltseq(pol));
> end function;
> testKL := function(x, w)
>     W := Parent(x);
>     rng<X> := PolynomialRing(Integers());
>     lenw := CoxeterLength(W, w);
>     lenx := CoxeterLength(W, x);
>
>     if (lenx gt lenw) then
>         return "Failed: l(x) > l(w) gives zero R and KL polynomials.";
>     end if;
>
>     /* Left hand side */
>     Pxw := KLPolynomial(x, w : Ring := rng);
>     lhs := signchange(Pxw, lenw - lenx);
>     if (Type(lhs) eq MonStgElt) then return lhs; end if;
>     lhs -:= Pxw;
>
>     /* Right hand side */
>     rhs := rng!0;
>     lvl := {w};
>     lvllen := lenw;
>     while (lvllen gt lenx and #lvl gt 0) do
>         for y in lvl do
>             rhs +:= RPolynomial(x,y : Ring := rng)*
>                         KLPolynomial(y, w : Ring := rng);
>         end for;
>         lvl := BruhatDescendants(lvl : z := x);
>         lvllen -:= 1;
>     end while;
>
>     /* Done */
>     printf "LHS: %o\n", lhs;
```

```
>     printf "RHS: %o\n", rhs;
>     return lhs eq rhs;
> end function;
> W := CoxeterGroup("D4");
> x := W.1*W.2*W.1;
> w := W.1*W.2*W.3*W.4*W.1*W.2;
> testKL(x,w);
LHS: X^3 + X^2 - X - 1
RHS: X^3 + X^2 - X - 1
true
```

`Exponents(R)`

The exponents of a Root datum R form a sequence of numbers $e_1, \ldots, e_r$, where r is the rank of R, such that the polynomial $\sum_{w \in W} X^{l(w)}$ decomposes as a product $\prod_{i=1}^{r} \sum_{j=0}^{e_i} X^j$. They are given in weakly increasing order.

Example H104E17

Exponents of A_3:

```
> W := CoxeterGroup("A3"); #W;
24
> P<X> := PolynomialRing(Integers());
> f := &+[ X^(CoxeterLength(W, w)) : w in W ]; f;
X^6 + 3*X^5 + 5*X^4 + 6*X^3 + 5*X^2 + 3*X + 1
> R := RootDatum("A3" : Isogeny := "SC");
> exp := Exponents(R); exp;
[ 1, 2, 3 ]
> g := [ &+[ X^j : j in [0..e] ] : e in exp ]; g;
[
    X + 1,
    X^2 + X + 1,
    X^3 + X^2 + X + 1
]
> f eq &*g;
true
```

`ToLiE(D)`

The LiE equivalent of the decomposition D.

`FromLiE(R, p)`

The decomposition of the representation over R that is equivalent to p, where p is a polynomial in LiE-syntax.

Example H104E18

Conversion to and from LiE-syntax

```
> R := RootDatum("B3" : Isogeny := "SC");
> D := LieRepresentationDecomposition(R, [1,2,3]);
> s := ToLiE(D); s;
1X[1,2,3]
> FromLiE(R, s):Maximal;
Highest weight decomposition of representation of:
     R: Simply connected root datum of dimension 3 of type B3
     Dimension of weight space:3
     Weights:
          [
               (1 2 3)
          ]
     Multiplicities:
          [ 1 ]
```

104.6.1 Operations Related to the Symmetric Group

In this section, we describe some functions taken from LiE for dealing with the Symmetric group.

`ConjugationClassLength(l)`

The order of the conjugation class of S_n of permutations of cycle type l (for n the sum of the elements of l).

`PartitionToWeight(l)`

Let n be the number of parts of l, then the function returns the weight for a group of type A_{n-1} corresponding to λ, expressed on the basis of fundamental weights.

`WeightToPartition(v)`

Let n be the length of v, then v is interpreted as a weight for a group of type A_n, and the expression of that weight in $n+1$ partition coordinates is returned. When v is dominant, this is a partition with $n+1$ parts.

`TransposePartition(l)`

The transpose partition of l.

104.6.2 FusionRules

In this section, we describe a function for computing fusion rules using the Kac-Walton formular, as described in Section 16.2 of [FMS97].

`WZWFusion(R, v, w, k)`

`ReturnForm` MonStgElt *Default :* "*Auto*"

Compute the fusion rules for weights $v \times w$ of R at level k using the Kac-Walton formula. The weights v and w may be given either as finite weights (i.e. vectors or sequences with rank(R) entries) or as affine weights (i.e. vectors or sequences with rank(R)+1 entries).

The optional argument `ReturnForm` should be "Auto" (in which case the weights returned will be finite weights or affine weights depending on what v and w are; the default), "Finite" (in which case the weights returned are finite weights), or "Affine" (in which case the weights returned are affine weights).

Note that R should be a weakly simply connected root datum.

`WZWFusion(D, E, k)`

Compute the fusion rules for representations D and E at level k.

Example H104E19

Fusion rules at level 3 for B_3 (using finite weights first, affine weights second)

```
> R := RootDatum("B3" : Isogeny := "SC");
> WZWFusion(R, [0,0,1],[1,0,1], 3);
{*
    (2 0 0),
    (1 1 0),
    (0 0 2),
    (1 0 2),
    (1 0 0),
    (0 1 0)
*}
> WZWFusion(R, [0,0,1],[1,0,1], 3 : ReturnForm := "Affine");
{*
    (2 0 0 1),
    (1 0 2 0),
    (0 1 0 1),
    (1 0 0 2),
    (0 0 2 1),
    (1 1 0 0)
*}
```

104.7 Subgroups of Small Rank

LiE contains a small database with the types of the maximal proper subgroups of complex reductive simply connected Lie groups g, where g is simple and of rank at most 8. We copied this list into MAGMA, and it can be accessed using the following functions.

`LiEMaximalSubgroups()`

All maximal subgroups as described above, as a sequence of pairs. Each pair consists of a string denoting the simple group at hand, and a sequence of strings denoting its maximal subgroups.

`MaximalSubgroups(G)`

The maximal subgroups of the complex reductive simply connected simple Lie group whose Cartan type is the string G, represented as a sequence of strings.

`RestrictionMatrix(G, H)`

`Index` RNGINTELT *Default :* -1

The restriction matrix for the maximal proper subgroup of type H of G. If more than one maximal subgroup of G is of type H, the parameter *Index* must be set to indicate which one is required.

Example H104E20

Using the subgroup database:

```
> MaximalSubgroups("E7");
[ A2, A1, A1, A1F4, G2C3, A1G2, A1A1, D6A1, A7, A5A2 ]
> M := RestrictionMatrix("E7", "A1" : Index := 2); M;
[26]
[37]
[50]
[72]
[57]
[40]
[21]
> R := RootDatum("E7" : Isogeny := "SC");
> S := RootDatum("A1" : Isogeny := "SC");
> D := AdjointRepresentationDecomposition(R);
> RepresentationDimension(D);
133
> E := Branch(S, D, M); #E;
8
> RepresentationDimension(E);
133
```

104.8 Subalgebras of su(d)

This section describes functions for studying irreducible simple subalgebras of the Lie algebra $\mathfrak{su}(d)$ (cf. [Dyn57]). The verbose flag `"SubSU"` may be set to show details and progress of the various computations.

The algorithms and the implementation in this package are due to Robert Zeier. For more information about some of the algorithms used and the results obtained using this package we refer to [ZSH11].

`IrreducibleSimpleSubalgebrasOfSU(N)`

A list of all irreducible simple subalgebras occurring in the Lie algebra $\mathfrak{su}(d)$, for $2 \leq d \leq N$.

`IrreducibleSimpleSubalgebraTreeSU(Q, d)`

The subalgebra tree for degree d as a directed graph whose vertex labels describe subalgebras, derived from the list Q of irreducible subalgebras. The vertex labels are records with three fields: `algebra`, a string containing the Cartan type of this subalgebra; `weights`, a sequence of highest weights (as sparse vectors) corresponding to irreducible representations (they are related by an outer automorphism if there is more than one highest weight); and `type`, an integer with values -1, 1, or 0 corresponding to irreducible representations of quaternionic, real, or complex type, respectively (the Frobenius-Schur indicator).

`PrintTreesSU(Q, F)`

`FromDegree`	RngIntElt	*Default :* 2
`ToDegree`	RngIntElt	*Default :* $\|Q\|$
`IncludeTrivial`	BoolElt	*Default :* `true`

Print the tree of subalgebras in the sequence Q (as obtained by a call to IrreducibleSimpleSubalgebrasOfSU) to the file with filename F. The file F will be overwritten.

The resulting file will be a LaTeX document that may be typeset using `latex` followed by `dvipdf`, for instance. If the resulting file is large, the main memory allocated to TEX may have to be increased (the `main_memory` directive in `texmf.cnf`). Contact your system administrator in case of difficulty.

The optional arguments `FromDegree` and `ToDegree` limit which degrees are output; `IncludeTrivial` may be set to `false` to remove "trivial" cases (i.e. trivial trees) from the output. For $d \geq 5$ and d even, $\mathfrak{su}(d)$ is considered trivial if it contains only the (proper) irreducible simple subalgebras $C_{d/2}$ (i.e. $\mathfrak{sp}(d/2)$), $D_{d/2}$ (i.e. $\mathfrak{so}(d)$), and A_1 (i.e. $\mathfrak{su}(2)$); for $d \geq 5$ and d is odd, $\mathfrak{su}(d)$ is considered trivial if it contains only $B_{(d-1)/2}$ (i.e. $\mathfrak{so}(d)$) and A_1.

The Lie algebras in the output are coloured according to type: red for -1, blue for 1, and black for 0 (see IrreducibleSimpleSubalgebraTreeSU).

Example H104E21

We investigate subalgebras of $\mathfrak{su}(d)$ for d up to 2^{10}.

```
> Q := IrreducibleSimpleSubalgebrasOfSU(2^10);
> t := IrreducibleSimpleSubalgebraTreeSU(Q, 12);
> t;
Digraph
Vertex  Neighbours
1       2 4 ;
2       3 ;
3       ;
4       ;
> r := VertexLabel(t, 1); r`algebra;
rec<recformat<algebra: MonStgElt, weights, type: IntegerRing()> |
    algebra := A11,
    weights := [
        Sparse matrix with 1 row and 11 columns over Integer Ring,
        Sparse matrix with 1 row and 11 columns over Integer Ring
    ],
    type := 0>
> r := VertexLabel(t, 2); r;
rec<recformat<algebra: MonStgElt, weights, type: IntegerRing()> |
    algebra := C6,
    weights := [
        Sparse matrix with 1 row and 6 columns over Integer Ring
    ],
    type := -1>
> [ Matrix(w) : w in r`weights ];
[
    [1 0 0 0 0 0]
]
> RepresentationDimension(RootDatum("C6"),[1,0,0,0,0,0]);
12
> r := VertexLabel(t, 3); r`algebra;
A1
> [ Matrix(w) : w in r`weights ];
[
    [11]
]
> RepresentationDimension(RootDatum("A1"),[11]);
12
> r := VertexLabel(t, 4); r`algebra;
D6
```

In this manner we have used IrreducibleSimpleSubalgebraTreeSU to obtain information about irreducible simple subalgebras of $\mathfrak{su}(12)$: A_{11} ($\mathfrak{su}(12)$) is the root of the tree, C_6 corresponds to a proper subalgebra of A_{11}, and A_1 is a proper subalgebra of C_6. In addition, we have used RepresentationDimension to verify the dimensions of the representations. Let us use

RepresentationDimension to see what other $\mathfrak{su}(d)$ the Lie algebra of type C_6 should at the very least occur in:

```
> V := RSpace(Integers(), 6);
> [ RepresentationDimension(RootDatum("C6"), v) : v in Basis(V) ];
[ 12, 65, 208, 429, 572, 429 ]
```

We compare that to the list of $\mathfrak{su}(d)$ it does occur in using IrreducibleSimpleSubalgebraTreeSU and obtain the weights for the case $\mathfrak{su}(78)$.

```
> [ i : i in [2..2^10] | exists{r : r in VertexLabels(
>    IrreducibleSimpleSubalgebraTreeSU(Q, i)) | r`algebra eq "C6"} ];
[ 12, 65, 78, 208, 364, 429, 560, 572 ]
> t := IrreducibleSimpleSubalgebraTreeSU(Q, 78);
> l := VertexLabels(t);
> [ r`algebra : r in l ];
[ A77, C39, A1, D39, B6, C6, E6, A2, A11, A12 ]
> r := l[6];
> [ Matrix(x) : x in r`weights ];
[
    [2 0 0 0 0 0]
]
```

104.9 Bibliography

[**And77**] C.M. Andersen. Clebsch-Gordan Series for symmetrized tensor products. *J. Math. Phys*, 8:988–997, 1977.

[**CMT04**] Arjeh M. Cohen, Scott H. Murray, and D. E. Taylor. Computing in groups of Lie type. *Math. Comp.*, 73(247):1477–1498, 2004.

[**dG01**] W. A. de Graaf. Constructing representations of split semisimple Lie algebras. *J. Pure Appl. Algebra*, 164(1-2):87–107, 2001. Effective methods in algebraic geometry (Bath, 2000).

[**Dyn57**] E. B. Dynkin. Maximal Subgroups of the Classical Groups. *Amer. Math. Soc. Transl. Ser. 2*, 6:245–378, 1957.

[**FMS97**] Philippe Di Francesco, P. Mathieu, and D. Sénéchal. *Conformal Field Theory*. Graduate texts in contemporary physics. Springer, 1997.

[**JK81**] G. James and A. Kerber. *The Representation Theory of the Symmetric Group*. Addison-Weasly, Reading MA, 1981.

[**KL79**] D. Kazhdan and G. Lusztig. Representations of Coxeter groups and Hecke algebras. *Inventiones Math.*, 53:165–184, 1979.

[**vLCL92**] M.A.A. van Leeuwen, A.M. Cohen, and B. Lisser. *LiE, A package for Lie Group Computations*. CAN, Amsterdam, 1992.

[**ZSH11**] Robert Zeier and Thomas Schulte-Herbrüggen. Symmetry principles in quantum systems theory. *Journal of Mathematical Physics*, 52(11):113510, 2011.

PART XIV
COMMUTATIVE ALGEBRA

105 GRÖBNER BASES

Chapter 105
GRÖBNER BASES

105.1 Introduction

This chapter describes the basics for configuring MAGMA's powerful *Gröbner basis* machinery, which lies at the heart of computations with ideals and modules over multivariate polynomial rings. Later chapters will describe the many functions and operations available to the user for working with ideal and modules.

Gröbner bases were introduced by Bruno Buchberger [Buc65] and at the heart of the theory is the *Buchberger algorithm* which computes a Gröbner basis of an ideal starting from an arbitrary basis (generating set) of the ideal. The two books *Ideals, Varieties and Algorithms* [CLO96] and *Gröbner Bases* [BW93] have also inspired much of the design and presentation of ideals of multivariate polynomial rings in MAGMA.

Since V2.11 (May 2004), MAGMA also contains a highly optimized implementation of the Faugère F_4 algorithm [Fau99], based on sparse linear algebra techniques, which usually performs dramatically better than the Buchberger algorithm (see [Ste04]).

Chapter 24 deals with the basics of multivariate polynomial rings and their elements (for which there are very many functions), so it is recommended that that chapter be perused before reading this one.

Permutation and matrix groups have a natural action on multivariate polynomial rings. This leads to the subject of invariant rings of finite groups, which is covered in Chapter 110. See also the chapters on affine algebras (Chapter 108) and on modules over affine algebras (Chapter 109), and the chapter on algebraically closed fields (Chapter 40), which allows one to compute the variety of an ideal over the algebraic closure of the base field.

105.2 Representation and Monomial Orders

Let P be the polynomial ring $R[x_1, \ldots, x_n]$ of rank n over a ring R. A *monomial* (or *power product*) of P is a product of powers of the variables (or indeterminates) of P, that is, an expression of the form $x_1^{e_1} \cdots x_n^{e_n}$ with $e_i \geq 0$ for $1 \leq i \leq n$. Multivariate polynomials in MAGMA are stored efficiently in distributive form, using arrays of coefficient-monomial pairs, where the coefficient is in the base ring R. The word 'term' will always refer to a coefficient multiplied by a monomial.

Monomial orders are of critical importance when dealing with Gröbner bases. Let M be the set of all monomials of P. A *monomial ordering* on M is a total order $<$ on M such that $1 \leq s$ for all $s \in M$, $s \leq t$ implies $su \leq tu$ for all $s, t, u \in M$, and M is a well-ordering (every non-empty subset of M possesses a minimal element w.r.t. $<$). Monomial orders can be naturally specified in terms of *weight vectors*: a vector W from $\mathbf{Q}^n$ with non-negative entries is called a weight vector since it weights a monomial s by the product $s.W$ (defined to be the dot product of the exponent vector of s with W); any sequence of

n linearly-independent weight vectors determines a monomial order on M (see the weight order below [subsection 105.2.9]). All monomial orderings can in fact be represented in terms of weight vectors.

Multivariate polynomial rings are constructed in MAGMA such that the monomials of any polynomial are sorted with respect to a specified monomial order, with the greatest monomial first. Gröbner basis computations are dramatically affected by the choice of monomial order. MAGMA provides an extensive choice of monomial orders. Currently, the intrinsic functions `PolynomialRing` (or `PolynomialAlgebra`), `ChangeOrder` and `VariableExtension` expect a monomial order; it is specified by a string giving the name, optionally followed by extra arguments for that order.

We now describe each of the monomial orders available in MAGMA. We suppose that s and t are monomials from P which has rank n. Any order on the monomials is then fully defined by just specifying exactly when $s < t$ with respect to that order. In the following, the argument(s) are described for an order as a list of expressions; that means that the expressions (without the parentheses) should be appended to any base arguments when any particular intrinsic function is called which expects a monomial order. See also [CLO96, Chap. 2, §2] for more details about the first three orders.

105.2.1 Lexicographical: `lex`

Definition: $s < t$ iff there exists $1 \leq i \leq n$ such that the first $i-1$ exponents of s and t are equal but the i-th exponent of s is less than the i-th exponent of t. The order is specified by the argument (`"lex"`).

The order is called "lexicographical" since it orders the monomials as if they were words in a dictionary. The i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. A Gröbner basis of an ideal with respect to the lexicographical order usually represents the most information about the ideal but can be hard to compute.

105.2.2 Graded Lexicographical: `glex`

Definition: $s < t$ iff the total degree of s is less than the total degree of t or the total degree of s is equal to the total degree of t and $s < t$ with respect to the lexicographical order. The order is specified by the argument (`"glex"`).

The order is called "graded lexicographical" since it first grades the monomials by total degree, and then decides ties by the lexicographical order. The i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. This order is rarely used because the `grevlex` order below is usually a better degree order (i.e., yields smaller Gröbner bases).

105.2.3 Graded Reverse Lexicographical: `grevlex`

Definition: $s < t$ iff the total degree of s is less than the total degree of t or the total degree of s is equal to the total degree of t and $s > t$ with respect to the lexicographical order applied to the exponents of s and t in reverse order. The order is specified by the argument (`"grevlex"`).

The order is called "graded reverse lexicographical" since it first grades the monomials by total degree, and then decides ties by the negation of the lexicographical order applied to the variables in reverse order. Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. A Gröbner basis of an ideal with respect to the graded reverse lexicographical order is usually the easiest to compute so it is recommended that this order be used when just any Gröbner basis for an ideal is desired.

105.2.4 Graded Reverse Lexicographical (Weighted): `grevlexw`

Definition (given a sequence W of n positive integer weights): $s < t$ iff the total weighted degree d_s of s w.r.t. W is less than the total degree d_t of t w.r.t. W or $d_s = d_t$ and $s > t$ with respect to the lexicographical order applied to the exponents of s and t in reverse order. The order is specified by the arguments (`"grevlexw", W`).

The order is called "graded reverse lexicographical (weighted)" since it first grades the monomials by weighted degree w.r.t. W, and then decides ties by the negation of the lexicographical order applied to the variables in reverse order. If $W = [1, 1, \ldots, 1]$, then this order is equal to the `grevlex` order. Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable.

This order is similar to the `grevlex` order, but is useful if an ideal is homogeneous with respect to the grading given by W, since the Gröbner basis of the ideal will tend to be smaller with this order.

105.2.5 Elimination (k): `elim`

Definition (given k with $1 \leq k \leq n-1$): $s < t$ iff $s_k < t_k$ with respect to the `grevlex` order or $s_k = t_k$ and $s_{k'} < t_{k'}$ with respect to the `grevlex` order where m_k denotes the monomial consisting of the first k exponents of m and $m_{k'}$ denotes the monomial consisting of the last $n-k$ exponents of m (this order is thus the concatenation of two block `grevlex` orders). The order is specified by the arguments (`"elim", k`).

The order is called "elimination" since the first k variables are "eliminated": if G is a Gröbner basis of an ideal I of the polynomial ring $K[x_1, \ldots, x_n]$ with respect to this order, then $G \cap K[x_{k+1}, \ldots, x_n]$ is a Gröbner basis of the k-th elimination ideal $I \cap K[x_{k+1}, \ldots, x_n]$. (It is usually easier to compute a Gröbner basis with respect to this order for any k than with respect to the full lexicographical order.) Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable.

105.2.6 Elimination List: `elim`

Definition (given sequences U and V such that U and V contain distinct integers in the range 1 to n and the sum of the lengths of U and V is n and U and V are disjoint): $s < t$ iff $s_U < t_U$ with respect to the `grevlex` order or $s_U = t_U$ and $s_V < t_V$ with respect to the `grevlex` order where m_L denotes the monomial consisting of the exponents of m corresponding to the entries of L in order. The order is specified by the arguments (`"elim", U, V`). V may be omitted if desired so the arguments are just (`"elim", U`); in this case V is chosen to be an appropriate sequence to complement U.

The order is called "elimination" since the variables in U are "eliminated". The order of the elements in U and V are significant since the ordering on the variables makes $U[1]$ greatest, then $U[2]$, etc., then $V[1]$, $V[2]$, etc.

105.2.7 Inverse Block: `invblock`

Definition (given sequences U and V such that U and V contain distinct integers in the range 1 to n and the sum of the lengths of U and V is n and U and V are disjoint): $s < t$ iff $s_V < t_V$ with respect to the `grevlex` order or $s_V = t_V$ and $s_U < t_U$ with respect to the `grevlex` order. The order is specified by the arguments (`"invblock", U, V`). V may be omitted if desired so the arguments are just (`"invblock", U`); in this case V is chosen to be an appropriate sequence to complement U.

The order is called "inverse block" since it applies a block ordering on the exponents on V then U which inverts the lists supplied to the elimination list order. Thus this is the same as the elimination order except that the lists U and V are swapped. See [BW93, p. 390] for the motivation for this order.

105.2.8 Univariate: `univ`

Definition (given i with $1 \leq i \leq n$): $s < t$ iff $s_L < t_L$ with respect to the `grevlex` order or $s_L = t_L$ and the i-th exponent of s is less than the i-th exponent of t, where L is the sequence `[1 .. n]` with i deleted. The order is specified by the arguments (`"univ", i`).

The order is called "univariate" since when monomials are compared, any monomial not containing the i-th variable is greater than any monomial containing the i-th variable. Thus all variables but the i-th are "eliminated" so that a Gröbner basis of a zero-dimensional ideal I with this ordering will contain the unique monic generator of the elimination ideal consisting of all the polynomials in I containing the i-th variable alone. The j-th variable is greater than the $(j+1)$-th variable for $1 \leq j < i$ and $i < j \leq n$ and the j-th variable is greater than the i-th variable for any $j \neq i$.

105.2.9 Weight: `weight`

Definition (given n weight vectors $W_1, \ldots W_n$ from $\mathbf{Q}^n$): $s < t$ iff there exists $1 \leq i \leq n$ such that $s.W_j = t.W_j$ for $1 \leq j < i$ and $s.W_i < t.W_i$. The order is specified by the arguments (`"weight", Q`) where Q is a sequence of n^2 non-negative integers or rationals describing the n weight vectors of length n (in row major order).

The n weight vectors must describe a vector space basis of $\mathbf{Q}^n$ (i.e., be linearly-independent), since otherwise this would not yield a total ordering on the monomials. The weight order allows one to specify any possible monomial order; any of the monomial orders mentioned above can be specified by an appropriate choice of weight vectors. However, using the in-built versions of the specialized orders above is much faster than constructing versions of them based on weight vectors. The next section contains an example in which a polynomial ring is constructed with a weight order for the monomials.

105.3 Polynomial Rings and Ideals

105.3.1 Creation of Polynomial Rings and Accessing their Monomial Orders

Multivariate polynomial rings are created from a coefficient ring, the number of variables, and a monomial order. If no order is specified, the monomial order is taken to be the lexicographical order. This section is briefly repeated from the section 24.2.1 in the multivariate polynomial rings chapter, so as to show how one can set up the polynomial ring in which to create an ideal.

Please note that the Gröbner basis of an ideal with respect to the lexicographical order is often much more complicated and difficult to compute than the Gröbner basis of the same ideal with respect to other monomial orders (e.g. the `grevlex` order), so it may be preferable to use another order if the Gröbner basis with respect to any order is desired (see also the function `EasyIdeal` below). Yet the lexicographical order is the most natural order and is often the desired order so that is why it is used by default if no specific order is given.

`PolynomialRing(R, n)`

`PolynomialAlgebra(R, n)`

`Global` BoolElt *Default :* `false`

Create a multivariate polynomial ring in $n > 0$ variables over the ring R. The ring is regarded as an R-algebra via the usual identification of elements of R and the constant polynomials. The lexicographical ordering on the monomials is used for this default construction (see next function).

By default, a *non-global* polynomial ring will be returned; if the parameter `Global` is set to `true`, then the unique global polynomial ring over R with n variables will be returned. This may be useful in some contexts, but a non-global result is returned by default since one often wishes to have several rings with the same numbers of variables but with different variable names (and create mappings between them, for example). Explicit coercion is always allowed between polynomial rings having the same number of variables (and suitable base rings), whether they are global or not, and the coercion maps the i-variable of one ring to the i-th variable of the other ring.

`PolynomialRing(R, n, order)`

`PolynomialAlgebra(R, n, order)`

Create a multivariate polynomial ring in $n > 0$ variables over the ring R with the given order *order* on the monomials. See the section on monomial orders for the valid values for the argument *order*.

`PolynomialRing(R, n, T)`

`PolynomialAlgebra(R, n, T)`

Create a multivariate polynomial ring in $n > 0$ variables over the ring R with the order given by the tuple T on the monomials. T must be a tuple whose components match the valid arguments for the monomial orders in Section 105.2 (or a tuple returned by the following function `MonomialOrder`).

`MonomialOrder(P)`

Given a polynomial ring P (or an ideal thereof), return a description of the monomial order of P. This is returned as a tuple which matches the relevant arguments listed for each possible order in Section 105.2, so may be passed as the third argument to the function `PolynomialRing` above.

`MonomialOrderWeightVectors(P)`

Given a polynomial ring P of rank n (or an ideal thereof), return the weight vectors of the underlying monomial order as a sequence of n sequences of n rationals. See, for example, [CLO98, p. 153] for more information.

Example H105E1

We show how one can construct different polynomial rings with different orders.

```
> Z := IntegerRing();
> // Construct polynomial ring with DEFAULT lex order
> P<a,b,c,d> := PolynomialRing(Z, 4);
> MonomialOrder(P);
<"lex">
> MonomialOrderWeightVectors(P);
[
    [ 1, 0, 0, 0 ],
    [ 0, 1, 0, 0 ],
    [ 0, 0, 1, 0 ],
    [ 0, 0, 0, 1 ]
]
> // Construct polynomial ring with grevlex order
> P<a,b,c,d> := PolynomialRing(Z, 4, "grevlex");
> MonomialOrder(P);
<"grevlex">
> MonomialOrderWeightVectors(P);
[
    [ 1, 1, 1, 1 ],
    [ 1, 1, 1, 0 ],
    [ 1, 1, 0, 0 ],
    [ 1, 0, 0, 0 ]
]
> // Construct polynomial ring with block elimination and a > d > b > c
```

```
> P<a,b,c,d> := PolynomialRing(Z, 4, "elim", [1, 4], [2, 3]);
> MonomialOrder(P);
<"elim", [ 1, 4 ], [ 2, 3 ]>
> MonomialOrderWeightVectors(P);
[
    [ 1, 0, 0, 1 ],
    [ 1, 0, 0, 0 ],
    [ 0, 1, 1, 0 ],
    [ 0, 1, 0, 0 ]
]
> a + b + c + d;
a + d + b + c
> a + d^10 + b + c^10;
d^10 + a + c^10 + b
> a + d^10 + b + c;
d^10 + a + b + c
> // Construct polynomial ring with weight order and x > y > z
> P<x, y, z> := PolynomialRing(Z, 3, "weight", [100,10,1, 1,10,100, 1,1,1]);
> MonomialOrder(P);
<"weight", [ 100, 10, 1, 1, 10, 100, 1, 1, 1 ]>
> MonomialOrderWeightVectors(P);
[
    [ 100, 10, 1 ],
    [ 1, 10, 100 ],
    [ 1, 1, 1 ]
]
> x + y + z;
x + y + z
> (x+y^2+z^3)^4;
x^4 + 4*x^3*y^2 + 4*x^3*z^3 + 6*x^2*y^4 + 12*x^2*y^2*z^3 +
    6*x^2*z^6 + 4*x*y^6 + 12*x*y^4*z^3 + 12*x*y^2*z^6 +
    4*x*z^9 + y^8 + 4*y^6*z^3 + 6*y^4*z^6 +
    4*y^2*z^9 + z^12
```

105.3.2 Creation of Graded Polynomial Rings

It is possible within MAGMA to assign weights to the variables of a multivariate polynomial ring. This means that monomials of the ring then have a *weighted degree* with respect to the weights of the variables. Such a multivariate polynomial ring is called *graded* or *weighted*. A polynomial of the ring whose monomials all have the same weighted degree is called *homogeneous*. The polynomial ring can be decomposed as the direct sum of graded homogeneous components.

Suppose a polynomial ring P has n variables $x_1, \ldots, x_n$ and the weights for the variables are $d_1, \ldots, d_n$ respectively. Then for a monomial $m = x_1^{e_1} \cdots x_n^{e_n}$ of P (with $e_i \geq 0$ for $1 \leq i \leq n$), the *weighted degree* of m is defined to be $\sum_{i=1}^{n} e_i d_i$.

A polynomial ring created without a specific weighting (using the default version of the PolynomialRing function or similar) has weight 1 for each variable so the weighted degree coincides with the total degree.

The following functions allow one to create and operate on elements of polynomial rings with specific weights for the variables.

```
PolynomialRing(R, Q)
```

```
PolynomialAlgebra(R, Q)
```

Given a ring R and a non-empty sequence Q of positive integers, create a multivariate polynomial ring in $n = \#Q$ variables over the ring R with the weighted degree of the i-th variable set to be $Q[i]$ for each i. The rank n of the polynomial is determined by the length of the sequence Q. (The angle bracket notation can be used to assign names to the variables, just like in the usual invocation of the `PolynomialRing` function.)

As of V2.15, the default monomial order chosen is the `grevlexw` order with weights given by Q, since the Gröbner basis of an ideal w.r.t. this order tends to be smaller if the ideal is homogeneous w.r.t. the grading.

```
Grading(P)
```

```
VariableWeights(P)
```

Given a graded polynomial ring P (or an ideal thereof), return the variable weights of P as a sequence of n integers where n is the rank of P. If P was constructed without specific weights, the sequence containing n copies of the integer 1 is returned.

105.3.3 Element Operations Using the Grading

```
Degree(f)
```

```
WeightedDegree(f)
```

Given a polynomial f of the graded polynomial ring P, this function returns the weighted degree of f, which is the maximum of the weighted degrees of all monomials that occur in f. The weighted degree of a monomial m depends on the weights assigned to the variables of the polynomial ring P — see the introduction of this section for details. Note that this is different from the natural total degree of f which ignores any weights.

```
LeadingWeightedDegree(f)
```

Given a polynomial f of the graded polynomial ring P, this function returns the leading weighted degree of f, which is the weighted degree of the leading monomial of f. The weighted degree of a monomial m depends on the weights assigned to the variables of the polynomial ring P — see the introduction of this section for details.

`IsHomogeneous(f)`

Given a polynomial f of the graded polynomial ring P, this function returns whether f is homogeneous with respect to the weights on the variables of P (i.e., whether the weighted degrees of the monomials of f are all equal).

`HomogeneousComponent(f, d)`

Given a polynomial f of the graded polynomial ring P, this function returns the *weighted* degree-d homogeneous component of f which is the sum of all the terms of f whose monomials have weighted degree d. d must be greater than or equal to 0. If f has no terms of weighted degree d, then the result is 0.

`HomogeneousComponents(f)`

Given a polynomial f of the graded polynomial ring P, this function returns the *weighted* degree-d homogeneous component of f which is the sum of all the terms of f whose monomials have weighted degree d. d must be greater than or equal to 0. If f has no terms of weighted degree d, then the result is 0.

`MonomialsOfDegree(P, d)`

Given a polynomial ring P and a non-negative integer d, return an indexed set consisting of all monomials in P with total degree d. If P is graded, the grading is ignored.

`MonomialsOfWeightedDegree(P, d)`

Given a graded polynomial ring P and a non-negative integer d, return an indexed set consisting of all monomials in P with weighted degree d. If P has the trivial grading, then this function is equivalent to the function `MonomialsOfDegree`.

Example H105E2

We create a simple graded polynomial ring and perform various simple operations on it.

```
> P<x, y, z> := PolynomialRing(RationalField(), [1, 2, 4]);
> P;
Graded Polynomial ring of rank 3 over Rational Field
Order: Grevlex with weights [1, 2, 4]
Variables: x, y, z
Variable weights: [1, 2, 4]
> VariableWeights(P);
[ 1, 2, 4 ]
> Degree(x);
1
> Degree(y);
2
> Degree(z);
4
> Degree(x^2*y*z^3); // Weighted total degree
```

```
16
> TotalDegree(x^2*y*z^3); // Natural total degree
6
> IsHomogeneous(x);
true
> IsHomogeneous(x + y);
false
> IsHomogeneous(x^2 + y);
true
> I := ideal<P | x^2*y + z, (x^4 + z)^2, y^2 + z>;
> IsHomogeneous(I);
true
> MonomialsOfDegree(P, 4);
{@
    x^4,
    x^3*y,
    x^3*z,
    x^2*y^2,
    x^2*y*z,
    x^2*z^2,
    x*y^3,
    x*y^2*z,
    x*y*z^2,
    x*z^3,
    y^4,
    y^3*z,
    y^2*z^2,
    y*z^3,
    z^4
@}
> MonomialsOfWeightedDegree(P, 4);
{@
    x^4,
    x^2*y,
    y^2,
    z
@}
```

105.3.4 Creation of Ideals and Accessing their Bases

Within the general context of ideals of polynomial rings, the term "basis" will refer to an *ordered* sequence of polynomials which generate an ideal. (Thus a basis can contain duplicates and zero elements so is not like a basis of a vector space.)

One normally creates an ideal by the `ideal` constructor or `Ideal` function, described below. But it is also possible to create an ideal with a specific basis U and then find the coordinates of polynomials from the polynomial ring with respect to U (see the function `Coordinates` below). This is done by specifying a *fixed basis* with the `IdealWithFixedBasis` intrinsic function. In this case, when MAGMA computes the Gröbner basis of the ideal (see below), extra information is stored so that polynomials of the ideal can be rewritten in terms of the original fixed basis. However, the use of this feature makes the Gröbner basis computation much more expensive so an ideal should usually **not** be created with a fixed basis.

`ideal< P | L >`

Given a multivariate polynomial ring P, return the ideal of P generated by the elements of P specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of P;

(b) A set or sequence of elements of P;

(c) An ideal of P;

(d) A set or sequence of ideals of P.

`Ideal(B)`

Given a set or sequence B of polynomials from a polynomial ring P, return the ideal of P generated by the elements of B with the given basis B. This is equivalent to the above `ideal` constructor, but is more convenient when one simply has a set or sequence of polynomials.

`Ideal(f)`

Given a polynomial f from a polynomial ring P, return the principal ideal of P generated by f.

`IdealWithFixedBasis(B)`

Given a sequence B of polynomials from a polynomial ring P, return the ideal of P generated by the elements of B with the given fixed basis B. When the function `Coordinates` is called, its result will be with respect to the entries of B instead of the Gröbner basis of I. **WARNING:** this function should *only* be used when it is desired to express polynomials of the ideal in terms of the elements of B, as the computation of the Gröbner basis in this case is *very* expensive, so it should be avoided if these expressions are not wanted.

`Basis(I)`

Given an ideal I, return the current basis of I. If I has a fixed basis, that is returned; otherwise the current basis of I (whether it has been converted to a Gröbner basis or not – see below) is returned.

`BasisElement(I, i)`

Given an ideal I together with an integer i, return the i-th element of the current basis of I. This the same as `Basis(I)[i]`.

105.4 Gröbner Bases

Computation in ideals of multivariate polynomial rings is possible because of the construction of Gröbner bases of such ideals. In MAGMA, it is possible to create ideals and compute their Gröbner bases for polynomial rings defined not only over fields but also over general Euclidean rings.

Different monomial orderings give different Gröbner bases for a fixed ideal. When an ideal I is created from a polynomial ring P or another ideal J, then the monomial order of I is taken to be the monomial order of P or J. Ideals can only be compatible if they have the same monomial order.

105.4.1 Gröbner Bases over Fields

Gröbner bases of ideals defined over fields have been studied for some time now, and there is a large literature concerning them.

For ideals defined over fields, a basis is called *minimal* if each polynomial in it is monic and not contained in the ideal generated by all the other polynomials [CLO96, Chap. 2, §7, Def. 4]. A basis is called *reduced* if each polynomial in it is monic and, for every monomial of each polynomial in the basis, that monomial is not divisible by the leading monomial of any other polynomial in the basis (equivalently, each leading monomial does not divide any monomial in any of the other polynomials) [CLO96, Chap. 2, §7, Def. 5].

For a given fixed monomial ordering, every ideal of a polynomial ring over a field possesses a **unique** sorted minimal reduced Gröbner basis (GB) [CLO96, Chap. 2, §7, Prop. 7]. This unique Gröbner basis (with respect to the order defined by the user) will be computed automatically when needed by MAGMA. Before this happens, an ideal will usually possess a basis which is not a Gröbner basis, but that will be changed into the unique Gröbner basis when needed. Thus the original basis will be discarded. See the procedure `Groebner` below for details on the algorithms available.

105.4.2 Gröbner Bases over Euclidean Rings

Since V2.8 (July 2001), MAGMA provides facilities for computing with Gröbner bases of ideals of polynomial rings over Euclidean rings (including the important case of the integer ring $\mathbf{Z}$). Such Gröbner bases are computed in MAGMA by an extension, due to Allan Steel (unpublished), of Jean-Charles Faugère's F_4 algorithm [Fau99], which uses sparse linear algebra.

The current Euclidean rings in MAGMA supported are: the integer ring $\mathbf{Z}$, the integer residue class rings $\mathbf{Z}_m$, the univariate polynomial rings $K[x]$ over any field K, Galois rings, p-adic quotient rings, and valuation rings.

We first outline some of the things which are peculiar to Gröbner bases defined over a Euclidean ring. Let I be an ideal of a polynomial ring defined over a Euclidean ring R. A subset G of I is called a *Gröbner basis* for I in MAGMA if, for every $f \in I$, there exists a $g \in G$ such that the leading **term** of g divides the leading **term** of f. Recall that "leading term" here means the leading coefficient times the leading monomial, so the leading coefficient of g must divide the leading coefficient of f in the base ring R. If R were a field, then obviously the leading coefficients would be insignificant and the Gröbner basis elements could be normalized (made monic) to yield an equivalent Gröbner basis. But if R is not a field, the leading coefficients are quite significant. For example, over the ring $\mathbf{Z}$, the set $\{x^2, 2x\}$ is a Gröbner basis and the polynomial x^2 is not redundant since 2 does not divide 1, but over $\mathbf{Q}$, the polynomial x^2 would be redundant.

Note that the definition here for a Gröbner basis in MAGMA is actually what some authors (e.g., [AL94, Def. 4.5.6]) call a *strong* Gröbner basis. *Weak* Gröbner bases have also been defined, but strong Gröbner bases satisfy stronger conditions, yield a simple effective normal form algorithm, provide more information about the ideal, are easier to get into a unique form, and are no more difficult to compute using the algorithm implemented in MAGMA. Thus MAGMA **always** computes a strong Gröbner basis, so the distinction between weak and strong is ignored. MAGMA also effectively computes a D-Gröbner basis as defined in [BW93, Def. 10.4, Table 10.1], although MAGMA also allows Euclidean rings which are not integral domains (i.e., which have zero divisors).

Over Euclidean rings, the definition of a minimal basis is practically the same as for fields (there must be no polynomial in the ideal generated by the others and each polynomial must be normalized), but the definition of a reduced basis is more subtle. A basis is called *reduced* if each polynomial in it is normalized and if, for every term $c \cdot s$ of every polynomial in the basis (where c is the coefficient and s is the monomial), then if some other polynomial in the basis has leading term $d \cdot t$, with t dividing s, then the Euclidean quotient of c by d must be zero (the remainder will be non-zero of course). Informally, this means that each polynomial is reduced modulo all the other polynomials, where each coefficient must be reduced modulo all other appropriate leading coefficients. As an example, suppose $f_1 = x^2 + 14xy$ and $f_2 = 5y + 9$ are in $\mathbf{Z}[x, y]$. Then $\{f_1, f_2\}$ is not reduced, since the second term of f_1 can be reduced by f_2 (y divides xy and the Euclidean quotient of 14 by 5 is 2, with remainder 4). But if we were to replace f_1 by $f_1 - 2xf_2 = x^2 + 4xy - 18x$, then $\{f_1, f_2\}$ would now be reduced.

MAGMA's extension of Faugère's algorithm depends on sparse linear algebra over Euclidean rings. (Note also that the advanced criteria for eliminating useless pairs in [Möl88]) are also implemented in this extension to work for general Euclidean rings as well.) MAGMA now contains an algorithm for computing a unique echelon form of a sparse matrix over such a ring; uniqueness is ensured because there is a unique Euclidean quotient-remainder algorithm for each Euclidean ring (and zero divisors are also handled properly). Consequently, based on this unique echelon form algorithm and some other techniques, MAGMA ensures that a Gröbner basis over a Euclidean ring is not only minimal (contains no re-

dundant polynomials), but it is also **reduced**, and **unique**.

Thus every ideal of a polynomial ring over a Euclidean ring possesses a **unique** sorted minimal reduced Gröbner basis (with respect to some fixed monomial ordering), just as for ideals defined over fields. Also, as for ideals defined over fields, this unique Gröbner basis will be computed automatically when needed by MAGMA, and before this happens, an ideal will usually possess a basis which is not a Gröbner basis, but that will be changed into the unique Gröbner basis when needed.

The uniqueness of the Gröbner basis also ensures that the normal form of an element with respect to an ideal for a fixed monomial order is always unique. All of this holds even for Euclidean rings which have zero divisors.

See the examples below for illustrations of the points made above, and also how one can effectively compute with Gröbner bases of ideals defined over rings which are not even Euclidean.

105.4.3 Construction of Gröbner Bases

The following functions and procedures allow one to construct Gröbner bases. Note that a Gröbner basis for an ideal will be automatically generated when necessary; the `Groebner` procedure below simply allows control of the algorithms used to compute the Gröbner basis.

NOTE: MAGMA applies a special monomial representation and a special variant of the F_4 algorithm if the ideal I is defined over $\mathbf{F}_2$ and the polynomials ${x_i}^2 + x_i$ for all i are present in the input basis of the ideal I. So if one wishes to solve a system of equations over $\mathbf{F}_2$, then one should include these polynomials in the input basis (they can be at any place and in any order; as long as there is at least one copy of ${x_i}^2 + x_i$ present for each i). Alternatively (since V2.15), one can create a boolean polynomial ring (via the function `BooleanPolynomialRing` below) and construct the ideal within this. See also Example H105E5 below.

`Groebner(I: parameters)`

(Procedure.) Explicitly force a Gröbner basis (GB) for the ideal I to be constructed. This procedure is normally not necessary, as MAGMA will automatically compute the GB when needed, but it does allow one to control how the GB is constructed by various parameters.

By default, the parameters are set to default values which tend to work best for the particular kinds of inputs which are given, but there exist many inputs for which setting at least one of the parameters to a non-default value will lead to a dramatic improvement. (A general strategy for the computation of GBs is very difficult to design.)

If I is defined over a Euclidean ring, then MAGMA always uses the extension of the Faugère algorithm directly, and of the parameters given below, only `Homogenize` is applicable. So the rest of this description assumes that I is defined over a field.

We call a GB algorithm **direct** if it takes the initial basis of the ideal I (with no structure) and computes the unique minimal reduced GB of I with respect to

some monomial order. Since V2.11 (May 2004), MAGMA has two direct algorithms for computing GBs over fields:

(1) The Faugère F_4 algorithm [Fau99], which works by specialized sparse linear algebra and is applicable to ideals defined over a finite field or the rational field;

(2) The Buchberger algorithm [CLO96, Chap. 2, §7] for ideals defined over any field.

Both direct algorithms use the advanced criteria for eliminating useless pairs in [Möl88]. MAGMA also uses two **order change** algorithms which both change the GB of an ideal with respect to one monomial order to the GB with respect to another monomial order:

(1) The FGLM algorithm [FGLM93], which works by efficient linear algebra and is only applicable if I is zero-dimensional;

(2) The Gröbner Walk algorithm [CKM97].

This parameter affects the main strategy:

`Al`	MonStgElt	*Default :* "*Default*"

The parameter `Al` may be set to one of: `"Default"`, `"Direct"`, `"FGLM"` or `"Walk"`. The value `"Direct"` specifies that MAGMA should compute the GB of I (with respect to the order of I) by a direct algorithm alone, so that an order-conversion algorithm is not used (the parameter `Faugere` below controls which direct algorithm is used).

The alternative strategy is to compute the GB first with respect to an "easy" order, and then to convert this to the GB with respect to the order of I. Setting `Al` to the values `"FGLM"` or `"Walk"` will cause this strategy to be used, where the order change algorithm will be the FGLM algorithm or Gröbner Walk algorithm, respectively.

If no algorithm is specified, or if `"Default"` is specified, an appropriate strategy is chosen by MAGMA, which is usually the FGLM method if the ideal is zero-dimensional and over a finite field or the rational field, and the Walk method otherwise.

The following parameters affect the direct algorithms:

`Faugere`	BoolElt	*Default :* `true`
`HomogeneousWeights`	BoolElt	*Default :* `true`
`Homogenize`	BoolElt	*Default :* `true`
`DegreeStart`	RngIntElt	*Default :* `true`

If the parameter `Faugere` is set to `true`, then the Faugère F_4 algorithm will be used (if the field is a finite field or the rational field); otherwise the Buchberger algorithm is used.

The current implementation of the Faugère algorithm is usually very much faster than the Buchberger algorithm and usually does not take much more memory, so that it is why it is now selected by default. However, there may be examples for

which it may be more desirable to use the Buchberger algorithm (particularly to save some memory).†

Since V2.12, if the input basis is not homogeneous, then MAGMA first attempts to find a weight vector W with respect to which the ideal is homogeneous; if such a W is found, then the "easy" order used internally for the direct algorithm (accessed by `EasyIdeal`) is taken to be the `grevlexw` order with respect to W (see subsection 105.2.4), since the GB is likely to be smaller with respect to this order. The selection of such an order may be suppressed by setting the parameter `HomogeneousWeights` to `false`.

If no appropriate `grevlexw` order is used, then setting `Homogenization` to `true` specifies that the ideal should first be *homogenized*: a GB of the homogenization of the ideal is computed and then the homogenization variable is removed and the final basis reduced. This parameter has the default value of `true` over the rational field and `false` over all other fields, since most computations are improved by these defaults.

If the parameter `DegreeStart` is set to an integer d, then any S-polynomial pairs of degree less than d will be ignored.

The following parameters affect the Faugère F_4 algorithm:

`AllPairs`	BOOLELT	*Default :* `false`
`PairsLimit`	BOOLELT	*Default :* 0
`ReversePairs`	BOOLELT	*Default :* `false`
`HFE`	BOOLELT	*Default :* `false`
`Boolean`	BOOLELT	*Default :* `false`
`Nthreads`	RNGINTELT	*Default :* 1

By default, the Faugère F_4 algorithm includes all pairs of the next degree at each step (see [Fau99, Sec.2.5]), since this usually produces the best performance. However, setting the parameter `AllPairs` to `true` will cause the algorithm to include *all* pairs currently in the queue at each new step; this generally makes the matrix larger and is usually less efficient, but for some inputs (e.g., inhomogeneous ideals where there are only a small number of pairs for each degree at each step) this option may yield a significant improvement.

Alternatively, setting the parameter `PairsLimit` to a positive integer n will cause the algorithm to include at most n pairs from the queue at each step; this will usually make the matrix smaller, thus saving memory, but will often also make the running time longer. Setting also the parameter `ReversePairs` to `true` will reverse the list of pairs of the current degree from which the restricted set of pairs is taken: this may help a lot for certain types of input, since this may lead to new polynomials of lower degree being found more quickly. (If there is no pairs limit, then the value

† If you encounter an example where the Faugère algorithm is significantly slower than the Buchberger algorithm, then please mail it to us (`magma@maths.usyd.edu.au`)!

of ReversePairs is irrelevant since all pairs of the current degree are taken at each step.)

If the input basis is an HFE system over $\mathbf{F}_2$ such that the secret degree d is less than or equal to 127, then one should set the HFE parameter to true. In this case, MAGMA can apply various optimizations which save memory and time (only pairs of degree of most 4 are considered, as this is sufficient for systems for which $d \leq 127$).

Since V2.18, if the base ring is the finite field $\mathbf{F}_p$, where p is a prime with $2 < p < 2^{23.5}$, then a multi-threaded version of the algorithm is available if POSIX threads are enabled in the current Magma version. In this case, setting the parameter Nthreads to a positive integer n will cause the F_4 algorithm to use n threads within the linear algebra phase of each step. One can alternatively use the procedure SetNthreads to set the global number of threads to a value n so that n threads are always used by default in this algorithm (unless overridden by the Nthreads parameter).

The following parameters affect the Buchberger algorithm:

ReduceInitial	BOOLELT	*Default :* true
RemoveRedundant	BOOLELT	*Default :* true
ReduceByNew	BOOLELT	*Default :* true

Setting ReduceInitial to true specifies that the basis of the ideal should be first reduced (see the function Reduce) before any S-polynomial pairs are considered. Setting RemoveRedundant to true specifies that redundant polynomials in the input (which reduce to zero with respect to the other polynomials) should first be removed. Setting ReduceByNew to true specifies that when a new polynomial f is inserted into the current GB being constructed, the current basis should be reduced by f (thus the basis stays close to being fully reduced throughout the algorithm).

Each of these control parameters usually have the default values of true (it depends on the coefficient ring).

The following parameters affect the Walk algorithm:

SigmaEpsilon	FLDRATELT	*Default :* $1/2$
TauEpsilon	FLDRATELT	*Default :* $1/n$
SigmaVectors	RNGINTELT	*Default :* n
TauVectors	RNGINTELT	*Default :* $\lceil n/2 \rceil$

The parameters SigmaEpsilon and TauEpsilon control the factor ϵ which is used in the Walk algorithm to perturb the initial weight vector σ and the final weight vector τ respectively. The parameters SigmaVectors and TauVectors determine how many weight vectors of the initial and final orders are used to perturb the initial weight vector σ and the final weight vector τ respectively. By default, the ϵ factor and number of weight vectors for σ are determined dynamically to be "optimal", while the ϵ factor for τ is taken to be $1/n$ and the number of weight vectors for τ is taken to be $\lceil n/2 \rceil$, where n is the rank of I.

`GroebnerBasis(I: parameters)`

Given an ideal I, force the Gröbner basis of I to be computed, and then return that. The parameters are the same as those for the procedure `Groebner`.

See also the function `GroebnerBasis(S,d)` below, which creates a truncated degree-d Gröbner basis.

`GroebnerBasis(S: parameters)`

Given a set or sequence S of polynomials, return the unique Gröbner basis of the ideal generated by S as a sorted sequence. This function is useful for computing Gröbner bases without the need to construct ideals. The parameters are the same as those for the procedure `Groebner`.

See also the function `GroebnerBasis(S,d)` below, which creates a truncated degree-d Gröbner basis.

`GroebnerBasisUnreduced(S: parameters)`

`Homogenize`	BoolElt	*Default :* `true`
`ReduceInitial`	BoolElt	*Default :* `true`
`ReduceByNew`	BoolElt	*Default :* `true`

Given a set or sequence S of polynomials, return an unreduced Gröbner basis of the ideal generated by S as a sorted sequence. This function is useful for computing Gröbner bases without the need to construct ideals and when the reduction of the Gröbner basis is very expensive. The parameters behave the same as for the procedure `Groebner`.

`GroebnerBasis(S, d: parameters)`

Given a set or sequence S of polynomials, return the degree-d Gröbner basis of the ideal generated by S, which is the truncated Gröbner basis obtained by ignoring S-polynomial pairs whose total degree is greater than d.

If the ideal is homogeneous, then it is guaranteed that the result G_d is equal to the set of all polynomials in the full Gröbner basis of the ideal whose total degree is less than or equal to d, and thus a polynomial whose total degree is less than or equal to d is in the ideal iff its normal form with respect to the degree-d Gröbner basis G_d is zero. But if the ideal is not homogeneous, these last properties may not hold, but it may be still useful to construct the truncated basis.

The parameters are the same as those for the procedure `Groebner`. See the section on graded polynomial rings below for an example. See also [BW93, section 10.2], for further discussion.

105.4.4 Related Functions

The following functions and procedures perform operations related to Gröbner bases.

`HasGroebnerBasis(I)`

Given an ideal I, return whether the Gröbner basis of I can be computed. This depends on the type of base ring of I: the base ring must currently be a field or a Euclidean ring.

`EasyIdeal(I)`

Given an ideal I, return the ideal E which is mathematically equal to I but whose basis is the Gröbner basis of I with respect to an "easy" order, together with an isomorphism f from I onto E. The easy order is usually the `grevlex` order or `grevlexw` order with suitable weights, and the easy basis (the Gröbner basis of the easy ideal) of I is used extensively by MAGMA in many of its internal algorithms; this function allows one to access this "easy" Gröbner basis directly.

`EasyBasis(I)`

Given an ideal I, return the Gröebner basis of the easy ideal of I.

`SmallBasis(I)`

Given an ideal I, return the basis of I with shortest length which is currently known. This may be the original basis with which I was constructed, or a Gröbner basis, but the result is always has the the same monomial order as the main monomial order of I.

`MarkGroebner(I)`

(Procedure.) Given an ideal I, mark the current basis of I to be *the* Gröbner basis of the ideal w.r.t. the monomial order of the ideal. Note that the current basis must exactly equal the unique (reverse) sorted minimal reduced Gröbner basis for the ideal, as returned by the function `GroebnerBasis`. This procedure is useful when one creates an ideal with a basis known to be the Gröbner basis of the ideal from a previous computation or for other reasons. If the basis is not the unique Gröbner basis, the results are unpredictable.

`IsGroebner(S)`
`IsGroebner(S)`

Given a set or sequence S of polynomials describing a basis of an ideal, return whether the basis is itself a (not necessarily minimal or reduced) Gröbner basis of the ideal.

`Coordinates(I, f)`

Given an ideal I of a polynomial ring P, together with a polynomial f in I, and supposing that I has basis $b_1, \ldots, b_k$, return a sequence $[g_1, \ldots, g_k]$ of elements of P so that $f = g_1 * b_1 + \ldots + g_k * b_k$. If I was created by `IdealWithFixedBasis`(B), then the fixed basis B is used as the basis $b_1, \ldots, b_k$; otherwise the (unique) Gröbner basis of I is used as the basis $b_1, \ldots, b_k$. The resulting sequence is not necessarily unique.

`CoordinateMatrix(I)`

Given an ideal I such that I has a fixed basis (i.e., such that I was created via the function `IdealWithFixedBasis`), return the coordinate matrix C of I. The i-th row of C gives the coordinates of the i-th element of the Gröebner basis of I w.r.t. the fixed basis of I. The Gröebner basis of I is first computed if it has not been already.

`NormalForm(f, I)`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return the unique normal form of f with respect to (the Gröbner basis of) I. The normal form of f is zero if and only if f is in I.

`NormalForm(f, S)`

Given a polynomial f from a polynomial ring P, together with a set or sequence S of polynomials from P, return a normal form g of f with respect to S. (This is not unique in general. If the normal form of f is zero then f is in the ideal generated by S, but the converse is false in general. In fact, the normal form is unique if and only if S forms a Gröbner basis.) If S is a sequence, one may also assign a second return value C which gives the coordinates of the reduction, so that $C[i] \cdot S[i]$ is subtracted from f for each i to yield g.

`SPolynomial(f, g)`

Given elements f and g from a polynomial ring P, return the S-polynomial of f and g.

`Reduce(S)`

Given a set or sequence S of polynomials, return the sequence consisting of the reduction of S. The reduction is obtained by reducing to normal form each element of S with respect to the other elements and sorting the resulting non-zero elements left. Note that all Gröbner bases returned by MAGMA are automatically reduced so that this function would usually only be used just to simplify a set or sequence of polynomials which is not a Gröbner basis.

`ReduceGroebnerBasis(S)`

Given a set or sequence S of polynomials which is assumed to be a (not necessarily minimal or reduced) Gröbner basis for an ideal, return the sequence consisting of the reduction of S. The reduction is obtained by first removing each redundant polynomial whose leading term is a multiple of another leading term and then reducing the remaining polynomials as in the function `Reduce`. This function would usually only be used to reduce a set or sequence of polynomials which is known to be a non-reduced Gröbner basis (created in some way other than by one of MAGMA's internal Gröbner basis construction algorithms).

105.4.5 Gröbner Bases of Boolean Polynomial Rings

Since V2.15, a special type of polynomial ring is available: the **boolean polynomial ring** in n variables. Such a ring is a multivariate polynomial ring defined over $\mathbf{F}_2$ but such that all monomials are reduced modulo the *field relations* $x_i^2 = x_i$ for each i (so a bit vector representation can be used for monomials). Technically, the ring is thus the quotient algebra

$$\mathbf{F}_2[x_1, \ldots, x_n]/\langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle.$$

Besides the basic creation and access functions for elements and ideals of such a ring, the main interest is to compute and examine a Gröbner basis of an ideal. Since the field relations are always present, an ideal represents a zero-dimensional system of multivariate polynomial equations over $\mathbf{F}_2$ with the solution components always lying in $\mathbf{F}_2$; these are particularly of interest for algebraic attacks on cryptosystems. Otherwise, there are not many other operations applicable to such rings and their elements.

Note that if one creates an ideal I of $\mathbf{F}_2[x_1, \ldots, x_n]$ such that the basis of I includes the *field polynomials* ($x_i^2 + x_i$ for each i), then MAGMA automatically uses the boolean polynomial ring representation internally, so this is basically equivalent to using the boolean polynomial ring type, except that MAGMA will have to move back to the original ring $\mathbf{F}_2[x_1, \ldots, x_n]$ at the end, and this may take much more time and memory. So it is preferable to use the boolean polynomial ring from the outset if one wishes to create the Gröbner basis of such an ideal and examine it (particularly if it does not collapse down to a sequence of linear polynomials).

See example H105E5 below for simple uses of boolean polynomial rings.

`BooleanPolynomialRing(n)`

Create the boolean polynomial ring with n variables (whose coefficients lie in $\mathbf{F}_2$). The default monomial order chosen is the lexicographical (`lex`) order.

`BooleanPolynomialRing(n, order)`

Create the boolean polynomial ring with n variables (whose coefficients lie in $\mathbf{F}_2$) and with the given order *order* on the monomials. Currently, *order* must be one of the following strings: `"lex"`, `"grevlex"`, `"glex"`.

`BooleanPolynomialRing(B, Q)`

Given a boolean polynomial ring B of rank n and a sequence Q of integers, create the boolean polynomial in B whose monomials are given by the entries of Q: each integer must be in the range $[0 \dots 2^n-1]$ and its binary expansion gives the exponents of the monomial in order (the resulting monomials are sorted w.r.t. the monomial order of B, so may be given in any order and duplicate monomials are added).

This function is simply provided so that boolean polynomials may be stored and read back in a compact form; otherwise, one can create a boolean polynomial in the usual way from the generators of B after B is created. Note also that if one prints B, an ideal of B, or an element of B with the `Magma` print level, then this function will be used to print the elements in a compact form.

105.4.6 Verbosity

This subsection describes the verbose flags available for the Gröbner basis algorithms. There are separate verbose flags for each algorithm (`Buchberger`, etc.), but the all-encompassing verbose flag `Groebner` includes all these flags implicitly.

For each procedure provided for setting one of these flags, the value `false` is equivalent to level 0 (nothing), and `true` is equivalent to level 1 (minimal verbosity). For each `Set-` procedure, there is also a corresponding `Get-` function to return the value of the corresponding flag.

`SetVerbose("Groebner", v)`

(Procedure.) Change the verbose printing level for all Gröbner basis algorithms to be v. This includes all of the algorithms whose verbosity is controlled by flags subsequently listed, as well as some other minor related algorithms. Currently the legal levels are 0, 1, 2, 3, or 4. One would normally set this flag to 1 for minimal verbosity for Gröbner basis-type computations, and possibly also set one or more of the following flags to levels higher than 1 for more verbosity.

`SetVerbose("Buchberger", v)`

(Procedure.) Change the verbose printing level for the Buchberger algorithm to be v. Currently the legal levels are 0, 1, 2, 3, or 4. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

`SetVerbose("Faugere", v)`

(Procedure.) Change the verbose printing level for the Faugère algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

`SetVerbose("FGLM", v)`

(Procedure.) Change the verbose printing level for the FGLM order change algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

`SetVerbose("GroebnerWalk", v)`

(Procedure.) Change verbose printing for the Gröbner Walk order change algorithm to be v. Currently the legal levels are 0, 1, 2, or 3. If the value w of the `Groebner` verbose flag is greater than v, then w is taken to be the current value of this flag.

Example H105E3

We compute the Gröbner basis of the "Cyclic-6" ideal with respect to the lexicographical order. The ideal is an ideal of the polynomial ring $\mathbf{Q}(x, y, z, t, u, v)$. We also note that the last polynomial in the Gröbner basis is univariate (since, in fact, the ideal is zero-dimensional and the monomial order is lexicographical) and observe that it has a nice factorization. Note especially that in this example, homogenizing at first and keeping the Gröbner basis reduced makes this computation very fast; without using these features (i.e., if the parameters `Homogenize := false` or `ReduceByNew := false` are given), the computation is much more expensive (takes hundreds of seconds on the same computer).

```
> Q := RationalField();
> P<x, y, z, t, u, v> := PolynomialRing(Q, 6);
> I := ideal<P |
>     x + y + z + t + u + v,
>     x*y + y*z + z*t + t*u + u*v + v*x,
>     x*y*z + y*z*t + z*t*u + t*u*v + u*v*x + v*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*v + t*u*v*x + u*v*x*y + v*x*y*z,
>     x*y*z*t*u + y*z*t*u*v + z*t*u*v*x + t*u*v*x*y + u*v*x*y*z + v*x*y*z*t,
>     x*y*z*t*u*v - 1>;
> time B := GroebnerBasis(I);
Time: 1.140
> #B;
17
> B[17];
v^48 - 2554*v^42 - 399710*v^36 - 499722*v^30 + 499722*v^18 + 399710*v^12 +
    2554*v^6 - 1
> time Factorization(B[17]);
[
    <v - 1, 1>,
    <v + 1, 1>,
    <v^2 + 1, 1>,
    <v^2 - 4*v + 1, 1>,
    <v^2 - v + 1, 1>,
    <v^2 + v + 1, 1>,
    <v^2 + 4*v + 1, 1>,
    <v^4 - v^2 + 1, 1>,
    <v^4 - 4*v^3 + 15*v^2 - 4*v + 1, 1>,
    <v^4 + 4*v^3 + 15*v^2 + 4*v + 1, 1>,
    <v^8 + 4*v^6 - 6*v^4 + 4*v^2 + 1, 1>,
    <v^8 - 6*v^7 + 16*v^6 - 24*v^5 + 27*v^4 - 24*v^3 +
        16*v^2 - 6*v + 1, 1>,
    <v^8 + 6*v^7 + 16*v^6 + 24*v^5 + 27*v^4 + 24*v^3 +
```

```
        16*v^2 + 6*v + 1, 1>
]
Time: 0.060
```

Example H105E4

We solve the system of equations Runge-Kutta 2 from the paper "Some Examples for Solving Systems of Algebraic Equations by Calculating Groebner Bases" by Boege, Gebauer, and Kredel (J. Symbolic Computation (1986) **1**, 83–98). The coefficient field K is the rational function field $\mathbf{Q}(c2, c3)$, and the polynomial ring $K[c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43]$ has 11 variables with the lexicographical ordering on monomials. The resulting Gröbner basis contains a linear polynomial for each variable so there is exactly one solution to the system.

```
> K<c2, c3> := FunctionField(IntegerRing(), 2);
> P<c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43> := PolynomialRing(K, 11);
> I := ideal<P |
>    b1 + b2 + b3 + b4 - 1,
>    b2*c2 + b3*c3 + b4*c4 - 1/2,
>    b2*c2^2 + b3*c3^2 + b4*c4^2 - 1/3,
>    b3*a32*c2 + b4*a42*c2 + b4*a43*c3 - 1/6,
>    b2*c2^3 + b3*c3^3 + b4*c4^3 - 1/4,
>    b3*c3*a32*c2 + b4*c4*a42*c2 + b4*c4*a43*c3 - 1/8,
>    b3*a32*c2^2 + b4*a42*c2^2 + b4*a43*c3^2 - 1/12,
>    b4*a43*a32*c2 - 1/24,
>    c2 - a21,
>    c3 - a31 - a32,
>    c4 - a41 - a42 - a43>;
> time Groebner(I);
Time: 0.110
> I;
Ideal of Polynomial ring of rank 11 over Multivariate rational function field
    of rank 2 over Integer Ring
Order: Lexicographical
Variables: c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43
Inhomogeneous, Dimension 0
Groebner basis:
[
    c4 - 1,
    b4 + (-6*c2*c3 + 4*c2 + 4*c3 - 3)/(12*c2*c3 - 12*c2 - 12*c3 + 12),
    b3 + (2*c2 - 1)/(12*c2*c3^2 - 12*c2*c3 - 12*c3^3 + 12*c3^2),
    b2 + (-2*c3 + 1)/(12*c2^3 - 12*c2^2*c3 - 12*c2^2 + 12*c2*c3),
    b1 + (-6*c2*c3 + 2*c2 + 2*c3 - 1)/(12*c2*c3),
    a21 - c2,
    a31 + (-4*c2^2*c3 + 3*c2*c3 - c3^2)/(4*c2^2 - 2*c2),
    a32 + (-c2*c3 + c3^2)/(4*c2^2 - 2*c2),
    a41 + (-12*c2^2*c3^2 + 12*c2^2*c3 - 4*c2^2 + 12*c2*c3^2 - 15*c2*c3 + 6*c2 -
        4*c3^2 + 5*c3 - 2)/(12*c2^2*c3^2 - 8*c2^2*c3 - 8*c2*c3^2 + 6*c2*c3),
    a42 + (-c2^2 + 4*c2*c3^2 - 5*c2*c3 + 3*c2 - 4*c3^2 + 5*c3 - 2)/(12*c2^3*c3 -
```

```
        8*c2^3 - 12*c2^2*c3^2 + 6*c2^2 + 8*c2*c3^2 - 6*c2*c3),
    a43 + (-2*c2^2*c3 + 2*c2^2 + 3*c2*c3 - 3*c2 - c3 + 1)/(6*c2^2*c3^2 -
        4*c2^2*c3 - 6*c2*c3^3 + 3*c2*c3 + 4*c3^3 - 3*c3^2)
]
```

Example H105E5

We demonstrate how one can solve a system of multivariate equations over $\mathbf{F}_2$. We construct a sequence B of 4 polynomials in 5 variables, and note that the Gröbner basis of B contains monomials having degrees greater than 1.

```
> P<a,b,c,d,e> := PolynomialRing(GF(2), 5);
> B := [a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1];
> GroebnerBasis(B);
[
    a + c^2*d + c + d^2*e,
    b*c + d^3*e^2 + d^3*e + d^2*e^2 + d*e + e + 1,
    b*e + d*e^2 + d*e + e,
    c*e + d^3*e^2 + d^3*e + d^2*e^2 + d*e,
    d^4*e^2 + d^4*e + d^3*e + d^2*e^2 + d^2*e + d*e + e
]
```

If one wanted to consider solutions over an algebraic closure of $\mathbf{F}_2$, then one would have to work with this ideal. But to solve over $\mathbf{F}_2$ itself, one can add the *field polynomials* $a^2 + a$, $b^2 + b$, etc. MAGMA recognizes these extra polynomials and uses an optimized representation; this makes the computation much faster for larger examples. The resulting polynomials (besides any remaining field polynomials) will always have degree at most 1 in each variable. In this example, we see that there are 2 solutions over $\mathbf{F}_2$ for the system.

```
> L := [P.i^2 + P.i: i in [1 .. Rank(P)]];
> BB := B cat L;
> BB;
[
    a*b + c*d + 1,
    a*c*e + d*e,
    a*b*e + c*e,
    b*c + c*d*e + 1,
    a^2 + a,
    b^2 + b,
    c^2 + c,
    d^2 + d,
    e^2 + e
]
> GroebnerBasis(BB);
[
    a + d + 1,
    b + 1,
    c + 1,
    d^2 + d,
```

```
    e
]
> I := ideal<P|BB>;
> Variety(I);
[ <0, 1, 1, 1, 0>, <1, 1, 1, 0, 0> ]
```

Since V2.15, an alternative way to solve the system over $\mathbf{F}_2$ is to use the boolean polynomial ring type as follows.

```
> P<a,b,c,d,e> := BooleanPolynomialRing(5, "grevlex");
> B := [a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1];
> I := Ideal(B);
> I;
Ideal of Boolean polynomial ring of rank 5 over GF(2)
Order: Graded Reverse Lexicographical (bit vector word)
Variables: a, b, c, d, e
Basis:
[
    a*b + c*d + 1,
    a*c*e + d*e,
    a*b*e + c*e,
    c*d*e + b*c + 1
]
> GroebnerBasis(I);
[
    a + d + 1,
    b + 1,
    c + 1,
    e
]
> Variety(I);
[ <0, 1, 1, 1, 0>, <1, 1, 1, 0, 0> ]
```

In general, if one wishes to solve a system over $\mathbf{F}_2$ from the outset, it is best to use the boolean polynomial ring type so as to save memory (and to avoid internal conversion to and from the bit vector representation for monomials). Note also that because of the implicit field relations, the Gröbner basis of an ideal generated by only one polynomial may have several polynomials. In the following example, the Gröbner basis of an ideal generated by just one polynomial has linear polynomials alone.

```
> R<[x]> := BooleanPolynomialRing(10, "grevlex");
> R;
Boolean polynomial ring of rank 10 over GF(2)
Order: Graded Reverse Lexicographical (bit vector word)
Variables: x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9], x[10]
> f := x[2]*x[3]*x[5]*x[7] + x[2]*x[4]*x[5]*x[8] + x[3]*x[4]*x[5]*x[9] +
>    x[3]*x[6]*x[7]*x[9] + x[2]*x[3]*x[5] + x[2]*x[4]*x[5] + x[2]*x[3]*x[7] +
>    x[2]*x[5]*x[7] + x[3]*x[5]*x[7] + x[3]*x[6]*x[7] + x[2]*x[4]*x[8] +
>    x[2]*x[5]*x[8] + x[4]*x[5]*x[8] + x[3]*x[6]*x[9] + x[3]*x[7]*x[9] +
>    x[6]*x[7]*x[9] + x[2]*x[3] + x[2]*x[4] + x[3]*x[5] + x[4]*x[5] +
```

```
>    x[3]*x[6] + x[2]*x[7] + x[5]*x[7] + x[6]*x[7] + x[2]*x[8] + x[4]*x[8] +
>    x[5]*x[8] + x[3]*x[9] + x[6]*x[9] + x[7]*x[9] + x[1]*x[10] + x[1] + x[4]
>    + x[6] + x[8] + x[9] + x[10];
> I := Ideal([f]);
> G := GroebnerBasis(I);
> #G;
38
> [Length(f): f in G];
[ 188, 50, 80, 82, 26, 22, 20, 26, 20, 20, 26, 32, 8, 8, 8, 8, 32, 32, 8, 8, 8,
8, 8, 8, 8, 8, 8, 32, 8, 8, 8, 8, 40, 5, 8, 8, 8, 8 ]
> G[38];
x[1]*x[4]*x[7]*x[10] + x[1]*x[5]*x[7]*x[10] + x[1]*x[4]*x[7] + x[1]*x[5]*x[7] +
    x[4]*x[7]*x[10] + x[5]*x[7]*x[10] + x[4]*x[7] + x[5]*x[7]
```

Example H105E6

This simple example illustrates some of the peculiarities of Gröbner bases over Euclidean rings. We first create a simple ideal I in $\mathbf{Z}[x, y, z]$ and compute its Gröbner basis.

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> I := ideal<P| x^2 - 1, y^2 - 1, 2*x*y - z>;
> GroebnerBasis(I);
[
    x^2 - 1,
    x*z - 2*y,
    2*x - y*z,
    y^2 - 1,
    z^2 - 4
]
```

Notice that the Gröbner basis contains polynomials whose leading terms are x^2, xz and $2x$, but the third cannot eliminate the first two since the leading coefficient 2 does not divide the other leading coefficients 1 and 1.

When we compute normal forms modulo I, x is clearly not reducible by any polynomial, while $2x$ can be reduced by the $2x - yz$ polynomial.

```
> NormalForm(x, I);
x
> NormalForm(2*x, I);
y*z
```

If we compute the normal form of $(-x)$ modulo I, then even though the x monomial cannot be reduced, the result is NOT the negative of the normal form of x, since one can use the $2x - yz$ polynomial and the fact that $((-1) \bmod 2)$ is 1 to reduce the polynomial to a unique normal form. This behaviour differs from that for ideals defined over fields, where the normal form of $-f$ will always be the negative of the normal form of f.

```
> NormalForm(-x, I);
```

```
x - y*z
```

If we reduce the Gröbner basis modulo various primes, we obtain familiar Gröbner bases over fields:

```
> GroebnerBasis(ChangeRing(I, GF(2)));
[
    x^2 + 1,
    y^2 + 1,
    z
]
> GroebnerBasis(ChangeRing(I, GF(3)));
[
    x + y*z,
    y^2 + 2,
    z^2 + 2
]
```

But if we reduce modulo 4, using the ring of integers modulo 4, then the Gröbner basis still has a structure not encountered when working over fields:

```
> GroebnerBasis(ChangeRing(I, IntegerRing(4)));
[
    x^2 + 3,
    x*z + 2*y,
    2*x + y*z,
    y^2 + 3,
    z^2,
    2*z
]
```

In fact, the new polynomial $2z$ has been included in this Gröbner basis.

Example H105E7

This example shows how one can use Gröbner bases over the integers to find the primes modulo which a system of equations has a solution, when the system has no solutions over the rationals.

We first form a certain ideal I in $\mathbf{Z}[x, y, z]$, and note that the Gröbner basis of I over $\mathbf{Q}$ contains 1, so there are no solutions over $\mathbf{Q}$ or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> I := ideal<P | x^2 - 3*y, y^3 - x*y, z^3 - x, x^4 - y*z + 1>;
> GroebnerBasis(ChangeRing(I, RationalField()));
[
    1
]
```

However, when we compute the Gröbner basis of I (defined over $\mathbf{Z}$), we note that there is a certain integer in the ideal which is not 1.

```
> GroebnerBasis(I);
```

```
[
    x + 170269749119,
    y + 2149906854,
    z + 170335012540,
    282687803443
]
```

Now for each prime p dividing this integer 282687803443, the Gröbner basis of I modulo p will be non-trivial and will thus give a solution of the original system modulo p.

```
> Factorization(282687803443);
[ <101, 1>, <103, 1>, <27173681, 1> ]
> GroebnerBasis(ChangeRing(I, GF(101)));
[
    x + 19,
    y + 48,
    z + 68
]
> GroebnerBasis(ChangeRing(I, GF(103)));
[
    x + 39,
    y + 8,
    z + 85
]
> GroebnerBasis(ChangeRing(I, GF(27173681)));
[
    x + 26637654,
    y + 3186055,
    z + 10380032
]
```

Of course, modulo any other prime the Gröbner basis is trivial so there are no other solutions. For example:

```
> GroebnerBasis(ChangeRing(I, GF(3)));
[
    1
]
```

Note that the problem can also be solved by using resultants, but this may yield many extraneous potential primes, while the Gröbner basis technique yields the exact list of primes for which there are modular solutions.

Example H105E8

This example shows how one can effectively compute in MAGMA with Gröbner bases over a ring which is not Euclidean (and may not even be a principal ideal ring), by starting with **Z** and adding appropriate defining relations. The input for this example is based on [AL94, Ex. 4.2.13].

Let $R = \mathbf{Z}[\sqrt{-5}]$. R is the maximal order of $\mathbf{Q}(\sqrt{-5})$ and is **NOT** a PIR. We consider the ideal I of $R[x, y]$ generated by $f_1 = 2xy + \sqrt{-5}y$ and $f_2 = (1 + \sqrt{-5})x^2 - xy$. To work over R, we

simply compute over $\mathbf{Z}$, introduce a new variable S to represent $\sqrt{-5}$, make sure that S is less than both x and y in the monomial order, and include the polynomial (S^2+5) in the ideal I. We then print out the Gröbner basis of I.

```
> P<x, y, S> := PolynomialRing(IntegerRing(), 3);
> f1 := 2*x*y + S*y;
> f2 := (1 + S)*x^2 - x*y;
> I := ideal<P | f1, f2, S^2 + 5>;
> GroebnerBasis(I);
[
    x^2*S + x^2 + 5*y^3 + 13*y*S - 25*y,
    6*x^2 + 5*y^2 + 3*y*S - 10*y,
    x*y + 5*y^3 + 13*y*S - 25*y,
    y^2*S + 5*y^2 - 15*y,
    10*y^2 + 5*y*S - 25*y,
    S^2 + 5
]
```

In [AL94, p. 224], a (weak) Gröbner basis for the ideal is given as $\{f_2, f_5, f_7, f_9\}$, where $f_5 = (5+\sqrt{-5})y^2 - 15y$, $f_7 = -2\sqrt{-5}y^2 + 5(1+\sqrt{-5})y$, and $f_9 = xy + \sqrt{-5}y^3 - 5\sqrt{-5}y^2 + 8\sqrt{-5}y$. We can easily verify that the ideal J generated by these 4 polynomials describes the same ideal as I (and so has the same Gröbner basis in MAGMA).

```
> f5 := (5 + S)*y^2 - 15*y;
> f7 := -2*S*y^2 + (5 + 5*S)*y;
> f9 := x*y + S*y^3 - 5*S*y^2 + 8*S*y;
> J := ideal<P |  f2, f5, f7, f9, S^2 + 5>;
> I eq J;
true
> GroebnerBasis(I) eq GroebnerBasis(J);
true
```

We can even write f_5, f_7 and f_9 as combinations of the Gröbner basis elements of I, as follows.

```
> Coordinates(I, f5);
[
    0, 0, 0, 1, 0, 0
]
> Coordinates(I, f7);
[
    0, 0, 0, -2, 1, 0
]
> Coordinates(I, f9);
[
    0, 0, 1, y, -y - 1, 0
]
```

We can see that these elements are fairly trivially derived from the Gröbner basis which MAGMA computes for I. But if we now create J again using the `IdealWithFixedBasis` function and the sequence $Q = [f_2, f_5, f_7, f_9, S^2+5]$, then we can see the coordinates of any element of $I = J$ as a

linear combination of the elements of Q. We find the coordinates of the second element of MAGMA's original Gröbner basis of I with respect to Q. The resulting coordinates are rather non-trivial.

```
> Q := [f2, f5, f7, f9, S^2 + 5];
> J := IdealWithFixedBasis(Q);
> J eq I;
true
> g := GroebnerBasis(I)[2];
> g;
6*x^2 + 5*y^2 + 3*y*S - 10*y
> C := Coordinates(J, g);
> C;
[
    -S + 1,
    -5*y + 1,
    -x - y^2*S + 7*y*S - 2*y - 7*S - 2,
    -2*y*S + 4*S + 6,
    x^2 + 5*y^3 - 13*y^2 + 3*y
]
```

We check that multiplying out the expression recovers g.

```
> &+[C[i]*Q[i]: i in [1 .. #C]] eq g;
true
```

Note that in the terminology of Adams and Loustaunau, MAGMA is here computing a "strong" Gröbner basis (for this representation which uses an extra variable for $\sqrt{-5}$), while these authors show that $\{f_2, f_5, f_7, f_9\}$ constitutes a "weak" Gröbner basis for I over the ring $\mathbf{Z}[\sqrt{-5}]$. The fact that the coordinates of g with respect to Q are rather non-trivial shows that MAGMA's strong Gröbner basis computation has computed a lot more information than the weak Gröbner basis (i.e., g, which must be included in the strong Gröbner basis, is not trivially derived from Q).

Most importantly of all, the fact that we have done all this by defining things over $\mathbf{Z}$ with the extra variable S has been no less powerful: we can still do full membership testing, normal forms, coordinate computations, etc. with this representation. Also, see below for an elimination computation which continues this example.

Gröbner bases over very many other general rings can be effectively handled in just the same way as that presented in this example! For example, if we need $\alpha = (1 + \sqrt{5})/2$, we can introduce a variable new A and the polynomial $(2A - 1)^2 - 5$.

Example H105E9

We construct an ideal I of the polynomial ring $P = \mathbf{Q}[x, y]$ with a specific fixed basis S, determine that I is the full polynomial ring P, and then find coordinates of the polynomial 1 of P with respect to S. Note that we use the function `IdealWithFixedBasis` to construct the ideal so that the fixed basis will be remembered.

```
> P<x, y> := PolynomialRing(RationalField(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
```

```
> 1 in I;
true
> C := Coordinates(I, P!1);
> C;
[
    -1/2*x^2*y^3 - 1/2*x^2*y^2 + 1/2*x^2*y + 1/2*x^2 + 1/2*x*y^3 +
        1/2*x*y^2 - 1/2*x*y - 1/2*y^4 - 1/2*y^3 + 1/2*y^2 + 1/2*y,
    1/2*x*y^3 + 1/2*x*y^2 - 1/2*x*y - 1/2*x - 1/2*y^3 - 1/2*y^2 + 1/2*y,
    -1/2*y^2 + 1
]
```

Now we check that multiplying out by the coordinates gives 1.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
1
```

Now we move the problem to being over the integer ring $\mathbf{Z}$.

```
> P<x, y> := PolynomialRing(IntegerRing(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
false
> GroebnerBasis(I);
[
    x + 1,
    y + 1,
    2
]
```

We note that 1 is not in the ideal this time, but 2 is! So we compute the coordinates of 2 with respect to I this time.

```
> C := Coordinates(I, P!2);
> C;
[
    x^2*y^2 - x^2*y - x^2 - x*y^2 + x*y + x + y^4 + y^3 - y^2 - y - 1,
    -x*y^2 + x*y + x + y^3 + y^2 - y - 1,
    -x^2 - x*y + y - 2
]
```

Note that C is the same as above, except that each polynomial has been scaled by 2 to make it integral. Finally we check again that multiplying out by the coordinates gives 2.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
2
```

Incidentally, we can see from the Gröbner basis of I over $\mathbf{Z}$ that the only solution to the system of equations described by S is the local solution $x = y = 1$ over $\mathbf{F}_2$.

Example H105E10

Gröbner bases can be constructed over any exact Euclidean ring in MAGMA, not just the ring of integers and its residue class rings.
We construct an ideal I of the polynomial ring $P = \mathbf{Q}[x, y]$ with a specific fixed basis S, determine that I is the full polynomial ring P, and then find coordinates of the polynomial 1 of P with respect to S. Note that we use the function `IdealWithFixedBasis` to construct the ideal so that the fixed basis will be remembered.

```
> P<x, y> := PolynomialRing(RationalField(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
true
> C := Coordinates(I, P!1);
> C;
[
    -1/2*x^2*y^3 - 1/2*x^2*y^2 + 1/2*x^2*y + 1/2*x^2 + 1/2*x*y^3 +
        1/2*x*y^2 - 1/2*x*y - 1/2*y^4 - 1/2*y^3 + 1/2*y^2 + 1/2*y,
    1/2*x*y^3 + 1/2*x*y^2 - 1/2*x*y - 1/2*x - 1/2*y^3 - 1/2*y^2 + 1/2*y,
    -1/2*y^2 + 1
]
```

Now we check that multiplying out by the coordinates gives 1.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
1
```

Now we move the problem to being over the integer ring $\mathbf{Z}$.

```
> P<x, y> := PolynomialRing(IntegerRing(), 2);
> S := [x^2 - y, x^3 + y^2, x*y^3 - 1];
> I := IdealWithFixedBasis(S);
> 1 in I;
false
> GroebnerBasis(I);
[
    x + 1,
    y + 1,
    2
]
```

We note that 1 is not in the ideal this time, but 2 is! So we compute the coordinates of 2 with respect to I this time.

```
> C := Coordinates(I, P!2);
> C;
[
    x^2*y^2 - x^2*y - x^2 - x*y^2 + x*y + x + y^4 + y^3 - y^2 - y - 1,
    -x*y^2 + x*y + x + y^3 + y^2 - y - 1,
    -x^2 - x*y + y - 2
```

```
]
```

Note that C is the same as above, except that each polynomial has been scaled by 2 to make it integral. Finally we check again that multiplying out by the coordinates gives 2.

```
> C[1]*S[1] + C[2]*S[2] + C[3]*S[3];
2
```

Incidentally, we can see from the Gröbner basis of I over $\mathbf{Z}$ that the only solution to the system of equations described by S is the local solution $x = y = 1$ over $\mathbf{F}_2$.

105.4.7 Degree-d Gröbner Bases

`GroebnerBasis(S, d : parameters)`

Given a set or sequence S of polynomials from a graded polynomial ring P, return the weighted degree-d Gröbner basis of the ideal generated by S, which is the truncated Gröbner basis obtained by ignoring S-polynomial pairs whose weighted degree (with respect to the grading on P) is greater than d.

If the ideal is homogeneous, then it is guaranteed that the result is equal to the set of all polynomials in the full Gröbner basis of the ideal whose weighted degree is less than or equal to d, and a polynomial whose weighted degree is less than or equal to d is in the ideal iff its normal form with respect to this truncated basis is zero. But if the ideal is not homogeneous, these last properties may not hold, but it may be still useful to construct the truncated basis.

The parameters are the same as those for the procedure `Groebner`. See also [BW93, section 10.2] for further discussion. Note that the base ring may be a field or Euclidean ring.

Example H105E11

We create a graded polynomial ring and compute the degree-d Gröbner basis of a sequence L of homogeneous polynomials for various d. Since the polynomials are homogeneous (with respect to the grading), we check that the result for each d contains the set of all polynomials in the full Gröbner basis of L having weighted degree less than or equal to d.

```
> P<a,b,c,d> := PolynomialRing(RationalField(), [4,3,2,1]);
> L := [a*b - c^2*d^3, b*c*d + c^3, c^2*d - d^5, a*d - b*c];
> [IsHomogeneous(f): f in L];
[ true, true, true, true ]
> [Degree(f): f in L];
[ 7, 6, 5, 5 ]
> G:=GroebnerBasis(L);
> G;
[
    a*b - d^7,
    a*c^3 + d^10,
    a*d - b*c,
```

```
    b^2*c - d^8,
    b*c^3 + d^9,
    b*c*d + c^3,
    b*d^5 + c^4,
    c^5 - d^10,
    c^2*d - d^5,
    c*d^7 - d^9
]
> #G;
10
> [Degree(f): f in G];
[ 7, 10, 5, 8, 9, 6, 8, 10, 5, 9 ]
> for D := 1 to 10 do
>     T := GroebnerBasis(L, D);
>     printf "D = %o, #GB = %o, contains all degree-D polynomials: %o\n",
>         D, #T, {f: f in G | Degree(f) le D} subset T;
> end for;
D = 1, #GB = 4, contains all degree-D polynomials: true
D = 2, #GB = 4, contains all degree-D polynomials: true
D = 3, #GB = 4, contains all degree-D polynomials: true
D = 4, #GB = 4, contains all degree-D polynomials: true
D = 5, #GB = 4, contains all degree-D polynomials: true
D = 6, #GB = 4, contains all degree-D polynomials: true
D = 7, #GB = 4, contains all degree-D polynomials: true
D = 8, #GB = 6, contains all degree-D polynomials: true
D = 9, #GB = 8, contains all degree-D polynomials: true
D = 10, #GB = 10, contains all degree-D polynomials: true
> GroebnerBasis(L, 5);
[
    a*b - d^7,
    a*d - b*c,
    b*c*d + c^3,
    c^2*d - d^5
]
> GroebnerBasis(L, 8);
[
    a*b - d^7,
    a*d - b*c,
    b^2*c - d^8,
    b*c*d + c^3,
    b*d^5 + c^4,
    c^2*d - d^5
]
```

105.5 Changing Coefficient Ring

The `ChangeRing` function enables the changing of the coefficient ring of a polynomial ring or ideal.

`ChangeRing(I, S)`

Given an ideal I of a polynomial ring $P = R[x_1, \ldots, x_n]$ of rank n with coefficient ring R, together with a ring S, construct the ideal J of the polynomial ring $Q = S[x_1, \ldots, x_n]$ obtained by coercing the coefficients of the elements of the basis of I into S. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. If R and S are fields and R is known to be a subfield of S and the current basis of I is a Gröbner basis, then the basis of J is marked automatically to be a Gröbner basis of J.

Example H105E12

It is better to find the Gröbner basis of an ideal over the smallest subfield possible (e.g. **Q**), then use `ChangeRing` to create the equivalent ideal over a splitting field to find the variety.

```
> P<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<P |
>     x + y + z + t + u,
>     x*y + y*z + z*t + t*u + u*x,
>     x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
>     x*y*z*t*u - 1>;
> Groebner(I);
> K<W> := CyclotomicField(5);
> J := ChangeRing(I, K);
> V := Variety(J);
> #V;
70
```

105.6 Changing Monomial Order

Often one wishes to change the monomial order of an ideal. MAGMA allows one to do this by use of the `ChangeOrder` function.

`ChangeOrder(I, Q)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a polynomial ring Q of rank n (with possibly a different order to that of P), return the ideal J of Q corresponding to I and the isomorphism f from P to Q. The map f simply maps $P.i$ to $Q.i$ for each i.

The point of the function is that one can change the order on monomials of I to be that of Q. When a Gröbner basis of J is needed to be calculated, MAGMA uses a conversion algorithm starting from a Gröbner basis of I if possible—this usually

makes order conversion much more efficient than by computing a Gröbner basis of J from scratch.

`ChangeOrder(I, order)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a monomial order *order* (see Section 105.2), construct the polynomial ring $Q = R[x_1, \ldots, x_n]$ with order *order*, and then return the ideal J of Q corresponding to I and the isomorphism f from P to Q. See the section on monomial orders for the valid values for the argument *order*. The map f simply maps $P.i$ to $Q.i$ for each i.

`ChangeOrder(I, T)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a tuple T, construct the polynomial ring $Q = R[x_1, \ldots, x_n]$ with the monomial order given by the tuple T on the monomials, and then return the ideal J of Q corresponding to I and the isomorphism f from P to Q. T must be a tuple whose components match the valid arguments for the monomial orders in Section 105.2 (or a tuple returned by the function `MonomialOrder`).

Example H105E13

We write a function `univgen` which, given a zero-dimensional ideal defined over a field, computes the univariate elimination ideal generator for a particular variable by changing order to the appropriate univariate order. Note that this function is the same as (and is in fact implemented in exactly the same way as) the intrinsic function `UnivariateEliminationIdealGenerator`. We then find the appropriate univariate polynomials for a particular ideal.

```
> function univgen(I, i)
>    // Make sure I has a Groebner basis so that
>    // the conversion algorithm will be used when
>    // constructing a Groebner basis of J
>    Groebner(I);
>    J := ChangeOrder(I, "univ", i);
>    Groebner(J);
>    return rep{f: f in Basis(J) | IsUnivariate(f, i)};
> end function;
>
> P<x, y, z> := PolynomialRing(RationalField(), 3, "grevlex");
> I := ideal<P |
>     1 - x + x*y^2 - x*z^2,
>     1 - y + y*x^2 + y*z^2,
>     1 - z - z*x^2 + z*y^2 >;
>
> univgen(I, 1);
x^21 - x^20 - 2*x^19 + 4*x^18 - 5/2*x^17 - 5/2*x^16 + 4*x^15 -
    15/2*x^14 + 129/16*x^13 + 11/16*x^12 - 103/8*x^11 +
    131/8*x^10 - 49/16*x^9 - 171/16*x^8 + 12*x^7 - 3*x^6 -
    29/8*x^5 + 15/4*x^4 - 17/16*x^3 - 5/16*x^2 + 5/16*x - 1/16
```

```
> univgen(I, 2);
y^14 - y^13 - 13/2*y^12 + 8*y^11 + 53/4*y^10 - 97/4*y^9 -
    45/8*y^8 + 33*y^7 - 25/2*y^6 - 18*y^5 + 107/8*y^4 + 5/8*y^3 -
    27/8*y^2 + 9/8*y - 1/8
> univgen(I, 3);
z^21 - z^20 - 2*z^19 + 4*z^18 - 5/2*z^17 - 5/2*z^16 + 4*z^15 -
    15/2*z^14 + 129/16*z^13 + 11/16*z^12 - 103/8*z^11 +
    131/8*z^10 - 49/16*z^9 - 171/16*z^8 + 12*z^7 - 3*z^6 -
    29/8*z^5 + 15/4*z^4 - 17/16*z^3 - 5/16*z^2 + 5/16*z - 1/16
```

105.7 Hilbert-driven Gröbner Basis Construction

MAGMA incorporates an implementation of the *Hilbert-driven Buchberger Algorithm* [Tra96]. This algorithm constructs the Gröbner basis of an homogeneous ideal I whose Hilbert series is known. The algorithm is often much more efficient than the conventional Buchberger algorithm since knowledge of the Hilbert series eliminates many unnecessary reductions of S-polynomials. The algorithm can also be used as an alternative to the Gröbner Walk algorithm for changing order since one can compute the Hilbert series of the ideal with respect to an easy monomial order, and then start again with the Hilbert-driven algorithm to compute the Gröbner basis with respect to the desired final order. Furthermore, the algorithm can sometimes be used to test whether an ideal has a particular Hilbert series and abort early if this is proven to be false. The algorithm is also used extensively internally in the Invariant Theory algorithms of MAGMA.

`HilbertGroebnerBasis(S, H)`

`HilbertGroebnerBasis(S, N)`

Let S be a set or sequence of homogeneous polynomials from the multivariate polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field, and let I be the ideal of P generated by S. Let either H be the Hilbert series $H_{P/I}(t)$ of I (as a rational function in $\mathbf{Z}(t)$) or let $N \in \mathbf{Z}[t]$ be a univariate integer polynomial such that the weighted numerator of the Hilbert series of I is N. This function attempts to construct the (reduced) Gröbner basis of I using the given Hilbert series. The weighted numerator of the Hilbert series of I is the Hilbert series $H_{P/I}(t)$ of I, multiplied by the denominator $\prod_{i=1}^{n} 1 - t^{d_i}$, where d_i is the weighted degree of the i-th variable x_i (this denominator is thus $(1-t)^n$ if P has the default grading).

If the function returns `false`, then H (or N) cannot be the correct Hilbert series (or weighted numerator of the Hilbert series) of I. Otherwise, the function returns `true` and a sequence B of polynomials which generates the same ideal as S; if H or N is correct, B will be the (reduced) Gröbner basis of I.

In more detail, let f_H be the power series corresponding to the true Hilbert series of I and let f_N be the power series corresponding to $N/(\prod_{i=1}^{n} 1 - t^{d_i})$. If $f_H = f_N$, then the function returns `true` and the correct (reduced) Gröbner basis of I. Otherwise, consider the first term at which f_N and f_H differ: if the coefficient

of f_N is greater than that of f_H, then the function returns `false` (since it will not be able to construct the extra Gröbner basis polynomials needed), otherwise the function will return `true` with a partial Gröbner basis (since it concludes that it has enough Gröbner basis polynomials when it hasn't). Consequently, the algorithm is usually used when the correct Hilbert series or weighted numerator of the Hilbert series is known, or when there is a weighted numerator which is known to be greater than or equal to the correct weighted numerator of the Hilbert series.

`SetVerbose("HilbertGroebner", v)`

Change verbose printing for the Hilbert-driven Buchberger algorithm to be v. Currently the legal values for v are `true`, `false`, 0, or 1.

Example H105E14

We illustrate a subalgorithm of the Invariant Theory module of MAGMA which uses the Hilbert-driven Buchberger Algorithm.
Let R be the invariant ring of the (permutation) cyclic group G of order 4 over the field $K = \mathbf{F}_2$. Suppose we have a sequence L of 4 homogeneous invariants of degrees $1, 2, 2$, and 4 respectively. We wish to determine efficiently whether the polynomials of L constitute primary invariants for R. To check this, the ideal generated by L must be zero-dimensional and the elements of L must be algebraically independent. This is equivalent to the condition that the weighted numerator of the Hilbert series of the ideal is the product $(1-t)(1-t^2)^2(1-t^4)$. If that is not the correct weighted numerator, it will be less than the correct weighted numerator so the algorithm will return whether the polynomials L do constitute primary invariants for R.

```
> K := GF(2);
> P<a,b,c,d> := PolynomialRing(K, 4);
> L := [
>     a + b + c + d,
>     a*b + a*d + b*c + c*d,
>     a*c + b*d,
>     a*b*c*d
> ];
> // Form potential Hilbert series weighted numerator
> T<t> := PolynomialRing(IntegerRing());
> N := &*[1 - t^Degree(f): f in L];
> N;
t^9 - t^8 - 2*t^7 + 2*t^6 + 2*t^3 - 2*t^2 - t + 1
> time l, B := HilbertGroebnerBasis(L, N);
Time: 0.000
> l;
true
> // Examine Groebner basis B of L:
> B;
[
    a + b + c + d,
    b^2 + d^2,
    b*c + b*d + c^2 + c*d,
```

```
    c^3 + c^2*d + c*d^2 + d^3,
    d^4
]
```

105.8 SAT solver

MAGMA V2.16 contains an interface to the *MiniSat* satisfiability (SAT) solver. Such a solver is given a system of boolean expressions in conjunctive normal form and determines whether there is an assignment in the variables such that all the expressions are satisfied. MAGMA supplies a function by which one may transform a system of boolean polynomial equations into an equivalent boolean system, and solve this via the SAT solver.

To use the interface function, the *MiniSat* program must currently be installed as a command external to MAGMA. At the time of writing (November 2009), the latest version of *MiniSat* can be installed as follows on most Unix/Linux systems:

(1) Download `http://minisat.se/downloads/minisat2-070721.zip` from the *MiniSat* website (`minisat.se`).

(2) Use the command `unzip minisat2-070721.zip` or equivalent to unzip the files.

(3) Change directory into `minisat/core` and run `make` there.

(4) Copy the produced executable `minisat` into a place which is in the current path when MAGMA is run.

`SAT(B)`

`Exclude`	[RngMPolElt]	*Default :* []
`Verbose`	BoolElt	*Default :* `true`

Given a sequence B of boolean polynomials in a rank-n boolean polynomial ring (or a rank-n polynomial ring over $\mathbf{F}_2$), call *MiniSat* on the associated boolean system and return whether the system is satisfiable, and if so, return also a solution S as a length-n sequence of elements of $\mathbf{F}_2$. (This assumes that *MiniSat* is in the executable path of external commands; see above for instructions for installing *MiniSat*).

The parameter `Exclude` may be set to a sequence $[e_1, \ldots e_k]$, where each e_i is a sequence of n elements of $\mathbf{F}_2$, specifying that the potential solutions in e_i are to be excluded (this is done by adding new relations to the system to exclude the e_i). The verbose information printed by *MiniSat* may be controlled by the parameter `Verbose`.

Example H105E15

In Example H105E5, we solved a boolean polynomial system via the standard Gröbner basis method (which the function `Variety` uses). Here we solve the same system via the SAT solver. Each time we obtain a solution, we can call the function again, but excluding the solution(s) already found. We can thus find all the solutions to the system. Of course, this is not worth doing when there are large numbers of solutions, but it may be of interest to find all solutions when it is expected there is a small number of solutions.

```
> P<a,b,c,d,e> := BooleanPolynomialRing(5, "grevlex");
> B := [a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1];
> l, S := SAT(B);
> l;
true
> S;
[ 1, 1, 1, 0, 0 ]
> Universe(S);
Finite field of size 2
> [Evaluate(f, S): f in B];
[ 0, 0, 0, 0 ]
> l, S2 := SAT(B: Exclude := [S]);
> l;
true
> S2;
[ 0, 1, 1, 1, 0 ]
> [Evaluate(f, S2): f in B];
[ 0, 0, 0, 0 ]
> l, S3 := SAT(B: Exclude := [S, S2]);
> l;
false
```

105.9 Bibliography

[**AL94**] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases*, volume 3 of *Graduate studies in mathematics*. American Mathematical Society, Providence, R.I., 1994.

[**Buc65**] Bruno Buchburger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, Austria, 1965.

[**BW93**] Thomas Becker and Volker Weispfenning. *Gröbner Bases*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.

[**CKM97**] Stephane Collart, Michael Kalkbrener, and Daniel Mall. Converting Bases with the Gröbner Walk. *J. Symbolic Comp.*, 24(3):465–469, 1997.

[**CLO96**] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms.* Undergraduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2nd edition, 1996.

[**CLO98**] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry.* Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.

[**Fau99**] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139 (1-3):61–88, 1999.

[**FGLM93**] Jean-Charles Faugère, Patrizia Gianni, Daniel Lazard, and Teo Mora. Efficient computations of zero-dimensional Gröbner bases by change of ordering. *J. Symbolic Comp.*, 16:329–344, 1993.

[**Möl88**] H.M. Möller. On the construction of Gröbner bases using syzygies. *J. Symbolic Comp.*, 6:345–359, 1988.

[**Ste04**] Allan Steel. Gröbner Basis Timings Page. URL:http://magma.maths.usyd.edu.au/users/allan/gb/, 2004.

[**Tra96**] Carlo Traverso. Hilbert Functions and the Buchberger Algorithm. *J. Symbolic Comp.*, 22(4):355–376, 1996.

106 POLYNOMIAL RING IDEAL OPERATIONS

Chapter 106
POLYNOMIAL RING IDEAL OPERATIONS

106.1 Introduction

This chapter describes the MAGMA functionality for ideals over polynomial rings. For the basics on multivariate polynomial rings and their elements, see Chapter 24. Most of the significant operations with ideals construct or utilise a previously-constructed Gröbner basis. The monomial ordering used for this basis can greatly affect the speed and memory usage of these operations. This ordering is attached to the polynomial ring in which the ideals are created. For information on Gröbner bases and the creation of polynomial rings with specified orders, see Chapter 105. That chapter also tells the user how to compute and return a Gröbner basis, or just to compute it internally for later use in the operations described below, with many additional configuration parameters to optimise the computation. Users may ignore the issue when creating the ambient polynomial rings by allowing MAGMA to make default choices. It is, however, highly recommended that users who wish to work with complicated ideals thoroughly acquaint themselves with the options available. MAGMA has an extremely powerful Gröbner basis engine and often makes sophisticated choices internally of alternative monomial orders for particular computations. Ultimately, however, the user may significantly speed up his work by a judicious choice of order. We note here that the default order is the lexicographical one, a total elimination order well suited to finding solutions of zero-dimensional systems of polynomial equations but tending to produce very large bases that can take much time and memory to compute. For homogeneous ideals of rings with the standard weighting (all variables have weight one), the grevlex order is usually the best in practise and there is theoretical justification for this. In the case that the ring has a different weighting and the ideal is homogeneous with respect to that, the weighted grevlex order is the best choice. In any case, the `EasyIdeal` and `EasyBasis` intrinsics of the Gröbner basis chapter return to the user a basis for an internally chosen good order and these "easy" bases are used in many internal functions if a basis with respect to the polynomial ring order has not already been computed and stored.

The functions and operations described here cover a wide range of commutative algebra functionality. This includes sums and intersections, colon ideals and saturations, elimination, radicals and primary decompositions, Noether normalisations and computation of Hilbert polynomials and Hilbert series.

Related chapters including other polynomial ring functionality relying on Gröbner bases are the chapter on invariant rings of finite group actions, Chapter 110, and the chapters on affine algebras (Chapter 108) and on modules over affine algebras (Chapter 109). The chapter on algebraically closed fields (Chapter 40) describes functions that allows one to compute the variety of an ideal over the algebraic closure of the base field. And, of course, the Algebraic Geometry component of MAGMA and parts of the Arithmetic Geometry are built upon the commutative algebra here.

106.2 Creation of Polynomial Rings and their Ideals

As noted in the introduction, for the basics on multivariate polynomial rings and their elements, including their creation, the user should refer to Chapter 24. For creation of polynomial rings with non-default (currently lexicographic) monomial ordering, the user should refer to Chapter 105. Similarly, the basic creation functions for ideals and additional basis options are described in Chapter 105. The commonest creation methods are the `ideal` constructor and the `Ideal` function.

106.3 First Operations on Ideals

In the following, note that since ideals of a full polynomial ring P are regarded as subrings of P, the ring P itself is a valid ideal as well (the ideal containing 1).

106.3.1 Simple Ideal Constructions

The following basic constructions involve no Gröbner basis computation.

`I + J`

Given ideals I and J of the same polynomial ring P, return the sum of I and J, which is the ideal generated by the generators of I and those of J.

`I * J`

Given ideals I and J of the same polynomial ring P, return the product of I and J, which is the ideal generated by the products of the generators of I and those of J.

`I ^ k`

Given an ideal I of the polynomial ring P, and an integer k, return the k-th power of I.

`I / J`

Given an ideal I of a polynomial ring P over a field and an ideal J of P, such that $J \subset I$, return the affine algebra I/J.

106.3.2 Basic Commutative Algebra Operations

The following important basic operations on ideals involve Gröbner basis computation and use the standard algorithms as described in Chapter 1.8 of [GP02], for example, unless otherwise stated.

`QuotientDimension(I)`

Given an ideal I of a polynomial ring P over a field K, return the dimension of P/I as a K-vector space. Note that this is quite different from the function `Dimension` below (which returns the Krull dimension of an ideal). If I is not of Krull dimension 0 then the vector space is infinite and `Infinity` is returned.

`ColonIdeal(I, J)`

`IdealQuotient(I, J)`

Given ideals I and J of the same polynomial ring P, return the colon ideal $I : J$ (or ideal quotient of I by J), consisting of the polynomials f of P such that $f * g$ is in I for all g in J.

`ColonIdeal(I, f)`

`IdealQuotient(I, f)`

Given an ideal I and an element f of a polynomial ring P, return the saturation (colon) ideal $I : f^\infty$, consisting of the polynomials g of P such that there exists an $i \geq 1$ with $f^i * g \in I$. An integer s with $s \geq 1$ is also returned such that $I : f^\infty = I : f^s$. Note that if s is not needed, only one return value of the function should be expected which increases the efficiency enormously. Note also that this function is *not* equivalent to taking the ideal quotient of I by the ideal of P generated by f. It is in some ways a more natural operation mathematically, corresponding to taking the full inverse image of the localised ideal I_f under the localisation map $P \to P_f$, and can be faster than the $I : f$ computation, if s is not required. In this case, the computation goes by the elimination of extra variable t from the ideal $< I, 1 - f * t >$.

`ColonIdealEquivalent(I, f)`

`Saturation(I, f)`

Given an ideal I and an element f of a polynomial ring P, return the saturation (colon) ideal $C = I : f^\infty$, and a polynomial $g \in P$ such that $C = I : \langle g \rangle$ and g is of minimal degree. The irreducible factors of g will be a subset of the irreducible factors of f (and the corresponding multiplicities may be greater or lesser, depending on how often an irreducible factor divides the ideal I).

`Saturation(I, J)`

Given ideals I and J of some polynomial ring P, return the saturation $(I : J^\infty)$: that is, the ideal $\{f \in P : \exists n > 0, f^n J \subseteq I\}$.

`Saturation(I)`

Given an ideal I of a polynomial ring P, return the saturation of I with respect to the irrelevant ideal of P – that is, the ideal of all elements of P having positive degree.

`Generic(I)`

Given an ideal I of a generic polynomial ring P, return P.

`LeadingMonomialIdeal(I)`

Given an ideal I, return the leading monomial ideal of I; that is, the ideal generated by all the leading monomials of I.

`I meet J`

Given ideals I and J of the same polynomial ring P, return the intersection of I and J.

`&meet S`

Given a set or sequence S of ideals of the same polynomial ring P, return the intersection of all the ideals of S.

`RegularSequence(I)`

`Homogeneous` BoolElt *Default :* `true`

Given an ideal I of a polynomial ring P over a field, computes and returns a maximal regular sequence in I. The algorithm used is that of Eisenbud and Sturmfels ([ES94]) that tries to construct a regular sequence of fairly sparse polynomials. If parameter `Homogeneous` is `true` (the default), and I is a homogeneous ideal with respect to the variable weights, then the regular sequence constructed will also consist of homogeneous polynomials.

`ReesIdeal(P, I)`

`ReesIdeal(P, J, I)`

`a` RngMPolElt *Default :* 1

`ReesIdeal(R, I)`

`a` RngMPolElt *Default :* 1

In each case P is a multivariate polynomial ring and I is an ideal of P. In the third case R is an affine quotient algebra of the form P/J. In the second case J is another ideal of P and we write R for the affine algebra P/J. In the first case, let $R = P$.

The Rees algebra $R(I)$ is the finitely-generated, graded polynomial algebra isomorphic to the algebra

$$R \oplus I \oplus I^2 \oplus I^3 \oplus \dots$$

where I gives the first graded part, I^2 the second etc. and the multiplication is the obvious one. Here I is thought of as an ideal of R, rather than P for the second and third signatures. *Proj* of this algebra represents the blow-up of the affine scheme $Spec(R)$ along the closed subscheme defined by I (see Chapter 2, Section 7 of [Har77]).

The function returns the *Rees ideal*, K, such that, if R_1 is the generic polynomial ring of K, then R_1/K is an affine algebra isomorphic to $R(I)/ < a - torsion >$, where a is an element of P (or R in the third case) that gives a non-zero divisor in R and is 1 by default. In the first case, any such a remains a non-zero divisor in $R(I)$, so is redundant. However, in the second and third cases, a can be specified to be not equal to 1 by use of the `a` parameter. Geometrically, dividing out by a-torsion gives the coordinate ring of the maximal closed subscheme of the blow-up that is flat over the generic point and the codimension one points defined by the vanishing of a, if these points are regular.

106.3.3 Ideal Predicates

`I eq J`

Given two ideals I and J of the same polynomial ring P, return whether I and J are equal. Involves the use of a Gröbner basis for each ideal.

`I ne J`

Given two ideals I and J of the same polynomial ring P, return whether I and J are not equal. Involves the use of a Gröbner basis for each ideal.

`I notsubset J`

Given two ideals I and J in the same polynomial ring P return whether I is not contained in J. Involves the use of a Gröbner basis for J.

`I subset J`

Given two ideals I and J in the same polynomial ring P return whether I is contained in J. Involves the use of a Gröbner basis for J.

`IsZero(I)`

Given an ideal I of the polynomial ring P, return whether I is the zero ideal (contains zero alone).

`IsProper(I)`

Given an ideal I of the polynomial ring P, return whether I is proper; that is, whether I is strictly contained in P, or whether the Gröbner basis of I does not contain 1 alone.

`IsHomogeneous(I)`

Given an ideal I of the polynomial ring P, this function returns whether I is homogeneous with respect to the weights on the variables of P (i.e., whether I possesses a basis consisting of homogeneous polynomials alone). Checks whether the current basis of I consists of homogeneous polynomials and, if not and the current basis isn't Gröbner, then whether an easy Gröbner basis consists of homogeneous elements.

`IsPrincipal(I)`

Given an ideal I of the polynomial ring P, return whether I is principal, and if so, return also a generator of I. This will be true if and only if an arbitrary Gröbner basis consists of a single (generating) element.

`IsPrimary(I)`

Given an ideal I of the polynomial ring P, return whether I is primary. An ideal I is primary if and only if for all $ab \in I$, either $a \in I$ or $b^n \in I$ for some $n \geq 1$. The restrictions on I are the same as for the function `PrimaryDecomposition`—see the description of that function. In general, this function computes or retrieves the primary decomposition and checks whether it has a unique element.

`IsPrime(I)`

Given an ideal I of the polynomial ring P, return whether I is prime. An ideal I is prime if and only if for all $ab \in I$, either $a \in I$ or $b \in I$. The restrictions on I are the same as for the function `PrimaryDecomposition`—see the description of that function. Again, this function computes the primary decomposition or uses the already stored one.

`IsMaximal(I)`

Given an ideal I of the polynomial ring P, return whether I is maximal. The restrictions on I are the same as for the function `PrimaryDecomposition`—see the description of that function. Checks first whether I is zero-dimensional (see below) and, if so, then checks whether it is prime. NB: given that I is of dimension 0, the prime/primary decomposition computation is relatively fast.

`IsRadical(I)`

Given an ideal I of the polynomial ring P, return whether I is radical; that is, whether the radical of I is I itself. The restrictions on I are the same as for the function `Radical`—see the description of that function. The function computes the radical or uses the already stored one.

`IsZeroDimensional(I)`

Given an ideal I of the polynomial ring P, defined over a field, return whether I is zero-dimensional (so the quotient of P by I has non-zero finite dimension as a vector space over the coefficient field – see the section on dimension for further details). Note that the full polynomial ring P as an ideal of itself has dimension -1, so it is not zero-dimensional.

`HasGrevlexOrder(I)`

Given an ideal I of the polynomial ring P, return whether the monomial order of I is the `grevlex` order.

Example H106E1

We construct some ideals in $\mathbf{Q}[x, y, z]$ and perform basic arithmetic on them.

```
> P<x,y,z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x*y - 1, x^3*z^2 - y^2, x*z^3 - x - 1>;
> J := ideal<P | x*y - 1, x^2*z - y, x*z^3 - x - 1>;
> A := I * J;
> A;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Basis:
[
    x^2*y^2 - 2*x*y + 1,
```

```
    x^3*y*z - x^2*z - x*y^2 + y,
    x^2*y*z^3 - x^2*y - x*y - x*z^3 + x + 1,
    x^4*y*z^2 - x^3*z^2 - x*y^3 + y^2,
    x^5*z^3 - x^3*y*z^2 - x^2*y^2*z + y^3,
    x^4*z^5 - x^4*z^2 - x^3*z^2 - x*y^2*z^3 + x*y^2 + y^2,
    x^2*y*z^3 - x^2*y - x*y - x*z^3 + x + 1,
    x^3*z^4 - x^3*z - x^2*z - x*y*z^3 + x*y + y,
    x^2*z^6 - 2*x^2*z^3 + x^2 - 2*x*z^3 + 2*x + 1
]
> M := I meet J;
> M;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Basis:
[
    x^4 + x^3 - x*z^2 + z^12 - 4*z^9 + 6*z^6 - z^4 - 4*z^3 + z + 1,
    x^5 + x^4 - x^2*z^2 + z^9 - 3*z^6 + 3*z^3 - z - 1,
    x*z^3 - x - 1,
    y - z^3 + 1
]
> A eq M;
true
> QuotientDimension(A);
24
> ColonIdeal(I, J);
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0
Basis:
[
    x*y - 1,
    x^3*z^2 - y^2,
    x*z^3 - x - 1
]
```

106.3.4 Element Operations with Ideals

`f in I`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is in I. The function computes the normal form of f relative to some Gröbner basis of I and checks if this is zero.

`f notin I`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is not in I. As with `in`, this performs a normal form computation.

`IsInRadical(f, I)`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is in the radical of I. Note that using this function is much quicker in general than actually computing the radical of I. It uses the algorithm described in section 1.8.6 of [GP02].

`JacobianIdeal(f)`

Return the ideal generated by all first partial derivatives of the polynomial f.

Example H106E2

We demonstrate the element operations with respect to an ideal of $\mathbf{Q}[x, y, z]$.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | (x + y)^3, (y - z)^2, y^2*z + z>;
> NormalForm(y^2*z + z, I);
0
> NormalForm(x^3, I);
-3*x^2*y - 3*x*z^4 - 6*x*z^2 + 1/2*z^3 + 3/2*z
> NormalForm(z^4 + y^2, I);
2*z^4 + 2*z^2
> x + y in I;
false
> IsInRadical(x + y, I);
true
> IsInRadical((x + y)^2, I);
true
> IsInRadical(z, I);
false
> SPolynomial(x^4 + y - z, x^2 + y - z);
-x^2*y + x^2*z + y - z
```

106.4 Computation of Varieties

The slightly non-standard term variety in this section refers to the (finite) solution set of the system of polynomials that make up a zero-dimensional ideal. It can also be thought of as the set of points of a zero-dimensional affine scheme over a specified field extension of the polynomial ring base field. For more general functionality for schemes of arbitrary dimension, see the chapters on Algebraic and Arithmetic Geometry. The functions here also work for higher-dimensional ideals if the base field is finite, when the solution set is again finite (over the base or a finite extension of the base).

The functions compute solutions over the base field of the polynomial ring or over an extension field L. MAGMA's algebraically closed fields (see Chapter 40) may be used to get all solutions if so desired when an explicit splitting field is not known for the system. L should be an exact field over which MAGMA has a root-finding algorithm for univariate polynomials or a real or complex field.

For the corresponding functions with argument a zero-dimensional scheme which may not be affine, see the Section 112.7 in the Schemes chapter.

`Variety(I)`

`Variety(I, L)`

`Digits` RNGINTELT *Default :* 38

Given a zero-dimensional ideal I of a polynomial ring P, return the variety of I over its coefficient field K as a sequence of tuples. Each tuple is of length n, where n is the rank of P, and corresponds to an assignment of the n variables of P (in order) such that all polynomials in I vanish with this assignment.

If K is not a finite field then the ideal must be the full polynomial ring or be zero-dimensional so that the variety is known to be finite. If a superfield L of K is also given, the variety is computed over L instead, so the entries of the tuples lie in L.

If the field over which the variety is computed is the free complex field, MAGMA uses a special root finding algorithm to ensure the precision of the results; in this case, the parameter `Digits` may be given (see the `Roots` function in the Real and Complex Fields chapter (Chapter 25)).

The function works in the zero-dimensional case by first computing a triangular or radical decomposition of I (see Section 106.11). This reduces the problem to successively computing roots of univariate polynomials.

`VarietySequence(I)`

`VarietySequence(I, L)`

`Digits` RNGINTELT *Default :* 38

Given a zero-dimensional ideal I of a polynomial ring P whose order is of lexicographic type, return the variety of I over its coefficient field K as a sequence of sequences of elements of K. Each inner sequence is of length n, where n is the rank of P, and corresponds to an assignment of the n variables of P (in order) such that all polynomials in I vanish with this assignment.

If K is not a finite field then the ideal must be the full polynomial ring or be zero-dimensional so that the variety is known to be finite. If a superfield L of K is also given, the variety is computed over L instead, so the entries of the sequences lie in L.

If the field over which the variety is computed is the free complex field, MAGMA uses a special root finding algorithm to ensure the precision of the results; in this case, the parameter `Digits` may be given (see the `Roots` intrinsic function in the Real and Complex Fields chapter (Chapter 25)).

The function works in the zero-dimensional case by first computing a triangular or radical decomposition of I (see Section 106.11). This reduces the problem to successively computing roots of univariate polynomials.

`VarietySizeOverAlgebraicClosure(I)`

Given a zero-dimensional ideal I of a polynomial ring P over a field K, return the size of the variety of I over the algebraic closure K' of K. The size is determined by finding the (prime) radical decomposition of I and placing each component of the decomposition into normal position so the size of the variety of the component over K' can be read off. Note that this function will usually be much faster than actually computing the variety of I over a suitable extension field of K.

Example H106E3

We construct an ideal I of the polynomial ring $\mathbf{F}_{27}[x, y]$, and then find the variety $V = V(I)$. We then check that I vanishes on V.

```
> K<w> := GF(27);
> P<x, y> := PolynomialRing(K, 2);
> I := ideal<P | x^8 + y + 2, y^6 + x*y^5 + x^2>;
> Groebner(I);
> I;
Ideal of Polynomial ring of rank 2 over GF(3^3)
Order: Lexicographical
Variables: x, y
Inhomogeneous, Dimension 0
Groebner basis:
[
    x + 2*y^47 + 2*y^45 + y^44 + 2*y^43 + y^41 + 2*y^39 + 2*y^38 + 2*y^37 +
        2*y^36 + y^35 + 2*y^34 + 2*y^33 + y^32 + 2*y^31 + y^30 + y^28 + y^27 +
        y^26 + y^25 + 2*y^23 + y^22 + y^21 + 2*y^19 + 2*y^18 + 2*y^16 + y^15 +
        y^13 + y^12 + 2*y^10 + y^9 + y^8 + y^7 + 2*y^6 + y^4 + y^3 + y^2 + y +
        2,
    y^48 + y^41 + 2*y^40 + y^37 + 2*y^36 + 2*y^33 + y^32 + 2*y^29 + y^28 +
        2*y^25 + y^24 + y^2 + y + 1
]
> V := Variety(I);
> V;
[ <w^14, w^12>, <w^16, w^10>, <w^22, w^4> ]
```

```
> // Check that the original polynomials vanish:
> [
>    <x^8 + y + 2, y^6 + x*y^5 + x^2> where x is v[1] where y is v[2]: v in V
> ];
[ <0, 0>, <0, 0>, <0, 0> ]
> // Note that the variety of I would be larger over an extension field of K:
> VarietySizeOverAlgebraicClosure(I);
48
```

106.5 Multiplicities

This section contains some useful invariants for an isolated singularity at the origin of a hypersurface given by a multivariate polynomial f.

`MilnorNumber(f)`

Given a polynomial $f \in K[x_1, \ldots, x_n]$, where K is a field, return the Milnor number of f at the origin. This is the dimension of the quotient by the ideal generated by the partials of f in the localization of $K[x_1, \ldots, x_n]$ at the origin. See [CLO98, p. 147] or [DL06, Remark 9.37].

`TjurinaNumber(f)`

Given a polynomial $f \in K[x_1, \ldots, x_n]$, where K is a field, return the Tjurina number of f at the origin. This is the dimension of the quotient by the ideal generated by f and the partials of f in the localization of $K[x_1, \ldots, x_n]$ at the origin. See [CLO98, p. 148] or [DL06, Def. 9.35].

Example H106E4

We compute some Milnor and Tjurina numbers, based on Exercise 12 of [CLO98, p. 177].

```
> P<x,y> := PolynomialRing(RationalField(), 2);
> MilnorNumber((x^2 + y^2)^3 - 4*x^2*y^2); // 4-leaved rose
13
> [MilnorNumber(y^2 - x^n): n in [1 .. 5]];
[ 0, 1, 2, 3, 4 ]
> P<x,y,z> := PolynomialRing(RationalField(), 3);
> [MilnorNumber(x*y*z + x^n + y^n + z^n): n in [1 .. 10]];
[ 0, 1, 8, 11, 14, 17, 20, 23, 26, 29 ]
> [TjurinaNumber(x*y*z + x^n + y^n + z^n): n in [1 .. 10]];
[ 0, 1, 8, 10, 13, 16, 19, 22, 25, 28 ]
```

A much larger example is given in [DL06, p. 254].

```
> f := y^2 - 2*x^28*y - 4*x^21*y^17 + 4*x^14*y^33 - 8*x^7*y^49 +
>      x^56 + 20*y^65 + 4*x^49*y^16;
> time TjurinaNumber(f);
2260
```

```
Time: 0.010
```

106.6 Elimination

Elimination theory plays an important role when working with ideals of multivariate polynomial rings. MAGMA provides an assortment of functions to perform various kinds of elimination easily. Elimination of variables is accomplished by computing a Gröbner basis with respect to a suitable elimination order (for more information about elimination orders, see Section 105.2 as well as comments in the function descriptions below).

All of the functions in this section may be applied to ideals over general Euclidean rings, not just over fields.

106.6.1 Construction of Elimination Ideals

`EliminationIdeal(I, k: parameters)`

Given an ideal I of a polynomial ring P of rank n with $P = R[x_1, \ldots, x_n]$, together with an integer k with $0 \leq k \leq n$, return the k-th elimination ideal I_k of I, which is defined to be $I \cap R[x_{k+1}, \ldots, x_n]$. Thus I_k consists of all polynomials of I which have the first k variables eliminated. If the elimination ideals I_k are to be computed for several different k, it is recommended first that a Gröbner basis with respect to lexicographical order for I first be computed as then the elimination ideals can be determined trivially. If I does not have a Gröbner basis stored with respect to lexicographical order, then a Gröbner basis computation will be necessary each time an elimination ideal is desired.

If $k = n$, then $I \cap R$ is returned, which, if R is a field, is always the full ring P or the empty ideal, according to whether I is the full polynomial ring or not. But if R is not a field, then this intersection will yield the ideal generated by the normalized smallest element of R which is in I (according to the Euclidean norm), which could be neither 0 nor 1.

The parameters are as for the `Groebner` procedure. Note that setting `Al := "Direct"` occasionally produces much better performance since the relevant elimination order may yield a better Gröbner basis than the default method of going via the `grevlex` order.

`EliminationIdeal(I, S)`

`EliminationIdeal(I, S)`

Given an ideal I of a polynomial ring P of rank n with $P = R[x_1, \ldots, x_n]$, together with a set S describing a subset U of the variables $\{x_1, \ldots x_n\}$, return the elimination ideal I_U of I, which is defined to be $I \cap R[U]$. Thus I_U consists of all polynomials of I which contain variables only found in U. U can be specified in two ways: either as a set S of integers in the range $1 \ldots n$ such the integer i corresponds to the i-th variable x_i, or as a set S of variables lying in P. S may be the empty set, in which case this is equivalent to `EliminationIdeal(I, n)`; see above.

Example H106E5

This example continues the example above which computed a Gröbner basis over a ring which was not even a PIR.

As before, $R = \mathbf{Z}[\sqrt{-5}]$ and I is the ideal of $R[x, y]$ generated by $f_1 = 2xy + \sqrt{-5}y$ and $f_2 = (1+\sqrt{-5})x^2 - xy$. As before, we compute over $\mathbf{Z}$, introduce a new variable S and include (S^2+5) in I, so we can effectively work over R.

```
> P<x, y, S> := PolynomialRing(IntegerRing(), 3);
> f1 := 2*x*y + S*y;
> f2 := (1 + S)*x^2 - x*y;
> I := ideal<P | f1, f2, S^2 + 5>;
> GroebnerBasis(I);
[
    x^2*S + x^2 + 5*y^3 + 13*y*S - 25*y,
    6*x^2 + 5*y^2 + 3*y*S - 10*y,
    x*y + 5*y^3 + 13*y*S - 25*y,
    y^2*S + 5*y^2 - 15*y,
    10*y^2 + 5*y*S - 25*y,
    S^2 + 5
]
```

In [AL94, Ex. 4.3.8], the elimination ideal $E_y = I \cap (\mathbf{Z}[\sqrt{-5}])[y]$ is shown to be generated by $f_5 = (5 + \sqrt{-5})y^2 - 15y$ and $f_7 = -2\sqrt{-5}y^2 + 5(1 + \sqrt{-5})y$. We can compute E_y in MAGMA easily using `EliminationIdeal`. We must be careful to include S in the second argument (the set of variables which we want), since S should be considered a 'constant' (member of R) in this context.

```
> Ey := EliminationIdeal(I, {y, S});
> GroebnerBasis(Ey);
[
    y^2*S + 5*y^2 - 15*y,
    10*y^2 + 5*y*S - 25*y,
    S^2 + 5
]
```

Obviously, the polynomials yielded are simply the last 3 polynomials of the full Gröbner basis given above. We check also that the ideal generated by f_5 and f_7 over R is the same as that given by MAGMA.

```
> f_5 := y^2*S + 5*y^2 - 15*y;
> f_7 := -2*y^2*S + 5*y*S + 5*y;
> E := ideal<P | f_5, f_7, S^2 + 5>;
> E eq Ey;
true
```

Finally, we also compute $E_x = I \cap (\mathbf{Z}[\sqrt{-5}])[x]$, which requires more effort this time, since it cannot be read off the Gröbner basis.

```
> Ex := EliminationIdeal(I, {x, S});
> GroebnerBasis(Ex);
```

```
[
    2*x^3*S + 2*x^3 + x^2*S - 5*x^2,
    12*x^3 + 6*x^2*S,
    S^2 + 5
]
```

From this, we see that E_x is generated by $(2+2\sqrt{-5})x^3+(-5+\sqrt{-5})x^2$ and $12x^3+6\sqrt{-5}x^2$.

106.6.2 Univariate Elimination Ideal Generators

`UnivariateEliminationIdealGenerator(I, i)`

Given a zero-dimensional ideal I of a polynomial ring P of rank n with $P = K[x_1,\ldots,x_n]$, together with an integer i with $1 \le i \le n$, return the unique monic generator of the univariate elimination ideal $I \cap K[x_i]$.

`UnivariateEliminationIdealGenerators(I)`

Given a zero-dimensional ideal I of a polynomial ring P of rank n with $P = K[x_1,\ldots,x_n]$, return the sequence of length n whose i-th element is the unique monic generator of the univariate elimination ideal $I \cap K[x_i]$.

Example H106E6

We construct an ideal I (derived from Neural networks theory) of the polynomial ring $\mathbf{Q}[x,y,z]$, and then find various elimination ideals of I.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P |
>     1 - x + x*y^2 - x*z^2,
>     1 - y + y*x^2 + y*z^2,
>     1 - z - z*x^2 + z*y^2 >;
> UnivariateEliminationIdealGenerator(I, 1);
x^21 - x^20 - 2*x^19 + 4*x^18 - 5/2*x^17 - 5/2*x^16 + 4*x^15 - 15/2*x^14 +
    129/16*x^13 + 11/16*x^12 - 103/8*x^11 + 131/8*x^10 - 49/16*x^9 -
    171/16*x^8 + 12*x^7 - 3*x^6 - 29/8*x^5 + 15/4*x^4 - 17/16*x^3 - 5/16*x^2
    + 5/16*x - 1/16
> UnivariateEliminationIdealGenerator(I, 2);
y^14 - y^13 - 13/2*y^12 + 8*y^11 + 53/4*y^10 - 97/4*y^9 - 45/8*y^8 + 33*y^7 -
    25/2*y^6 - 18*y^5 + 107/8*y^4 + 5/8*y^3 - 27/8*y^2 + 9/8*y - 1/8
> E :=  EliminationIdeal(I, {y, z});
> E;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Basis:
[
    z^21 - z^20 - 2*z^19 + 4*z^18 - 5/2*z^17 - 5/2*z^16 + 4*z^15 - 15/2*z^14 +
        129/16*z^13 + 11/16*z^12 - 103/8*z^11 + 131/8*z^10 - 49/16*z^9 -
```

```
        171/16*z^8 + 12*z^7 - 3*z^6 - 29/8*z^5 + 15/4*z^4 - 17/16*z^3 - 5/16*z^2
        + 5/16*z - 1/16,
    y + 141944208/7806653*z^20 - 42803108/7806653*z^19 - 290535348/7806653*z^18
        + 309392460/7806653*z^17 - 164881460/7806653*z^16 -
        331099258/7806653*z^15 + 203830442/7806653*z^14 - 894960798/7806653*z^13
        + 622205873/7806653*z^12 + 1352184655/31226612*z^11 -
        4746138097/31226612*z^10 + 5122044359/31226612*z^9 +
        991547639/31226612*z^8 - 830598655/7806653*z^7 + 1472712995/15613306*z^6
        - 59983627/15613306*z^5 - 486698319/15613306*z^4 + 174173263/7806653*z^3
        - 30672252/7806653*z^2 - 735083/664396*z + 30093391/31226612
]
```

Example H106E7

We write a simple function ZRadical to compute the radical of a zero dimensional ideal defined over a field using the univariate elimination ideal generators. See [BW93, p. 345].

```
> function ZRadical(I)
>     // Find radical of zero dimensional ideal I
>     P := Generic(I);
>     n := Rank(P);
>     G := UnivariateEliminationIdealGenerators(I);
>     N := {};
>
>     for i := 1 to n do
>         // Set FF to square-free part of the i-th univariate
>         // elimination ideal generator
>         F := G[i];
>         FF := F;
>         while true do
>             D := GCD(FF, Derivative(FF, 1, i));
>             if D eq 1 then
>                 break;
>             end if;
>             FF := FF div D;
>         end while;
>         // Include FF in N if FF is a proper divisor of F
>         if FF ne F then
>             Include(~N, FF);
>         end if;
>     end for;
>
>     // Return the sum of I and N
>     if #N eq 0 then
>       return I;
>     else
>       return ideal<P | I, N>;
>     end if;
```

```
> end function;
```

We now apply ZRadical to an ideal of $\mathbf{Q}[x, y, z]$.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | (x+1)^3*y^4, x*(y-z)^2+1, z^3-z^2>;
> R := ZRadical(I);
> Groebner(I);
> Groebner(R);
> I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0, Non-radical
Groebner basis:
[
    x - 4*y^9 + 21*y^8 - 32*y^7 + 7*y^6 + 432*y^5*z^2 - 546*y^5*z + 120*y^5 -
        137*y^4*z^2 + 288*y^4*z - 146*y^4 - 956*y^3*z^2 + 1088*y^3*z - 128*y^3 +
        393*y^2*z^2 - 576*y^2*z + 186*y^2 + 498*y*z^2 - 540*y*z + 44*y - 220*z^2
        + 288*z - 67,
    y^10 - 6*y^9 + 12*y^8 - 8*y^7 + 288*y^5*z^2 - 348*y^5*z + 60*y^5 -
        110*y^4*z^2 + 192*y^4*z - 82*y^4 - 624*y^3*z^2 + 696*y^3*z - 72*y^3 +
        273*y^2*z^2 - 384*y^2*z + 111*y^2 + 322*y*z^2 - 348*y*z + 26*y - 150*z^2
        + 192*z - 42,
    y^6*z - y^6 - 6*y^5*z^2 + 6*y^5*z - 3*y^4*z + 3*y^4 + 12*y^3*z^2 - 12*y^3*z
        + 3*y^2*z - 3*y^2 - 6*y*z^2 + 6*y*z - z + 1,
    z^3 - z^2
> R;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0, Radical
Groebner basis:
[
    x + 1,
    y^2 - 2*y*z + z - 1,
    z^2 - z
]
> I subset R;
true
> R subset I;
false
> IsInRadical(x + 1, I);
true
```

106.6.3 Relation Ideals

`RelationIdeal(Q)`

`RelationIdeal(Q, T)`

Given a sequence Q of k polynomials of a polynomial ring P over a ring S (not necessarily a field), return the relation ideal U of Q which is an ideal of the polynomial ring of rank k over S containing all algebraic relations between the elements of Q. That is, U consists of all polynomials $r \in S[y_1, \ldots, y_k]$ such that $r(Q[1], \ldots, Q[k]) = 0$. If U is desired to be an ideal of a particular polynomial ring T of rank k (to obtain predetermined names of variables, for example), then T may be passed as a second argument.

The computation is the same as that for the image of an affine polynomial map, which this basically is, thinking of the polynomials in Q as giving a map from n-dimensional affine space (n = rank of P) to k-dimensional affine space. k new variables y_i and relations $y_i - Q[i]$ are added and then the original variables x_i of P are eliminated in the usual way.

Example H106E8

We construct an ideal I of the polynomial ring $\mathbf{F}_2[x, y, z]$, and discover that the ideal is the full polynomial ring. Suppose we then wish to write $1 \in I$ as an (algebraic) expression in terms of the original generators. We use `RelationIdeal` to find that expression.

```
> P<x, y, z> := PolynomialRing(GF(2), 3, "grevlex");
> S := [(x + y + z)^2, (x^2 + y^2 + z^2)^3 + x + y + z + 1];
> I := ideal<P | S>;
> Groebner(I);
> I;
Ideal of Polynomial ring of rank 3 over GF(2)
Graded Reverse Lexicographical Order
Variables: x, y, z
Groebner basis:
[
    1
]
> Q<a, b> := PolynomialRing(GF(2), 2);
> U := RelationIdeal(S, Q);
> U;
Ideal of Polynomial ring of rank 2 over GF(2)
Order: Lexicographical
Variables: a, b
Inhomogeneous, Dimension >0
Basis:
[
    a^6 + a + b^2 + 1
```

```
]
```

Finally, we check the algebraic expression, evaluating it at the original polynomials:

```
> S[1]^6 + S[1] + S[2]^2;
1
```

106.7 Variable Extension of Ideals

Often one wishes to introduce new variables temporarily to a polynomial ring. MAGMA allows one to do this by use of the `VariableExtension` function, and also to restrict again to the original ring with elimination performed automatically.

`VariableExtension(I, k, b)`

`VariableExtension(I, k, b, order)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, create a polynomial ring Q as a k-variable extension of P, the ideal J of Q corresponding to I, and the embedding map $f : P \to Q$, and return J and f.

If the argument b (standing for "before") is `true`, the k variables are inserted before the current variables of P, so Q is defined to be $R[y_1, \ldots, y_k, x_1, \ldots, x_n]$ and f maps $P.i$ to $Q.(k+i)$ (so the x_i variables of P are mapped to the x_i variables of Q).

If the argument b is `false`, the k variables are inserted after the current variables of P, so Q is defined to be $R[x_1, \ldots, x_n, y_1, \ldots, y_k]$ and f maps $P.i$ to $Q.i$ (so the x_i variables of P are mapped to the x_i variables of Q).

If the argument *order* is given, then Q is constructed with the specified order; otherwise, the `grevlex` order is used for Q by default. See the section on monomial orders (Section 105.2) for the valid values for the argument *order*.

The image under f of a polynomial of P is the corresponding polynomial of Q, while the image under f of an ideal of P is the corresponding ideal of Q. The inverse image under f of a polynomial of Q is only defined if none of the extension variables of Q occur in that polynomial, in which case the inverse image is just the restriction back to P, while the inverse image under f of an ideal H of Q is always defined and is the restriction back to P of the elimination ideal $H \cap R[x_1, \ldots, x_n]$.

106.8 Homogenization of Ideals

MAGMA allows one to homogenize a polynomial ring or ideal by use of the `Homogenization` function, and also to restrict again to the original ring with elimination performed automatically.

`Homogenization(I, b)`

`Homogenization(I, b, order)`

`Homogenization(I)`

`Homogenization(I, order)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, create a polynomial ring H as a single variable extension of P, the homogenized ideal J of H corresponding to I, and the homogenization map $f : P \to H$, and return J and f.

If the argument b (standing for "before") is `true`, the homogenization variable is inserted before the current variables of P, so H is defined to be $R[h, x_1, \ldots, x_n]$ and f maps $P.i$ to $H.(k+i)$ (so the x_i variables of P are mapped to the x_i variables of H).

If the argument b is `false`, the homogenization variable is inserted after the current variables of P, so H is defined to be $R[x_1, \ldots, x_n, h]$ and f maps $P.i$ to $H.i$ (so the x_i variables of P are mapped to the x_i variables of H).

If the argument b is omitted, it is taken to be `false`, so the homogenization variable is introduced after the current variables of P.

If the argument *order* is given, then H is constructed with the specified order; otherwise, the `grevlex` order is used for H by default. See the section on monomial orders (Section 105.2) for the valid values for the argument *order*.

The image under f of a polynomial of P is the homogenization of f in H, while the image under f of an ideal of P is the homogenization ideal I^h in H. The inverse image under f of a polynomial of H is the restriction back to P (obtained by setting the homogenization variable to 1), while the inverse image under f of an ideal J of H is the restriction back to P of the ideal obtained by setting the homogenization variable to 1.

106.9 Extension and Contraction of Ideals

MAGMA allows the extension to and contraction from the ring of quotients of an ideal, defined over a field, with respect to certain variables. See [BW93, pp. 54–58 and 388–397] for the relevant definitions and theory.

`Extension(I, U)`

Given an ideal I of the polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field, together with a sequence U of integers each between 1 and n, create the (ring of quotients) extension Q of P, and return the ideal J of Q, together with the map $f : P \to Q$.

If U has length k and the values (in order) of U are $u_1, \ldots, u_k$, then first the rational function field $F = K(x_{u_1}, \ldots, x_{u_k})$ is constructed, then the list $v_1, \ldots, v_{n-k}$ is constructed as the list $1, \ldots, n$ with the u_i removed, and finally the extension Q of P is defined to be the polynomial ring $F[x_{v_1}, \ldots, x_{v_{n-k}}] = K(x_{u_1}, \ldots, x_{u_k})[x_{v_1}, \ldots, x_{v_{n-k}}]$.

The map f is constructed in the obvious way so that x_i is mapped to the appropriate variable in F if i is in U, or the appropriate variable in Q otherwise. The image under f of an ideal of P is just the appropriate ideal of Q whose basis is obtained by taking the image under f of each of the polynomials in the basis of I.

The inverse image under f of a polynomial of Q is obtained by first making the polynomial monic, then multiplying by the LCM of the denominators ("clearing the denominators"), then mapping each variable back to the appropriate one in P—this is possible since there are no proper denominators. The inverse image under f of an ideal H of Q is defined to be the ideal of P generated by the inverse images under f of the polynomials in the basis of H (note that this is not always equal to the contraction of H—see [BW93, p. 389], for a simple algorithm to compute the contraction of an ideal of Q).

106.10 Dimension of Ideals

Let I be an ideal of the polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field. Let X be the set $\{x_1, \ldots, x_n\}$ of variables of P. A subset U of X is called *independent modulo* I if $I \cap K[U] = \emptyset$. A subset U of X is called *maximally independent modulo* I if U is independent modulo I, and no proper superset of U is independent modulo I. The *dimension* of I is defined to be the maximum of the cardinalities of all the independent sets modulo I. It is not too hard to see in this case that this coincides with the more abstract commutative algebra definition of the Krull dimension of the *quotient algebra* P/I as the maximal length of a chain of prime ideals.

Note that the definition given above of zero-dimensionality (as the case when the quotient of P by I has finite dimension as a vector space over the coefficient field) coincides with the definition of zero-dimensionality as dimension 0.

`Dimension(I)`

Given an ideal I of a polynomial ring P defined over a field, return the dimension d of I, together with a (sorted) sequence U of integers of length d such that the variables of P corresponding to the integers of U constitute a maximally independent set modulo I. If I is the full polynomial ring P, the dimension is defined to be -1, and the second return value is not set. The algorithm implemented is that given in [BW93, p. 449].

106.11 Radical and Decomposition of Ideals

MAGMA has algorithms for computing the full radical and the primary decomposition of ideals. See [BW93, chapter 8], for the relevant definitions and theory. The implementation of the algorithms presented here in MAGMA was based on the algorithms presented in that chapter. Currently these algorithms work only for ideals of polynomial rings over fields (Euclidean rings will be supported in the future).

There are also functions for some easier decompositions than the full primary decomposition: radical decompositions, equidimensional decompositions and triangular decompositions for zero-dimensional ideals. The theory behind these is discussed in the relevant function description.

106.11.1 Radical

`Radical(I)`

Given an ideal I of a polynomial ring P over a field, return the radical of I. The radical R of I is defined to be the set of all polynomials $f \in P$ such that $f^n \in I$ for some $n \geq 1$. The radical R is also an ideal of P, containing I. The function works for an ideal defined over any field over which polynomial factorization is available.

Example H106E9

We compute the radical of an ideal of $\mathbf{Q}[x, y, z, t, u]$ (which is not zero-dimensional).

```
> P<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<P |
> x + y + z + t + u,
> x*y + y*z + z*t + t*u + u*x,
> x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
> x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
> x*y*z*t*u>;
> R := Radical(I);
> Groebner(R);
> R;
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension >0, Radical
Groebner basis:
[
    x + y + z + t + u,
    y^2 + y*t - z*u - u^2,
    y*z,
    y*u + z*u + u^2,
    z^2*u + z*u^2,
    z*t,
    t*u
]
```

```
> // Check that t*u is in the radical of I by another method:
> IsInRadical(t*u, I);
true
```

106.11.2 Primary Decomposition

`PrimaryDecomposition(I)`

Given an ideal I of a polynomial ring over a field, return the primary decomposition of I, and also the sequence of associated prime ideals. See IsPrimary for the definition of a primary ideal. The primary decomposition of I is returned as two parallel sequences Q and P of ideals, both of length k, having the following properties:

(a) The ideals of Q are primary.

(b) The intersection of the ideals of Q is I.

(c) The ideals of P are the associated primes of Q; i.e., $P[i]$ is the radical of $Q[i]$ (so $P[i]$ is prime) for $1 \leq i \leq k$.

(d) Q is minimal: no ideal of Q contains the intersection of the rest of the ideals of Q and the associated prime ideals in P are distinct.

(e) Q and P are sorted so that P is always unique and Q is unique if I is zero-dimensional. If I is not zero-dimensional, then an embedded component of Q (one whose associated prime contains another associated prime from P) will not be unique in general. Yet MAGMA will always return the same values for Q and P, given the same I.

The function works for an ideal defined over any field over which polynomial factorization is available (inseparable field extensions are handled by an algorithm due to Allan Steel [Ste05]).

NB: if one only wishes to compute the prime components P, then the next function RadicalDecomposition should be used, since this may be much more efficient.

`RadicalDecomposition(I)`

Given an ideal I of a polynomial ring over a field, return the prime decomposition of the radical of I. This is equivalent to applying the function PrimaryDecomposition to the radical of I, but may be a much more efficient than using that method. The (prime) radical decomposition of I is returned as a sequence P of ideals of length k having the following properties:

(a) The ideals of P are prime.

(b) The intersection of the ideals of P is the radical of I.

(c) P is minimal: no ideal of P contains the intersection of the rest of the ideals of P.

(e) P is sorted so that P is always unique. Thus MAGMA will always return the same values for P, given the same I.

The function works for an ideal defined over any field over which polynomial factorization is available.

`ProbableRadicalDecomposition(I)`

Given an ideal I of a polynomial ring P over a field, return a probabilistic prime decomposition of the radical of I as a sequence of ideals of P. This function is like the function `RadicalDecomposition` except that the ideals returned may not be prime, but the time taken may be much less.

`MinimalDecomposition(S)`

Given a set or sequence S of ideals of a polynomial ring over a field, with $I = \cap_{J \in S} J$ (so that S describes a decomposition of I), return a minimal decomposition M of I as a subset of S such that $I = \cap_{J \in M} J$ also (so none of the ideals in the decomposition M are redundant).

`SetVerbose("Decomposition", v)`

Change verbose printing for the (Primary/Radical) Decomposition algorithm to be v. Currently the legal values for v are `true`, `false`, 0, 1, or 2.

Example H106E10

We compute the primary decomposition of the same ideal of $\mathbf{Q}[x, y, z, t, u]$ (which is not zero-dimensional).

```
> P<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<P |
> x + y + z + t + u,
> x*y + y*z + z*t + t*u + u*x,
> x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
> x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
> x*y*z*t*u>;
> IsZeroDimensional(I);
false
> Q, P := PrimaryDecomposition(I);
```

We next print out the primary components Q and associated primes P.

```
> Q;
[
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
    Groebner basis:
    [
        x + 1/2*z + 1/2*u,
        y + 1/2*z + 1/2*u,
        z^2 + 2*z*u + u^2,
```

```
    t
],
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
Groebner basis:
[
    x + 2*z + t,
    y - z,
    z^2,
    u
],
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
Groebner basis:
[
    x + z + 2*u,
    y,
    t - u,
    u^2
],
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
Groebner basis:
[
    x - u,
    y + t + 2*u,
    z,
    u^2
],
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Homogeneous, Dimension 1, Non-radical, Primary, Non-prime
Groebner basis:
[
    x,
    y + 2*t + u,
    z - t,
    t^2
],
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
```

```
Variables: x, y, z, t, u
Homogeneous, Dimension 0, Non-radical, Primary, Non-prime
Size of variety over algebraically closed field: 1
Groebner basis:
[
    x + y + z + t + u,
    y^2 + y*t + 2*y*u - z*t + z*u + u^2,
    y*z^2 - y*z*t + y*t*u - y*u^2 + z^2*t - z^2*u + z*t*u - 2*z*u^2 + t^2*u
        + t*u^2 - u^3,
    y*z*t^2 - 2*y*z*u^2 + 3*y*t*u^2 - 2*y*u^3 + z^3*t - z^3*u - z^2*t^2 +
        4*z^2*t*u - 4*z^2*u^2 + z*t^2*u + 2*z*t*u^2 - 5*z*u^3 + 3*t^2*u^2 +
        2*t*u^3 - 2*u^4,
    y*z*t*u + y*z*u^2 - y*t^2*u - 4*y*t*u^2 + 3*y*u^3 - z^3*t + z^3*u +
        z^2*t^2 - 3*z^2*t*u + 4*z^2*u^2 - 2*z*t*u^2 + 6*z*u^3 - t^3*u -
        5*t^2*u^2 - 3*t*u^3 + 3*u^4,
    y*z*u^3 - 5/2*y*t*u^3 + 3/2*y*u^4 + 1/4*z^3*t^2 + 1/2*z^3*u^2 -
        3/4*z^2*t^3 + 5/4*z^2*t^2*u - 1/4*z^2*t*u^2 + 9/4*z^2*u^3 -
        9/4*z*t^3*u + 1/4*z*t^2*u^2 - 3/4*z*t*u^3 + 13/4*z*u^4 - t^3*u^2 -
        5/2*t^2*u^3 - 7/4*t*u^4 + 3/2*u^5,
    y*t^3*u - 17/4*y*t*u^3 + 13/4*y*u^4 + 1/8*z^3*t^2 + 5/4*z^3*u^2 -
        19/8*z^2*t^3 + 13/8*z^2*t^2*u - 5/8*z^2*t*u^2 + 33/8*z^2*u^3 -
        33/8*z*t^3*u - 7/8*z*t^2*u^2 - 31/8*z*t*u^3 + 49/8*z*u^4 + t^4*u +
        1/2*t^3*u^2 - 15/4*t^2*u^3 - 31/8*t*u^4 + 13/4*u^5,
    y*t^2*u^2 - 3/4*y*t*u^3 - 1/4*y*u^4 + 3/8*z^3*t^2 - 1/4*z^3*u^2 -
        1/8*z^2*t^3 + 7/8*z^2*t^2*u + 1/8*z^2*t*u^2 - 5/8*z^2*u^3 -
        3/8*z*t^3*u + 11/8*z*t^2*u^2 - 5/8*z*t*u^3 - 5/8*z*u^4 + 1/2*t^3*u^2
        + 3/4*t^2*u^3 - 5/8*t*u^4 - 1/4*u^5,
    y*t*u^4 - 2/3*z^2*t^4 + 13/15*z^2*t^2*u^2 - 1/5*z^2*t*u^3 -
        31/15*z*t^4*u + 3/5*z*t^3*u^2 - 2/5*z*t^2*u^3 + 23/15*z*t*u^4 -
        3/5*t^4*u^2 + 2/15*t^3*u^3 - 1/3*t^2*u^4 + t*u^5,
    y*u^5 - 1/2*z^2*t^4 - 1/2*z^2*t^2*u^2 + 1/2*z^2*t*u^3 + 1/2*z^2*u^4 -
        3/2*z*t^4*u - 3*z*t^3*u^2 + 5/2*z*t*u^4 + 3/2*z*u^5 - 1/2*t^4*u^2 -
        2*t^3*u^3 - 2*t^2*u^4 + 1/2*t*u^5,
    z^7,
    z^4*t - z^4*u - z^3*t^2 - 3*z^3*u^2 + 2*z^2*t^3 + 2*z^2*t^2*u -
        9*z^2*t*u^2 - 3*z^2*u^3 + 7*z*t^3*u + 2*z*t^2*u^2 - z*u^4 +
        2*t^3*u^2 - t^2*u^3 + t*u^4,
    z^4*u^2 + 7/3*z^2*t^4 - 40/3*z^2*t^2*u^2 + 8*z^2*t*u^3 - 3*z^2*u^4 +
        22/3*z*t^4*u - 20*z*t^3*u^2 + 2*z*t^2*u^3 + 31/3*z*t*u^4 - 2*z*u^5 +
        t^4*u^2 - 41/3*t^3*u^3 - 10/3*t^2*u^4 + 2*t*u^5,
    z^3*t^3 + 1/3*z^2*t^4 + 2/3*z^2*t^2*u^2 + z^2*t*u^3 + 1/3*z*t^4*u -
        2*z*t^3*u^2 - z*t^2*u^3 + 1/3*z*t*u^4 - 2/3*t^3*u^3 - 1/3*t^2*u^4,
    z^3*t*u - z^2*t^3 + 3*z^2*t*u^2 - 3*z*t^3*u + z*t*u^3 - t^3*u^2,
    z^3*u^3 - 1/3*z^2*t^4 + 7/3*z^2*t^2*u^2 - 2*z^2*t*u^3 + 2*z^2*u^4 -
        4/3*z*t^4*u + 7*z*t^3*u^2 - 2*z*t^2*u^3 - 13/3*z*t*u^4 + z*u^5 +
        14/3*t^3*u^3 + 4/3*t^2*u^4 - t*u^5,
    z^2*t^5 - 3*z*t*u^5 + 17/2*t^5*u^2 + 33/2*t^4*u^3 + 9*t^3*u^4 +
        15/2*t^2*u^5,
```

```
        z^2*t^3*u - z^2*t^2*u^2 + z*t^3*u^2,
        z^2*t^2*u^3 - 4/5*z*t*u^5 + 16/5*t^5*u^2 + 59/10*t^4*u^3 -
            11/10*t^3*u^4,
        z^2*t*u^4 - 4/5*z*t*u^5 + 47/10*t^5*u^2 + 42/5*t^4*u^3 - 31/10*t^3*u^4 -
            1/2*t^2*u^5,
        z^2*u^5 + 6*z*t*u^5 - 2*t^5*u^2 - 4*t^4*u^3 - 4*t^3*u^4 - 7*t^2*u^5,
        z*t^5*u + z*t*u^5 - 5/2*t^5*u^2 - 11/2*t^4*u^3 - 3*t^3*u^4 -
            5/2*t^2*u^5,
        z*t^4*u^2 + 2/5*z*t*u^5 - 11/10*t^5*u^2 - 17/10*t^4*u^3 - 1/5*t^3*u^4 -
            1/2*t^2*u^5,
        z*t^3*u^3 + 1/5*z*t*u^5 - 3/10*t^5*u^2 - 3/5*t^4*u^3 - 1/10*t^3*u^4 -
            1/2*t^2*u^5,
        z*t^2*u^4 + 2/5*z*t*u^5 - 8/5*t^5*u^2 - 16/5*t^4*u^3 - 1/5*t^3*u^4,
        t^6,
        t^5*u^3,
        t^4*u^4,
        t^3*u^5,
        u^6
    ]
]
> P;
[
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x,
        y,
        z + u,
        t
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x + t,
        y,
        z,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension >0, Radical, Prime
```

```
    Groebner basis:
    [
        x + z,
        y,
        t,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x,
        y + t,
        z,
        u
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 1, Radical, Prime
    Groebner basis:
    [
        x,
        y + u,
        z,
        t
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Homogeneous, Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 1
    Groebner basis:
    [
        x,
        y,
        z,
        t,
        u
    ]
]
```

Notice that $P[6]$ contains other ideals of P so $Q[6]$ is an embedded primary component of I. Thus the first 5 ideals of Q would the same be in any primary decomposition of I, while $Q[6]$ could be different in another primary decomposition of I. Finally, notice that the prime decomposition of the radical of I is the same as P except for the removal of $P[6]$ to satisfy the uniqueness condition.

The structure of the variety of I can be easily understood by examining the prime decomposition of the radical.

```
> RP := RadicalDecomposition(I);
> #RP;
5
> Set(RP) eq { P[i]: i in [1 .. 5] };
true
```

106.11.3 Triangular Decomposition

Let T be a zero-dimensional ideal of the polynomial ring $K[x_1, \ldots, x_n]$, where K is a field. T is called *triangular* if its Gröbner basis G has length n and the initial term of the i-th polynomial of G is of the form $x_i^{e_i}$ for each i. Any *radical* zero-dimensional ideal has a decomposition as an intersection of triangular ideals. The algorithm in MAGMA for primary decomposition now (since V2.4) first computes a triangular decomposition and then decomposes each triangular component to primary ideals since the computation of a triangular decomposition is usually fast. See [Laz92] for further discussion of triangular ideals.

`TriangularDecomposition(I)`

Given a zero-dimensional ideal I of a polynomial ring over a field with *lexicographical* order, return a triangular decomposition of I as a sequence Q of ideals such that the intersection of the ideals of Q equals I and for each ideal J of Q which is radical, J is triangular (see above for the definition of a triangular ideal). A second return value indicates whether I is proven to be radical. If I is radical, all entries of Q are triangular. Computing a triangular decomposition will often be faster than computing the full primary decomposition and may yield sufficient information for a specific problem. The algorithm implemented is that given in [Laz92].

Example H106E11

We compute the triangular decomposition of the (radical) Cyclic-5 roots ideal and compare it with the full primary decomposition of the same ideal.

```
> R<x, y, z, t, u> := PolynomialRing(RationalField(), 5);
> I := ideal<R |
>     x + y + z + t + u,
>     x*y + y*z + z*t + t*u + u*x,
>     x*y*z + y*z*t + z*t*u + t*u*x + u*x*y,
>     x*y*z*t + y*z*t*u + z*t*u*x + t*u*x*y + u*x*y*z,
>     x*y*z*t*u - 1>;
> IsRadical(I);
true
> time T := TriangularDecomposition(I);
Time: 0.000
```

```
> time Q, P := PrimaryDecomposition(I);
Time: 0.010
> #T;
9
> #Q;
20
```

So we notice that although I decomposes into 9 triangular ideals, some of these ideals must decompose further since the primary decomposition consists of 20 prime ideals. We examine the first entry of T. Notice that it is at least triangular (it has 5 polynomials and for each variable there is a polynomial whose leading monomial is a power of that variable).

```
> T[1];
Ideal of Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: x, y, z, t, u
Inhomogeneous, Dimension 0
Groebner basis:
[
    x - 6/5*t^5 - 4*t^4 - 3*t^3 - 3*t^2 - 3*t - 9/5,
    y - 2/5*t^5 - 2*t^4 - 3*t^3 - 2*t^2 - 2*t - 8/5,
    z + 8/5*t^5 + 6*t^4 + 6*t^3 + 5*t^2 + 6*t + 22/5,
    t^6 + 4*t^5 + 5*t^4 + 5*t^3 + 5*t^2 + 4*t + 1,
    u - 1
]
> IsPrimary(T[1]);
false
> D := PrimaryDecomposition(T[1]);
> #D;
2
> D;
[
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Inhomogeneous, Dimension 0, Radical, Prime
    Size of variety over algebraically closed field: 2
    Groebner basis:
    [
        x - 1,
        y - 1,
        z + t + 3,
        t^2 + 3*t + 1,
        u - 1
    ],
    Ideal of Polynomial ring of rank 5 over Rational Field
    Order: Lexicographical
    Variables: x, y, z, t, u
    Inhomogeneous, Dimension 0, Radical, Prime
```

```
    Size of variety over algebraically closed field: 4
    Groebner basis:
    [
        x + t^3 + t^2 + t + 1,
        y - t^3,
        z - t^2,
        t^4 + t^3 + t^2 + t + 1,
        u - 1
    ]
]
```

106.11.4 Equidimensional Decomposition

`EquidimensionalPart(I)`

`EquidimensionalDecomposition(I)`

`FineEquidimensionalDecomposition(I)`

Let I be an ideal of a polynomial ring P over a field. Currently for the two decomposition functions, it is *assumed* that I has no embedded associated primes (e.g., when I is radical). In this case, it can be much faster to compute an equidimensional decomposition rather than a full primary or radical one. The equidimensional decomposition is the set of ideals which are the intersections of all primary components of I associated to primes of the same dimension. This decomposition (often trivial) is useful for certain constructions involving the Jacobian ideal.

The first function just computes the highest-dimensional decomposition component. The second performs the straight decomposition. The third gives a slightly finer decomposition for the convenience of some applications. In it, each equidimensional component is possibly further split so that, for each final equidimensional factor there is a single set of variables which constitute a maximally independent set of every primary component of the factor (*cf* `Dimension` on page 3244). A sequence of pairs consisting of each factor and the indices of its set of variables is returned.

The algorithm from [GP02] is used in the general case. When I is homogeneous, a faster, more module-theoretic method is employed for the first two functions. This involves first expressing P/I as a finite module M over a linear Noether Normalisation (described in the next section) S of I. Then if $E(I)$ is the equidimensional part of I, $E(I)/I$ as a submodule of M is equal to the kernel of the natural map of M to its double dual over S, $Hom_S(Hom_S(M,S),S)$. Working with modules over S rather than over P here allows the "reduction to dimension 0". We could directly over P, doing a similar computation but with Hom_S replaced by some Ext^i_P (see [EHV92]).

Example H106E12

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> P1 := ideal<P|x*y+y*z+z*x>; // dimension 2 prime
> P2 := ideal<P|x^2+y,y*z+2>; // dimension 1 prime
> P3 := ideal<P|x*y-1,y+z>;   // dimension 1 prime
> I := P1 meet P2 meet P3;
> time rd := RadicalDecomposition(I);
Time: 3.720
> time ed := EquidimensionalDecomposition(I);
Time: 0.070
> #ed;
2
> ed[1] eq P1;
true
> ed[2] eq (P2 meet P3);
true
```

106.12 Normalisation and Noether Normalisation

Suppose I is an ideal of $P = K[x_1, \ldots, x_n]$ with K a field, and I has dimension d.

A Noether normalisation of I is given by a set of d polynomials $f_1, \ldots, f_d$ of P, algebraically independent over K, for which $K[f_1, \ldots, f_d] \cap I = 0$ and $K[f_1, \ldots, f_d] \to P/I$ is an integral extension. These always exist and if K is an infinite field, the f_i can be chosen to be linear expressions in the x_i.

If I is radical, then the normalisation of I here will refer to the integral closure of the affine ring P/I in its total ring of fractions. If $I = \bigcap P_i$ with P_i prime, then the normalisation is equal to the finite direct product of the normalisations of the P_i as affine rings. It will be specified by a list of pairs (I_i, ϕ_i) where I_i is a prime ideal with generic ring G_i, a multivariate polynomial ring over K, and ϕ_i a homomorphism from P to G_i. The pairs represent the normalisation of each P_i and the inclusion $P/I \to \prod G_i/I_i$ induced by the ϕ_i makes the RHS the integral closure of P/I.

106.12.1 Noether Normalisation

`NoetherNormalisation(I)`

`NoetherNormalization(I)`

This function attempts to compute a Noether Normalisation for I, as described above, using *linear* combinations of the variables. The function is guaranteed to work if K has characteristic zero but may fail in unlucky cases in small characteristic.

The algorithm followed is basically that given in [GP02] but with a simpler test for homogeneous ideals I, which gives a speed-up in that case. Also, subsets of the full sets of variables are considered before more general linear combinations.

The return values are

1) the sequence $[f_1, \ldots, f_d]$.
2) h, an automorphism $P \to P$ given by a linear change of variables which maps the f_i to the last d variables of P. Thus $x_{n-d+1}, \ldots, x_n$ are a corresponding Noether normalising set of polynomials for $h(I)$.
3) the inverse of h.

Example H106E13

```
> P<x,y> := PolynomialRing(RationalField(),2);
> I := ideal<P | x*y+x+2>;
> fs,h,hinv := NoetherNormalisation(I);
> fs;
[
    x + y
]
> J := ideal<P | [h(b) : b in Basis(I)]>; J;
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: x, y
Basis:
[
    -x^2 + x*y + x + 2
]
> // clearly x is integral over the last variable y in P/J
```

106.12.2 Normalisation

`Normalisation(I)`

`Normalization(I)`

`UseFF`	BoolElt	*Default :* `true`
`FFMin`	BoolElt	*Default :* `true`
`UseMax`	BoolElt	*Default :* `false`

This function computes the normalisation of the ideal I and returns the result as a list of pairs as described above. The ideal I must be radical - this is not checked in the function. Also the base field K must be perfect.

There are several options. The general algorithm used is that of De Jong as described in [GP02]. However, if the generic polynomial ring P of I has rank 2 then Magma's powerful function field machinery can be applied to give a generally much faster algorithm. This is the default behaviour but can be bypassed by setting the parameter `UseFF` to `false`.

When the function field machinery is used, a correct result can be obtained extremely quickly, but the generic spaces of the solution ideals can be of quite high

dimension. The default behaviour, controlled by the parameter `FFMin`, is to use a filtration by Riemann-Roch spaces to try to find a roughly minimal number of generators of the algebras G_i/I_i and return the corresponding ideal in the smaller number of variables as a more optimal presentation of the solution. This takes more time but is still usually faster than the general algorithm and tends to produce much nicer results. In some cases, the minimised solution is the same as the basic one but takes longer to generate. The minimising stage can be cut out by setting `FFMin` to `false`.

The general algorithm can avoid doing some work if it is known that certain conditions on I hold. One standard condition is that $I \subseteq M =< x_1, \ldots, x_n >$ and that P/I is locally normal away from M. This holds, for example, if the affine variety defined by I in K^n is non-singular except at the origin. If this is known, then parameter `UseMax` can be set to `true` which will usually speed up the general algorithm (it has no effect if the function field method is used). However, if P/I is locally non-normal at other primes then this will produce an incorrect result.

Example H106E14

```
> P<x,y> := PolynomialRing(RationalField(),2);
> // we begin with a very simple example (prime ideal)
> I := Ideal((x - y^2)^2 - x*y^3);
> time Js := Normalisation(I); // function field method
Time: 0.010
> #Js;
1
> N := Js[1][1];
> N<[a]> := N;
> N;
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: a[1], a[2]
Basis:
[
    -a[1]*a[2] + a[2]^2 - 2*a[2] + 1
]
> // Now try the basic function field method
> time Js := Normalisation(I: FFMin:=false);
Time: 0.010
> //get the same result here either way
> N := Js[1][1];
> N<[a]> := N;
> N;
Ideal of Polynomial ring of rank 2 over Rational Field
Order: Lexicographical
Variables: a[1], a[2]
Basis:
```

```
[
    -a[1]*a[2] + a[2]^2 - 2*a[2] + 1
]
> time Js := Normalisation(I:UseFF:=false); // try the general method
Time: 0.120
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: $.1, $.2, $.3, $.4
Groebner basis:
[
    $.1^2 + 2*$.1 + $.2 - 1,
    $.1*$.2 - 2*$.1 + 2*$.2 - $.3 + $.4 - 4,
    $.1*$.3 + $.3*$.4 + 2*$.3 - $.4^2,
    $.1*$.4 - $.2 + 2,
    $.2^2 - 4*$.2 - $.3*$.4 - 2*$.3 + $.4^2 + 4,
    $.2*$.3 + $.3*$.4^2 + 2*$.3*$.4 - 2*$.3 - $.4^3,
    $.2*$.4 + $.3 - 2*$.4,
    $.3^2 - $.3*$.4^3 - 2*$.3*$.4^2 + $.4^4
]
> // try the general method with UseMax (which applies here)
> time Js := Normalisation(I:UseFF:=false,UseMax:=true);
Time: 0.040
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: $.1, $.2, $.3
Groebner basis:
[
    $.1^2 - 4*$.1 - $.2*$.3 - 2*$.2 + $.3^2 + 4,
    $.1*$.2 + $.2*$.3^2 + 2*$.2*$.3 - 2*$.2 - $.3^3,
    $.1*$.3 + $.2 - 2*$.3,
    $.2^2 - $.2*$.3^3 - 2*$.2*$.3^2 + $.3^4
]
> // now try a harder case - a singular affine form of modular curve X1(11)
> I := ideal<P | (x-y)*x*(y+x^2)^3-y^3*(x^3+x*y-y^2)>;
> time Js := Normalisation(I: FFMin := false);
Time: 0.110
> #Js;
1
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 5 over Rational Field
```

```
Lexicographical Order
Variables: $.1, $.2, $.3, $.4, $.5
Groebner basis:
[
    $.1*$.3 - $.1 - 6*$.3 + $.4*$.5^2 - 4*$.4*$.5 + 6*$.4 - $.5^5 + $.5^4 +
        11*$.5^3 - 16*$.5^2 + 2*$.5 + 6,
    $.1*$.4 + 2*$.3 - $.4*$.5^2 + 2*$.4*$.5 - 2*$.4 + $.5^4 - 4*$.5^3 + 4*$.5^2
        - 2,
    $.1*$.5 - 2*$.3 + $.4 + $.5^3 - 2*$.5^2 + $.5 + 1,
    $.2 - $.3 + $.5^3 - $.5^2,
    $.3^2 + 3*$.3 - 2*$.4*$.5^2 + 4*$.4*$.5 - 4*$.4 - $.5^6 + 2*$.5^5 + $.5^4 -
        10*$.5^3 + 10*$.5^2 - 4,
    $.3*$.4 - $.3 - $.4*$.5^3 + $.4*$.5^2 - $.4*$.5 + $.4 - $.5^4 + 2*$.5^3 -
        2*$.5^2 + 1,
    $.3*$.5 + $.3 - $.4 - $.5^4 + 2*$.5^2 - $.5 - 1,
    $.4^2 - 2*$.4*$.5^2 + $.4*$.5 + $.4 - $.5^5
]
> time Js := Normalisation(I);
Time: 1.110
> J := Js[1][1];
> Groebner(J);
> J;
Ideal of Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: $.1, $.2
Groebner basis:
[
    $.1^2*$.2 + 2*$.1*$.2 + $.1 - $.2^2 + 2*$.2 + 1
]
> // Minimised result is a cubic equation in K[x,y] - as good as we could get!
> // This example takes MUCH longer with the general method - even setting
> // UseMax := true.
```

106.13 Hilbert Series and Hilbert Polynomial

Let I be a *homogeneous* ideal of the graded polynomial ring $P = K[x_1, \ldots, x_n]$, where K is a field. Then the quotient ring P/I is a *graded* vector space in the following way: P/I is the direct sum of the vector spaces V_d for $d = 0, 1, \ldots$ where V_d is the K-vector space consisting of all homogeneous polynomials in P/I (i.e., reduced residues of polynomials of P with respect to I) of weighted degree d. The *Hilbert Series* of the graded vector space P/I is the generating function

$$H_{P/I}(t) = \sum_{d=0}^{\infty} \dim(V_d) t^d.$$

The Hilbert series can be written as a rational function in the variable t.

If the weights on the variables of P are all 1, then there also exists the Hilbert polynomial $F_{P/I}(d)$ corresponding to the Hilbert series $H_{P/I}(t)$ which is a univariate polynomial in $\mathbf{Q}[d]$ such that $F_{P/I}(i)$ is equal to the coefficient of t^i in the Hilbert series for all $i \geq k$ for some fixed k.

`HilbertSeries(I)`

Given an homogeneous ideal I of a polynomial ring P over a field, return the Hilbert series $H_{P/I}(t)$ of the quotient ring P/I as an element of the univariate function field $\mathbf{Z}(t)$ over the ring of integers. The algorithm implemented is that given in [BS92].

Note that this is equivalent to `HilbertSeries(QuotientModule(I))`, while if one wishes the Hilbert series of I considered as a P-module, one should call `HilbertSeries(Submodule(I))`.

`HilbertSeries(I, p)`

Given an homogeneous ideal I of a polynomial ring P over a field, return the Hilbert series $H_{P/I}(t)$ of the quotient ring P/I as a power series to precision p.

`HilbertDenominator(I)`

Given an homogeneous ideal I of a polynomial ring P over a field, return the unreduced Hilbert denominator D of P/I (as a univariate polynomial over the ring of integers). The denominator D equals

$$\prod_{i=1}^{n}(1 - t^{w_i}),$$

where n is the rank of P and w_i is the weight of the i-th variable (1 by default).

`HilbertNumerator(I)`

Given an homogeneous ideal I of a polynomial ring P over a field, return the unreduced Hilbert numerator N of P/I (as a univariate polynomial over the ring of integers). The numerator N equals $D \times \times H_{P/I}(t)$, where D is the unreduced Hilbert denominator above. Computing with the unreduced numerator is often more convenient.

`HilbertPolynomial(I)`

Given an homogeneous ideal I of a polynomial ring P over a field with weight 1 for each variable, return the Hilbert polynomial $H(d)$ of the quotient ring P/I as an element of the univariate polynomial ring $Q[d]$, together with the index of regularity of P/I (the minimal integer $k \geq 0$ such that $H(d)$ agrees with the Hilbert function of P/I at d for all $d \geq k$).

Example H106E15

We compute the Hilbert series and Hilbert polynomial for an ideal corresponding to the square of a matrix (see [BS92]).

```
> MatSquare := function(n)
>     P := PolynomialRing(RationalField(), n * n, "grevlex");
>     AssignNames(
>         ~P,
>         ["x" cat IntegerToString(i) cat IntegerToString(j): i, j in [1..n]]
>     );
>     M := MatrixRing(P, n);
>     X := M ! [P.((i - 1) * n + j): i, j in [1 .. n]];
>     Y := X^2;
>     return ideal<P | [Y[i][j]: i, j in [1 .. n]]>;
> end function;
> I := MatSquare(4);
> I;
Ideal of Polynomial ring of rank 16 over Rational Field
Order: Graded Reverse Lexicographical
Variables: x11, x12, x13, x14, x21, x22, x23, x24, x31, x32,
    x33, x34, x41, x42, x43, x44
Homogeneous
Basis:
[
    x11^2 + x12*x21 + x13*x31 + x14*x41,
    x11*x12 + x12*x22 + x13*x32 + x14*x42,
    x11*x13 + x12*x23 + x13*x33 + x14*x43,
    x11*x14 + x12*x24 + x13*x34 + x14*x44,
    x11*x21 + x21*x22 + x23*x31 + x24*x41,
    x12*x21 + x22^2 + x23*x32 + x24*x42,
    x13*x21 + x22*x23 + x23*x33 + x24*x43,
    x14*x21 + x22*x24 + x23*x34 + x24*x44,
    x11*x31 + x21*x32 + x31*x33 + x34*x41,
    x12*x31 + x22*x32 + x32*x33 + x34*x42,
    x13*x31 + x23*x32 + x33^2 + x34*x43,
    x14*x31 + x24*x32 + x33*x34 + x34*x44,
    x11*x41 + x21*x42 + x31*x43 + x41*x44,
    x12*x41 + x22*x42 + x32*x43 + x42*x44,
    x13*x41 + x23*x42 + x33*x43 + x43*x44,
    x14*x41 + x24*x42 + x34*x43 + x44^2
]
> S<t> := HilbertSeries(I);
> S;
(t^12 - 7*t^11 + 20*t^10 - 28*t^9 + 14*t^8 + 15*t^7 - 20*t^6 +
    19*t^5 - 22*t^4 + 7*t^3 + 20*t^2 + 8*t + 1)/(t^8 - 8*t^7 +
    28*t^6 - 56*t^5 + 70*t^4 - 56*t^3 + 28*t^2 - 8*t + 1)
> H<d>, k := HilbertPolynomial(I);
> H, k;
```

```
1/180*d^7 + 7/90*d^6 + 293/360*d^5 + 61/36*d^4 + 1553/360*d^3 +
    851/180*d^2 + 101/30*d + 1
5
> // Check that evaluations of H for d >= 5 match coefficients of S:
> L<u> := LaurentSeriesRing(IntegerRing());
> L;
Laurent Series Algebra over Integer Ring
> L ! S;
1 + 16*u + 120*u^2 + 575*u^3 + 2044*u^4 + 5927*u^5 + 14832*u^6 +
    33209*u^7 + 68189*u^8 + 130642*u^9 + 236488*u^10 + 408288*u^11 +
    677143*u^12 + 1084929*u^13 + 1686896*u^14 + 2554659*u^15 +
    3779609*u^16 + 5476772*u^17 + 7789144*u^18 + 10892530*u^19 +
    O(u^20)
> Evaluate(H, 5);
5927
> Evaluate(H, 6);
14832
> Evaluate(H, 19);
10892530
```

106.14 Syzygies

The main functions to compute syzygies work with or return modules. See Chapter 109 for these. This section contains a variant that returns a basis of syzygies of a polynomial sequence as rows of a matrix.

`SyzygyMatrix(Q)`

Given a sequence Q of polynomials from a multivariate polynomial ring P, return the module of syzygies of Q as a matrix S. This an r by k matrix, where k is the length of Q, whose rows span the space of all vectors v such that the sum of $v[i] * Q[i]$ for $i = 1, \ldots k$ is zero. The algorithm used is the standard one, computing a module Gröbner basis with respect to a particular elimination order (see section 2.5 of [GP02], for example). The base ring may be a field or Euclidean ring.

Example H106E16

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> SyzygyMatrix([x + y, x - y, x*z + y*z]);
[             z              0             -1]
[ 1/2*x - 1/2*y -1/2*x - 1/2*y              0]
```

106.15 Maps between Rings

MAGMA includes functions for working with maps between multivariate polynomial rings. Let $R = K_1[x_1, \ldots, x_n]$ and $S = K_2[y_1, \ldots, y_m]$ be a polynomial rings over the fields K_1, K_2, and $f : R \to S$ a ring homomorphism.

`PolyMapKernel(f)`

Return the kernel of the map f as an ideal in the domain R, i.e., the set $\{a \in R | f(a) = 0\}$. This is basically the computation of the relation ideal for the polynomials defining the map and is as described in `RelationIdeal`.

`IsInImage(f, p)`

Given an polynomial p in S, return whether p is in the image of the map f. The algorithm is the one described on p. 82 of [AL94].

`IsSurjective(f)`

Return whether the map f is surjective. Uses the function above to check whether each codomain variable lies in the image.

`Extension(phi, I)`

The extension of the ideal I by ϕ, where ϕ is a homomorphism from the generic of I. That is, the ideal generated by the image of I under ϕ.

`Implicitization(phi)`

Suppose the polynomial map $\phi : K^n \to K^m$ is a parametrization of a variety V, i.e., V is the image of ϕ in K^m. This function constructs the ideal of S corresponding to V.

The map ϕ maps $(z_1, \ldots, z_n) \mapsto (f_1(z_1), \ldots, f_m(z_m))$ where the z_i are the coordinates of K^n. Let $f : S \to R$ be the map of polynomial rings defined by $(y_1, \ldots, y_m) \mapsto (f_1(y_1), \ldots, f_m(y_m))$. Then `Implicitization(f)` is the ideal of S corresponding to V.

If V is not a true variety, the function returns the smallest variety containing V (the Zariski closure of V).

The algorithm used is given on p. 97 of [CLO96]

Example H106E17

We demonstrate the use of the function `Implicitization` for the variety defined by $\phi : Q[x, y] \to Q[r, u, v, w]$, $(x, y) \mapsto (x^4, x^3y, xy^3, y^4)$. This example is taken from [AL94, Ex. 2.5.4].

```
> R<x, y> := PolynomialRing(Rationals(), 2);
> S<r, u, v, w> := PolynomialRing(Rationals(), 4);
> f := hom<S -> R |x^4, x^3*y, x*y^3, y^4>;
> Implicitization(f);
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: r, u, v, w
```

```
Basis:
[
    -r^2*v + u^3,
    r*v^2 - u^2*w,
    -u*w^2 + v^3,
    -r*w + u*v
]
```

106.16 Symmetric Polynomials

MAGMA includes functions for working with symmetric polynomials.

`ElementarySymmetricPolynomial(P, k)`

Given a polynomial ring P of rank n, and an integer k with $1 \leq k \leq n$, return the k-th elementary symmetric polynomial of P.

`IsSymmetric(f)`

`IsSymmetric(f, S)`

Given a polynomial f from a polynomial ring P of rank n, return whether f is a symmetric polynomial of P (i.e., is symmetric in all the n variables of P). If the answer is true, a polynomial g from a new polynomial ring of rank n is returned such that $f = g(e_1, \ldots, e_n)$, where e_i is the i-th elementary symmetric polynomial of P. If g is desired to be a member of a particular polynomial ring S of rank n (to obtain predetermined names of variables, for example), then S may also be passed.

Example H106E18

We create a symmetric polynomial from $\mathbf{Q}[a, b, c, d]$ and express it in terms of the elementary symmetric polynomials.

```
> P<a, b, c, d> := PolynomialRing(RationalField(), 4, "grevlex");
> f :=
> a^2*b^2*c*d + a^2*b*c^2*d + a*b^2*c^2*d + a^2*b*c*d^2 + a*b^2*c*d^2 +
>     a*b*c^2*d^2 - a^2*b^2*c - a^2*b*c^2 - a*b^2*c^2 - a^2*b^2*d -
>     3*a^2*b*c*d - 3*a*b^2*c*d - a^2*c^2*d - 3*a*b*c^2*d - b^2*c^2*d -
>     a^2*b*d^2 - a*b^2*d^2 - a^2*c*d^2 - 3*a*b*c*d^2 - b^2*c*d^2 -
>     a*c^2*d^2 - b*c^2*d^2 + a + b + c + d;
> // Check orbit under Sym(4) has size one:
> #(f^Sym(4));
1
> Q<e1, e2, e3, e4> := PolynomialRing(RationalField(), 4);
> l, E := IsSymmetric(f, Q);
> l;
true
> E;
```

```
e1 - e2*e3 + e2*e4
```

In the following example, we use a rational function field to define parameters a and b which occur as coefficients of the symmetric polynomial f.

```
> F<a,b> := FunctionField(RationalField(), 2);
> P<x1,x2,x3,x4,x5> := PolynomialRing(F, 5, "grevlex");
> y1 := x1^4 + x1^2*a + x1*b;
> y2 := x2^4 + x2^2*a + x2*b;
> y3 := x3^4 + x3^2*a + x3*b;
> y4 := x4^4 + x4^2*a + x4*b;
> y5 := x5^4 + x5^2*a + x5*b;
> f := y1*y2 + y1*y3 + y1*y4 + y1*y5 + y2*y3 + y2*y4 +
>      y2*y5 + y3*y4 + y3*y5 + y4*y5;
> Q<e1,e2,e3,e4,e5> := PolynomialRing(F,  5);
> l,E := IsSymmetric(f, Q);
> l, E;
true b*e1^3*e2 - 2*a*e1^3*e3 - 4*e1^3*e5 + a*e1^2*e2^2 +
    4*e1^2*e2*e4 + 2*e1^2*e3^2 - b*e1^2*e3 + 2*a*e1^2*e4 -
    4*e1*e2^2*e3 - 3*b*e1*e2^2 + 4*a*e1*e2*e3 + 8*e1*e2*e5 +
    a*b*e1*e2 - 8*e1*e3*e4 - 2*a^2*e1*e3 + b*e1*e4 - 6*a*e1*e5 +
    e2^4 - 2*a*e2^3 - 4*e2^2*e4 + a^2*e2^2 + 4*e2*e3^2 +
    5*b*e2*e3 + 2*a*e2*e4 + b^2*e2 - 3*a*e3^2 - 4*e3*e5 -
    3*a*b*e3 + 6*e4^2 + 2*a^2*e4 - 5*b*e5
```

106.17 Functions for Polynomial Algebra and Module Generators

The following functions work with collections of polynomials which are considered as generators for subalgebras or submodules of a polynomial ring. They have particular use in invariant theory.

`MinimalAlgebraGenerators(L)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose L is a set or sequence of k polynomials $f_1, \ldots, f_k$ in R. Let $A = K[f_1, \ldots, f_k]$ be the subalgebra (*not* ideal) of R generated by L. This function returns a minimal generating set of the algebra A as a (sorted) sequence of elements taken from L.

`HomogeneousModuleTest(P, S, F)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose F is an element of R. This function returns whether F is in the module M (considered as a submodule of R).

If the result is `true`, the function also returns a sequence $C = [c_1, \ldots, c_r]$ of length r with $c_i \in K[t_1, \ldots, t_k]$ such that $F = \sum_{i=1}^{r} c_i(p_1, \ldots, p_k) \cdot s_i$. (The polynomial ring $K[t_1, \ldots, t_r]$ is constructed separately but automatically with the print names `t1`, `t2`, etc.)

The grading of the polynomial ring R is used to determine the (weighted) degrees of all the polynomials in P, S and the polynomial F.

The function works as follows: it first splits F into its homogeneous components, and then, for each homogeneous component of (weighted) degree d, it constructs a basis for the K-space of all polynomials of the module M of degree d and then determines by linear algebra whether the component lies in that space.

The function is most often used with an invariant ring: P is the sequence of primary invariants, S is the sequence of secondary invariants, and F is a general invariant which one wishes to express in terms of the module generators S over the algebra generated by P. Also, if one wishes to test only for membership in the algebra $A = K[p_1, \ldots, p_k]$, then the sequence `[R!1]` should be passed for S.

`HomogeneousModuleTest(P, S, L)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose L is a sequence of length l of elements of R which are all homogeneous of (weighted) degree d. This function returns parallel sequences B and V with the following properties:

(a) B is sequence of length l of booleans such that for $1 \leq i \leq l$, $B[i]$ is true iff $L[i]$ is in the module M.

(b) V is a sequence of length l consisting of sequences of length r and consisting of polynomials in the polynomial ring $T = K[t_1, \ldots, t_r]$. (The polynomial ring $T = K[t_1, \ldots, t_r]$ is constructed separately but automatically with the print names `t1`, `t2`, etc.) If $B[i]$ is false (so $L[i]$ is not in M), $V[i]$ is a sequence of r zero polynomials. Otherwise $V[i]$ is a sequence of r polynomials $c_{i,1}, \ldots, c_{i,r}$ in T such that that $L[i] = \sum_{j=1}^{r} c_{i,j}(p_1, \ldots, p_k) \cdot s_j$.

The grading of the polynomial ring R is used to determine the (weighted) degrees of all the polynomials in P, S and L.

The function works as follows: it constructs a basis for the K-space of all polynomials of the module M of degree d and then, for each i with $1 \leq i \leq l$, determines by linear algebra whether $L[i]$ lies in the space. Only one echelonization of the space is needed to determine all the values of B and V so it is *much* more efficient to use this function if possible with many polynomials in L of the same homogeneous degree instead of calling the previous function separately for each polynomial since that will need to construct the basis for the homogeneous space and perform an echelonization each time.

Again, this function is most often used with an invariant ring: P is the sequence of primary invariants, S is the sequence of secondary invariants, and L is a sequence of general invariants which one wishes to express in terms of the module generators S over the algebra generated by P. Also, if one wishes to test only for membership in the algebra $A = K[p_1, \ldots, p_k]$, then the sequence $[R!1]$ should be passed for S.

`HomogeneousModuleTestBasis(P, S, L)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose L is a sequence of length l of elements of R which are all homogeneous of (weighted) degree d. Let U be the K-subspace of R consisting of all polynomials of the module M of (weighted) degree d and let V be the K-subspace of R generated by the elements of L. This function returns a sequence I of integer indices such that the sequence elements of L corresponding to the indices in I forms a basis for a K-subspace W of R such that $U + V = U \oplus W$. That is, I selects a subsequence of L which yields an *extension* of any basis of U to a basis of $U + V$.

Using this function, one can extend a minimal module generating set in S to include new elements of increasing degree, while ensuring that the module generators are minimalized (i.e., there is no redundancy amongst them).

The grading of the polynomial ring R is used to determine the (weighted) degrees of all the polynomials in P, S and L.

Example H106E19

We demonstrate simple uses of the function `HomogeneousModuleTest`. See also the example `HomogeneousModuleTest2` in the Invariant Rings chapter which demonstrates the use of the function `HomogeneousModuleTest` in invariant theory.

```
> R<x, y, z> := PolynomialRing(RationalField(), 3);
> P := [x^2 + y^2, z];
> S := [1, x + y + z];
> L := [x^2 + y^2, (x+y+z)^2-z^2-2*x*y, x*y];
> B, V := HomogeneousModuleTest(P, S, L);
> B;
[ true, true, false ]
```

```
> V;
[
    [
        t1,
        0
    ],
    [
        t1 - 2*t2^2,
        2*t2
    ],
    [
        0,
        0
    ]
]
> // Thus L[1] is P[1]*S[1] and
> // L[2] is (P[1] - 2*P[2]^2)*S[1] + 2*P[2]*S[2].
> L[1] eq P[1]*S[1];
true
> (P[1] - 2*P[2]^2)*S[1] + 2*P[2]*S[2] eq L[2];
true
> // Determine subsequence of [x^3, y^3, z^3] which forms
> // extension basis of module generated by P and S.
> L := [x^3, y^3, z^3];
> HomogeneousModuleTestBasis(P, S, L);
[ 1, 2 ]
> // Thus x^2 and y^2 could be appended to S to preserve
> // minimality.
```

106.18 Bibliography

[**AL94**] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases*, volume 3 of *Graduate studies in mathematics*. American Mathematical Society, Providence, R.I., 1994.

[**BS92**] David Bayer and Michael Stillman. Computation of Hilbert Functions. *J. Symbolic Comp.*, 14(1):31–50, 1992.

[**BW93**] Thomas Becker and Volker Weispfenning. *Gröbner Bases*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.

[**CLO96**] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties and Algorithms*. Undergraduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 2nd edition, 1996.

[**CLO98**] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.

[**DL06**] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry*, volume 16 of *Algorithms and Computation in Mathematics*. Springer, New York–Berlin–Heidelberg, 2006.

[**EHV92**] D. Eisenbud, C. Huneke, and W. Vasconcelas. Direct Methods for Primary Decomposition. *Inv. math.*, 110:207–235, 1992.

[**ES94**] Eisenbud and Sturmfels. Finding sparse systems of parameters. *Journal of Pure and Applied Algebra*, 94:143–157, 1994.

[**GP02**] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer-Verlag, Berlin–Heidelberg–New York, 2002.

[**Har77**] Robin Hartshorne. *Algebraic Geometry, GTM 52*. Springer, ASpringer, 1977.

[**Laz92**] Daniel Lazard. Solving Zero-dimensional Algebraic Systems. *J. Symbolic Comp.*, 13(2):117–131, 1992.

[**Ste05**] Allan Steel. Conquering Inseparability: Primary Decomposition and Multivariate Factorization over Algebraic Function Fields of Positive Characteristic. *J. Symbolic Comp.*, 40(3):1053–1075, 2005.

107 LOCAL POLYNOMIAL RINGS

Chapter 107
LOCAL POLYNOMIAL RINGS

107.1 Introduction

This chapter describes local polynomial rings. Let R be the multivariate polynomial ring $K[x_1, \ldots, x_n]$, where K is a field. We denote by

$$K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$$

the collection of all rational functions f/g of $x_1, \ldots, x_n$ with $g(p) \neq 0$, where $p = (0, \ldots, 0)$. Such a ring is **local** (has a unique maximal ideal) and we will call it a **local polynomial ring** in MAGMA. Such a ring is always multivariate and is related to the corresponding multivariate polynomial ring $K[x_1, \ldots, x_n]$ which will we will call **global** (when distinguishing it from the local case). We will also call $K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$ the **localization** of $K[x_1, \ldots, x_n]$ (this is always understood to be at the prime ideal generated by $x_1, \ldots, x_n$, corresponding to the origin).

Much of the theory for multivariate polynomial rings and their ideals carry over to local polynomial rings, so the reader should first be familiar with multivariate polynomial rings and their ideals (see Chapters 24 and 105).

Corresponding to a Gröbner basis of an ideal of a global multivariate ring is a **standard basis** of an ideal of a local polynomial ring. See [CLO98, Chapter 4] or [GP02, Chapter 1] for the basics of the theory and algorithms.

The other facilities are currently basic but will be expanded in coming versions. But note that computations with $R-$modules, where R is a local polynomial ring, are fully supported: see Chapter 109.

107.2 Elements and Local Monomial Orders

Elements of a local polynomial ring are multivariate polynomials just like the usual (global) multivariate polynomials, except that the monomials are sorted (again with the greatest first) with respect to a **local monomial order**, which is in general the negation of a standard global monomial order. Thus the monomial 1 is less than all other monomials and the polynomials are like multivariate formal power series (written, for example, as $1 + x + x^2y + y^4$). But most arithmetic-like operations allowed for global polynomials also carry over automatically for elements of a local polynomials so we will not list them in detail in this chapter (see Chapter 24).

Note that in the strict mathematical definition of $R = K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$, elements of R may have non-trivial denominators, but this is currently not supported in MAGMA: the elements in MAGMA must always be strict polynomials. The main purpose of supporting

such rings is for standard bases of ideals (see below), and this restriction does not matter there, since units are automatically removed from the elements of a standard basis.

We now describe the current local monomial orders available in MAGMA. First the reader should see Section 105.2 for the fundamental points about (global) monomial orders for multivariate polynomial rings.

The fundamental difference in the local case is that for a local polynomial ring R of rank n, the monomial order is the negation of a global monomial order. More precisely, let M be the monomials of R. A **local monomial ordering** on M is a total order $<$ on M such that $s \leq 1$ for all $s \in M$, $s \leq t$ implies $su \leq tu$ for all $s, t, u \in M$, and M is a well-ordering (every non-empty subset of M possesses a minimal element w.r.t. $<$). See [CLO98, Sec. 4.3], [DL06, Sec. 9.1], or [GP02, Sec. 1.2] for more information.

We now list each of the monomial orders available in MAGMA (these will be expanded in future versions). As in the global case, we suppose that s and t are monomials from a ring R of rank n. Any order on the monomials is then fully defined by just specifying exactly when $s < t$ with respect to that order. In the following, the argument(s) are described for an order as a list of expressions; that means that the expressions (without the parentheses) should be appended to any base arguments when any particular intrinsic function is called which expects a monomial order.

107.2.1 Local Lexicographical: `llex`

Definition: $s < t$ iff there exists $1 \leq i \leq n$ such that all of the j-th exponents of s and t are equal for $i < j \leq n$, but the i-th exponent of s is greater than the i-th exponent of t. The order is specified by the argument (`"llex"`).

This order is the negation of the global lexicographical order, but with the reverse order for the variables. Thus the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable.

107.2.2 Local Graded Lexicographical: `lglex`

Definition: $s < t$ iff the total degree of s is greater than the total degree of t or the total degree of s is equal to the total degree of t and $s > t$ with respect to the (glocal) lexicographical order. The order is specified by the argument (`"lglex"`).

This order is the negation of the global `glex` order.

107.2.3 Local Graded Reverse Lexicographical: `lgrevlex`

Definition: $s < t$ iff the total degree of s is greater than the total degree of t or the total degree of s is equal to the total degree of t and $s < t$ with respect to the (global) lexicographical order applied to the exponents of s and t in reverse order. The order is specified by the argument (`"grevlex"`).

This order is the negation of the global `grevlex` order.

107.3 Local Polynomial Rings and Ideals

107.3.1 Creation of Local Polynomial Rings and Accessing their Monomial Orders

Local polynomial rings are created from a coefficient field, the number of variables, and a monomial order. If no order is specified, the monomial order is taken to be the local lexicographical order.

`LocalPolynomialRing(K, n)`

Create a local polynomial ring in $n > 0$ variables over the field K. The *local lexicographical* ordering on the monomials is used for this default construction.

`LocalPolynomialRing(K, n, order)`

`LocalPolynomialAlgebra(K, n, order)`

Create a local polynomial ring in $n > 0$ variables over the ring R with the given order *order* on the monomials. See the above section on local monomial orders for the valid values for the argument *order*.

`LocalPolynomialRing(K, n, T)`

Create a local polynomial ring in $n > 0$ variables over the field K with the order given by the tuple T on the monomials. T must be a tuple whose components match the valid arguments for the monomial orders in Section 107.2. Such a tuple is also returned by the next function.

`MonomialOrder(R)`

Given a local polynomial ring R (or an ideal thereof), return a description of the monomial order of R. This is returned as a tuple which matches the relevant arguments listed for each possible order in Section 107.2, so may be passed as the third argument to the function `LocalPolynomialRing` above.

`MonomialOrderWeightVectors(R)`

Given a polynomial ring R of rank n (or an ideal thereof), return the weight vectors of the underlying monomial order as a sequence of n sequences of n rationals. See, for example, [CLO98, p. 153] for more information.

`Localization(R)`

Given a (global) multivariate polynomial ring $R = K[x_1, \ldots, x_n]$ (or an ideal I of such an R), return the localization $K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$ of R (or the ideal of the localization of R which corresponds to I). The print names for the variables of R are carried over.

Example H107E1

We show how one can construct local polynomial rings with different orders. Note the order on the monomials for elements of the rings.

```
> K := RationalField();
> R<x,y,z> := LocalPolynomialRing(K, 3);
> R;
Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
> MonomialOrder(R);
<"llex">
> MonomialOrderWeightVectors(R);
[
    [ 0, 0, -1 ],
    [ 0, -1, 0 ],
    [ -1, 0, 0 ]
]
> 1 + x + y + z + x^7 + x^8*y^7 + y^5 + z^10;
1 + x + x^7 + y + y^5 + x^8*y^7 + z + z^10
> R<x,y,z> := LocalPolynomialRing(K, 3, "lgrevlex");
> R;
Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Graded Reverse Lexicographical
Variables: x, y, z
> MonomialOrder(R);
<"lgrevlex">
> MonomialOrderWeightVectors(R);
[
    [ -1, -1, -1 ],
    [ -1, -1, 0 ],
    [ -1, 0, 0 ]
]
> 1 + x + y + z + x^7 + x^8*y^7 + y^5 + z^10;
1 + z + y + x + y^5 + x^7 + z^10 + x^8*y^7
```

107.3.2 Creation of Ideals and Accessing their Bases

As for global polynomial rings, within the general context of ideals of local polynomial rings, the term "basis" will refer to an *ordered* sequence of polynomials which generate an ideal. (Thus a basis can contain duplicates and zero elements so is not like a basis of a vector space.)

`ideal< R | L >`

Given a local polynomial ring R, return the ideal of R generated by the elements of R specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of R;

(b) A set or sequence of elements of R;

(c) An ideal of R;

(d) A set or sequence of ideals of R.

`Ideal(B)`

Given a set or sequence B of polynomials from a local polynomial ring R, return the ideal of R generated by the elements of B with the given basis B. This is equivalent to the above `ideal` constructor, but is more convenient when one simply has a set or sequence of polynomials.

`Ideal(f)`

Given a polynomial f from a local polynomial ring R, return the principal ideal of R generated by f.

`Basis(I)`

Given an ideal I, return the current basis of I. This will be the standard basis of I if it is computed; otherwise it will be the original basis.

`BasisElement(I, i)`

Given an ideal I together with an integer i, return the i-th element of the current basis of I. This the same as `Basis(I)[i]`.

107.4 Standard Bases

Computation in ideals of local polynomial rings is possible because of the construction of **standard bases** of such ideals. These are the counterpart to Gröbner bases for ideals of global polynomial rings. Currently, standard bases may only be computed for ideals defined over fields.

MAGMA computes a standard basis of an ideal using the Mora normal form and standard basis algorithms (with the homogenization technique): see [CLO98, Sec. 4.4] for an overview.

In contrast to the global case, for a given fixed monomial ordering a standard basis of an ideal is not unique in general because it can be difficult to get the lower order terms of polynomials in the standard basis into a unique form. But the **leading monomials** of a standard basis are always sorted in MAGMA and **are unique**.

107.4.1 Construction of Standard Bases

The following functions and procedures allow one to construct standard bases. Note that a standard basis for an ideal will be automatically generated when necessary; the `Groebner` procedure below simply allows control of the algorithms used to compute the standard basis. The verbose flags are shared with those for global Gröbner basis construction (since the standard basis algorithms reduce to those), so see Section 105.4.6 for details on these.

`StandardBasis(I)`

Given an ideal I, force the standard basis of I to be computed, and then return that.

`StandardBasis(S)`

Given a set or sequence S of polynomials of a local polynomial ring R, return a standard basis of the ideal generated by S as a sorted sequence.

Example H107E2

We compute the standard basis of the ideal given in [CLO98, p.167].

```
> Q := RationalField();
> R<x,y,z> := LocalPolynomialRing(Q, 3);
> I := Ideal([x^5 - x*y^6 + z^7, x*y + y^3 + z^3, x^2 + y^2 - z^2]);
> I;
Ideal of Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
Basis:
[
    x^5 - x*y^6 + z^7,
    x*y + y^3 + z^3,
    x^2 + y^2 - z^2
]
> StandardBasis(I);
[
    x^2 + y^2 - z^2,
    x*y + y^3 + z^3,
    y^3 - x*y^3 - y*z^2 - x*z^3,
    x*z^4 + 3*y^2*z^4 + 4*x*y^4*z^4 + y*z^5 + 5*x*y^3*z^5 - 2*z^6 + 4*y^2*z^6 +
        x*y^2*z^6 + z^7 - 2*x*z^7 + 7*y*z^7 + 4*x*y*z^7 - y^2*z^7 + 3*z^8 +
        4*x*z^8,
    y^2*z^4 + 3*y^5*z^5 - z^6 + 3/2*x*z^6 + 2*y^2*z^6 + y^4*z^6 - x*z^7 +
        7/2*y*z^7 - x*y^2*z^7 - y^3*z^7 + 3/2*z^8 - 3/2*x*z^8 - 2*y*z^8 +
        2*x*y*z^8 + 3/2*y^2*z^8,
    y*z^7 + 1/2*x*z^8 + 23/4*y*z^8 - 9/4*x*y^3*z^8 + 3/2*y^4*z^8,
    z^9
```

```
]
```

We note that no elements of the standard basis have factors which are units in R.

```
> [Factorization(f): f in $1];
[
    [
        <x^2 + y^2 - z^2, 1>
    ],
    [
        <x*y + y^3 + z^3, 1>
    ],
    [
        <y^2 - x*y^2 - y*z + x*y*z - x*z^2, 1>,
        <y + z, 1>
    ],
    [
        <z, 4>,
        <x + 3*y^2 + 4*x*y^4 + y*z + 5*x*y^3*z - 2*z^2 + 4*y^2*z^2 + x*y^2*z^2 +
            z^3 - 2*x*z^3 + 7*y*z^3 + 4*x*y*z^3 - y^2*z^3 + 3*z^4 + 4*x*z^4, 1>
    ],
    [
        <z, 4>,
        <y^2 + 3*y^5*z - z^2 + 3/2*x*z^2 + 2*y^2*z^2 + y^4*z^2 - x*z^3 +
            7/2*y*z^3 - x*y^2*z^3 - y^3*z^3 + 3/2*z^4 - 3/2*x*z^4 - 2*y*z^4 +
            2*x*y*z^4 + 3/2*y^2*z^4, 1>
    ],
    [
        <z, 7>,
        <y + 1/2*x*z + 23/4*y*z - 9/4*x*y^3*z + 3/2*y^4*z, 1>
    ],
    [
        <z, 9>
    ]
]
```

Example H107E3

We note that starting from an ideal I of a global polynomial ring, the standard basis of the localization of I may be much simpler than the Gröbner basis of I.

```
> Q := RationalField();
> R<x,y,z> := PolynomialRing(Q, 3);
> I := Ideal([x^2 - x*y^3 + z^3, x*y + y^2 + z, x + y^2 - z^2]);
> Groebner(I); I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0
```

```
Groebner basis:
[
    x - y*z + 127/1052*z^9 + 585/1052*z^8 + 233/263*z^7 + 1273/1052*z^6 +
        695/526*z^5 + 223/1052*z^4 + 569/526*z^3 + 35/1052*z^2 - z,
    y^2 + y*z - 127/1052*z^9 - 585/1052*z^8 - 233/263*z^7 - 1273/1052*z^6 -
        695/526*z^5 - 223/1052*z^4 - 569/526*z^3 - 1087/1052*z^2 + z,
    y*z^2 + 51/263*z^9 + 208/263*z^8 + 308/263*z^7 + 476/263*z^6 + 465/263*z^5 +
        278/263*z^4 + 691/263*z^3 + 217/263*z^2,
    z^10 + 5*z^9 + 10*z^8 + 15*z^7 + 16*z^6 + 9*z^5 + 12*z^4 + 13*z^3 + 2*z^2
]
> QuotientDimension(I);
12
>
> IL := Localization(I);
> StandardBasis(IL);
[
    x + y^2,
    y^2 - y^3 + z,
    z^2
]
> QuotientDimension(IL);
4
```

107.5 Operations on Ideals

In the following, note that since ideals of a full polynomial ring P are regarded as subrings of P, the ring P itself is a valid ideal as well (the ideal containing 1).

107.5.1 Basic Operations

`I + J`

Given ideals I and J of the same polynomial ring P, return the sum of I and J, which is the ideal generated by the generators of I and those of J.

`I * J`

Given ideals I and J of the same polynomial ring P, return the product of I and J, which is the ideal generated by the products of the generators of I and those of J.

`I ^ k`

Given an ideal I of the polynomial ring P, and an integer k, return the k-th power of I.

`QuotientDimension(I)`

Given an ideal I of a local polynomial ring R over a field K, return the dimension of P/I as a K-vector space. Note that this is quite different from the function `Dimension` below (which returns the Krull dimension of an ideal).

`Generic(I)`

Given an ideal I of a generic local polynomial ring R, return R.

`LeadingMonomialIdeal(I)`

Given an ideal I, return the leading monomial ideal of I; that is, the ideal generated by all the leading monomials of I.

`I meet J`

Given ideals I and J of the same polynomial ring P, return the intersection of I and J.

`&meet S`

Given a set or sequence S of ideals of the same local polynomial ring R, return the intersection of all the ideals of S.

107.5.2 Ideal Predicates

`I eq J`

Given two ideals I and J of the same polynomial ring P, return whether I and J are equal.

`I ne J`

Given two ideals I and J of the same polynomial ring P, return whether I and J are not equal.

`I notsubset J`

Given two ideals I and J in the same polynomial ring P return whether I is not contained in J.

`I subset J`

Given two ideals I and J in the same polynomial ring P return whether I is contained in J.

`IsZero(I)`

Given an ideal I of the local polynomial ring R, return whether I is the zero ideal (contains zero alone).

`IsProper(I)`

Given an ideal I of the local polynomial ring R, return whether I is proper; that is, whether I is strictly contained in R (or whether the standard basis of I does not contain 1 alone).

`IsZeroDimensional(I)`

Given an ideal I of the local polynomial ring R, return whether I is zero-dimensional (so the quotient of P by I has non-zero finite dimension as a vector space over the coefficient field – see the section on dimension for further details). Note that the ring R has dimension -1, so it is not zero-dimensional.

Example H107E4

We construct some ideals in $\mathbf{Q}[x, y, z]$ and perform basic arithmetic on them.

```
> R<x,y,z> := LocalPolynomialRing(RationalField(), 3);
> I := ideal<R | x*y - z, x^3*z^2 - y^2, x*z^3 - x - y>;
> J := ideal<R | x*y - z, x^2*z - y, x*z^3 - x - y>;
> A := I * J;
> _ := StandardBasis(A);
> A;
Ideal of Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension 0
Standard basis:
[
    x^2 - y^2 + 2*x^3*z,
    x*y + y^2 - x^3*z,
    y^3,
    x*z + y*z,
    y*z,
    z^2
]
> M := I meet J;
> M;
Ideal of Localization of Polynomial Ring of rank 3 over Rational Field
Order: Local Lexicographical
Variables: x, y, z
Homogeneous
Basis:
[
    x + y,
    y^2,
    z
]
> A eq M;
```

```
false
> A subset M;
true
```

107.5.3 Operations on Elements of Ideals

`f in I`

Given a polynomial f from a local polynomial ring R, together with an ideal I of R, return whether f is in I.

`NormalForm(f, I)`

Given a polynomial f from a local polynomial ring R, together with an ideal I of R, return a normal form of f with respect to (the standard basis of) I. The normal form of f is zero if and only if f is in I.

`f notin I`

Given a polynomial f from a polynomial ring P, together with an ideal I of P, return whether f is not in I.

Example H107E5

We demonstrate the element operations with respect to an ideal of the localization of $\mathbf{Q}[x, y, z]$.

```
> R<x,y,z> := LocalPolynomialRing(RationalField(), 3);
> I := ideal<R | (x + y)^3, (y - z)^2, y^2*z + z>;
> NormalForm(y^2*z + z, I);
0
> NormalForm(x^3, I);
-3*x^2*y
> x + y in I;
false
```

107.6 Changing Coefficient Ring

The `ChangeRing` function enables the changing of the coefficient ring of a local polynomial ring or ideal.

`ChangeRing(I, L)`

Given an ideal I of a local polynomial ring $R = K[x_1, \ldots, x_n]$ of rank n with coefficient ring K, together with a field L, construct the ideal J of the polynomial field $S = L[x_1, \ldots, x_n]$ obtained by coercing the coefficients of the elements of the basis of I into L. It is necessary that all elements of the old coefficient field K can be automatically coerced into the new coefficient field L. If K and L are fields and K is known to be a subfield of L and the current basis of I is a standard basis, then the basis of J is marked automatically to be a standard basis of J.

107.7 Changing Monomial Order

Often one wishes to change the monomial order of an ideal. MAGMA allows one to do this by use of the `ChangeOrder` function.

`ChangeOrder(I, Q)`

Given an ideal I of the local polynomial ring $R = K[x_1, \ldots, x_n]$, together with a local polynomial ring S of rank n (with possibly a different order to that of R), return the ideal J of S corresponding to J and the isomorphism f from R to S. The map f simply maps $R.i$ to $S.i$ for each i.

`ChangeOrder(I, order)`

Given an ideal I of the polynomial ring $P = R[x_1, \ldots, x_n]$, together with a monomial order *order* (see Section 107.2), construct the polynomial ring $Q = R[x_1, \ldots, x_n]$ with order *order*, and then return the ideal J of Q corresponding to I and the isomorphism f from P to Q. See the section on monomial orders for the valid values for the argument *order*. The map f simply maps $P.i$ to $Q.i$ for each i.

107.8 Dimension of Ideals

Let I be an ideal of the local polynomial ring $K[x_1, \ldots, x_n]_{\langle x_1, \ldots, x_n \rangle}$, where K is a field. As for polynomial rings, the dimension of the ideal I can be defined as the the maximum of the cardinalities of all the independent sets modulo I (see Section 107.8 for details).

`Dimension(I)`

Given an ideal I of a local polynomial ring R defined over a field, return the dimension d of I, together with a (sorted) sequence U of integers of length d such that the variables of P corresponding to the integers of U constitute a maximally independent set modulo I. If I is the full local polynomial ring R, the dimension is defined to be -1, and the second return value is not set. The algorithm implemented is that given in [BW93, p. 449].

107.9 Bibliography

[**BW93**] Thomas Becker and Volker Weispfenning. *Gröbner Bases*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1993.

[**CLO98**] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.

[**DL06**] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry*, volume 16 of *Algorithms and Computation in Mathematics*. Springer, New York–Berlin–Heidelberg, 2006.

[**GP02**] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer-Verlag, Berlin–Heidelberg–New York, 2002.

108 AFFINE ALGEBRAS

Chapter 108
AFFINE ALGEBRAS

108.1 Introduction

An affine algebra in MAGMA is simply the quotient ring of a multivariate polynomial ring $P = R[x_1, \ldots, x_n]$ by an ideal J of P. Such rings arise commonly in commutative algebra and algebraic geometry. They can also be viewed as generalizations of number fields and algebraic function fields, when R is a field.

The elements of affine algebras are simply multivariate polynomials which are always kept reduced to normal form modulo the ideal J of "relations". Practically all operations which are applicable to multivariate polynomials are also applicable in MAGMA to elements of affine algebras (when meaningful).

If the ideal J of relations defining an affine algebra $A = R[x_1, \ldots, x_n]/J$ is *maximal* and R is a field, then A is a field and may be used with any algorithms in MAGMA which work over fields. Factorization of polynomials over such affine algebras is also supported (including fields of small characteristic, since V2.10).

If an affine algebra defined over a field has finite dimension considered as a vector space over the coefficient field, extra special operations are available on its elements.

Currently the base ring R may be a field or a Euclidean ring. Further operations for affine algebras over Euclidean rings will be supported in the future.

An affine algebra has type `RngMPolRes` and its elements type `RngMPolResElt`.

108.2 Creation of Affine Algebras

One can create an affine algebra simply by forming the quotient of a multivariate polynomial ring by an ideal (`quo` constructor or `/` function). A special constructor `AffineAlgebra` is also provided to remove the need to create the base polynomial ring.

`quo< P | J >`

`quo< P | a`$_1$`, ..., a`$_r$ `>`

Given a multivariate polynomial ring P and an ideal J of P, return the quotient ring P/J. The ideal J may either be specified as an ideal or by a list a_1, a_2, ..., a_r, of generators which all lie in P. The angle bracket notation can be used to assign names to the indeterminates: `Q<q, r> := quo< I | I.1 + I.2, I.2^2 - 2, I.3^2 + I.4 >;`.

`P / J`

Given a multivariate polynomial ring P and an ideal J of P, return the quotient affine algebra P/J.

`AffineAlgebra< R, X | L >`

Given a ring R, a list X of n identifiers, and a list L of polynomials (relations) in the n variables X, create the affine algebra of rank n with base ring R with given quotient relations; i.e., return $R[X]/\langle L\rangle$. The angle bracket notation can be used to assign names to the indeterminates.

Example H108E1

One can create a relative extension of an algebraic number field as an affine algebra. The multivariate representation will often be more efficient than an absolute representation because of the sparsity of the elements in the field.

```
> Q := RationalField();
> A<x, y> := AffineAlgebra<Q, x, y | x^2 - y^2 + 2, y^3 - 5>;
> A;
Affine Algebra of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Quotient relations:
[
    x^2 - y^2 + 2,
    y^3 - 5
]
> x^2;
y^2 - 2
> x^-1;
2/17*x*y^2 + 5/17*x*y + 4/17*x
> P<z> := PolynomialRing(Q);
> MinimalPolynomial(x);
z^6 + 6*z^4 + 12*z^2 - 17
> MinimalPolynomial(x^-1);
z^6 - 12/17*z^4 - 6/17*z^2 - 1/17
> MinimalPolynomial(y);
z^3 - 5
```

Another important construction is to create an affine algebra over a rational function field to obtain an algebraic function field:

```
> F<t> := FunctionField(IntegerRing());
> A<x, y> := AffineAlgebra<F, x, y | t*x^2 - y^2 + t + 1, y^3 - t>;
> P<z> := PolynomialRing(F);
> x^-1;
(-t^2 - t)/(t^3 + 2*t^2 + 3*t + 1)*x*y^2 - t^2/(t^3 + 2*t^2 + 3*t + 1)*x*y
    + (-t^3 - 2*t^2 - t)/(t^3 + 2*t^2 + 3*t + 1)*x
> MinimalPolynomial(x);
z^6 + (3*t + 3)/t*z^4 + (3*t^2 + 6*t + 3)/t^2*z^2 + (t^3 + 2*t^2 + 3*t +
    1)/t^3
> MinimalPolynomial(x^-1);
z^6 + (3*t^3 + 6*t^2 + 3*t)/(t^3 + 2*t^2 + 3*t + 1)*z^4 + (3*t^3 +
```

```
3*t^2)/(t^3 + 2*t^2 + 3*t + 1)*z^2 + t^3/(t^3 + 2*t^2 + 3*t + 1)
```

In this example we can consider y as a cube root of the transcendental indeterminate t.
Note that in general the (Krull) dimension of the ideal defining the relations may be anything; it need not be 0 or 1 as it is in these examples.

108.3 Operations on Affine Algebras

This section describes operations on affine algebras. Most of the operations are very similar to those for multivariate polynomial rings; such operations are done by mapping the computation to the preimage ideal and then by mapping the result back into the affine algebra. See the corresponding functions for the multivariate polynomial rings for details.

`Q . i`

Given an affine algebra Q, return the i-th indeterminate of Q as an element of Q.

`CoefficientRing(Q)`

Return the coefficient ring of the affine algebra Q.

`Rank(Q)`

Return the rank of the affine algebra Q (the number of indeterminates of Q).

`DivisorIdeal(I)`

Given an ideal I of an affine algebra Q which is the quotient ring P/J, where P is a polynomial ring and J an ideal of P, return the ideal J.

`PreimageIdeal(I)`

Given an ideal I of an affine algebra Q which is the quotient ring P/J, where P is a polynomial ring and J an ideal of P, return the ideal I' of P such that the image of I' under the natural epimorphism $P \to Q$ is I.

`PreimageRing(Q)`

Given an affine algebra Q which is the quotient ring P/J, where P is a polynomial ring and J an ideal of P, return the polynomial ring P.

`OriginalRing(Q)`

Return the generic polynomial ring P such that Q is P/J for some ideal J of P.

`I eq J`

Given two ideals I and J of the same affine algebra Q, return `true` if and only if I and J are equal.

`I subset J`

Given two ideals I and J of the same affine algebra Q, return `true` if and only if I is contained in J.

`I + J`

Given two ideals I and J of the same affine algebra Q, return the sum $I + J$.

`I * J`

Given two ideals I and J of the same affine algebra Q, return the product $I * J$.

`I ^ n`

Given an ideal I of an affine algebra Q and an integer n, return the power I^n.

`I meet J`

Given two ideals I and J of the same affine algebra Q, return the intersection $I \cap J$.

`IsProper(I)`

Given an ideal I of the affine algebra Q, return whether I is proper; that is, whether I is strictly contained in Q.

`IsZero(I)`

Given an ideal I of the affine algebra Q, return whether I is the zero ideal. Note that this is equivalent to whether the preimage ideal of I is the divisor ideal of Q.

`IsPrime(I)`

Given an ideal I of the affine algebra Q, return whether I is a prime ideal.

`IsPrimary(I)`

Given an ideal I of the affine algebra Q, return whether I is a primary ideal.

`IsRadical(I)`

Given an ideal I of the affine algebra Q, return whether I is a radical ideal.

`PrimaryDecomposition(I)`

Given an ideal I of the affine algebra Q, return the primary decomposition of I, together with the associated primes.

`RadicalDecomposition(I)`

Given an ideal I of the affine algebra Q, return the (prime) decomposition of the radical of I.

Example H108E2

We illustrate the operations on ideals of affine algebras.

```
> Q := RationalField();
> A<x,y,z> := AffineAlgebra<Q,x,y,z | x^2 - y + 1, y^3 + z - 1>;
> A;
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
> I := ideal<A | x^3*y*z^2>;
> IsRadical(I);
false
> Radical(I);
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
Generating basis:
[
    x*y^2 + x*y - x*z + x,
    y*z,
    z^2 - z
]
> PQ, PP := PrimaryDecomposition(I);
> #PQ;
3
> PQ[1];
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
Generating basis:
[
    y + 5/81*z^3 + 1/9*z^2 + 1/3*z - 1,
    x*z^3,
```

```
    y + 5/81*z^3 + 1/9*z^2 + 1/3*z - 1,
    z^4
]
> PP[1];
Affine Algebra of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Quotient relations:
[
    x^2 - y + 1,
    y^3 + z - 1
]
Generating basis:
[
    x,
    y - 1,
    z
]
```

108.4 Maps between Affine Algebras

MAGMA includes functions for working with maps between affine algebras.

`AffineAlgebraMapKernel(phi)`

Return the kernel of the homomorphism ϕ of affine algebras.

108.5 Finite Dimensional Affine Algebras

If an affine algebra is defined over a field and has finite dimension considered as a vector space over its coefficient field, extra special operations are available on its elements.

Similar operations for affine algebras defined over general Euclidean rings will be supported in the future.

`HasFiniteDimension(Q)`

Given an affine algebra Q defined over a field, return whether Q has finite dimension.

`Dimension(Q)`

Given a finite dimensional affine algebra Q defined over a field, return the dimension of Q.

`VectorSpace(Q)`

Given a finite dimensional affine algebra Q defined over a field, construct the vector space V isomorphic to Q, and return V together with the isomorphism f from Q onto V.

`MonomialBasis(Q)`

Given a finite dimensional affine algebra Q defined over a field, return the basis B of monomials of Q. This is a sequence of monomials in Q of length d, such that that the image $f(B[i]) = V.i$ where V and f are the return values of `VectorSpace` above.

`MatrixAlgebra(Q)`

Given a finite dimensional affine algebra Q defined over a field, construct the matrix algebra A isomorphic to Q, and return A together with the isomorphism f from Q onto A.

`RepresentationMatrix(f)`

Given an element f of a finite dimensional affine algebra Q defined over a field, return the representation matrix of f, which is a d by d matrix over the coefficient field of Q (where d is the dimension of Q) which represents f.

`IsUnit(f)`

Given an element f of a finite dimensional affine algebra Q defined over a field, return whether f is a unit.

`IsNilpotent(f)`

Given an element f of a finite dimensional affine algebra Q defined over a field, return whether f is nilpotent, and if so, return also the smallest q such that $f^q = 0$.

`MinimalPolynomial(f)`

Given an element f of a finite dimensional affine algebra Q defined over a field, return the minimal polynomial of f as a univariate polynomial over the coefficient field of Q.

Example H108E3

Suppose we wish to find the minimal polynomial of $\theta = \sqrt{2} + \sqrt[3]{5}$ over $\mathbf{Q}$. To do this we can just compute the minimal polynomial of (the coset of) $x + y$ over $\mathbf{Q}$ in the affine algebra $Q[x, y]/(x^2 - 2, y^3 - 5)$.

```
> Q := RationalField();
> A<x, y> := AffineAlgebra<Q, x, y | x^2 - 2, y^3 - 5>;
> UP<z> := PolynomialRing(Q);
> MinimalPolynomial(x + y);
z^6 - 6*z^4 - 10*z^3 + 12*z^2 - 60*z + 17
```

108.6 Affine Algebras which are Fields

If the ideal J of relations defining an affine algebra $A = K[x_1, \ldots, x_n]/J$, where K is a field, is *maximal*, then A is a field and may be used with any algorithms in MAGMA which work over fields. Factorization of polynomials over such affine algebras is also supported (in any characteristic, since V2.10). The examples below will demonstrate some of the applications available.

Note that an affine algebra defined over a field which itself is a field also has finite dimension when considered as a vector space over its coefficient field, so all of the operations in the previous section are also available.

Example H108E4

We create the function field $F = \mathbf{Q}(a, b, x)$ and then the affine algebra $A = F[y]/ < y^2 - (x^3 + ax + b) >$ (which is also equivalent to an algebraic function field). This then allows us to create a generic elliptic curve E over A and compute the coordinates of multiples of a generic point easily.

```
> Q := RationalField();
> F<x, a, b> := FunctionField(Q, 3);
> A<y> := AffineAlgebra<F, y | y^2 - (x^3 + a*x + b)>;
> IsField(A);
true
> y^2;
x^3 + x*a + b
> y^-1;
1/(x^3 + x*a + b)*y
> E := EllipticCurve([A | a, b]);
> E;
Elliptic Curve defined by y^2 = x^3 + a*x + b over Affine Algebra of rank 1 over
    Rational function field of rank 3 over Rational Field
    Variables: x, a, b
> p := E ! [x, y];
> p;
(x : y : 1)
> q := 2*p;
> q;
((1/4*x^4 - 1/2*x^2*a - 2*x*b + 1/4*a^2)/(x^3 + x*a + b) : (1/8*x^6 +
5/8*x^4*a + 5/2*x^3*b - 5/8*x^2*a^2 - 1/2*x*a*b - 1/8*a^3 - b^2)/(x^6
+ 2*x^4*a + 2*x^3*b + x^2*a^2 + 2*x*a*b + b^2)*y : 1)
> c := LeadingCoefficient(q[2]);
> Denominator(c);
x^6 + 2*x^4*a + 2*x^3*b + x^2*a^2 + 2*x*a*b + b^2
> Factorization($1);
[
    <x^3 + x*a + b, 2>
]
```

Example H108E5

Starting with the same affine algebra $A = \mathbf{Q}(a, b, x)F[y]/ < y^2 - (x^3 + ax + b) >$ as in the last example, we factor some univariate polynomials over A. A is of course isomorphic to an absolute field, but the presentation given may be much more convenient to the user.

```
> Q := RationalField();
> F<x, a, b> := FunctionField(Q, 3);
> A<y> := AffineAlgebra<F, y | y^2 - (x^3 + a*x + b)>;
> P<z> := PolynomialRing(A);
> f := z^2 - (x^3 + a*x + b);
> f;
z^2 + -x^3 - x*a - b
> time Factorization(f);
[
    <z - y, 1>,
    <z + y, 1>
]
Time: 0.019
```

Example H108E6

In this final example, A is isomorphic to an algebraic number field, but its presentation may be more convenient than an absolute presentation (and may lead to sparser expressions for elements).

```
> Q := RationalField();
> A<a,b,c> := AffineAlgebra<Q, a,b,c | a^2 - b*c + 1, b^2 - c + 1, c^2 + 2>;
> P<x> := PolynomialRing(A);
> time Factorization(x^2 + 2);
[
    <x - c, 1>,
    <x + c, 1>
]
Time: 0.080
> time Factorization(x^2 - b*c + 1);
[
    <x - a, 1>,
    <x + a, 1>
]
Time: 0.090
> MinimalPolynomial(a);
x^8 + 4*x^6 + 2*x^4 - 4*x^2 + 9
> time Factorization(P ! $1);
[
    <x - a, 1>,
    <x + a, 1>,
    <x - 1/3*a*b*c - 2/3*a*b + 1/3*a*c - 1/3*a, 1>,
    <x + 1/3*a*b*c + 2/3*a*b - 1/3*a*c + 1/3*a, 1>,
    <x^4 + 2*x^2 - 2*c - 1, 1>
```

```
]
Time: 2.809
```

108.7 Rings and Fields of Fractions of Affine Algebras

Given any affine algebra $Q = K[x_1, \ldots, x_n]/J$, where K is a field, one may create the *ring of fractions* R of Q. This is the set of fractions a/b, where $a, b \in Q$ and b is invertible, and it forms a ring.

The defining ideal J does not need to be zero-dimensional. The ring of fractions R is itself represented internally by an affine algebra over an appropriate rational function field, but has the appearance to the user of the set of fractions, so one may access the numerator and denominator of elements of R, for example.

If the ideal J is prime, then R is the field of fractions of A and may be used with any algorithms in MAGMA which work over fields. For example, factorization of polynomials over such fields of fractions is supported (in any characteristic).

Rings of fractions have type `RngFunFrac` and their elements `RngFunFracElt`.

`RingOfFractions(Q)`

`FieldOfFractions(Q)`

Given an affine algebra Q over a field K, return the ring of fractions of Q. The only difference between the two functions is that for `FieldOfFractions`, the defining ideal of Q must be prime.

`Numerator(a)`

`Denominator(a)`

Given an element a from the ring of fractions of an affine algebra Q, return the numerator (resp. denominator) of a as an element of Q.

Example H108E7

We create the field of fractions of an affine algebra and note the basic operations.

```
> A<x,y> := AffineAlgebra<RationalField(), x,y | y^2 - x^3 - 1>;
> IsField(A);
false
> F<a,b> := FieldOfFractions(A);
> F;
Ring of Fractions of Affine Algebra of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Quotient relations:
[
    x^3 - y^2 + 1
]
```

```
> a;
a
> b;
b
> a^-1;
> a^-1;
1/(b^2 - 1)*a^2
> b^-1;
1/b
> c := b/a;
> c;
b/(b^2 - 1)*a^2
> Numerator(c);
x^2*y
> Denominator(c);
y^2 - 1
> P<X> := PolynomialRing(F);
> time Factorization(X^3 - b^2 + 1);
[
    <X - a, 1>,
    <X^2 + a*X + a^2, 1>
]
Time: 0.000
> P<X,Y> := PolynomialRing(F, 2);
> time Factorization((X + Y)^3 - b^2 + 1);
[
    <X + Y - a, 1>,
    <X^2 + 2*X*Y + a*X + Y^2 + a*Y + a^2, 1>
]
Time: 0.030
> time Factorization((b*X^2 - a)*(a*Y^3 - b + 1)*(X^3 - b^2 + 1));
[
    <Y^3 - 1/(b + 1)*a^2, 1>,
    <X - a, 1>,
    <X^2 - 1/b*a, 1>,
    <X^2 + a*X + a^2, 1>
]
Time: 0.010
```

Example H108E8

This example shows the internal operations underlying the method of constructing the field of fractions. If the ideal of relations has dimension d, then the sequence L of d maximally independent variables is passed to the extension/contraction construction, which creates a rational function field with d variables such that the ideal of relations over this field now becomes zero dimensional. Appropriate maps are set up, too.

```
> Q := RationalField();
```

```
> A<x,y> := AffineAlgebra<RationalField(), x,y | y^2 - x^3 - 1>;
> IsField(A);
false
> I := DivisorIdeal(A);
> I;
Ideal of Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Groebner basis:
[
    x^3 - y^2 + 1
]
> d, L := Dimension(I);
> d;
1
> L;
[ 2 ]
> E, f := Extension(I, L);
> E;
Ideal of Polynomial ring of rank 1 over Multivariate rational function
    field of rank 1 over Integer Ring
Graded Reverse Lexicographical Order
Variables: x
Basis:
[
    x^3 - y^2 + 1
]
> F := Generic(E)/E;
Affine Algebra of rank 1 over Multivariate rational function field of
    rank 1 over Integer Ring
Graded Reverse Lexicographical Order
Variables: x
Quotient relations:
[
    x^3 - y^2 + 1
]
> g := map<A -> F | x :-> F!f(x)>;
>
> g(x);
x
> g(y);
y
> g(x)^-1;
1/(y^2 - 1)*x^2
> g(y)^-1;
1/y
> g(x^2 + x*y);
x^2 + y*x
```

```
> g(x^2 + x*y)^-1;
y^2/(y^5 + y^4 - y^3 - 2*y^2 + 1)*x^2 + 1/(y^3 + y^2 - 1)*x - y/
    (y^3 + y^2 - 1)
> $1 * $2;
1
```

109 MODULES OVER MULTIVARIATE RINGS

Chapter 109
MODULES OVER MULTIVARIATE RINGS

109.1 Introduction

This chapter describes modules over multivariate polynomial rings and related rings. The fundamental tool for computing with such modules is the construction of Gröbner bases for modules, since these rings are not principal ideal rings in general (so standard matrix echelonization algorithms are not applicable).

In this chapter, unless otherwise indicated, a **ring** R will refer to one of the following:

(a) **Multivariate Polynomial Ring** (Chapters 24 and 105). Currently the coefficient ring of such a ring may be a field or Euclidean ring (even operations such as syzygy modules or free resolutions work over modules whose coefficient rings are Euclidean but not fields).

(b) **Local Polynomial Ring** (Localization of a Multivariate Polynomial Ring: Chapter 107; new in V2.15). Currently the coefficient ring of such a ring must be a field.

(c) **Affine Algebra** (Chapter 108). Currently the coefficient ring of such a ring must be a field.

(d) **Exterior Algebra** (Chapter 82; new in V2.15). Currently the coefficient ring of such a ring must be a field. Strictly speaking, this is a skew-commutative ring, so is not a commutative ring, and the associated modules are left R-modules, but the operations on R-modules in this chapter are practically all applicable if R is such an algebra also, so the term 'a ring R' will include such an algebra in this chapter.

In this chapter, the term "**module**" will always refer to an R-module, where R is one of the above types of ring, and such a module will have type `ModMPol` (or may have type `ModMPolGrd` if graded; see below). So we assume that the reader is generally familiar with such base rings and their ideals in MAGMA; see the relevant chapters for background. Many of the concepts and tools of Gröbner basis theory carry over from these types of rings.

109.2 Module Basics: Embedded and Reduced Modules

All of the modules considered in this chapter are ambient modules or embedded in such a module. We call an R-module **ambient** if it has the **explicit** presentational form $R^k/\langle relations\rangle$, where the relations are elements of R^k (and they may be zero or not even determined, initially). Elements of an ambient R-module M are represented explicitly as vectors in R^k, and M is always generated by the k unit vectors. The **degree** of M is k.

An arbitrary module S may have a representation as a submodule of such an ambient A, which is referred to as its **ambient module**. Hence the most general definition of a module is as a sub-quotient of a free module. If A has no relations then S is just a

submodule of a free module (namely, A). However, in this case, S will often also have an *internal* representation in presentational form that is essential for much of its fundamental functionality. In any case, the primary representation of elements of such an embedded module S is as vectors in the ambient.

As with vector spaces, there are two basic ways that modules can be defined in MAGMA: as embedded or reduced modules. A general subquotient as described above is in embedded form, but ambients may also be defined of either reduced or embedded type. The type primarily affects the way submodules and quotient modules are created. Briefly, submodules and quotient modules of embedded modules stay in embedded form (as generally proper submodules of an ambient) whereas submodules or quotients of reduced modules are always returned in presentational form as ambients, with connecting homomorphisms to link them explicitly to the original module. The two types are described in a bit more detail below. For illustration, see the examples at the end of Section 109.6.

Embedded modules are created in general via the function `EModule`, which returns a free embedded module, and in principle mimic the embedded R-spaces (as created by the function `RSpace(R, k)` in Chapter 54). Such modules are always presented with their elements and bases lying in an ambient module $R^k/\langle relations\rangle$. The modules are basically implemented as extensions of the multivariate polynomial ideal type (or affine algebra type if non-zero relations are present), where columns are internally added to monomials in a polynomial to represent a vector. Many operations applicable to ideals, including various Gröbner basis operations, naturally extend to such modules.

Starting with an ambient embedded module $M = R^k/\langle relations\rangle$, when a submodule S of M is created, the ambient module of S is still M, so the elements of S are represented as elements of R^k (modulo the relations if present); this therefore also applies to elements of any basis of S, including the Gröbner basis of S. Thus S itself may be not ambient and this is the only situation in which non-ambients can occur. Similarly, when a quotient module Q of M is constructed, the elements of Q appear as elements of R^k, while Q simply gains more relations than M, but its generators are usually not minimally reduced.

Reduced modules are created in general via the function `RModule`, which returns a free reduced module, and are more abstract and mimic the reduced modules with action over fields and Euclidean rings (as created by the function `RModule(R, k)` in Chapter 54). Such modules are **always ambient**, so always have the abstract form $R^n/\langle relations\rangle$, and the relationships between such modules are managed by **morphisms** lying in the background. The Gröbner basis techniques and properties are also hidden from the user in general.

Starting from a reduced module $M = R^k/\langle relations\rangle$, when a submodule S (having s generators $v_1, \ldots, v_s$) of M is created, S is generally created as $R^s/\langle relations_S\rangle$ (where the relations for S are initially unknown and are only computed when needed) and a morphism is stored from S to M, which maps the i-th unit vector of S to v_i in M. Similarly, a quotient module Q of M is constructed as another ambient module, usually with minimal generators, and a morphism from M onto Q is stored in the background. All morphisms between modules can be accessed via the function `Morphism`.

For any module M, there exists an isomorphic reduced **presentation module** P, which is always ambient, since P is reduced. If M is embedded, then P is a reduced

module equivalent to M (and morphisms in the background allow automatic coercion between M and P). Otherwise, M is already reduced so P is simply identical to M. Some functions (such as `FreeResolution`) always move to the presentation of M, since it is more natural to work only with ambient modules in that context.

Embedded modules are generally preferable when one wishes to work explicitly with Gröbner bases at a very low level, while reduced modules are generally preferable for homological computations since the ambient presentation form is more convenient (particularly for the relevant maps).

Technically, there is little difference in practice between an ambient embedded module and a reduced module, if each module is considered in isolation. The concepts basically refer to how submodules and quotient modules are derived from a given module (and the fact that embedded modules allow non-ambient submodules).

Finally, there is a subclass of reduced modules with the special type `ModMPolGrd`: these are **graded**, which means that they are always generated by homogeneous elements (with respect to the relevant grading). The main distinctive of this type is simply that when one creates a submodule or quotient module of a module of type `ModMPolGrd`, then the generators must be homogeneous, thus ensuring that the new derived module is also graded so will be of type `ModMPolGrd` also. In the future, more functions will be developed which will take modules of type `ModMPolGrd` explicitly. Note also that since the type `ModMPolGrd` ISA `ModMPol` via the type 'ISA' relation, any operation applicable to a module of type `ModMPol` is also applicable to a module of type `ModMPolGrd`.

109.3 Monomial Orders

In this section we describe each of the **module monomial orders** available in MAGMA. If the user wishes to work with reduced modules only (particularly for homology computations), then the underlying monomial orders and Gröbner bases will probably be rarely of interest to the user, so this section may be skipped. The monomial orders are mostly of interest if one wishes to work with embedded modules with special orders so that the relevant Gröbner bases have special properties. In either case, elements of the module are represented by vectors in an ambient and we refer to the vector component positions as *columns* in analogy to matrix terminology: a presentational module is often just defined by a matrix of relations, the rows giving vectors generating the relation module and the column numbering labelling the components of the vectors. For our modules there can be a non-trivial column weighting, which we think of as applying a shift to the degree of a homogeneous polynomial that occurs as the corresponding vector component of the module element. This is used to define homogeneity and degree of the overall vector.

Given an R-module M, suppose that the underlying monomial order of R is $<_R$. A **module monomial** of M is a monomial-column pair consisting of a monomial s of R and a column number c (with $c \geq 1$), written as $s[c]$ in the following. Monomial-column pairs give an (infinite) basis for the elements in a free module R^k and a vector representing an element of M can be decomposed into a sum of scalar multiples of monomial-column pairs just as elements of the polynomial ring R can be written as a sum of scalar multiples of plain monomials.

Now suppose that $s_1[c_1]$ and $s_2[c_2]$ are module monomials from M. Any order on the pairs is then fully defined by just specifying exactly when $s_1[c_1] < s_2[c_2]$ with respect to that order. As for multivariate polynomial rings, in the following the argument(s) are described for an order as a list of expressions; that means that the expressions (without the parentheses) should be appended to any base arguments when any particular intrinsic function is called which expects a module monomial order. See [AL94, Sec. 3.5] and [CLO98, Def. 2.4] for motivation and further discussion.

109.3.1 Term Over Position: `TOP`

Definition: $s_1[c_1] < s_2[c_2]$ iff $s_1 <_R s_2$ or $s_1 = s_2$ and $c_2 > c_1$. The order is specified by the argument (`"top"`).

This order is called "TOP" (term over position) since it first compares the underlying monomials (terms with the coefficients ignored†) and then compares the columns (the positions). The column comparison is ordered so that the first column is the greatest. A Gröbner basis of a module with respect to the `TOP` order is usually the easiest to compute, and corresponds to the `grevlex` order for polynomial rings in a certain way (i.e., the order favours the 'size' of monomials and only gives priority to the columns in a secondary way).

109.3.2 Term Over Position (Weighted): `TOPW`

Definition (given a sequence W of k integer weights, where k is the degree of the ambient module): write $d_i = \text{Degree}_W(s_i[c_i]) = \text{Degree}(s_i) + W[c_i]$; then $s_1[c_1] < s_2[c_2]$ iff $d_1 < d_2$ or $d_1 = d_2$ and $s_1 <_R s_2$ or $d_1 = d_2$, $s_1 = s_2$ and $c_2 > c_1$. The order is specified by the arguments (`"topw", W`). The weights need not be positive (but must be small integers).

This order first compares the degrees of the monomial-coefficient pairs using both the weights of the underlying ring R and the weights on the columns given by W and then proceeds as for the `TOP` order. If there is a natural grading W on the columns of the module, then it is preferable to use this order with W, particularly if submodules of interest are homogeneous or graded w.r.t. W, since then the GB w.r.t. this order will tend to be smaller and easier to compute. Normally one would also make the base order $<_R$ to be one of the `grevlex` or `grevlexw` degree orders (see Subsections 105.2.3, 105.2.3), so that the order $<$ extends the degree order $<_R$ to a degree order on the module.

109.3.3 Position Over Term: `POT`

Definition: $s_1[c_1] < s_2[c_2]$ iff $c_2 > c_1$ or $c_1 = c_2$ and $s_1 <_R s_2$. The order is specified by the argument (`"pot"`).

This order is called "POT" (position over term) since it first compares the columns and then compares the underlying monomials. The column comparison is ordered so that the first column is the greatest. A Gröbner basis of a module with respect to the `POT` order is like an echelon form of a matrix, since the order gives priority to the columns but this is in general rather harder to compute than the GB w.r.t. the `TOP` order.

---

† Some authors apply the terms 'monomial' and 'term' in opposite senses to how we do here, so that is why there are the established names 'TOP' and 'POT'; we follow this instead of using 'MOP' and 'POM'!

109.3.4 Position Over Term (Permutation): POTPERM

Definition (given a sequence P of k integers describing a permutation of $[1..k]$, where k is the degree of the ambient module): $s_1[c_1] < s_2[c_2]$ iff $P[c_2] > P[c_1]$ or $c_1 = c_2$ and $s_1 <_R s_2$. The order is specified by the arguments ("potperm", P).

This order first compares the columns using the given permutation, and then compares the underlying monomials.

109.3.5 Block TOP-TOP: TOPTOP

Definition (given a integer k): say that a column c is in the 1st block if $c \leq k$ and in the 2nd block if $c > k$; then $s_1[c_1] < s_2[c_2]$ iff c_2 is in the 1st block and c_1 is in the 2nd block, or if the columns are in the same block and $s_1[c_1] < s_2[c_2]$ w.r.t. the TOP order.

This order is a block order, like an elimination order for polynomial rings: comparison is first made on the blocks in which the columns lie, and then the TOP order is applied within each block. A GB w.r.t. this order is easier in general to compute than the POT order and so is useful when one wishes to 'eliminate' the first k columns only in a GB.

109.3.6 Block TOP-POT: TOPPOT

Definition (given a integer k): say that a column c is in the 1st block if $c \leq k$ and in the 2nd block if $c > k$; then $s_1[c_1] < s_2[c_2]$ iff c_2 is in the 1st block and c_1 is in the 2nd block, or if the columns are in the same block and $s_1[c_1] < s_2[c_2]$ w.r.t. the TOP/POT order (respective to the 1st/2nd blocks).

This order is a block order, like an elimination order for polynomial rings: comparison is first made on the blocks in which the columns lie, and then the TOP order is applied within the 1st block and the POT order is applied within the 2nd block. This is similar to the TOPTOP order, but it may be preferable to order the 2nd block w.r.t. the POT order. Note: POTPOT would equal to POT, and POTTOP does not seem to be useful.

109.4 Basic Creation and Access

An ambient free module $M = R^k$ is created by giving the base ring R (see introduction above), the degree r or a sequence W of r integers for the column weights, and, optionally, an argument specifying the type of module monomial order.

109.4.1 Creation of Ambient Embedded Modules

The following functions create ambient embedded modules.

`EModule(R, k)`

Given a ring R, create the ambient embedded module R^k with the default TOP module monomial order.

`EModule(R, k, order)`

Given a ring R, create the ambient embedded module R^k with the module monomial order described by the given order *order*. See Section 109.3 for the valid values for *order*.

`EModule(R, W)`

Given a ring R and a sequence W of k integers, create the ambient embedded module R^k with column weights given by W and with the `TOPW` module monomial order with weights W.

`EModule(R, W, order)`

Given a ring R and a sequence W of k integers, create the ambient embedded module R^k with column weights given by W and with the module monomial order described by the given order *order*. See Section 109.3 for the valid values for *order*.

109.4.2 Creation of Reduced Modules

The following functions create reduced modules, which are always ambient.

`RModule(R, k)`

Given a ring R, create the reduced module R^k with zero column weights.

`RModule(R, W)`

Given a ring R and a sequence W of k integers, create the reduced module R^k with column weights given by W.

`GradedModule(R, k)`

Given a ring R, create the reduced graded module R^k with zero column weights. The resulting module has type `ModMPolGrd`, so submodules and quotient modules of it may only be generated by homogeneous elements.

Note also that in general it is preferable if possible that the base ring R has a degree ordering (such as the `grevlex` or `grevlexw` orders) so that associated Gröbner bases of derived modules will be easier to compute.

`GradedModule(R, W)`

Given a ring R and a sequence W of k integers, create the reduced graded module R^k with column weights given by W. The resulting module has type `ModMPolGrd`, so submodules and quotient modules of it may only be generated by homogeneous elements.

109.4.3 Localization

`Localization(M)`

Given an R-module M, where $R = K[x_1, \ldots, x_n]$ for a field K, return the corresponding S-module $M_{\langle x_1,\ldots,x_n\rangle}$, where $S = K[x_1, \ldots, x_n]_{\langle x_1,\ldots,x_n\rangle}$ is the localization of R. See Chapter 107 for more information.

109.4.4 Basic Invariants

The following functions access simple defining invariants of a module M.

`Ambient(M)`
`Generic(M)`

Given a module M, return the ambient (or generic) module A in which M is embedded. The only case in which A differs from M is when M is a proper submodule of an ambient embedded module. So if M is reduced, A will always equal M.

`IsAmbient(M)`

Given a module M, return whether M is ambient.

`IsEmbedded(M)`

Given a module M, return whether M is embedded.

`IsReduced(M)`

Given a module M, return whether M is reduced.

`IsRoot(M)`

Given a module M, return whether M is a root (an independent module, not derived via sub- or quotient constructions from another module).

`CoefficientRing(M)`
`BaseRing(M)`

Given an R-module M, return the base ring R over which M is defined. Note that one can then call `BaseRing(R)` to obtain the underlying ring S in which the base coefficients of elements R lie.

`Degree(M)`

Given an R-module M, return the degree of M, which is the k such that the ambient module of M equals $R^k/\langle relations \rangle$. Note that if M is free and ambient, then the degree of M equals the rank of M, but otherwise in general the rank of M may be less than the degree of M (see the function `Rank` below).

`ColumnWeights(M)`
`Grading(M)`

Given a module M of degree k, return the grading of M, which is a sequence of k integers giving the grading on the columns of M.

`RelationModule(M)`

Given an R-module M of degree k, return the submodule of the embedded module R^k which is generated by the defining relations of M.

`Relations(M)`

Given an R-module M of degree k, return the defining relations of M as a sorted sequence of elements of the embedded module R^k.

`RelationMatrix(M)`

Given a module M, return the relation matrix of M, which is the matrix whose rows are the defining relations of M.

`Presentation(M)`

Given an R-module M, return the presentation module P of M. This is a reduced module isomorphic to M (and such that automatic coercion between M and P is allowed). If M is reduced, then P is identical to M.

`IsGraded(M)`

`IsHomogeneous(M)`

Given a module M, return whether M is graded (or equivalently, homogeneous), w.r.t. the grading of M (given by the weights on the columns of M and the variables of the base ring of M). This is true iff the Gröbner basis of M consists of homogeneous elements only (always true if M is reduced) and the Gröbner basis of the relation module of M consists of homogeneous elements alone. Note that a module of type `ModMPolGrd` is always graded.

109.4.5 Creation of Module Elements

Module elements (internally, multivariate polynomials with columns attached to the monomials) are constructed in general by giving a sequence or vector of elements from the coefficient ring R.

`M ! Q`

Suppose M is an R-module of degree r. Given a sequence $Q = [a_1, \ldots, a_r]$ of ring elements such that the a_i are coercible into R, construct the element of M corresponding to Q.

`M ! v`

Suppose M is an R-module of degree r. Given a vector v from the R-space R^r, construct the element of M corresponding to v.

`M ! 0`

`Zero(M)`

Create the zero element of the module M.

`UnitVector(M, i)`

Suppose M is an R-module of degree r. Given an integer i in the range $[1..r]$, construct the i-th unit vector of M (the vector with 1 in the i-th column and 0 elsewhere) whose parent is the ambient module of M (since it may not lie in M itself). Note that this *not* the same as the function `BasisElement` (below) which depends on the current basis of M.

109.4.6 Element Operations

The following functions allow simple access and operations on module elements. Some of them use the module structure and refer to the column structure of an element; others use the polynomial structure and ignore the column structure.

109.4.6.1 Access

`Eltseq(f)`

Given an element f of the R-module of degree r, return the sequence $[f_1, \ldots, f_r]$ of r elements from R corresponding to f.

`Vector(f)`

Given an element f of the module M over R and of degree r, return the element of the R-space of degree r over R corresponding to f.

`f[i]`

Given an element f of the R-module of degree r, together with an integer i in the range $[1..r]$, return the i-th component of f as an element of R.

109.4.6.2 Arithmetic

The following functions act on elements of R-modules. The operations are similar to those for multivariate polynomials or vectors, whenever meaningful. For the binary operations, the elements must be **compatible**; that is, their parents must have the same ambient module. Note that if quotient relations for M are present, then the result is reduced to the unique normal form modulo the quotient relations, but if the determination of the relations is delayed, then an element may have a non-unique representation, but all the predicates on elements below do not depend on the representation.

`f + g` `f - g` `- f` `r * f` `f * r`

Basic arithmetic operations. The element r lies in the base ring R.

`f div s`

Given a scalar ring element s and an element f of the module M, such that s is coercible into R s divides all components of f, return the quotient of f by s.

`SPolynomial(f, g)`

Given elements f and g of the module M such that the leading module monomials of f and g have the same column, return the S-polynomial of f and g. Note that the result is always reduced to the unique normal form modulo the quotient relations of M.

`Normalize(f)`

Given an element f of the module M, return the normalized form of f (so that the leading module monomial of f is normalized).

`NormalForm(f, S)`

Given an element f of the module M, together with a compatible module S, return the normal form of f with respect to S. This is unique if the base ring R is not local. In general, S will be a non-ambient embedded module for this to be useful (otherwise any f would already be in S so the result would always be zero).

`Coordinates(f, M)`

Given an element f of the R-module S, together with a compatible R-module M such that f is in M, return the coordinates of f with respect to the basis of M (whose components lie in R).

109.4.6.3 Accessing the Underlying Representation

The following functions access simple properties of module elements which are to do with the underlying representation.

`Coefficients(f)` `Monomials(f)` `Terms(f)`
`LeadingCoefficient(f)` `LeadingMonomial(f)`
`LeadingTerm(f)`
`CoefficientsAndMonomials(f)`

These functions are equivalent to the access functions for multivariate polynomials and access the underlying distributed polynomial representation (with columns added to the monomials); see Section 24.4.4 for details.

`Column(f)`

Given a single-term element f of a module M, return the column c of the single monomial-column pair (module monomial) $s[c]$ which f has.

`Degree(f)`
`WeightedDegree(f)`

Given an element f of a module M, return the weighted degree (abbreviated to 'degree' in this chapter) of f, which is the maximum of the weighted degrees of the monomial-column pairs of f. The weighted degree of a monomial-column $s[c]$ is the weighted degree of s (in the base ring R) plus the degree of column c in the grading of M.

`IsHomogeneous(f)`

Given an element f of a module M, return whether f is homogeneous; that is, whether the weighted degrees of all the monomial-columns of f are equal. (Note that the grading of M is thus significant.)

109.4.6.4 Predicates

`IsZero(f)`

Given an element f of the module M, return whether f is the zero element of M. Note that if the relations of M are non-zero this operation may be non-trivial (especially if the relations are not yet computed, but they will be automatically computed if needed).

`f eq g`

Given elements f and g of the module M, return whether f and g are equal. Note that this may be non-trivial (see the remarks above).

`f lt g`

Given elements f and g of the module M, return whether $f < g$ w.r.t. the underlying module monomial order. The operators `le`, `gt`, `ge` are similarly defined.

`f in M`

Given an element f of a module S together with a compatible module M, return whether f is in M.

Example H109E1

We illustrate simple modules over a multivariate polynomial ring. We construct simple ambient embedded modules over $\mathbf{Q}[x, y, z]$. The first module has default weights 0 on its columns, while the second has weights 1, 2, and 3 respectively on its columns.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3, "grevlex");
> M := EModule(R, 3);
> M;
Free Embedded Module R^3
Order: Module TOP: Graded Reverse Lexicographical
> f := M![x, y, z^2];
> g := M![z, y^3, x + 1];
> f;
[x, y, z^2]
> g;
[z, y^3, x + 1]
> f + g;
[x + z, y^3 + y, z^2 + x + 1]
> Terms(f);
[
    [0, 0, z^2],
    [x, 0, 0],
    [0, y, 0]
]
> Degree(f);
2
> [Degree(m): m in Monomials(f)];
```

```
[ 2, 1, 1 ]
> LeadingMonomial(f);
[0, 0, z^2]
> M2 := EModule(R, [10, 5, 1]);
Free Embedded Module R^3 with grading [10, 5, 1]
Order: Module TOP with column weights [10, 5, 1]: Graded Reverse Lexicographical
> f := M2![x, y, z^2];
> f;
[x, y, z^2]
> Terms(f);
[
    [x, 0, 0],
    [0, y, 0],
    [0, 0, z^2]
]
> Degree(f);
11
> [Degree(m): m in Monomials(f)];
[ 11, 6, 3 ]
```

Similar operations can be done with reduced modules. There is no difference for the elements.

```
> M := RModule(R, 3);
> M;
Free RModule R^3
> M := GradedModule(R, [10, 5, 1]);
> M;
Free Graded Module R^3 with grading [10, 5, 1]
> Grading(M);
[ 10, 5, 1 ]
> f := M![x, y^6, z^10];
> f;
[x, y^6, z^10]
> IsHomogeneous(f);
true
```

109.5 The Homomorphism Type

Magma has a special type for a homomorphism between two R-modules. The type of such a homomorphism is `ModMPolHom`. In general, functions such as `Morphism` return a homomorphism of type `ModMPolHom`, while the boundary maps of complexes are also of type `ModMPolHom` (see the function `FreeResolution`).

A homomorphism $f : M \to N$ is represented by a matrix A. There are two ways in which A can be defined:

(a) A is an **ambient** matrix: in this case, A gives the explicit map on the *ambient* modules of M and N. Thus A is $m \times n$, where $m =$`Degree(M)`, $n =$`Degree(N)`.

(b) A is a **presentation** matrix: in this case, A gives the explicit map on the *presentation* modules of M and N. Thus A is $m \times n$, where $m =$`Degree(Presentation(M))`, $n =$`Degree(Presentation(N))`.

If M and N are reduced (a common case), then they equal their respective presentation modules, so there is no difference between the above two cases (the ambient matrix and the presentation matrix are identical). So the only difference between (a) and (b) occurs when at least one of M and N is a non-ambient (proper) submodule of an embedded module.

When M and N are graded - that is, generated by elements homogeneous with respect to the ambient column weightings and with a relation module that is also generated by homogeneous elements - all homomorphisms as non-graded modules are still allowed. However there are functions to test if a given homomorphism preserves the gradings on the domain and codomain up to a constant degree shift. See `IsHomogeneous` and `Degree` below.

`Homomorphism(M, N, A)`

`Presentation` BoolElt *Default :* `true`

Given R-modules M and N and an $m \times n$ matrix A over R, construct the homomorphism $f : M \to N$ (with type `ModMPolHom`) defined by A.

By default, A is assumed to be a presentation matrix (see the comments above), in which case m and n must equal the degrees of the presentation modules of M and N, respectively. Alternatively, setting the parameter `Presentation` to `false` specifies that A is an ambient matrix; in this case, m and n must equal the degrees of M and N, respectively.

`Domain(f)`

Given a module homomorphism $f : M \to N$, return the domain M.

`Codomain(f)`

Given a module homomorphism $f : M \to N$, return the codomain N.

`PresentationMatrix(f)`

`Matrix(f)`

Given a module homomorphism $f : M \to N$, return the presentation matrix A_P of f as an $m \times n$ matrix corresponding to the presentation modules of M and N, respectively. This presentation matrix is always well-defined and computed, even if f is constructed via an ambient matrix.

`AmbientMatrix(f)`

`Matrix(f)`

Given a module homomorphism $f : M \to N$, return the ambient matrix A_A of f as an $m \times n$ corresponding to the ambient modules of M and N, respectively. If M and N are reduced (as commonly happens), this will be the same as the presentation matrix above. But if M and N are not reduced and f is constructed via a presentation matrix, then an error may result (since it may be impossible to give a matrix over the base ring R which gives the mapping for the ambient modules).

`f(v)`

`v * f`

Given a module homomorphism $f : M \to N$ and an element v of M, return the image of v under f, as an element of N.

`f[i]`

Given a module homomorphism $f : M \to N$ and an integer i, return the element of N corresponding to the i-th row of the ambient matrix of f.

`Image(f)`

Given a module homomorphism $f : M \to N$, return the image of f as a submodule of N (which will be reduced iff N is).

`Kernel(f)`

Given a module homomorphism $f : M \to N$, return the kernel of f as a submodule of M (which will be reduced iff M is).

`Cokernel(f)`

Given a module homomorphism $f : M \to N$, return the cokernel of f as a quotient module of N (which will be reduced iff N is).

`IsZero(f)`

Given a module homomorphism $f : M \to N$, return whether f is the zero map. Note that f may be the zero map even if the presentation or ambient matrices of f are non-zero.

`IsInjective(f)`

Given a module homomorphism $f : M \to N$, return whether f is injective (whether the kernel of f is the zero module).

`IsSurjective(f)`

Given a module homomorphism $f : M \to N$, return whether f is surjective (whether the image of f equals N).

`IsBijective(f)`

Given a module homomorphism $f : M \to N$, return whether f is bijective (injective and surjective).

`IsGraded(f)`
`IsHomogeneous(f)`

Given a module homomorphism $f : M \to N$, where M and N are graded modules, return whether f is homogeneous of some degree d; that is, whether for every pure degree element $v \in M$, $f(v) = 0$ or `Degree`$(f(v))$ equals `Degree`$(v) + d$.

`Degree(f)`

Given a module homomorphism $f : M \to N$, return the degree of f, which is the maximum d such that an element of M of degree e is mapped via f to zero or an element of degree $e + d$. If f is homogeneous, then the 'maximum' concept is unnecessary, since the degree will be consistent for all elements of M (see the previous function).

Example H109E2

We illustrate some homomorphism functionality by looking at the explicit inclusion homomorphism between two submodules of a rank 3 free module over $\mathbf{Q}[x, y]$. We define this in non-presentational form by the identity matrix. Then we can retrieve the corresponding defining matrix for the map between the internal presentations of the two submodules. The two submodules being graded submodules, we check that the inclusion is indeed homogeneous of degree 0 (as it must be, obviously preserving degrees of elements).

```
> R<x,y> := PolynomialRing(RationalField(), 2, "grevlex");
> F := EModule(R, 3);
> // get a submodule M1 generated by a single non-zero element of F
> M1 := sub<F|[x^2,y^2,x*y]>;
> // and a second submodule M2 containing M1
> M2 := sub<F|[x,0,y],[0,y,0]>;
> incl_hm :=  Homomorphism(M1,M2,IdentityMatrix(R,3) :
>               Presentation := false);
> incl_hm;
Module homomorphism (3 by 3)
Ambient matrix:
[1 0 0]
```

```
[0 1 0]
[0 0 1]
```

Now the corresponding presentation matrix of the inclusion map is the obvious one coming from the expression of the natural generator of M1 in terms of the two natural generators of M2

```
>  PresentationMatrix(incl_hm);
[y x]
> // check homogeneity of incl_hm
> IsHomogeneous(incl_hm);
true
> Degree(incl_hm);
0
```

109.6 Submodules and Quotient Modules

The following functions allow the construction of submodules and quotient modules and access to essential properties.

109.6.1 Creation

`sub< M | L >`

Given a module M over a ring R, return the submodule of M (with the same quotient relations as M) generated by the elements of M specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of M;

(b) A set or sequence of elements of M;

(c) A submodule of M;

(d) A set or sequence of submodules of M.

A morphism is stored from the resulting submodule S into M, such that $S.i$ is mapped to the i-th generator given in the above list.

`quo< M | L >`

Given a module M over a ring R, return the quotient module of M by the elements of M specified by the list L. Each term of the list L must be an expression defining an object of one of the following types:

(a) An element of M;

(b) A set or sequence of elements of M;

(c) A submodule of M;

(d) A set or sequence of submodules of M.

A morphism is stored from M onto the resulting quotient module Q.

`Morphism(M, N)`

Given modules M and N, related by a chain of stored sub and quo morphisms as mentioned above, returns the resulting morphism matrix map from M to N. If no known sub/quo relationship chain exists between M and N then an error is returned.

`Submodule(I)`

Given an ideal I of a polynomial ring R, return the submodule of R^1 generated by I.

`QuotientModule(I)`

Given an ideal I of a polynomial ring R, return the quotient module R^1/I.

`GradedModule(I)`

Given a homogeneous ideal I of a ring R, return the graded quotient module R^1/I.

109.6.2 Module Bases

The following functions allow one to manipulate the bases of modules. Note that a Gröbner basis for a module will be automatically generated when necessary; the `Groebner` procedure just allows explicit immediate construction of the Gröbner basis.

`Basis(M)`

Given a module M, return the current basis (whether it has been converted to a Gröbner basis or not) of M.

`BasisElement(M, i)`

Given a module M together with an integer i, return the i-th element of the current basis of M. Note that this is *not* the same as $M.i$.

`BasisMatrix(M)`

Given a module M, return the basis matrix of M, which is a k by r matrix over R, where k is the length of the basis of M and r is the degree of M.

`Groebner(M)`

(Procedure.) Explicitly force a Gröbner basis for the module M to be constructed.

Example H109E3

We construct simple submodules and quotient modules of an embedded module and consider some of their basic properties.

```
> R<x, y, z> := PolynomialRing(RationalField(), 3);
> M := EModule(R, 3);
> S := sub<M | [1, x, x^2+y], [z, y, x*y^2+1],
>              [y, z, x+z]>;
> Groebner(S);
> S;
Embedded Submodule of R^3
Order: Module TOP: Lexicographical
Groebner basis:
[ -x*z + y^2 + y, x*y^2 - x*y + z,         y^3 + z],
[  x*y - y*z - 1,   x*z - x - z^2,        -y - z^2],
[             y,               z,           x + z],
[       y^3 - z,       y^2*z - y,       y^2*z - 1]
> a := M ! [y, z, x+z];
> a;
[y, z, x + z]
> a in S;
true
> BasisElement(S, 1);
[-x*z + y^2 + y, x*y^2 - x*y + z, y^3 + z]
> Q := quo<M | [x, y, z]>;
> Q;
Embedded Module R^3/<relations>
Order: Module TOP: Lexicographical
Relations (Groebner basis):
[x, y, z]
> a := Q![x, y, 0];
> b := Q![0, 0, z];
> a;
[0, 0, -z]
> b;
[0, 0, z]
> a+b;
[0, 0, 0]
> Q ! [x,y,z];
[0, 0, 0]
> QQ := quo<Q | [x^2, 0, y+z]>;
> QQ;
Embedded Module R^3/<relations>
Order: Module TOP: Lexicographical
Relations (Groebner basis):
[          0,         x*y, x*z - y - z],
[          x,           y,           z]
> SL := Localization(S);
```

```
> SL;
Embedded Submodule of R^3 (local)
Order: Module TOP: Local Lexicographical
Basis:
[        1,         x,   x^2 + y],
[        z,         y, 1 + x*y^2],
[        y,         z,     x + z]
```

Example H109E4

We construct simple submodules and quotient modules of a reduced module and consider some of their basic properties.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> M := RModule(R, 3);
> S := sub<M | [1, x, x^2+y], [z, y, x*y^2+1]>;
> M;
Free Reduced Module R^3
> S;
Reduced Module R^2/<relations>
> Morphism(S, M);
Module homomorphism (2 by 3)
Ambient matrix:
[        1         x   x^2 + y]
[        z         y x*y^2 + 1]
> RelationMatrix(S);
Matrix with 0 rows and 2 columns
> S;
Free Reduced Module R^2
> M.1;
[1, 0, 0]
> M!S.1;
[1, x, x^2 + y]
> M!S.2;
[z, y, x*y^2 + 1]
> M.1 in S;
false
> Q := quo<M | [1, x^2, y]>;
> Q;
Free Reduced Module R^2
> RelationMatrix(Q);
Matrix with 0 rows and 2 columns
> Morphism(M, Q);
Module homomorphism (3 by 2)
Ambient matrix:
[-x^2   -y]
[   1    0]
[   0    1]
```

```
> Morphism(S, Q);
Module homomorphism (2 by 2)
Ambient matrix:
[       -x^2 + x             x^2]
[     -x^2*z + y x*y^2 - y*z + 1]
> Q!M.1;
[-x^2, -y]
> M!Q.1;
[0, 1, 0]
> M!Q.2;
[0, 0, 1]
> Q!M!Q.2;
[0, 1]
```

109.7 Basic Module Constructions

The following functions give some fundamental basic constructions with modules.

`M + N`

Given compatible modules M and N (ie, embedded in the same ambient module), return the sum of M and N; that is, the submodule of the ambient generated by M and N.

`M meet N`

Given compatible modules M and N (ie, embedded in the same ambient module), return the intersection of M and N in the ambient. This uses the standard algorithm for intersecting two modules of a free module (see Section 2.8.3 of [GP02]). If the ambient is the quotient of a free module F by non-trivial relations, the intersection performed is effectively that of the inverse images of M and N in F.

`f * M`
`M * f`

Given an R-module M and an element $f \in R$, return the submodule of M generated by $\{f \cdot v : v \in M\}$ or $\{v \cdot f : v \in M\}$, respectively.

`I * M`
`M * I`

Given an R-module M and an ideal I of R, return the submodule of M generated by $\{f \cdot v : f \in I, v \in M\}$ or $\{v \cdot f : f \in I, v \in M\}$, respectively.

`M / N`

Given compatible modules M and N (ie, embedded in the same ambient module), return the quotient module $M/(M \cap N)$. This has the same effect as using the `quo` constructor.

`DirectSum(M, N)`

Given R-modules M and N, return the direct sum $D = M \oplus N$ and two sequences of corresponding homomorphisms giving the injections into and projections from D, respectively.

`DirectSum(S)`

A sequence or list L of R-modules, return their direct sum D and two sequences of corresponding homomorphisms giving the injections into and projections from D, respectively.

`Twist(M, d)`

Given a graded module M, and an integer d, return the Serre twist $M(d)$ and an isomorphism $f : M \to M(d)$. The twisted module is simply an isomorphic copy of M, but with the grading twisted by d (so d is subtracted from each weight of M). f has degree $-d$.

109.8 Predicates

`IsZero(M)`

Given a module M, return whether M is the zero module.

`M subset N`

Given compatible modules M and N (ie, embedded in the same ambient module), return whether M is a submodule of N. This will generally involve module Gröbner basis and normal form computations to check that the generators of M lie in N.

`M eq N`

Given compatible modules M and N (ie, embedded in the same ambient module), return whether M equals N. The function checks that appropriate module Gröbner bases of M and N are equal.

`IsFree(M)`

Given an R-module M, return whether M is free. M is free iff M is isomorphic to the module R^k for some k. Such a k need not equal the degree of M but will equal the rank of M (as defined in the next section) if M is free. The function checks whether a minimised presentation of M has trivial relations or not.

109.9 Module Operations

The following functions perform some fundamental module operations.

`MinimalBasis(M)`

Given an R-module M, return a minimal basis B of M. If M is graded, or if R is a local ring, then the cardinality of B (the rank) is guaranteed to be unique (so is the absolutely minimal number of elements needed to generate M).

Otherwise the cardinality of B is not unique: B will only satisfy the rule that the i-th element of B is not in the submodule generated by elements 1 to $i-1$ of B.

In the graded case or local cases, a minimal basis is computed in the usual way starting from any basis B consisting of homogeneous elements. B gives a particular presentation whose relation matrix R consists of homogeneous polynomials. If R contains a non-zero constant term (or more generally a unit in the local case), an element of B can be eliminated and R recalculated. This can be continued until all non-zero terms of R have positive degree.

`MinimalBasis(S)`

Given a set or sequence S of homogeneous module elements from a module M, return a minimal basis of the submodule of M generated by S.

`Rank(M)`

Given an R-module M, return the rank of M. This is simply defined to be the cardinality of the minimal basis of M, returned by the function `MinimalBasis`. Thus if M is graded, or if R is a local ring, then the rank is guaranteed to be unique (and is the absolutely minimal number of elements needed to generate M). Otherwise the result is not an invariant of M, but simply reflects the minimum as found by the `MinimalBasis` algorithm.

`ColonModule(M, J)`

Given an R-module M and an ideal J of R, return the colon module $M : J$ which is the submodule of the ambient module A of M consisting of all $f \in A$ such that $f \cdot g \in M$ for all $g \in J$. When J is generated by a single element, this easily reduces to a syzygy computation in A and in the general case, we intersect the colon modules for a set of generators of J.

`ColonIdeal(M, N)`

Given an R-modules M and N which are both submodules of a common supermodule, return the colon ideal $M : N$, which is the ideal of R consisting of all $f \in R$ such that $f \cdot N \subset M$. The algorithm used is as described in section 2.8.4 of [GP02].

`Annihilator(M)`

Given an R-module M, return the annihilator ideal of M. This is the ideal I of R consisting of all $f \in R$ such that $f \cdot M = 0$ (which can be seen to equal the ideal $0_M : M$, where 0_M is the zero submodule of M, so is a special case of `ColonIdeal`).

`FittingIdeal(M, i)`

Given an R-module M of degree r and an integer $i \geq 0$, return the i-th Fitting ideal of M, which is the ideal of R generated by the $(r-i)$-th minors of the presentation matrix of M, where r is the degree of M. See [CLO98, p.229] or [Eis95, Sec. 20.2].

`FittingIdeals(M)`

Given an R-module M of degree r, return the Fitting ideals (for from 0 to r) as a sequence of ideals of R.

`SyzygyModule(M)`

Given a module M, return the syzygy module S of M. If the basis B of M has length k, the syzygy module S has degree k and elements of S express a syzygy amongst the k elements of the basis B. Note that the degree of the resulting module thus depends on the current basis of M.

`MinimalSyzygyModule(M)`

Given a homogeneous module M, return the syzygy module S of the minimal basis of M. If the minimal basis B of M has length k, the syzygy module S has degree k and elements of S express a syzygy amongst the k elements of the minimal basis B.

`SyzygyModule(Q)`

Given a sequence Q of polynomials from a multivariate polynomial ring P, return the module of syzygies of Q. This is a module over P of degree k, where k is the length of Q, consisting of all vectors v such that the sum of $v[i] * Q[i]$ for $i = 1, \ldots k$ is zero.

Example H109E5

In this example we note that a certain module M has rank 3 (equal to its degree 3), since no generator is redundant. If we move to the localization of M, then $(1 + x - z)$ becomes a unit, so the first generator becomes redundant.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3, "grevlex");
> F := RModule(R, 3);
> M := quo<F | [x + 1, y, z], [z, y, 0]>;
> M;
Reduced Module R^3/<relations>
Relations:
[x + 1,     y,     z],
[    z,     y,     0]
> Degree(M);
3
> Rank(M);
3
> ML := Localization(M);
> ML;
```

```
Reduced Module R^3/<relations> (local)
Relations:
[1 + x - z,         0,         z],
[        z,         y,         0]
> Rank(ML);
2
> MinimalBasis(ML);
[
    [0, 1, 0],
    [0, 0, 1]
]
```

109.10 Changing Ring

The `ChangeRing` function enables the changing of the polynomial ring over which a module is defined.

`ChangeRing(M, S)`

Given an R-module M, where R is a polynomial ring, and another polynomial ring S, construct the S-module N obtained by coercing the coefficients of the elements of the basis and relations of M into S. It is necessary that all elements of the old coefficient ring R can be automatically coerced into the new coefficient ring S. Note that S itself must be polynomial ring having the same rank as R, so S does not specify the new ring for the underlying coefficients (one can use `ChangeRing` for polynomial rings to do that first).

109.11 Hilbert Series

The following functions compute the Hilbert series information of graded or (homogeneous) modules. This depends on the column weights, just as in graded polynomial rings.

`HilbertSeries(M)`

Given a graded R-module M, return the Hilbert series $H_M(t)$ of M (as a univariate function field over the ring of integers. The i-th coefficient of the series gives the vector-space dimension of the degree-i graded piece of M. The algorithm implemented is that given in [BS92].

Note that if I is an ideal of the ring R, then the corresponding function for ideals `HilbertSeries` applied to I gives the Hilbert series of the affine algebra (quotient) R/I, so this is equivalent to `HilbertSeries(QuotientModule(I))`.

`HilbertSeries(M, p)`

Given a graded R-module M, return the Hilbert series $H_M(t)$ of M as a Laurent series to precision p. (A Laurent series is required in general, since negative powers may occur when there are negative values in the grading of M.)

`HilbertDenominator(M)`

Given a graded R-module M, return the unreduced Hilbert denominator D of the Hilbert series $H_M(t)$ of M (as a univariate polynomial over the ring of integers). The denominator D equals `HilbertDenominator(R)` which is simply

$$\prod_{i=1}^{n}(1-t^{w_i}),$$

where n is the rank of R and w_i is the weight of the i-th variable (1 by default).

`HilbertNumerator(M)`

Given a graded R-module M, return the unreduced Hilbert numerator N of the Hilbert series $H_M(t)$ of M (as a univariate polynomial over the ring of integers) and a valuation shift s. The numerator N equals $D \times t^s \times H_M(t)$, where D is the unreduced Hilbert denominator above. Computing with the unreduced numerator is often more convenient. Note that s will only be non-zero when M has negative weights in its grading.

`HilbertPolynomial(I)`

Given a graded R-module M, return the Hilbert polynomial $H(d)$ of M as an element of the univariate polynomial ring $\mathbf{Q}[d]$, together with the index of regularity of M (the minimal integer $k \geq 0$ such that $H(d)$ agrees with the Hilbert function of M at d for all $d \geq k$).

Example H109E6

We apply the Hilbert series functions to a simple quotient module.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> F := GradedModule(R, 3);
> M := quo<F | [x,0,0], [0,y^2,0]>;
> M;
Graded Module R^3/<relations>
Relations:
[  x,   0,   0],
[  0, y^2,   0]
> HilbertSeries(M);
(t^2 + t - 3)/(t^3 - 3*t^2 + 3*t - 1)
> HilbertSeries(M, 10);
3 + 8*s + 14*s^2 + 21*s^3 + 29*s^4 + 38*s^5 + 48*s^6 + 59*s^7 + 71*s^8 + 84*s^9
    + O(s^10)
> HilbertNumerator(M);
-x^2 - x + 3
0
> HilbertDenominator(M);
-x^3 + 3*x^2 - 3*x + 1
```

```
> HilbertPolynomial(M);
1/2*x^2 + 9/2*x + 3
0
> [Evaluate(HilbertPolynomial(F), i): i in [0..10]];
[ 3, 9, 18, 30, 45, 63, 84, 108, 135, 165, 198 ]
```

If the module has negative weights, then denominator may include extra powers of t, so the shift for the numerator will be non-zero.

```
> F := GradedModule(R, [-1]);
> F;
Free Graded Module R^1 with grading [-1]
> HilbertSeries(F);
-1/(t^4 - 3*t^3 + 3*t^2 - t)
> HilbertSeries(F, 10);
s^-1 + 3 + 6*s + 10*s^2 + 15*s^3 + 21*s^4 + 28*s^5 + 36*s^6 + 45*s^7 + O(s^8)
> HilbertNumerator(F);
1
1
> HilbertDenominator(F);
-x^3 + 3*x^2 - 3*x + 1
> HilbertPolynomial(F);
1/2*x^2 + 5/2*x + 3
-1
> [Evaluate(HilbertPolynomial(F), i): i in [-1..10]];
[ 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78 ]
```

109.12 Free Resolutions

The functions in this section deal with free resolutions and associated properties. Free resolutions are returned as chain complexes (see Chapter 56).

109.12.1 Constructing Free Resolutions

`FreeResolution(M)`

`Minimal`	BoolElt	*Default :* `true`
`Limit`	RngIntElt	*Default :* 0
`Homogenize`	BoolElt	*Default :* `true`
`Al`	MonStgElt	*Default :* "*LaScala*"

Given an R-module M, return a free resolution M as a complex C, and a comparison homomorphism $f : C_0 \to M$ (where C_0 is the term of C of degree 0).

By default, the free resolution will be **minimal**. Setting the parameter `Minimal` to `false` will construct a non-minimal resolution (which is constructed via a sequence of successive syzygy modules, with no minimization).

Magma has two algorithms for computing resolutions:

(1) The **La Scala** (LS) [SS98] algorithm, which works with a homogeneous module. The Magma implementation involves an extension of this algorithm which uses techniques from the Faugère F_4 [Fau99] algorithm to compute many normal forms together in a block.

(2) The **Iterative** algorithm, which simply computes successive syzygy modules progressively (minimizing as it goes if and only a minimal resolution is desired).

By default the LS algorithm is used if M is homogeneous and the coefficient ring of R is a finite field or the rational field, since this tends to be faster in general. But for some inputs the iterative algorithm may be significantly faster, particularly for some modules over the rationals. So one may set the parameter `Al` to `"Iterative"` to select the iterative algorithm. Uniqueness of the terms in the resolution is as follows.

(1) If M is homogeneous or defined over a local ring R, then the resulting complex C is guaranteed to be minimal, so the ranks of the terms in C and the associated Betti numbers will be unique.

(2) If M is non-homogeneous and over a global ring R, then the boundary maps of C will not have any entries which are units, but C cannot be guaranteed to be an absolutely minimal free resolution, so the ranks of the terms and the associated Betti numbers will not be unique in general. Also, Magma may choose to compute C by computing the free resolution C_H of a homogenization M_H of M, and then specializing C_H to yield C, since this method is usually faster (since the LS algorithm can then be used). One may set the parameter `Homogenize` to `true` or `false` to force Magma to use this homogenization technique or not.

If the parameter `Limit` is set to a non-zero value l, then at most l terms (plus the term corresponding to the free module) are computed. If R is an affine algebra or exterior algebra of rank n, then by default the limit is set to n, since the resolution is not finite in general.

`SetVerbose("Resolution", v)`

(Procedure.) Change the verbose printing level for the free resolution algorithm and related functions to be v.

Example H109E7

We construct the module $M = R^1/I$ where I is the ideal of the twisted cubic and then construct a minimal free resolution of M and note simple properties of this.

```
> R<x,y,z,t> := PolynomialRing(RationalField(), 4, "grevlex");
> B := [
>     -x^2 + y*t, -y*z + x*t, x*z - t^2,
>     x*y - t^2, -y*z + x*t, -x^2 + z*t
> ];
> M := GradedModule(Ideal(B));
```

```
> M;
Graded Module R^1/<relations>
Relations:
[-x^2 + y*t],
[-y*z + x*t],
[ x*z - t^2],
[ x*y - t^2],
[-y*z + x*t],
[-x^2 + z*t]
> C := FreeResolution(M);
> C;
Chain complex with terms of degree 4 down to -1
Dimensions of terms: 0 1 5 5 1 0
> Terms(C);
[
    Free Graded Module R^0,
    Free Graded Module R^1 with grading [5],
    Free Graded Module R^5 with grading [3, 3, 3, 3, 3],
    Free Graded Module R^5 with grading [2, 2, 2, 2, 2],
    Free Graded Module R^1,
    Free Graded Module R^0
]
> B := BoundaryMaps(C);
> B;
[*
    Graded module homomorphism (0 by 1),
    Graded module homomorphism (1 by 5) of degree 0
    Ambient matrix:
    [ x*z - t^2  x^2 - z*t -y*t + z*t  y*z - x*t -x*y + t^2],
    Graded module homomorphism (5 by 5) of degree 0
    Ambient matrix:
    [-y  x  0 -t  0]
    [ 0 -z  y  0  t]
    [ t -z  0  x  0]
    [ 0 -t  t  0  x]
    [-z  0  x -t  z],
    Graded module homomorphism (5 by 1) of degree 0
    Ambient matrix:
    [x^2 - z*t]
    [x*y - t^2]
    [x*z - t^2]
    [y*z - x*t]
    [y*t - z*t],
    Graded module homomorphism (1 by 0)
*]
> B[2]*B[3];
Module homomorphism (1 by 5)
Ambient matrix:
```

```
[0 0 0 0 0]
> B[3]*B[4];
Module homomorphism (5 by 1)
Ambient matrix:
[0]
[0]
[0]
[0]
[0]
> Image(B[3]) eq Kernel(B[4]);
true
```

Example H109E8

Following [CLO98, p.248], we compute the ideal I of $\mathbf{Q}[x, y]$ whose affine variety is a certain list of 6 pairs.

```
> R<x,y> := PolynomialRing(RationalField(), 2, "grevlex");
> L := [<0, 0>, <1, 0>, <0, 1>, <2, 1>, <1, 2>, <3, 3>];
> I := Ideal(L, R);
> I;
Ideal of Polynomial ring of rank 2 over Rational Field
Graded Reverse Lexicographical Order
Variables: x, y
Inhomogeneous, Dimension 0
Groebner basis:
[
    x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y,
    x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y,
    x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y,
    y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y
]
```

I is not homogeneous, and we compute a non-minimal free resolution of the module R/I.

```
> M := QuotientModule(I);
> M;
Reduced Module R^1/<relations>
Relations:
[  x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y],
[x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y],
[x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y],
[  y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y]
> C := FreeResolution(M: Minimal := false);
> C;
Chain complex with terms of degree 3 down to -1
Dimensions of terms: 0 3 4 1 0
> B := BoundaryMaps(C);
> B;
```

```
[*
    Module homomorphism (0 by 3),
    Module homomorphism (3 by 4)
    Ambient matrix:
    [-y + 5  x - 8      6     -2]
    [     4 -y - 8  x + 8     -4]
    [     2     -6 -y + 8  x - 5],
    Module homomorphism (4 by 1)
    Ambient matrix:
    [  x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y]
    [x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y]
    [x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y]
    [  y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y],
    Module homomorphism (1 by 0)
*]
> IsZero(B[2]*B[3]);
true
```

As noted in [CLO98], the 3 by 3 minors of the boundary map from R^3 to R^4 generate the ideal I again, and this is due to the Hilbert-Burch Theorem.

```
> U := Minors(Matrix(B[2]), 3);
> U;
[
    y^3 - 2*x^2 + 2*x*y - 5*y^2 + 2*x + 4*y,
    x*y^2 - 4*x^2 + 3*x*y - 5*y^2 + 4*x + 5*y,
    x^2*y - 5*x^2 + 3*x*y - 4*y^2 + 5*x + 4*y,
    x^3 - 5*x^2 + 2*x*y - 2*y^2 + 4*x + 2*y
]
> Ideal(U) eq I;
true
```

109.12.2 Betti Numbers and Related Invariants

Each of the functions in this section compute numerical properties of a free resolution of a module M. Each function takes the same parameters as the function `FreeResolution` (not repeated here), thus allowing control of the construction of the underlying resolution.

In particular, by default the **minimal** free resolution of M is used (so the Betti numbers correspond to that), so the relevant invariant is guaranteed to be unique if M is graded or over a local ring R. Otherwise, one may set the parameter `Minimal` to `false` to give the Betti numbers for a non-minimal resolution.

Note: If M is graded and the LS algorithm is used (which will be the case by default), then computing any of the invariants to do with Betti numbers in this section may be quicker than computing the full resolution (since minimization of the actual resolution is needed for the latter). Thus it is preferable just to use one of the following functions instead of `FreeResolution` if only the numerical invariants are desired.

`BettiNumbers(M)`

Given a module M, return the Betti numbers of M, which is simply the sequence of integers consisting of the degrees of the non-zero terms of the free resolution of M. See the discussion above concerning the parameters. Since the underlying resolution is minimal by default, if M is graded or over a local ring, then the result is unique.

`BettiNumber(M, i, j)`

Given a module M and integers $i, j \geq 0$, return the graded Betti number $\beta_{i,j}$ of M as an integer. This is the number of generators of degree j in the i-th term F_i of the free resolution of M.

`MaximumBettiDegree(M, i)`

Given a module M and an integer $i \geq 0$, return the maximum degree of the generators in the i-th term of the free resolution of M. Equivalently, this is the maximum j such that `BettiNumber(M, i, j)` is non-zero.

`BettiTable(M)`

Given a module M, return the Betti table of M as a sequence S of sequences of integers, and a shift s. This is designed so that if M is non-zero, then `S[1, 1]` is always non-zero and `S[i, j]` equals `BettiNumber(M, i, j - i + s)`. (So the degrees are shifted by s.)

`Regularity(M)`

Given an R-module M which is either graded or over a local ring, return the Castelnuovo-Mumford regularity. This is the least r such that in a minimal free resolution of M, the maximum of the degrees of the generators of the i-th term F_i is at most $i + r$. A simple consequence of this is that M is generated by elements of degree at most r. See [Eis95, Sec. 20.5] or [DL06, p. 167].

`HomologicalDimension(M)`

Given a module M, return the homological dimension of M. This is just the length of a minimal free resolution of M (the number of non-zero boundary maps).

Example H109E9

For an integer n, we can construct a Koszul complex as the free resolution of R/I, where I is the ideal of $R = K[x_1, \ldots, x_n]$ generated by the n variables.

```
> Q := RationalField();
> n := 3;
> R<[x]> := PolynomialRing(Q, n);
> I := Ideal([R.i: i in [1 .. n]]);
> M := QuotientModule(I);
> M;
Graded Module R^1/<relations>
Relations:
```

```
[x[1]],
[x[2]],
[x[3]]
> C := FreeResolution(M);
> C;
Chain complex with terms of degree 4 down to -1
Dimensions of terms: 0 1 3 3 1 0
> BoundaryMaps(C);
[*
    Module homomorphism (0 by 1),
    Module homomorphism (1 by 3)
    Ambient matrix:
    [ x[3] -x[2]  x[1]],
    Module homomorphism (3 by 3)
    Ambient matrix:
    [-x[2]  x[1]     0]
    [-x[3]     0  x[1]]
    [    0 -x[3]  x[2]],
    Module homomorphism (3 by 1)
    Ambient matrix:
    [x[1]]
    [x[2]]
    [x[3]],
    Module homomorphism (1 by 0)
*]
```

In general, the i-th Betti number is $\binom{n}{i}$. We can see this for $n = 10$. Each boundary map consists of linear relations alone, so the regularity is zero.

```
> n := 10;
> R<[x]> := PolynomialRing(Q, n);
> I := Ideal([R.i: i in [1 .. n]]);
> M := QuotientModule(I);
> time C := FreeResolution(M);
Time: 0.060
> C;
Chain complex with terms of degree 11 down to -1
Dimensions of terms: 0 1 10 45 120 210 252 210 120 45 10 1 0
> Terms(C);
[
    Free Graded Module R^0,
    Free Graded Module R^1 with grading [10],
    Free Graded Module R^10 with grading [9, 9, 9, 9, 9, 9, 9, 9, 9, 9],
    Free Graded Module R^45 with grading [8^^45],
    Free Graded Module R^120 with grading [7^^120],
    Free Graded Module R^210 with grading [6^^210],
    Free Graded Module R^252 with grading [5^^252],
    Free Graded Module R^210 with grading [4^^210],
    Free Graded Module R^120 with grading [3^^120],
```

```
    Free Graded Module R^45 with grading [2^^45],
    Free Graded Module R^10 with grading [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    Free Graded Module R^1,
    Free Graded Module R^0
]
> B := BoundaryMaps(C);
> B: Minimal;
[*
    Graded module homomorphism (0 by 1),
    Graded module homomorphism (1 by 10) of degree 0,
    Graded module homomorphism (10 by 45) of degree 0,
    Graded module homomorphism (45 by 120) of degree 0,
    Graded module homomorphism (120 by 210) of degree 0,
    Graded module homomorphism (210 by 252) of degree 0,
    Graded module homomorphism (252 by 210) of degree 0,
    Graded module homomorphism (210 by 120) of degree 0,
    Graded module homomorphism (120 by 45) of degree 0,
    Graded module homomorphism (45 by 10) of degree 0,
    Graded module homomorphism (10 by 1) of degree 0,
    Graded module homomorphism (1 by 0)
*]
> [Binomial(n, i): i in [0 .. n]];
[ 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1 ]
> BettiTable(M);
[
    [ 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1 ]
]
> $1 eq [[Binomial(n, i): i in [0 .. n]]];
true
> Regularity(M);
0
```

Example H109E10

We can construct the same type of ideal and module as in the last example for $n = 3$, but over an exterior algebra. The free resolution is infinite here, but we can construct the resolution partially (by default, a bound is set on the number of terms). In this general construction, the i-th Betti number will be $\binom{i}{n-1}$.

```
> Q := RationalField();
> n := 3;
> R<[x]> := ExteriorAlgebra(Q, n);
> I := Ideal([R.i: i in [1 .. n]]);
> M := QuotientModule(I);
> M;
Reduced Module R^1/<relations>
Relations:
[x[1]],
```

```
[x[2]],
[x[3]]
> BettiNumbers(M);
[ 1, 3, 6, 10, 15 ]
> [Binomial(i + n - 1, n - 1): i in [0..4]];
[ 1, 3, 6, 10, 15 ]
> C := FreeResolution(M);
> C;
Chain complex with terms of degree 5 down to -1
Dimensions of terms: 0 15 10 6 3 1 0
> BoundaryMaps(C);
[*
    Graded module homomorphism (0 by 15),
    Graded module homomorphism (15 by 10) of degree 0
    Ambient matrix:
    [x[3]    0    0    0    0    0    0    0    0    0]
    [   0 x[3]    0 x[2]    0    0    0    0    0    0]
    [   0    0 x[2]    0    0    0    0    0    0    0]
    [   0 x[2] x[3]    0    0    0    0    0    0    0]
    [x[2]    0    0 x[3]    0    0    0    0    0    0]
    [   0    0    0    0 x[3]    0    0    0    0 x[1]]
    [   0    0    0    0    0 x[2]    0    0 x[1]    0]
    [   0    0    0    0 x[2] x[3]    0 x[1]    0    0]
    [   0    0    0    0    0    0 x[1]    0    0    0]
    [   0    0    0    0    0 x[1] x[2]    0    0    0]
    [   0    0    0    0 x[1]    0 x[3]    0    0    0]
    [   0    0    0 x[1]    0    0    0 x[3]    0 x[2]]
    [   0    0 x[1]    0    0    0    0    0 x[2]    0]
    [   0 x[1]    0    0    0    0    0 x[2] x[3]    0]
    [x[1]    0    0    0    0    0    0    0    0 x[3]],
    Graded module homomorphism (10 by 6) of degree 0
    Ambient matrix:
    [x[3]    0    0    0    0    0]
    [   0 x[3] x[2]    0    0    0]
    [   0 x[2]    0    0    0    0]
    [x[2]    0 x[3]    0    0    0]
    [   0    0    0 x[3]    0 x[1]]
    [   0    0    0 x[2] x[1]    0]
    [   0    0    0 x[1]    0    0]
    [   0    0 x[1]    0 x[3] x[2]]
    [   0 x[1]    0    0 x[2]    0]
    [x[1]    0    0    0    0 x[3]],
    Graded module homomorphism (6 by 3) of degree 0
    Ambient matrix:
    [x[3]    0    0]
    [   0 x[2]    0]
    [x[2] x[3]    0]
    [   0    0 x[1]]
```

```
    [   0 x[1] x[2]]
    [x[1]    0 x[3]],
    Graded module homomorphism (3 by 1) of degree 0
    Ambient matrix:
    [x[3]]
    [x[2]]
    [x[1]],
    Graded module homomorphism (1 by 0)
*]
```

Example H109E11

We construct a non-homogeneous quotient module M of $\mathbf{Q}^3$. As expected, the Betti numbers of the localization of M are smaller than the Betti numbers of M.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3, "grevlex");
> R3 := RModule(R, 3);
> B := [R3 | [x*y, x^2, z], [x*z^3, x^3, y], [y*z, z, x],
>       [z, y*z, x], [y, z, x]];
> M := quo<R3 | B>;
> M;
Reduced Module R^3/<relations>
Relations:
[  x*y,   x^2,     z],
[x*z^3,   x^3,     y],
[  y*z,     z,     x],
[    z,   y*z,     x],
[    y,     z,     x]
> BettiNumbers(M);
[ 3, 5, 4, 2 ]
> BettiNumbers(Localization(M));
[ 3, 5, 3, 1 ]
```

Since M is non-homogeneous, the Betti numbers are not unique. If we create a second module M_2 which is equivalent to M and compute the Betti numbers this time without homogenization (in the internal free resolution algorithm), then we obtain different Betti numbers for M_2. But since the Betti numbers over a local ring are unique, we get the same result for the localization of M_2.

```
> M2 := quo<R3 | B>;
> BettiNumbers(M2: Homogenize :=false);
[ 3, 6, 5, 2 ]
> BettiNumbers(Localization(M2): Homogenize:=false);
[ 3, 5, 3, 1 ]
```

Example H109E12

Suppose M is a graded R-module. Given the graded Betti numbers $\beta_{i,j}$ of M, one can compute the Hilbert series $H_M(t)$ of M via the formula ([Eis95, Thm. 1.13] or [DL06, Thm. 1.22]):

$$H_M(t) = \frac{\sum_{i,j}(-1)^i \beta_{i,j} t^j}{D},$$

where D is the Hilbert denominator of M: this depends on the underlying ring R and equals

$$\prod_{i=1}^{n}(1 - t^{w_i}),$$

where n is the rank of R and w_i is the weight of the i-th variable (1 by default). We can thus write a simple function to compute the Hilbert series numerator via this formula.

```
> function HilbertNumeratorBetti(M)
>    P<t> := PolynomialRing(IntegerRing());
>    return &+[
>        (-1)^i*BettiNumber(M, i, j)*t^j:
>           j in [0 .. MaximumBettiDegree(M, i)],
>           i in [0 .. #BettiNumbers(M)]
>    ];
> end function;
```

We then check that this function agrees with the MAGMA internal function `HilbertNumerator` for some modules. (Since the modules do not have negative gradings, we do not have to worry about the denominator shift which is 0 for these modules.) First we try the Twisted Cubic.

```
> Q := RationalField();
> R<x,y,z,t> := PolynomialRing(Q, 4, "grevlex");
> B := [
>     -x^2 + y*t, -y*z + x*t, x*z - t^2,
>     x*y - t^2, -y*z + x*t, -x^2 + z*t
> ];
> M := GradedModule(Ideal(B));
> HilbertNumeratorBetti(M);
-t^5 + 5*t^3 - 5*t^2 + 1
> HilbertNumerator(M);
-t^5 + 5*t^3 - 5*t^2 + 1
0
```

Now we apply the function to the module $M = R^1/I$ where I is the ideal generated by the 2×2 minors of a generic 4×4 matrix. Computing the Hilbert series numerator via the Betti numbers takes a little time since the resolution is non-trivial. Note the components of the Betti table which contribute to the terms of the Hilbert series numerator.

```
> n := 4;
> R<[x]> := PolynomialRing(Q, n^2, "grevlex");
> A := Matrix(n, [R.i: i in [1 .. n^2]]);
```

```
> A;
[ x[1]  x[2]  x[3]  x[4]]
[ x[5]  x[6]  x[7]  x[8]]
[ x[9] x[10] x[11] x[12]]
[x[13] x[14] x[15] x[16]]
> I := Ideal(Minors(A, 2));
> #Basis(I);
36
> M := QuotientModule(I);
> time HilbertNumeratorBetti(M);
-t^12 + 36*t^10 - 160*t^9 + 315*t^8 - 288*t^7 + 288*t^5 - 315*t^4 + 160*t^3 -
    36*t^2 + 1
Time: 0.470
> time HilbertNumerator(M);
-t^12 + 36*t^10 - 160*t^9 + 315*t^8 - 288*t^7 + 288*t^5 - 315*t^4 + 160*t^3 -
    36*t^2 + 1
0
Time: 0.000
> assert $1 eq $2;
> BettiNumbers(M);
[ 1, 36, 160, 315, 388, 388, 315, 160, 36, 1 ]
> BettiTable(M);
[
    [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
    [ 0, 36, 160, 315, 288, 100, 0, 0, 0, 0 ],
    [ 0, 0, 0, 0, 100, 288, 315, 160, 36, 0 ],
    [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
]
0
```

Example H109E13

Given a graded module $M = R^1/I$, one can obtain an upper bound on the regularity of M by computing the regularity of $M_L = R^1/I_L$, where I_L is the leading monomial ideal of I. This will be faster in general (since the associated free resolution will be easier to compute).

```
> wts := [ 1, 5, 9, 13, 17, 5, 1, 1, 1 ];
> K := GF(32003);
> R<x0,x1,x2,x3,x4,y0,y1,u,t> := PolynomialRing(K, wts);
> I := Ideal([
>     x0*y0 - y1^3*u^3 - x1*t,
>     x1*y1 - x0*u^5 - t^6,
>     x1^2 - x0*x2 + y1^2*u^3*t^5,
>     x2^2 - x1*x3 + y0*y1*u^8*t^4,
>     x3^2 - x2*x4 + y0^2*u^13*t^3,
>     x3*y0 - u^18 - x4*t,
>     x4*y1 - x3*u^5 - y0^3*t^3,
>     x1*x2 - x0*x3 + y0*y1^2*u^3*t^4 + y1*u^8*t^5,
```

```
>     x2^2 - x0*x4 + y0*y1*u^8*t^4 + u^13*t^5,
>     x2*x3 - x1*x4 + y0^2*y1*u^8*t^3 + y0*u^13*t^4,
>     x1*y0 - y1^2*u^8 - x2*t,
>     x2*y0 - y1*u^13 - x3*t,
>     x2*y1 - x1*u^5 - y0*t^5,
>     x3*y1 - x2*u^5 - y0^2*t^4]);
> IsHomogeneous(I);
true
> M := GradedModule(I);
> time Regularity(M);
67
Time: 3.360
> IL := LeadingMonomialIdeal(I);
> ML := GradedModule(IL);
> time Regularity(ML);
92
Time: 0.530
> BettiNumbers(M);
[ 1, 14, 45, 72, 76, 58, 29, 8, 1 ]
> BettiNumbers(ML);
[ 1, 42, 210, 505, 723, 659, 388, 144, 31, 3 ]
```

Example H109E14

The following example shows how to explicitly use the resolution and syzygy functions to compute the ideal of a random space curve (in P^3) of genus 11. The construction is described in Section 1.2 of [ST02] and an equivalent form of the following computation is used by MAGMA's `RandomCurveByGenus` function to produce such curves.

We work over the field $GF(101)$, which will be referred to as K and the polynomial ring R will be the 4 variable polynomial ring over K. The construction begins by choosing a random 8 x 3 matrix with entries given by random linear and quadratic polynomials of R in appropriate positions. The minimal free resolution of the reduced module having this as the matrix of relations is computed. The image of the second boundary map of the resolution is the module referred to as $\mathcal{G}^*$ in the above reference. Taking the submatrix of rows of a certain weighting of the matrix defining this map, we multiply by a 6 x 8 matrix with random entries in K. The resulting matrix represents a map from a free module F of rank 6 to $\mathcal{G}^*$, whose kernel is isomorphic to R as a submodule of F. The 6 coordinates of a generator of the kernel generate the desired ideal I. This kernel is computed with a syzygy computation (note: we could also use `Kernel` for the matrix giving the map). We also check that the quotient module of I has a minimal free resolution of the right form.

```
> K := GF(101);
> R<x,y,z,t> := PolynomialRing(GF(101),4,"grevlex");
> v := [1,1,1,1,1,1,2,2];
> // generate the base random relations with appropriate linear and quadratic
> // entries using the Random function for multivariate polynomials.
> rels := [[Random(i,R,0): j in [1..3]] : i in v];
> Matrix(8,3,[TotalDegree(e) : e in &cat(rels)]);
```

```
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
[2 2 2]
[2 2 2]
> // get the quotient module
> F := RModule(R,3);
> M := quo<F|rels>;
```

Get the minimal free resolution and check that it has the correct Betti table.

```
> res := MinimalFreeResolution(M);
> BettiTable(res);
[
    [ 3, 6, 0, 0, 0 ],
    [ 0, 2, 8, 0, 0 ],
    [ 0, 0, 3, 10, 4 ]
]
0
```

Get the 2nd boundary map matrix and then the 8 x 8 submatrix of linear and quadratic entry rows.

```
> mat :=  Matrix(BoundaryMap(res,2));
> Nrows(mat); Ncols(mat);
11
8
> u := [1,1,2,2,2,2,2,2];
> mat := Matrix(R,[ri : i in [1..11] |
>          &and[(ri[j] eq 0) or (TotalDegree(ri[j]) eq u[j]):
>             j in [1..8]] where ri is Eltseq(mat[i])]);
> Nrows(mat); Ncols(mat);
8
8
> Matrix(8,8,[TotalDegree(m) :m in Eltseq(mat)]);
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
[1 1 2 2 2 2 2 2]
```

Now generate the random 6 x 8 matrix over K, compute the kernel of the composition via syzygies and generate the matrix I.

```
> mat1 := Matrix(R,6,8,[Random(K) : i in [1..48]]);
```

```
> matc := mat1*mat;
> F1 := EModule(R,[2-x : x in u]);
> syz := SyzygyModule(sub<F1|RowSequence(matc)>);
> B := MinimalBasis(syz);
> #B;
1
> I := ideal<R|Eltseq(B[1])>;
```

Finally, check I has the right dimension (2) and degree (12) and that R/I has the correct minimal free resolution with Betti table as given in [ST02].

```
> Dimension(I); Degree(I);
2 [ 3, 4 ]
12
> OC := QuotientModule(I);
> BettiTable(MinimalFreeResolution(OC));
[
    [ 1, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 0, 0, 0 ],
    [ 0, 6, 2, 0 ],
    [ 0, 0, 6, 3 ]
]
0
```

109.13 The Hom Module and Ext

`Hom(M, N)`

Given $R-$modules M and N, return $H = \mathrm{Hom}_R(M, N)$ as an abstract reduced module and a transfer map $f : H \to S$, where S is the set of all homomorphisms (of type `ModMPolHom`) from M to N.

Thus H is a module representing the set of all homomorphisms from M to N, while f maps an element $h \in H$ to an actual homomorphism from M to N (and the inverse image of an element of S under f gives a corresponding element of H).

If M and N are graded, then H is graded also, and the degree d_f of an element $f \in H$ is the degree of the corresponding homomorphism (so an element in M of degree d will be mapped by f to zero or an element of degree $d_f + d$ in N).

`Hom(C, N)`

Given a complex C of R-modules and an R-module N, return $\mathrm{Hom}_R(C, N)$. This is a new complex whose i-th term is $\mathrm{Hom}_R(C_i, N)$ (where C_i is the i-th term of C); the boundary maps are also derived from those of C in the natural way via the functor $\mathrm{Hom}_R(-, N)$ (see [Eis95, p.63]). Note that the direction of arrows in this complex is opposite to that of C.

`Ext(i, M, N)`

Given an integer $i \geq 0$ and R-modules M and N, return $\mathrm{Ext}^i(M, N)$. This is the homology at the i-th term of the complex $\mathrm{Hom}_R(C, N)$ where C is a free resolution of M.

Example H109E15

We construct a Hom module and explicit homomorphisms derived from it.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> M := quo<GradedModule(R, 3) |
>     [x*y, x*z, y*z], [y, x, y],
>     [0, x^3 - x^2*z, x^2*y - x*y*z], [y*z, x^2, x*y]>;
> N := quo<GradedModule(R, 2) |
>     [x^2, y^2], [x^2, y*z], [x^2*z, x*y^2]>;
> M;
Graded Module R^3/<relations>
Relations:
[          x*y,           x*z,           y*z],
[            y,             x,             y],
[            0,   x^3 - x^2*z, x^2*y - x*y*z],
[          y*z,           x^2,           x*y]
> N;
Graded Module R^2/<relations>
Relations:
[  x^2,   y^2],
[  x^2,   y*z],
[x^2*z, x*y^2]
> H, f := Hom(M, N);
> H;
Graded Module R^7/<relations> with grading [1, 2, 1, 1, 1, 1, 1]
Relations:
[x, 0, 0, -z, 0, x, 0],
[y, 0, x, 0, y, 0, 0],
[y, 0, x, 0, 0, y, 0],
[0, 0, 0, 0, 0, 0, y],
[-y, 0, -x, 0, -z, 0, z],
[x, 0, 0, -y, x, 0, 0],
[x*y, y, 0, 0, 0, 0, x*y],
[-x*y + x*z, -y + z, 0, 0, 0, 0, x*z - z^2],
[x*z, x, 0, 0, 0, 0, 0],
[0, y, 0, y^2, -z^2, 0, z^2],
[0, y - z, 0, 0, 0, 0, z^2]
> h := f(H.1);
> h;
Module homomorphism (3 by 2) of degree 1
Presentation matrix:
[0 z]
```

```
[x 0]
[0 0]
> $1 @@ f;
[1, 0, 0, 0, 0, 0, 0]
> Degree(M.1);
0
> h(M.1);
[0, z]
> Degree(h(M.1));
1
> f(Basis(H));
[
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [0 z]
    [x 0]
    [0 0],
    Module homomorphism (3 by 2) of degree 2
    Presentation matrix:
    [   0 -z^2]
    [   0  y*z]
    [   0    0],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0  0]
    [-y  0]
    [ x  0],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0  0]
    [ 0 -y]
    [ 0  x],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0 -z]
    [ 0  0]
    [ 0  y],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [ 0 -z]
    [ 0  0]
    [ 0  z],
    Module homomorphism (3 by 2) of degree 1
    Presentation matrix:
    [    0 y - z]
    [    0     0]
    [    0     0]
```

```
]
```

109.14 Tensor Products and Tor

`TensorProduct(M, N)`

Given $R-$modules M and N, return the tensor product $M \otimes_R N$ as an ambient module T, together with the associated map $f : M \times N \to T$. If M and N are graded, then T is graded also.

`TensorProduct(C, N)`

Given a complex C of R-modules and an R-module N, return $C \otimes_R N$. This is a new complex whose i-th term is $C_i \otimes_R N$ (where C_i is the i-th term of C); the boundary maps are also derived from those of C in the natural way via the functor $- \otimes_R N$ (see [Eis95, p.64]).

`Tor(i, M, N)`

Given an integer $i \geq 0$ and R-modules M and N, return $\mathrm{Tor}^i(M, N)$. This is the homology at the i-th term of the complex $C \otimes_R N$ where C is a free resolution of M.

Example H109E16

We construct a tensor product and some Tor modules for the same modules from the previous example.

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> M := quo<GradedModule(R, 3) |
>     [x*y, x*z, y*z], [y, x, y],
>     [0, x^3 - x^2*z, x^2*y - x*y*z], [y*z, x^2, x*y]>;
> N := quo<GradedModule(R, 2) |
>     [x^2, y^2], [x^2, y*z], [x^2*z, x*y^2]>;
> T, f := TensorProduct(M, N);
> T;
Graded Module R^6/<relations>
Relations (Groebner basis):
[x^2, y*z, 0, 0, 0, 0],
[0, 0, 0, 0, x^2, y*z],
[0, 0, 0, 0, 0, x*y*z - y*z^2],
[x*y - y*z, 0, 0, 0, 0, 0],
[0, x*y - y*z, 0, 0, 0, 0],
[y*z, 0, 0, -y*z, x*y, 0],
[y, 0, x, 0, y, 0],
[0, y, 0, x, 0, y],
[0, y^2 - y*z, 0, 0, 0, 0],
[0, 0, 0, y^2 - y*z, 0, 0],
```

```
[0, 0, 0, 0, 0, y^2 - y*z],
[y*z^2, 0, 0, -y*z^2, 0, -y*z^2],
[0, y*z^2, 0, y*z^2, 0, y*z^2]
```

Note that f maps the cartesian product of M and N into T.

```
> f(<M.1, N.1>);
[1, 0, 0, 0, 0, 0]
> [f(<m, n>): n in Basis(N), m in Basis(M)];
[
    [1, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 1]
]
```

Finally we construct associated Tor modules.

```
> Tor(0, M, N);
Graded Module R^6/<relations>
Relations:
[y, 0, x, 0, y, 0],
[0, y, 0, x, 0, y],
[0, 0, 0, 0, x*y - y*z, 0],
[0, 0, 0, 0, 0, x*y - y*z],
[y*z, x^2, 0, 0, 0, 0],
[x*y*z - y*z^2, 0, 0, 0, 0, 0],
[y^2 - y*z, 0, 0, 0, 0, 0],
[0, 0, y*z, x^2, 0, 0],
[0, 0, x*y*z - y*z^2, 0, 0, 0],
[0, 0, y^2 - y*z, 0, 0, 0],
[0, 0, 0, 0, y*z, x^2],
[0, 0, 0, 0, y^2 - y*z, 0],
[0, 0, 0, 0, x*y*z - y*z^2, 0]
> Tor(1, M, N);
Graded Module R^2/<relations> with grading [3, 3]
Relations:
[y - z,     0],
[    z,    -y],
[  z^2,  -x*y],
[    0,     0]
> Tor(2, M, N);
Free Reduced Module R^0
```

109.15 Cohomology Of Coherent Sheaves

We have implemented functions to compute the dimensions of cohomology groups of coherent sheaves on ordinary projective space over an (exact) field. The sheaves are represented by graded modules over the coordinate ring of the ambient projective space. The sheaf may arise naturally as one supported on a particular closed subscheme (*eg*, the structure sheaf of a projective variety) but it is a matter of indifference whether the sheaf is considered as lying on the subscheme or the entire ambient space (equivalently, whether the representing module is considered as a module over the coordinate ring of the ambient or the quotient coordinate ring of the subscheme) because the cohomology groups are naturally isomorphic. We plan to add a fuller package of functionality for coherent sheaves, but it is convenient to add the cohomology function now as the algorithm we use works equally efficiently (roughly speaking) when applied to any two graded modules that represent the same coherent sheaf.

The algorithm we have implemented is that of Decker, Eisenbud, Floystad and Schreyer which uses the Beilinson-Gelfand-Gelfand (BGG) correspondence to reduce the computation of the cohomology groups of the sheaf and its Serre twists to that of various graded free modules in the projective resolution of a module over a finite exterior (alternating) algebra.

`CohomologyDimension(M,r,n)`

Verbose `Cohom` *Maximum :* 1

M is a graded module over $P = k[x_0, .., x_m]$ with k an exact field. Let $\tilde{M}$ be the corresponding coherent sheaf on $Proj(P) = \mathbf{P}_k^m$. The function returns the k-dimension of the cohomology group $H^r(\mathbf{P}_k^m, \tilde{M}(n))$ where $\tilde{M}(n)$ is the nth Serre twist of $\tilde{M}$. n can be any integer and r a non-negative integer.

The algorithm used is based on the BGG correspondence. Details can be found in [EFS03] or see [DE02] for a slightly more computational description. Let A be the finite exterior algebra with $m+1$ generators, which is of dimension 2^{m+1} over k. The *Tate resolution* of $\tilde{M}$ is a doubly infinite exact sequence of graded free A-modules. Each cohomology group of a twist of $\tilde{M}$ is isomorphic as a k vector space to a particular graded piece of a particular term in the Tate resolution. In fact, we never need to explicitly compute the terms of the resolution of index $\geq reg(M)$ (the regularity of M) because they are pure graded of dimension given by the Hilbert polynomial of M.

The algorithm computes two consecutive terms in the Tate resolution at indices $\geq reg(M)$, and the A-homomorphism between them, from two corresponding graded pieces of M and the linear maps between them coming from multiplication by the base variables. Then the resolution is extended backwards as far as necessary by computing the A-projective resolution of the kernel of this A-homomorphism. The projective resolution is efficiently determined by non-commutative Gröbner basis computations. This uses the new MAGMA machinery for exterior algebras and their modules. The projective resolution information is cached so that repeated calls to the function for the same module M will require either no extra work or only an

extension to the part of the resolution already computed.

Example H109E17

We consider a random surface X in a family of Enriques surfaces of degree 9 in $\mathbf{P}^4$. It is defined by 15 degree 5 polynomials and we work over $\mathbf{F}_{17}$ to keep the input of reasonable size (it is still fairly large!).
The surface is non-singular with arithmetic genus (p_a), geometric genus (g) and irregularity q all zero. These are related generally for a non-singular surface by $g = p_a + q$ and p_a can be computed without cohomology machinery (from Hilbert polynomials). But cohomology of the structure sheaf of X and Serre duality is the easiest way to get g or q.

```
> R<x,y,z,t,u> := PolynomialRing(GF(17),5,"grevlex");
> I := ideal<R |[
>  2*x^3*z*t + 5*x^2*y*z*t + 14*x^2*z^2*t + x^3*z*u + 5*x^2*y*z*u +
>   10*x^2*z^2*u + 8*x*y*z^2*u + 15*y^2*z^2*u + 4*x*z^3*u + 2*y*z^3*u +
>   9*x^3*t*u + 14*x^2*y*t*u + 16*x^2*z*t*u + 10*x*y*z*t*u + 10*x*z^2*t*u +
>   y*z^2*t*u + 13*x^3*u^2 + 14*x^2*y*u^2 + 11*x^2*z*u^2 + 15*x*y*z*u^2 +
>   10*y^2*z*u^2 + 8*x*z^2*u^2 + 8*y*z^2*u^2 + x^2*t*u^2 + 11*x*y*t*u^2 +
>   16*x*y*u^3 + 9*y^2*u^3 + 4*x*z*u^3 + 2*y*z*u^3 + 10*x*t*u^3 + y*t*u^3 +
>   8*x*u^4 + 8*y*u^4,
>  5*x^3*z*t + x^2*z^2*t + 5*x^3*z*u + 11*x^2*z^2*u + 15*x*y*z^2*u + 2*x*z^3*u
>   + 14*x^3*t*u + 5*x^2*z*t*u + x*z^2*t*u + 14*x^3*u^2 + 10*x*y*z*u^2 +
>   8*x*z^2*u^2 + 15*x^2*t*u^2 + 11*x^2*u^3 + 9*x*y*u^3 + 2*x*z*u^3 +
>   x*t*u^3 + 8*x*u^4,
>  14*x^3*z*t + x^2*y*z*t + 13*x^2*z^2*t + 7*x^2*z*t^2 + 3*x^3*z*u +
>   16*x^2*y*z*u + 4*x^2*z^2*u + 6*x^3*t*u + 16*x^2*y*t*u + 9*x^2*z*t*u +
>   9*x^2*t^2*u + 11*x^3*u^2 + x^2*y*u^2 + 14*x^2*z*u^2 + 2*x*z^2*u^2 +
>   11*y*z^2*u^2 + 6*z^3*u^2 + 4*x^2*t*u^2 + 4*x*z*t*u^2 + 14*y*z*t*u^2 +
>   6*z^2*t*u^2 + 15*x*t^2*u^2 + 10*z*t^2*u^2 + 3*x^2*u^3 + 11*x*z*u^3 +
>   16*y*z*u^3 + 4*z^2*u^3 + 16*x*t*u^3 + 3*y*t*u^3 + 14*z*t*u^3 + 6*t^2*u^3
>   + 13*x*u^4 + 7*y*u^4 + 16*z*u^4 + 11*t*u^4 + 10*u^5,
>  15*x^3*z^2 + 12*x^2*y*z^2 + 3*x^2*z^3 + 12*x^3*z*u + 8*x^2*y*z*u +
>   11*x^2*z^2*u + x^3*u^2 + 14*x^2*y*u^2 + 3*x^2*z*u^2 + 11*x^2*u^3,
>  12*x^3*z^2 + 16*x^2*z^3 + 8*x^3*z*u + x^2*z^2*u + 14*x^3*u^2 + 16*x^2*z*u^2
>   + x^2*u^3,
>  2*x^3*y*z + 5*x^2*y^2*z + 14*x^2*y*z^2 + 13*x^3*y*u + 12*x^2*y^2*u +
>   8*x^3*z*u + 4*x^2*y*z*u + 12*x^2*z^2*u + 3*x*y*z^2*u + 14*y^2*z^2*u +
>   15*x*z^3*u + 3*y*z^3*u + 15*x^2*y*t*u + 4*x^2*z*t*u + 15*x*y*z*t*u +
>   11*x*z^2*t*u + 10*y*z^2*t*u + 2*x^3*u^2 + 12*x^2*y*u^2 + 3*x^2*z*u^2 +
>   14*x*y*z*u^2 + 13*x*z^2*u^2 + 10*y*z^2*u^2 + x^2*t*u^2 + 15*x*y*t*u^2 +
>   16*y*z*t*u^2 + 12*x*y*u^3 + 3*y^2*u^3 + 15*x*z*u^3 + 4*y*z*u^3 +
>   11*x*t*u^3 + 6*y*t*u^3 + 13*x*u^4 + 14*y*u^4,
>  5*x^3*y*z + x^2*y*z^2 + 10*x^2*z^2*t + 4*x^4*u + 12*x^3*y*u + 5*x^3*z*u +
>   12*x^2*y*z*u + 14*x*y*z^2*u + 3*x*z^3*u + 15*x^3*t*u + 16*x^2*z*t*u +
>   10*x*z^2*t*u + 11*x^3*u^2 + 4*x^2*y*u^2 + 13*x^2*z*u^2 + 10*x*z^2*u^2 +
>   4*x^2*t*u^2 + 16*x*z*t*u^2 + 13*x^2*u^3 + 3*x*y*u^3 + 4*x*z*u^3 +
>   6*x*t*u^3 + 14*x*u^4,
>  10*x^2*z^3 + 8*x^2*z^2*u + 5*x^2*z*u^2 + 11*x^2*u^3,
```

```
>  16*x^3*z^2 + 12*x^2*y*z^2 + 7*x^2*z^3 + 9*x*y*z^3 + 2*y^2*z^3 + 13*x*z^4 +
>   15*y*z^4 + 13*x^3*z*t + 12*x^2*y*z*t + 7*x^2*z^2*t + 7*x*y*z^2*t +
>   7*x*z^3*t + 16*y*z^3*t + 4*x^3*z*u + 3*x^2*y*z*u + 6*x^2*z^2*u +
>   2*x*y*z^2*u + 7*y^2*z^2*u + 9*x*z^3*u + 9*y*z^3*u + 16*x^3*t*u +
>   3*x^2*y*t*u + 13*x^2*z*t*u + 6*x*y*z*t*u + x*y*z*u^2 + 8*y^2*z*u^2 +
>   13*x*z^2*u^2 + 15*y*z^2*u^2 + 6*x^2*t*u^2 + 7*x*z*t*u^2 + 16*y*z*t*u^2 +
>   9*x*z*u^3 + 9*y*z*u^3,
>  12*x^3*z^2 + 6*x^2*z^3 + 2*x*y*z^3 + 15*x*z^4 + 12*x^3*z*t + 11*x^2*z^2*t +
>   16*x*z^3*t + 3*x^3*z*u + 7*x*y*z^2*u + 9*x*z^3*u + 3*x^3*t*u +
>   3*x^2*z*t*u + 6*x^2*z*u^2 + 8*x*y*z*u^2 + 15*x*z^2*u^2 + 16*x^2*t*u^2 +
>   16*x*z*t*u^2 + 9*x*z*u^3,
>  x^3*y*z + 5*x^2*y^2*z + 10*x^2*y*z^2 + 8*x*y^2*z^2 + 15*y^3*z^2 + 4*x*y*z^3
>   + 2*y^2*z^3 + 13*x^3*y*t + 2*x^2*y^2*t + 9*x^3*z*t + 12*x^2*y*z*t +
>   10*x*y^2*z*t + 5*x^2*z^2*t + 7*x*y*z^2*t + 4*y^2*z^2*t + 2*x*z^3*t +
>   14*y*z^3*t + 2*x^2*y*t^2 + 13*x^2*z*t^2 + 2*x*y*z*t^2 + 6*x*z^2*t^2 +
>   7*y*z^2*t^2 + 13*x^3*y*u + 14*x^2*y^2*u + 11*x^2*y*z*u + 15*x*y^2*z*u +
>   10*y^3*z*u + 8*x*y*z^2*u + 8*y^2*z^2*u + 15*x^3*t*u + 6*x^2*y*t*u +
>   11*x*y^2*t*u + 14*x^2*z*t*u + 3*x*y*z*t*u + 4*x*z^2*t*u + 7*y*z^2*t*u +
>   16*x^2*t^2*u + 2*x*y*t^2*u + y*z*t^2*u + 16*x*y^2*u^2 + 9*y^3*u^2 +
>   4*x*y*z*u^2 + 2*y^2*z*u^2 + 15*x*y*t*u^2 + 15*y^2*t*u^2 + 2*x*z*t*u^2 +
>   13*y*z*t*u^2 + 6*x*t^2*u^2 + 11*y*t^2*u^2 + 8*x*y*u^3 + 8*y^2*u^3 +
>   4*x*t*u^3 + 3*y*t*u^3,
>  5*x^3*y*z + 11*x^2*y*z^2 + 15*x*y^2*z^2 + 2*x*y*z^3 + 13*x^4*t + 2*x^3*y*t +
>   7*x^3*z*t + 6*x^2*y*z*t + 16*x^2*z^2*t + 4*x*y*z^2*t + 14*x*z^3*t +
>   2*x^3*t^2 + 9*x^2*z*t^2 + 7*x*z^2*t^2 + 14*x^3*y*u + 5*x^3*z*u +
>   4*x^2*y*z*u + 10*x*y^2*z*u + 12*x^2*z^2*u + 8*x*y*z^2*u + 15*x*z^3*u +
>   6*y*z^3*u + 11*z^4*u + 16*x^3*t*u + 15*x^2*y*t*u + 13*x^2*z*t*u +
>   3*x*z^2*t*u + 3*y*z^2*t*u + 11*z^3*t*u + 13*x^2*t^2*u + 3*x*z*t^2*u +
>   7*z^2*t^2*u + 7*x^3*u^2 + 7*x^2*y*u^2 + 9*x*y^2*u^2 + x^2*z*u^2 +
>   2*x*y*z*u^2 + 6*x*z^2*u^2 + y*z^2*u^2 + 13*z^3*u^2 + 12*x^2*t*u^2 +
>   15*x*y*t*u^2 + 14*x*z*t*u^2 + 14*y*z*t*u^2 + 3*z^2*t*u^2 + 11*x*t^2*u^2
>   + 11*z*t^2*u^2 + 9*x^2*u^3 + 8*x*y*u^3 + 4*x*z*u^3 + 10*y*z*u^3 +
>   z^2*u^3 + 3*x*t*u^3 + 6*z*t*u^3 + 7*z*u^4,
>  4*x^4*z + 3*x^3*y*z + 10*x^3*z^2 + x^2*z^3 + 14*x*y*z^3 + 3*x*z^4 +
>   15*x^3*z*t + 8*x^2*z^2*t + 10*x*z^3*t + 14*x^3*y*u + 15*x^3*z*u +
>   11*x^2*z^2*u + 10*x*z^3*u + 16*x^2*z*t*u + 16*x*z^2*t*u + 6*x^3*u^2 +
>   14*x^2*z*u^2 + 3*x*y*z*u^2 + 4*x*z^2*u^2 + 6*x^2*t*u^2 + 6*x*z*t*u^2 +
>   15*x^2*u^3 + 14*x*z*u^3,
>  5*x^3*z^2 + 4*x^2*y*z^2 + 12*x^2*z^3 + 15*x*z^4 + 6*y*z^4 + 11*z^5 +
>   14*x^3*z*t + x^2*y*z*t + 7*x^2*z^2*t + 13*x*z^3*t + 3*y*z^3*t + 11*z^4*t
>   + 12*x^2*z*t^2 + 2*x*z^2*t^2 + 7*z^3*t^2 + 7*x^3*z*u + 13*x^2*y*z*u +
>   x^2*z^2*u + 6*x*z^3*u + y*z^3*u + 13*z^4*u + 6*x^3*t*u + 16*x^2*y*t*u +
>   9*x^2*z*t*u + x*z^2*t*u + 14*y*z^2*t*u + 3*z^3*t*u + 6*x^2*t^2*u +
>   11*z^2*t^2*u + 9*x^2*z*u^2 + 4*x*z^2*u^2 + 10*y*z^2*u^2 + z^3*u^2 +
>   15*x^2*t*u^2 + 6*z^2*t*u^2 + 7*z^2*u^3,
>  13*x^5 + 2*x^4*y + 7*x^4*z + 6*x^3*y*z + 16*x^3*z^2 + 4*x^2*y*z^2 +
>   14*x^2*z^3 + 2*x^4*t + 9*x^3*z*t + 7*x^2*z^2*t + 16*x^4*u + 15*x^3*y*u +
>   x^3*z*u + 11*x^2*z^2*u + 3*x*y*z^2*u + 14*x*z^3*u + 13*x^3*t*u +
```

```
>   3*x^2*z*t*u + 7*x*z^2*t*u + 14*x^3*u^2 + 15*x^2*y*u^2 + 6*x^2*z*u^2 +
>   14*x*y*z*u^2 + 10*x*z^2*u^2 + 11*x^2*t*u^2 + 11*x*z*t*u^2 + 3*x*z*u^3]>;
> X := Scheme(Proj(R),I); // define the surface
> // The structure sheaf O_X of X is represented by graded module R/I
> O_X := GradedModule(I); // R/I
```

We first compute the dimension of $H^0(O_X)$. This is fairly uninteresting (it's just 1) but after this computation, the cached data will allow the following cohomology calls to execute practically instantaneously.

```
> CohomologyDimension(O_X,0,0);  // dim H^0(O_X)
1
> // get the geometric genus and irregularity
> time CohomologyDimension(O_X,2,0); // dim H^2(O_X) = g
0
Time: 0.000
> time CohomologyDimension(O_X,1,0); // dim H^1(O_X) = q
0
Time: 0.000
> // => p_a(X)=0. Verify this.
> ArithmeticGenus(X);
0
```

We now compute a module representative of the canonical sheaf K_X of X. We can just take $Ext^2_R(M, R(-5))$. Then we check again that $H^0(K_X) = g$ is 0. Note that here the module representing K_X is maximal so that $H^0(K_X(n))$ is just the dimension of its nth graded part for any n. However, in other cases (where X is again *not* arithmetically Cohen-Macaulay) the Ext computation for K_X may not give the maximal representing module, so its 0th graded piece might have dimension less than g.

```
> K_X := Ext(2,O_X,RModule(R,[5]));
> K_X;
Reduced Module R^6/<relations> with grading [1, 1, 1, 1, 1, 1]
Quotient Relations:
[12*z + 8*t + 7*u, 5*y + 7*z + 8*t + 6*u, 16*z + 4*u, 16*z + 7*u, 15*u,
    7*u],
[14*u, 8*u, 13*x + 2*y + 4*t + 14*u, 4*y + 8*t + 2*u, 16*t + 6*u, 3*u],
[15*z + 9*t + 11*u, 2*y + 13*z + 8*t + 14*u, 7*t + 16*u, 8*x + 14*t + 6*
    11*t + 15*u, 11*z + 14*t + 16*u],
[12*z + 15*t + 6*u, 5*y + 7*z + 4*t + 9*u, t + u, 2*t + 2*u, 7*x + 4*t +
    12*z + t + 2*u],
[12*y + 10*z + 14*t + 14*u, 7*y + 15*z + 8*t, 4*u, 8*u, 16*u, 5*x + 15*z
    + 9*u],
[15*x, 15*x, 4*u, 8*u, 16*u, 0],
[14*z + 15*t + 9*u, 3*y + 11*z + 7*t + 13*u, 0, 11*z + 6*u, 2*z + 14*u,
    10*t + 4*u],
[10*z + 14*t + 14*u, 4*x + 12*y + 15*z + 8*t, 0, 0, 0, 15*z + 16*t + 9*u
[5*z + 16*t + 4*u, 12*y + 10*z + 15*t + 5*u, 0, 2*z + 15*u, 6*u, 15*z +
    7*u],
[13*z + t + 15*u, 4*y + 9*z + 16*t + 16*u, 0, 0, 0, 5*z + 11*t + 3*u]
```

```
> CohomologyDimension(K_X,0,0);
0
```

Finally, we verify some more cases of Serre duality which gives

$$\dim H^r(O_X(n)) = \dim H^{2-r}(K_X(-n)).$$

```
> [CohomologyDimension(K_X,0,i) eq CohomologyDimension(O_X,2,-i) :
>    i in [-1..5]];
[ true, true, true, true, true, true, true ]
```

109.16 Bibliography

[**AL94**] William Adams and Philippe Loustaunau. *An introduction to Gröbner bases*, volume 3 of *Graduate studies in mathematics*. American Mathematical Society, Providence, R.I., 1994.

[**BS92**] David Bayer and Michael Stillman. Computation of Hilbert Functions. *J. Symbolic Comp.*, 14(1):31–50, 1992.

[**CLO98**] David Cox, John Little, and Donal O'Shea. *Using Algebraic Geometry*. Graduate Texts in Mathematics. Springer, New York–Berlin–Heidelberg, 1998.

[**DE02**] Wolfram Decker and David Eisenbud. Sheaf algorithms using the Exterior algebra. In Eisenbud et al., editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of *Springer Algorithms and Computation in Mathematics Series*, pages 215–247. Springer-Verlag, 2002.

[**DL06**] Wolfram Decker and Christoph Lossen. *Computing in Algebraic Geometry*, volume 16 of *Algorithms and Computation in Mathematics*. Springer, New York–Berlin–Heidelberg, 2006.

[**EFS03**] Eisenbud, Floystad, and Schreyer. Sheaf Cohomology and Free Resolutions over Exterior Algebras. *Trans. Am. Maths. Soc.*, 355:4397–4426, 2003.

[**Eis95**] David Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry*, volume 150 of *Graduate Texts in Mathematics*. Springer, New York–Berlin–Heidelberg, 1995.

[**Fau99**] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139 (1-3):61–88, 1999.

[**GP02**] G.-M. Greuel and G. Pfister. *A Singular Introduction to Commutative Algebra*. Springer-Verlag, Berlin–Heidelberg–New York, 2002.

[**SS98**] Roberto La Scala and Michael Stillman. Strategies for Computing Minimal Free Resolutions. *J. Symbolic Comp.*, 26(4):409–431, 1998.

[**ST02**] Frank-Olaf Schreyer and Fabio Tonoli. Needles in a Haystack: Special Varieties via Small Fields. In Eisenbud et al., editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of *Springer Algorithms and Computation in Mathematics Series*, pages 251–277. Springer-Verlag, 2002.

110 INVARIANT THEORY

Chapter 110
INVARIANT THEORY

110.1 Introduction

MAGMA contains a powerful module for computing with invariant rings and fields of finite groups and algebraic groups. The algorithms for invariant theory of finite groups in MAGMA are based on those in the *Invar* package written in Maple, implemented by G. Kemper [Kem96], but also include many new ideas and improvements which are described in detail in a subsequent paper [KS97]. Since a detailed understanding of the latter paper is useful for better understanding of many of the functions in the chapter, it is recommended the paper be perused by anyone wishing to make serious applications of the functions.

Since V2.14, MAGMA also has algorithms for invariant theory of linear algebraic groups. In particular, Derksen's algorithm [Der99] and the algorithm by Beth and Müller-Quade [MQB99] have been implemented. These additions use code written by G. Kemper.

The primary goal of invariant theory in MAGMA is the computation of generators of the invariant ring or field of a given group, which may be finite or algebraic. The ground field may have arbitrary characteristic. In invariant theory of finite groups, the *modular case*, i.e., the case where the characteristic of the ground field K divides the group order, is of particular interest, since in that case there are still many theoretical questions unanswered. MAGMA also contains easy algorithms to calculate properties of modular invariant rings, such as the Hilbert series, the Cohen-Macaulay property, depth, and free resolutions.

The approach to calculating the invariant ring of a finite group is broken up into two major steps: first a system of *primary invariants* is constructed, i.e., homogeneous invariants $f_1, \ldots, f_n$ which are algebraically independent, such that the invariant ring is a finitely generated module over $A = K[f_1, \ldots, f_n]$. In the next step we calculate *secondary invariants*, which are generators of the invariant ring as an A-module.

Throughout this chapter, K will be a field and G is a group acting linearly on the n-dimensional vector space $V \cong K^n$ with basis $x_1, \ldots, x_n$. G may be a linear algebraic group, in which case K is assumed to be algebraically closed, or a finite matrix group, or a permutation group. G also acts on the symmetric algebra $K[V] = S(V)$, which is the multivariate polynomial ring $K[x_1, \ldots, x_n]$ in the variables $x_1, \ldots, x_n$. The invariant ring $R = \{f \in K[V] \mid f^\sigma = f \ \forall \sigma \in G\}$ is denoted by $K[V]^G$. The G-action extends naturally to the rational function field $K(V)$ on V, leading to the analogous definition of the invariants field $K(V)^G$.

Sections 110.2.1 through 110.7 describe the general setup of invariant theory in MAGMA. Section 110.8 is about computing invariants of specified degree. Sections 110.9 through 110.16 deal with functions for invariant rings of finite groups. The following Sections 110.17 and 110.18 present some functions whose scope is not limited to the context of invariant theory. Sections 110.20 and 110.21.3 are about functions for invariant rings

of algebraic groups and for invariant fields, respectively. The Section 110.19 gives information about some low-level control of the data structures associated to invariant theory. Finally, since V2.14, the Section 110.20 deals with invariant rings of algebraic groups and the Section 110.21.3 deals with invariant fields.

110.2 Invariant Rings of Finite Groups

110.2.1 Creation

Let G be a finite matrix or permutation group acting on the polynomial ring $P = K[x_1, \ldots, x_n]$ over the field K. MAGMA allows the construction of the invariant ring $R = K[V]^G$. The invariant ring R is a special structure which contains references to the group G and polynomial ring P. When the invariant ring R is created using the `InvariantRing` function, no explicit calculations are done until specifically invoked (e.g., by the `PrimaryInvariants` function). The elements of R are the polynomials of P which are invariant under the action of G. Note that the parent of such polynomials is still P – the invariant ring R is just a special structure which contains all the information about the invariant ring. The category of invariant rings is `RngInvar`.

`InvariantRing(G)`

`InvariantRing(G, K)`

Construct the invariant ring $R = K[V]^G$ of the finite matrix or permutation group G over the field K. For a matrix group G, G alone should be supplied, while for a permutation group G, G should be supplied, together with the field K. The appropriate multivariate polynomial ring P is automatically constructed. No other explicit calculations are done (e.g. computation of primary invariants).

110.2.2 Access

The following functions allow simple access to basic properties of invariant rings.

`Group(R)`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the group G.

`CoefficientRing(R)`

`CoefficientField(R)`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the coefficient field K.

`PolynomialRing(R)`

Given an invariant ring $R = K[V]^G$ of the group G of degree n over the field K, return the polynomial ring $P = K[x_1, \ldots, x_n]$ in which the invariants of R lie. P has the print names `"x1", "x2"`, etc. – the angle bracket notation or the . operator should be used to assign the variables of P to actual MAGMA variables.

`f in R`

Return whether the polynomial f is in $R = K[V]^G$. Note that the parent of f is always the polynomial ring P, never R, so a `true` result does not mean that the parent of f is R.

110.3 Group Actions on Polynomials

This section describes in detail the actions which groups have on multivariate polynomial rings.

110.4 Permutation Group Actions on Polynomials

If P is a polynomial ring in n indeterminates $x_1, \ldots, x_n$, over any coefficient ring, $\mathrm{Sym}(n)$ acts on P by permuting the indices of the indeterminates. Thus, the polynomial $f(x_1, \ldots, x_n)$ is mapped into the polynomial $f(x_{g(1)}, \ldots, x_{g(n)})$.

`f ^ g`

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation g belonging to a subgroup of $\mathrm{Sym}(\{1, \ldots, n\})$, return the image of f under g.

`f ^ G`

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation group G contained in $\mathrm{Sym}(\{1, \ldots, n\})$, return the orbit of f under G.

`IsInvariant(f, g)`

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation g of degree n or an element of a matrix group of degree n whose coefficient ring is the same as that of f, return whether f is an invariant of g, i.e., whether $f^g = f$.

`IsInvariant(f, G)`

Given a polynomial f belonging to a polynomial ring having n indeterminates, and a permutation group G of degree n or a matrix group of degree n whose coefficient ring is the same as that of f, return whether f is an invariant of G, i.e., whether $f^g = f$ for all $g \in G$.

110.5 Matrix Group Actions on Polynomials

If P is a polynomial ring in n indeterminates $x_1, \ldots, x_n$, over the ring S, then $\mathrm{GL}(n, S)$ acts on P as follows: Let $\mathbf{x}$ denote the vector $(x_1, \ldots, x_n)$. Then the image g of a polynomial f of P under the action of a matrix a of $\mathrm{GL}(n, S)$ is defined by $g(\mathbf{x}) = f(\mathbf{x} * a)$.

`f ^ a`

Given a polynomial f belonging to a polynomial ring having n indeterminates and coefficient ring S, and a matrix a belonging subgroup G of $\mathrm{GL}(n, S)$, return the image of f under a.

`f ^ G`

Given a polynomial f belonging to a polynomial ring having n indeterminates and coefficient ring S, and a to a subgroup of $\mathrm{GL}(n, S)$, return the orbit of f under G.

Example H110E1

We act on the polynomial ring in two indeterminates over the field $K = Q(\sqrt{2})$, by a cyclic subgroup of $\mathrm{GL}(2, K)$.

```
> K := QuadraticField(2);
> Aq := [ x / K.1 : x in [1, 1, -1, 1]];
> G := MatrixGroup<2, K | Aq>;
> P<x, y> := PolynomialRing(K, 2);
> f := x^2 + x * y + y^2;
> g := f^G.1;
> g;
1/2*x^2 + 3/2*y^2
> f^G;
{
   1/2*x^2 + 3/2*y^2,
   x^2 - x*y + y^2,
   x^2 + x*y + y^2,
   3/2*x^2 + 1/2*y^2
}
```

110.6 Algebraic Group Actions on Polynomials

In the invariant theory package of MAGMA, a linear algebraic group G is given by polynomials, say in variables $t_1, \ldots, t_m$, defined over some field K that is representable in MAGMA, as the affine variety over the algebraic closure $\bar{K}$ of K given by these polynomials. A G-module is given by a matrix $A \in K[t_1, \ldots, t_m]^{n \times n}$ such that a group element $(\eta_1, \ldots, \eta_m) \in G$ acts on $\bar{K}^n$ by the matrix obtained by substituting $(\eta_1, \ldots, \eta_m)$ into the polynomials occurring in the matrix A.

G then also acts on the ring of polynomials on $\bar{K}^n$ by

$$\sigma(f) = f \circ \sigma^{-1}$$

for $\sigma \in G$ and $f \in \bar{K}[x_1, \ldots, x_n]$. Since the algorithms in MAGMA do not work with the algebraic closure, single group elements are never dealt with. In fact, all relevant algorithms ony involve field elements of K, the field of definition.

110.7 Verbosity

The following procedure allows verbose information for the Invariant Theory algorithms to be displayed.

```
SetVerbose("Invariants", v)
```

(Procedure.) Set the verbose printing level for the Invariant Theory algorithms of MAGMA to be v. Currently the legal values for v are `true`, `false`, 0, 1, 2, 3, or 4 (`false` has the same effect as 0, and `true` has the same effect as 1). Level 1 gives a minimal amount of useful information during the running of all the algorithms while higher levels give more detailed information. For the primary invariants computation, the verbose output displays each possible degree list (degrees of the potential primary invariants) before and then tries to find primary invariants corresponding to this degree list. For the secondary invariants computation, in the non-modular case the algorithm loops over the necessary degrees in increasing order and computes the relevant new invariants; in the modular case the algorithm finds secondary invariants with respect to a subgroup and then performs a module syzygy computation. Full details of the algorithms are found in [KS97].

110.8 Construction of Invariants of Specified Degree

Let $R = K[V]^G$ be the invariant ring of the group G over the field K. Let $d \geq 0$ be a fixed integer. The homogeneous invariants in R of degree d form a vector space R_d over K.

There are two ways of explicitly constructing homogeneous invariants in R of degree d: the *Reynolds operator* method and the *linear algebra* method. Both methods are described in detail in [KS97].

The Reynolds operator method only works fro finite groups in the non-modular case. It takes a monomial of degree d and yields either the zero polynomial or a non-zero invariant of degree d. By applying it to several different monomials, a complete basis of R_d can be

constructed. If G is a permutation group, a simplified version of the Reynolds operator can always be used which is independent of the field K (and thus whether we are in the modular case or not).

The linear algebra method works in both the modular and non-modular cases and, with appropriate modifications, also for linear algebraic groups. It simply finds a basis for R_d in one step – it is not possible to find a single invariant alone by this method.

MAGMA provides the function `InvariantsOfDegree` to automatically compute a basis of R_d by a default appropriate method – the method can also be selected by a parameter. The function `InvariantsOfDegree` can also be given a positive integer k which is less than or equal to the dimension of R_d: in such a case, only k linearly independent invariants are computed. See also the functions `MonomialsOfDegree` and `MonomialsOfWeightedDegree` in the Ideal Theory chapter.

```
ReynoldsOperator(f, G)
```

Given a polynomial f and a matrix group G such that G can act on f, return the application of the Reynolds operator of G to f. (f need not be a monomial but may be a non-homogeneous polynomial.)

```
InvariantsOfDegree(R, d)
InvariantsOfDegree(G, d)
InvariantsOfDegree(G, K, d)
InvariantsOfDegree(G, P, d)
```

`Invariants` MONSTGELT *Default :* "*Both*"

Construct a K-basis of the space R_d of the homogeneous invariants of degree d in the invariant ring $R = K[V]^G$ of the group G over the field K as a sequence of polynomials. Either the invariant ring R, the group G (if a matrix group), or the group G (if a permutation group) together with the field K may be passed. A specific polynomial ring P compatible with G and K may be passed so that the returned invariants lie in P. The parameter `Invariants` may be supplied to select the method of the construction of the invariants: `"Reynolds"` (use the Reynolds operator), `"Linear"` (use the linear algebra method), or `"Both"` (use an appropriate combination of both methods). The default is `"Both"`.

```
InvariantsOfDegree(R, d, k)
InvariantsOfDegree(G, d, k)
InvariantsOfDegree(G, K, d, k)
InvariantsOfDegree(G, P, d, k)
```

`Invariants` MONSTGELT *Default :* "*Both*"

Construct k linearly independent homogeneous invariants of degree d in the invariant ring $R = K[V]^G$ of the group G over the field K as a sequence of polynomials, where k must be greater than or equal to 1 and less than or equal to the dimension of the

space R_d. Either the invariant ring R, the group G (if a matrix group), or the group G (if a permutation group) together with the field K may be passed. A specific polynomial ring P compatible with G and K may be passed so that the returned invariants lie in P. The parameter `Invariants` may be supplied to select the method of the construction of the invariants – see the last function.

Example H110E2

We demonstrate elementary uses of `ReynoldsOperator` and `InvariantsOfDegree`.

```
> K<z> := CyclotomicField(5);
> w := -z^3 - z^2;
> G := MatrixGroup<3,K |
>    [1,0,-w, 0,0,-1, 0,1,-w],
>    [-1,-1,w, -w,0,w, -w,0,1]>;
> P<x1,x2,x3> := PolynomialRing(K, 3);
> time ReynoldsOperator(x1^4, G);
(-z^3 - z^2 + 1)*x1^4 + (12/5*z^3 + 12/5*z^2 -
    4/5)*x1^3*x2 + (12/5*z^3 + 12/5*z^2 - 4/5)*x1^3*x3
    + (-14/5*z^3 - 14/5*z^2 + 14/5)*x1^2*x2^2 +
    (4/5*z^3 + 4/5*z^2 + 4/5)*x1^2*x2*x3 + (-14/5*z^3 -
    14/5*z^2 + 14/5)*x1^2*x3^2 + (12/5*z^3 + 12/5*z^2 -
    4/5)*x1*x2^3 + (4/5*z^3 + 4/5*z^2 + 4/5)*x1*x2^2*x3
    + (4/5*z^3 + 4/5*z^2 + 4/5)*x1*x2*x3^2 + (12/5*z^3
    + 12/5*z^2 - 4/5)*x1*x3^3 + (-z^3 - z^2 + 1)*x2^4 +
    (12/5*z^3 + 12/5*z^2 - 4/5)*x2^3*x3 + (-14/5*z^3 -
    14/5*z^2 + 14/5)*x2^2*x3^2 + (12/5*z^3 + 12/5*z^2 -
    4/5)*x2*x3^3 + (-z^3 - z^2 + 1)*x3^4
Time: 0.090
> time I20_1 := InvariantsOfDegree(G, 20, 1);
0.259
> time I20 := InvariantsOfDegree(G, 20);
3.589
> [LeadingMonomial(f): f in I20];
[
    x1^20,
    x1^18*x2^2,
    x1^16*x2^4,
    x1^15*x2^5,
    x1^14*x2^6,
    x1^13*x2^7,
    x1^12*x2^8
]
> G := CyclicGroup(4);
> K := GF(2);
> InvariantsOfDegree(G, K, 4);
[
    x1^4 + x2^4 + x3^4 + x4^4,
```

```
    x1^3*x2 + x1*x4^3 + x2^3*x3 + x3^3*x4,
    x1^3*x3 + x1*x3^3 + x2^3*x4 + x2*x4^3,
    x1^3*x4 + x1*x2^3 + x2*x3^3 + x3*x4^3,
    x1^2*x2^2 + x1^2*x4^2 + x2^2*x3^2 + x3^2*x4^2,
    x1^2*x2*x3 + x1*x2*x4^2 + x1*x3^2*x4 + x2^2*x3*x4,
    x1^2*x2*x4 + x1*x2^2*x3 + x1*x3*x4^2 + x2*x3^2*x4,
    x1^2*x3^2 + x2^2*x4^2,
    x1^2*x3*x4 + x1*x2^2*x4 + x1*x2*x3^2 + x2*x3*x4^2,
    x1*x2*x3*x4
]
```

`SetAllInvariantsOfDegree(R, d, Q)`

(Procedure.) Given an invariant ring $R = K[V]^G$, an integer $d \geq 0$, and a sequence Q consisting of k degree-d homogeneous invariants of G, set the internal list of all linearly-independent homogeneous invariants of degree d of R to be Q. Thus the elements of Q must describe a basis of the space of all homogeneous invariants of degree d of R. If the Hilbert Series of R is known, it will be used to check that the length of Q (the dimension of the basis) is correct.

Example H110E3

We demonstrate a simple use of `SetAllInvariantsOfDegree`.

```
> R := InvariantRing(CyclicGroup(4), GF(2));
> P<x1,x2,x3,x4> := PolynomialRing(R);
> L := [
>     x1^2 + x2^2 + x3^2 + x4^2,
>     x1*x2 + x1*x4 + x2*x3 + x3*x4,
>     x1*x3 + x2*x4
> ];
> SetAllInvariantsOfDegree(R, 2, L);
> InvariantsOfDegree(R, 2);
[
    x1^2 + x2^2 + x3^2 + x4^2,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4
]
> PrimaryInvariants(R);
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
```

The following sections 110.9 through 110.16 all deal with invariant rings of finite groups.

110.9 Construction of G-modules

This section describes how one can create a finite-dimensional G-module corresponding to the action of a finite group G on a polynomial ring P. There are two ways one can create a finite-dimensional action: the action on the space of homogeneous polynomials of a fixed degree, or the action on the quotient space of polynomials by a zero-dimensional ideal (so the quotient has finite-dimension as a vector space). The functions in this section are also found in the chapter on general modules but are also included here since they are useful in Invariant Theory.

`GModule(G, P, d)`

Given a finite permutation or matrix group G of degree n, a polynomial ring $P = K[x_1, \ldots, x_n]$ over a field K, and a non-negative integer d, create the $K[G]$-module M corresponding to the action of G on the space of homogeneous polynomials of degree d of the polynomial ring P. The function also returns the isomorphism f between the space of homogeneous polynomials of degree d of P and M, together with an indexed set of monomials of degree d of P which correspond to the columns of M.

`GModule(G, I, J)`

Given a finite permutation or matrix group G of degree n, an ideal I of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ over a field K, and a zero-dimensional subideal J of I, create the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient I/J. The function also returns the isomorphism f between the quotient space I/J and M, together with an indexed set of monomials of P, forming a (vector space) basis of I/J, and which correspond to the columns of M.

`GModule(G, Q)`

Given a finite permutation or matrix group G of degree n, and a finite-dimensional quotient ring $Q = I/J$ of a multivariate polynomial ring $P = K[x_1, \ldots, x_n]$ over a field K, create the $K[G]$-module M corresponding to the action of G on the finite-dimensional quotient Q. The function also returns the isomorphism f between the quotient ring Q and M, together with an indexed set of monomials of P, forming a (vector space) basis of Q, and which correspond to the columns of M.

Example H110E4

We demonstrate simple uses of the `GModule` function.

```
> q := 5;
> K := GF(q);
> G := GL(3, K);
> P<x, y, z> := PolynomialRing(K, 3);
> I := ideal< P | x^5 - x,y^5 - y, z^5 - z >;
> Q, rho := quo< P | I >;
> f := x^3 + x^2*y + y^3;
> M, phi:= GModule(G, P, I);
```

```
> Constituents(M);
[
    GModule of dimension 1 over GF(5),
    GModule of dimension 3 over GF(5),
    GModule of dimension 3 over GF(5),
    GModule of dimension 6 over GF(5),
    GModule of dimension 6 over GF(5),
    GModule of dimension 10 over GF(5),
    GModule of dimension 10 over GF(5),
    GModule of dimension 15 over GF(5),
    GModule of dimension 15 over GF(5),
    GModule of dimension 18 over GF(5),
    GModule of dimension 18 over GF(5),
    GModule of dimension 19 over GF(5)
]
> N := sub<M | phi(f)>;
> N;
GModule N of dimension 10 over GF(5)
> M5 := GModule(G, P, 5);
> M5;
GModule M5 of dimension 21 over GF(5)
> Constituents(M5);
[
    GModule of dimension 3 over GF(5),
    GModule of dimension 18 over GF(5)
]
```

110.10 Molien Series

Let $R = K[V]^G$ be the invariant ring of the finite group G over the field K. If G is a finite matrix group in the non-modular case or a permutation group (in *either* the modular or non-modular case) then the Molien series of G yields the Hilbert Series of R.

`MolienSeries(G)`

The Molien series of G, returned as an element of the rational function field $\mathbf{Z}(t)$. If G is a permutation group, the Molien series always exists and equals the Hilbert series of the invariant ring of G for any field. If G is a matrix group, the characteristic of the coefficient field of G must be coprime with the order of G.

`MolienSeriesApproximation(G, n)`

The Molien series of a permutation group G, or more precisely, an approximation to it, as a Laurent series with n known coefficients. In contrast to the `MolienSeries` function above, approximations can be computed for far larger groups.

Example H110E5

We compute the Molien series of a matrix G and verify that the coefficients of the corresponding power series match the number of independent invariants for each degree.

```
> K<z> := CyclotomicField(5);
> w := -z^3 - z^2;
> G := MatrixGroup<3,K |
>    [1,0,-w, 0,0,-1, 0,1,-w],
>    [-1,-1,w, -w,0,w, -w,0,1]>;
> M<t> := MolienSeries(G);
> M;
(-t^8 - t^7 + t^5 + t^4 + t^3 - t - 1)/(t^11 + t^10 -
    t^9 - 2*t^8 - t^7 + t^4 + 2*t^3 + t^2 - t - 1)
> P<u> := PowerSeriesRing(IntegerRing());
> P ! M;
1 + u^2 + u^4 + 2*u^6 + 2*u^8 + 3*u^10 + 4*u^12 +
    4*u^14 + u^15 + 5*u^16 + u^17 + 6*u^18 + u^19 +
    O(u^20)
> Coefficients(P ! M);
[ 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 4, 0, 4, 1, 5, 1,
6, 1 ]
> time [#InvariantsOfDegree(G, i): i in [0 .. 19]];
[ 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 4, 0, 4, 1, 5, 1,
6, 1 ]
```

110.11 Primary Invariants

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. A set of *primary invariants* of R is a set $\{f_1, \ldots, f_n\}$ of n algebraically independent homogeneous invariants of R such that the invariant ring R is a finitely generated module over $A = K[f_1, \ldots, f_n]$. A set of primary invariants always exists for any invariant ring R. The invocation `PrimaryInvariants(R)` allows automatic construction of primary invariants of R. The primary invariants are stored in R and recalled as necessary in subsequent computations.

The latest algorithm in MAGMA to compute primary invariants, due to G. Kemper [Kem99], now guarantees that the degrees of the primary invariants found by the algorithm are optimal (with respect to their product and then their sum).

`PrimaryInvariants(R)`

Construct optimal primary invariants for the invariant ring $R = K[V]^G$ as a sorted sequence (with increasing degrees) of n polynomials of R where n is the degree of G.

Example H110E6

We compute primary invariants for the "first A_5 in $\mathrm{SL}(\mathbf{F}_2)$", discussed in [AM94, p. 116]. The resulting degrees $3, 5, 8$, and 12 are necessarily optimal (see [Kem96]).

```
> K := GF(2);
> G := MatrixGroup<4, K |
>     [0,1,0,0, 1,1,0,0, 0,0,1,1, 0,0,1,0],
>     [1,0,0,0, 0,1,0,0, 1,0,1,0, 0,1,0,1],
>     [1,0,1,0, 0,1,0,1, 0,0,1,0, 0,0,0,1]>;
> R := InvariantRing(G);
> time p := PrimaryInvariants(R);
Time: 1.399
> [TotalDegree(f): f in p];
[ 3, 5, 8, 12 ]
```

110.12 Secondary Invariants

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. If $\{f_1, \ldots, f_n\}$ is a set of primary invariants for R then R can be viewed as a finitely generated module over the algebra $A = K[f_1, \ldots, f_n]$. A set of *secondary invariants* for R with respect to these primary invariants is set of module generators over A. The invocation `SecondaryInvariants(R)` allows automatic construction of secondary invariants of R. The secondary invariants are stored in R and recalled as necessary in subsequent computations. Different algorithms are needed for the modular and non-modular cases – see [KS97] for details.

`SecondaryInvariants(R)`

Construct secondary invariants for the invariant ring $R = K[V]^G$ (with respect to the current primary invariants of R, constructed automatically first if necessary) as a sorted sequence (with increasing degrees) of polynomials of R. The secondary invariants are *minimal*; i.e. they are a minimal generating set for R considered as a module over the algebra generated by the primary invariants.

`SecondaryInvariants(R, H)`

Construct secondary invariants for the *modular* invariant ring $R = K[V]^G$ (with respect to the current primary invariants of R), using the subgroup H. This function can only be used if R is a modular invariant ring. H must be a subgroup of the group G; first, secondary invariants are computed for $K[V]^H$ using the current primary invariants for G and then these secondary invariants are used in the manner described in [KS97]. The function `SecondaryInvariants(R)` (taking just the invariant ring R) follows a default strategy in which it tries to use this function with the best subgroup H appropriate. Thus usually using this function to specify a particular subgroup is not more helpful than the one-argument function but occasionally it may be.

`IrreducibleSecondaryInvariants(R)`

Return the irreducible secondary invariants of the invariant ring $R = K[V]^G$ (with respect to the current primary invariants of R, constructed automatically first if necessary) as a sequence of polynomials of R. These, together with the primary invariants of R, generate R as an algebra over K. In the modular case, these will be the same as the secondary invariants of R (excluding the polynomial 1) but in the non-modular case they may form a proper subsequence of the secondary invariants. Note that the expression of the secondary invariants in terms of the irreducible secondary invariants is given as the second return value of the function `Algebra` (see the section on the algebra of an invariant ring and algebraic relations below).

Example H110E7

We construct primary and then secondary invariants for the invariant ring R of the group G over $\mathbf{F}_2$, where G is the (permutation) cyclic group of order 4. Note that in this example Noether's degree bound (which holds for characteristic 0) is violated.

```
> K := GF(2);
> G := CyclicGroup(4);
> R := InvariantRing(G, K);
> time PrimaryInvariants(R);
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
Time: 0.040
> time SecondaryInvariants(R);
[
    1,
    x1*x2*x3 + x1*x2*x4 + x1*x3*x4 + x2*x3*x4,
    x1^2*x3 + x1^2*x4 + x1*x2^2 + x1*x3^2 + x2^2*x4 +
        x2*x3^2 + x2*x4^2 + x3*x4^2,
    x1^2*x3^2 + x1^2*x3*x4 + x1*x2^2*x4 + x1*x2*x3^2 +
        x2^2*x4^2 + x2*x3*x4^2,
    x1^3*x3*x4 + x1^2*x2^2*x3 + x1^2*x2^2*x4 +
        x1^2*x2*x3^2 + x1^2*x2*x3*x4 + x1^2*x2*x4^2 +
        x1^2*x3^2*x4 + x1^2*x3*x4^2 + x1*x2^3*x4 +
        x1*x2^2*x3^2 + x1*x2^2*x3*x4 + x1*x2^2*x4^2 +
        x1*x2*x3^3 + x1*x2*x3^2*x4 + x1*x2*x3*x4^2 +
        x1*x3^2*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 +
        x2*x3^2*x4^2 + x2*x3*x4^3
]
Time: 0.080
```

110.13 Fundamental Invariants

Let $R = K[V]^G$ be the invariant ring of the group G over the field K and suppose the degree of G is n. A set of *fundamental invariants* for R is a generating set of R as an algebra over K.

`FundamentalInvariants(R)`

`Al`	MonStgElt	*Default :* "*King*"
`MaxDegree`	RngIntElt	*Default :* 0

Construct fundamental invariants for the invariant ring $R = K[V]^G$ as a sorted sequence (with increasing degrees) of polynomials of R.

As of V2.15, if R is non-modular, then by default the fundamental invariants are computed via the algorithm of S.King [Kin07]; the alternative algorithm (always used in the modular case), which computes the fundamental invariants by minimalizing the union of the primary and secondary invariants of R, may be selected by setting the parameter `Al` to `"MinPrimSec"`.

If the fundamental invariants are known to be bounded by degree d, then the parameter `MaxDegree` may be set to d to assist the King algorithm with an early stopping condition in the non-modular case.

Example H110E8

We construct fundamental invariants for the invariant ring R of the group G over $\mathbf{Q}$, where G is permutation group consisting of two parallel copies of S_3 in degree 6. Notice that the sequence of fundamental invariants is shorter and simpler than the sequence consisting of the primary invariants combined with the secondary invariants.

```
> K := RationalField();
> G := PermutationGroup<6 | (1,2,3)(4,5,6), (1,2)(4,5)>;
> R := InvariantRing(G, K);
> PrimaryInvariants(R);
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x4^3 + x5^3 + x6^3
]
> SecondaryInvariants(R);
[
    1,
    x1*x4 + x2*x5 + x3*x6,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x1^2*x4^2 + 2*x1*x2*x4*x5 + 2*x1*x3*x4*x6 + x2^2*x5^2 + 2*x2*x3*x5*x6 +
        x3^2*x6^2,
```

```
    x1^3*x4^3 + x1^2*x2*x4*x5^2 + x1^2*x3*x4*x6^2 + x1*x2^2*x4^2*x5 +
        x1*x3^2*x4^2*x6 + x2^3*x5^3 + x2^2*x3*x5*x6^2 + x2*x3^2*x5^2*x6 +
        x3^3*x6^3
]
> FundamentalInvariants(R);
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x1*x4 + x2*x5 + x3*x6,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x4^3 + x5^3 + x6^3
]
```

Example H110E9

As in [Kin07], we compute fundamental invariants for the invariant rings for all transitive groups of degree 7 (in characteristic zero). For each group, we print its order and a summary of the degrees (where the i-th element of the sequence gives the number of fundamental invariants of degree i).

```
> function deg_summary(B)
>     degs := [TotalDegree(f): f in B];
>     return [#[j: j in degs | j eq d]: d in [1 .. Max(degs)]];
> end function;
>
> d := 7;
> time for i := 1 to NumberOfTransitiveGroups(d) do
>     G := TransitiveGroup(d, i);
>     R := InvariantRing(G, RationalField());
>     F := FundamentalInvariants(R);
>     printf "%o: Order: %o, Degrees: %o\n", i, #G, deg_summary(F);
> end for;
1: Order: 7, Degrees: [ 1, 3, 8, 12, 12, 6, 6 ]
2: Order: 14, Degrees: [ 1, 3, 4, 6, 6, 3, 3 ]
3: Order: 21, Degrees: [ 1, 1, 4, 5, 8, 8, 6 ]
4: Order: 42, Degrees: [ 1, 1, 2, 3, 4, 7, 7, 5, 1 ]
5: Order: 168, Degrees: [ 1, 1, 2, 2, 2, 2, 2 ]
6: Order: 2520, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
7: Order: 5040, Degrees: [ 1, 1, 1, 1, 1, 1, 1 ]
Time: 1.610
```

Instead of computing over the rational field, for each group G we can instead compute over $\mathbf{F}_p$, where p is the smallest prime which does not divide the order of G. This is faster, and it

is conjectured that the resulting degrees are always the same as for the computation over the rationals.

```
> d := 7;
> time for i := 1 to NumberOfTransitiveGroups(d) do
>     G := TransitiveGroup(d, i);
>     p := rep{p: p in [2 .. #G] | IsPrime(p) and #G mod p ne 0};
>     R := InvariantRing(G, GF(p));
>     F := FundamentalInvariants(R);
>     printf "%o: Order: %o, Degrees: %o\n", i, #G, deg_summary(F);
> end for;
1: Order: 7, Degrees: [ 1, 3, 8, 12, 12, 6, 6 ]
2: Order: 14, Degrees: [ 1, 3, 4, 6, 6, 3, 3 ]
3: Order: 21, Degrees: [ 1, 1, 4, 5, 8, 8, 6 ]
4: Order: 42, Degrees: [ 1, 1, 2, 3, 4, 7, 7, 5, 1 ]
5: Order: 168, Degrees: [ 1, 1, 2, 2, 2, 2, 2 ]
6: Order: 2520, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
7: Order: 5040, Degrees: [ 1, 1, 1, 1, 1, 1, 1 ]
Time: 0.790
```

Finally, we can do the same for all transitive groups of degree 8 in about 2 minutes.

```
> d := 8;
> time for i := 1 to NumberOfTransitiveGroups(d) do
>     G := TransitiveGroup(d, i);
>     p := rep{p: p in [2 .. #G] | IsPrime(p) and #G mod p ne 0};
>     R := InvariantRing(G, GF(p));
>     F := FundamentalInvariants(R);
>     printf "%o: Order: %o, Degrees: %o\n", i, #G, deg_summary(F);
> end for;
1: Order: 8, Degrees: [ 1, 4, 10, 18, 16, 8, 4, 4 ]
2: Order: 8, Degrees: [ 1, 5, 9, 16, 8 ]
3: Order: 8, Degrees: [ 1, 7, 7, 7 ]
4: Order: 8, Degrees: [ 1, 6, 8, 12, 5 ]
5: Order: 8, Degrees: [ 1, 4, 10, 19, 15, 7 ]
6: Order: 16, Degrees: [ 1, 4, 5, 9, 8, 4, 2, 2 ]
7: Order: 16, Degrees: [ 1, 3, 7, 12, 13, 9, 4, 4 ]
8: Order: 16, Degrees: [ 1, 3, 6, 11, 12, 7, 2, 2 ]
9: Order: 16, Degrees: [ 1, 5, 5, 8, 4 ]
10: Order: 16, Degrees: [ 1, 4, 6, 11, 7, 2 ]
11: Order: 16, Degrees: [ 1, 4, 6, 11, 7, 3 ]
12: Order: 24, Degrees: [ 1, 2, 4, 8, 11, 12, 7 ]
13: Order: 24, Degrees: [ 1, 3, 3, 7, 8, 11, 7 ]
14: Order: 24, Degrees: [ 1, 3, 3, 8, 7, 9, 6, 1, 1 ]
15: Order: 32, Degrees: [ 1, 3, 4, 7, 6, 4, 2, 2 ]
16: Order: 32, Degrees: [ 1, 3, 5, 8, 7, 7, 4, 4 ]
17: Order: 32, Degrees: [ 1, 3, 4, 7, 6, 4, 2, 2 ]
18: Order: 32, Degrees: [ 1, 4, 4, 7, 3 ]
```

```
19: Order: 32, Degrees: [ 1, 3, 3, 7, 6, 7, 5, 1 ]
20: Order: 32, Degrees: [ 1, 3, 5, 9, 6, 4, 2, 1 ]
21: Order: 32, Degrees: [ 1, 4, 4, 6, 4, 3, 2, 1 ]
22: Order: 32, Degrees: [ 1, 4, 4, 7, 3, 1 ]
23: Order: 48, Degrees: [ 1, 2, 3, 5, 6, 6, 5, 2 ]
24: Order: 48, Degrees: [ 1, 3, 3, 6, 4, 3, 1 ]
25: Order: 56, Degrees: [ 1, 1, 1, 4, 6, 13, 18, 23, 18, 6 ]
26: Order: 64, Degrees: [ 1, 3, 3, 5, 3, 3, 2, 3, 1 ]
27: Order: 64, Degrees: [ 1, 3, 5, 8, 6, 4, 2, 2 ]
28: Order: 64, Degrees: [ 1, 3, 3, 5, 4, 4, 2, 2 ]
29: Order: 64, Degrees: [ 1, 3, 3, 6, 3, 2, 1 ]
30: Order: 64, Degrees: [ 1, 3, 3, 5, 3, 2, 3, 4, 3, 2, 1, 1 ]
31: Order: 64, Degrees: [ 1, 4, 4, 6, 3, 1 ]
32: Order: 96, Degrees: [ 1, 2, 2, 4, 3, 5, 4, 2, 2, 1, 1, 1 ]
33: Order: 96, Degrees: [ 1, 2, 2, 4, 3, 6, 5, 5, 3 ]
34: Order: 96, Degrees: [ 1, 2, 2, 5, 2, 5, 4, 3, 3 ]
35: Order: 128, Degrees: [ 1, 3, 3, 5, 3, 2, 1, 1 ]
36: Order: 168, Degrees: [ 1, 1, 1, 2, 2, 5, 6, 8, 10, 11, 8 ]
37: Order: 168, Degrees: [ 1, 1, 1, 3, 1, 5, 5, 8, 9, 9, 7 ]
38: Order: 192, Degrees: [ 1, 2, 2, 3, 3, 5, 4, 3, 2, 1, 1, 1 ]
39: Order: 192, Degrees: [ 1, 2, 2, 4, 2, 2, 1 ]
40: Order: 192, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1, 1, 3, 3, 2, 2, 1, 1, 1 ]
41: Order: 192, Degrees: [ 1, 2, 2, 4, 2, 3, 2, 2, 1 ]
42: Order: 288, Degrees: [ 1, 2, 2, 3, 2, 3, 2, 2, 1, 1 ]
43: Order: 336, Degrees: [ 1, 1, 1, 2, 1, 3, 3, 5, 4, 6, 5, 4, 2 ]
44: Order: 384, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1 ]
45: Order: 576, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1, 0, 0, 0, 1 ]
46: Order: 576, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1 ]
47: Order: 1152, Degrees: [ 1, 2, 2, 3, 2, 2, 1, 1 ]
48: Order: 1344, Degrees: [ 1, 1, 1, 2, 1, 2, 2, 2, 1, 1 ]
49: Order: 20160, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ]
50: Order: 40320, Degrees: [ 1, 1, 1, 1, 1, 1, 1, 1 ]
Time: 128.030
```

Example H110E10

We compute fundamental invariants of a degree-10 representation of S_5 acting on pairs. See [Kin07, p.11–12]. First we compute the fundamental invariants mod 7 of the permutation representation (very difficult in practice hitherto).

```
> G := PermutationGroup<10 | (2,5)(3,6)(4,7),(1,5,8,10,4)(2,6,9,3,7)>;
> #G;
120
> R := InvariantRing(G, GF(7));
> time F := FundamentalInvariants(R);
Time: 29.310
> {* Degree(f): f in F *};
```

```
{* 1, 2^^2, 3^^4, 4^^7, 5^^10, 6^^13, 7^^13, 8^^4, 9^^2 *}
```

Finally, we can compute a matrix representation of G as a direct sum of irreducible representations of degrees 1, 4 and 5. We then compute the fundamental invariants of the invariant ring of this representation mod 7.

```
> Q := RationalField();
> R0 := InvariantRing(G, Q);
> P0 := PolynomialRing(R0);
> M := GModule(G, Q);
> Gl := MatrixGroup(M);
> C := CharacterTable(Gl);
> Pi := [&+[Q!Integers()!c(g)*MatrixAlgebra(Q, 10)!g: g in Gl]/#G: c in C];
> Pi := [p: p in Pi | p ne 0];
> L := [sub<M | Image(p)>: p in Pi];
> G := MatrixGroup(DirectSum(DirectSum(L[1],L[2]),L[3]));
> G;
MatrixGroup(10, Rational Field)
Generators:
    [   1    0    0    0    0    0    0    0    0    0]
    [   0    1  1/3  1/3  1/3    0    0    0    0    0]
    [   0    0  1/3 -2/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3  1/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3 -2/3  1/3    0    0    0    0    0]
    [   0    0    0    0    0    1    0    0    0    0]
    [   0    0    0    0    0    0    0    0    1    0]
    [   0    0    0    0    0    0    0    0    0    1]
    [   0    0    0    0    0    0    1    0    0    0]
    [   0    0    0    0    0    0    0    1    0    0]
    [   1    0    0    0    0    0    0    0    0    0]
    [   0    0  1/3 -2/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3  1/3 -2/3    0    0    0    0    0]
    [   0    0 -2/3 -2/3  1/3    0    0    0    0    0]
    [   0    1  1/3  1/3  1/3    0    0    0    0    0]
    [   0    0    0    0    0   -1   -1    1    1    0]
    [   0    0    0    0    0   -1    0    0    0    1]
    [   0    0    0    0    0   -1    0    1    0    0]
    [   0    0    0    0    0    0   -1    0    0    0]
    [   0    0    0    0    0    0   -1    1    0    0]
> Gp := ChangeRing(G, GF(7));
> #Gp;
120
> Rp := InvariantRing(Gp);
> time Fp := FundamentalInvariants(Rp);
Time: 35.380
> {* Degree(f): f in Fp *};
{* 1, 2^^2, 3^^4, 4^^7, 5^^10, 6^^13, 7^^13, 8^^4, 9^^2 *}
> [Degree(f): f in F] eq [Degree(f): f in Fp];
```

```
true
```

110.14 The Module of an Invariant Ring

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. Suppose also that primary invariants $\{f_1, \ldots, f_n\}$ for R have been constructed, together with minimal secondary invariants $S = \{g_1, \ldots, g_m\}$ for R with respect to these primary invariants. (These secondary invariants may possess non-trivial module syzygies.) Then R can be considered as a module over the algebra $A = K[f_1, \ldots, f_n]$ with the minimal (module) generating set S.

To compute with this module structure of R easily, MAGMA automatically constructs the graded multivariate polynomial algebra $A' = K[t_1, \ldots, t_n]$ (with the weighted degree of the variable t_i defined to be the degree of f_i) which is isomorphic to A, and then constructs the graded module $M = A'^m/Q$ over A' with the quotient relations Q given by the syzygies of the g_i (and with the weighted degree of column i equal to the degree of g_i). The algebra A' is isomorphic to A under the map $t_i \mapsto f_i$, and the module M is isomorphic to R (considered as a module) under the map $M.i \mapsto g_i$ (extended by the isomorphism from A' onto A). (See the chapter on modules over $K[x_1, \ldots, x_n]$ for details on how to compute with the module M and an explanation of quotient relations, the unit vectors $M.i$, etc.) Once the module M is created, together with the isomorphism $f : R \to M$, one can apply f to a general element h of R to obtain the element of M corresponding to h. This effectively yields a representation of h as a sum $\sum_{i=1} ka_i g_i$ with $a_i \in A$ in terms of the primary and secondary invariants. This representation is also unique up to the relations given by the syzygies of the g_i.

When creating the module M, the coefficient ring A' of M is assigned the print names `"t1", "t2"`, etc. – the angle bracket notation or the . operator should be used to assign the variables of A' to actual MAGMA variables.

`Module(R)`

The module M isomorphic to $R = K[V]^G$, together with the isomorphism $f : R \to M$.

Example H110E11

We create the module M corresponding to the invariant ring R of the group G generated by the 4 by 4 Jordan block over $\mathbf{F}_3$.

```
> K := GF(3);
> G := MatrixGroup<4,K | [1,0,0,0, 1,1,0,0, 0,1,1,0, 0,0,1,1]>;
> R := InvariantRing(G);
> P<x1,x2,x3,x4> := PolynomialRing(R);
> p := PrimaryInvariants(R);
> s := SecondaryInvariants(R);
> [TotalDegree(f): f in p];
[ 1, 2, 3, 9 ]
```

```
> [TotalDegree(f): f in s];
[ 0, 3, 4, 5, 6, 7, 8, 9 ]
> M, f := Module(R);
> M;
Full Quotient Module of degree 8
TOP Order
Column weights: 0 3 4 5 6 7 8 9
Coefficient ring:
    Graded Polynomial ring of rank 4 over GF(3)
    Lexicographical Order
    Variables: t1, t2, t3, t4
    Variable weights: 1 2 3 9
Quotient Relations:
[
    t1[7] + 2*t2[6] + t3[5],
    t1[4] + 2*t2[3] + t3[2]
]
> h := x1^5*x2 + 2*x1^3*x3^3 + 2*x2^6;
> h;
x1^5*x2 + 2*x1^3*x3^3 + 2*x2^6
> m := f(h);
> m;
t1^4*t2[1] + t1^3[2] + t2^3[1]
> // Evaluate in the primaries and secondaries:
> p[1]^4*p[2]*s[1] + p[1]^3*s[2] + p[2]^3*s[1];
x1^5*x2 + 2*x1^3*x3^3 + 2*x2^6
```

110.15 The Algebra of an Invariant Ring and Algebraic Relations

Let $R = K[V]^G$ be the invariant ring of a finite group G over the field K and suppose the degree of G is n. Suppose also that primary invariants $\{f_1, \ldots, f_n\}$ for R have been constructed, together with minimal secondary invariants $S = \{g_1, \ldots, g_m\}$ for R with respect to these primary invariants. Suppose also that the irreducible secondary invariants for R are $S = \{h_1, \ldots, h_r\}$ so that the g_i are power products of the h_i. We write $g_i = p_i(h_i)$ where the p_i are monomials of the indeterminates $t_1, \ldots, t_r$. Then R is generated as an algebra over K by the primary invariants $f_1, \ldots, f_n$ and the irreducible secondary invariants $h_1, \ldots, h_r$. MAGMA allows the construction of a polynomial algebra A with indeterminate names `"f1"`, `"f2"`, etc. corresponding to the primary invariants and indeterminate names `"h1"`, `"h2"`, etc. corresponding to the irreducible secondary invariants. Thus R can be regarded as an homomorphic image of A and finding the algebraic relations between these (algebra) generators of R yields a presentation of R as a quotient of a polynomial algebra. The functions in this section construct the algebra A and the algebraic relations for R. When creating the algebra A, the algebra A is assigned the print names `"f1", "f2",`

`"h1", "h2"`, etc. – the angle bracket notation or the . operator should be used to assign the variables of A to actual MAGMA variables.

`Algebra(R)`

Given an invariant ring $R = K[V]^G$, return the polynomial algebra $A = K[f_1, \ldots, f_n, h_1, \ldots, h_r]$ of which R is an homomorphic image. This function also returns a sequence Q giving the secondary invariants in terms of the irreducible secondary invariants as monomials in A. Thus $Q[i]$ is the monomial $p_i(t_i)$ mentioned in the introduction to this section. Note that the secondary invariant 1 is *not* an irreducible secondary invariant so no h-variable corresponds to it (the polynomial 1 in A simply corresponds to it).

`Relations(R)`

Given an invariant ring $R = K[V]^G$, return a (sorted) sequence L giving the algebraic relations amongst the algebra generators of R as elements of the algebra A corresponding to R. Thus R is isomorphic as an algebra (or ring) to the quotient of A by the ideal of A generated by the relations in L.

`RelationIdeal(R)`

Given an invariant ring $R = K[V]^G$, return the ideal of algebraic relations corresponding to R. This is simply the same as taking the ideal generated by the algebra A by the sequence L returned by the function `Relations(R)`.

`PrimaryAlgebra(R)`

Given an invariant ring $R = K[V]^G$, return the algebra corresponding to the primary invariants of R as a graded polynomial ring (with the weights corresponding to the degrees of the primary invariants).

`PrimaryIdeal(R)`

Given an invariant ring $R = K[V]^G$, return the ideal generated by the primary invariants of R (this is stored in R).

Example H110E12

We create the invariant ring $R = K[V]^G$ where G is a degree-6 permutation representation of the direct product $C_3 \times C_3$ of two cyclic groups both of order 3 and K is the rational field. We construct the algebra A and the sequence Q giving the secondary invariants in terms of the irreducible secondary invariants. We then note that the degree-6 secondary invariant is obtained as the product of two degree-3 irreducible secondary invariants. We then construct the list L of algebraic relations in A for R. Thus R is isomorphic to the quotient ring $A/ < L >$. We then construct an homomorphism h from A onto R and check that the relations in L are correct. Finally, we check that the Hilbert series of the (quotient by the) ideal of A generated by L is the same as the Hilbert series of R as expected.

```
> G := PermutationGroup<6 | (1, 2, 3), (4, 5, 6)>;
> R := InvariantRing(G, RationalField());
```

```
> P := PrimaryInvariants(R);
> P;
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x4^3 + x5^3 + x6^3
]
> S := SecondaryInvariants(R);
> S;
[
    1,
    x1^2*x2 + x1*x3^2 + x2^2*x3,
    x4^2*x5 + x4*x6^2 + x5^2*x6,
    x1^2*x2*x4^2*x5 + x1^2*x2*x4*x6^2 + x1^2*x2*x5^2*x6 +
        x1*x3^2*x4^2*x5 + x1*x3^2*x4*x6^2 + x1*x3^2*x5^2*x6 +
        x2^2*x3*x4^2*x5 + x2^2*x3*x4*x6^2 + x2^2*x3*x5^2*x6
]
> H := IrreducibleSecondaryInvariants(R);
> H;
[
    x1^2*x2 + x1*x3^2 + x2^2*x3,
    x4^2*x5 + x4*x6^2 + x5^2*x6
]
> A, Q := Algebra(R);
> A;
Graded Polynomial ring of rank 8 over Rational Field
Lexicographical Order
Variables: f1, f2, f3, f4, f5, f6, h1, h2
Variable weights: 1 1 2 2 3 3 3 3
> Q;
[
    1,
    h1,
    h2,
    h1*h2
]
> // Thus S[4] must be H[1]*H[2]:
> S[4];
x1^2*x2*x4^2*x5 + x1^2*x2*x4*x6^2 + x1^2*x2*x5^2*x6 +
    x1*x3^2*x4^2*x5 + x1*x3^2*x4*x6^2 + x1*x3^2*x5^2*x6 +
    x2^2*x3*x4^2*x5 + x2^2*x3*x4*x6^2 + x2^2*x3*x5^2*x6
> H[1];
x1^2*x2 + x1*x3^2 + x2^2*x3
> H[2];
x4^2*x5 + x4*x6^2 + x5^2*x6
```

```
> H[1]*H[2] eq S[4];
true
> L := Relations(R);
> L;
[
    -1/24*f1^6 + 3/8*f1^4*f3 - 1/3*f1^3*f5 - 9/8*f1^2*f3^2 +
        2*f1*f3*f5 + f1*f3*h1 + 1/8*f3^3 - f5^2 - f5*h1 - h1^2,
    -1/24*f2^6 + 3/8*f2^4*f4 - 1/3*f2^3*f6 - 9/8*f2^2*f4^2 +
        2*f2*f4*f6 + f2*f4*h2 + 1/8*f4^3 - f6^2 - f6*h2 - h2^2
]
> // Construct homomorphism h from A onto (polynomial ring of) R:
> h := hom<A -> PolynomialRing(R) | P cat H>;
> // Check images of L under h are zero so that elements of L are relations:
> h(L);
[
    0,
    0
]
> // Create relation ideal and check its Hilbert series equals that of R:
> I := RelationIdeal(R);
> I;
Ideal of Graded Polynomial ring of rank 8 over Rational Field
Lexicographical Order
Variables: f1, f2, f3, f4, f5, f6, h1, h2
Variable weights: 1 1 2 2 3 3 3 3
Basis:
[
    f1^6 - 9*f1^4*f3 + 8*f1^3*f5 + 27*f1^2*f3^2 - 48*f1*f3*f5 -
        24*f1*f3*h1 - 3*f3^3 + 24*f5^2 + 24*f5*h1 + 24*h1^2,
    f2^6 - 9*f2^4*f4 + 8*f2^3*f6 + 27*f2^2*f4^2 - 48*f2*f4*f6 -
        24*f2*f4*h2 - 3*f4^3 + 24*f6^2 + 24*f6*h2 + 24*h2^2
]
> HilbertSeries(I);
(t^4 - 2*t^3 + 3*t^2 - 2*t + 1)/(t^10 - 4*t^9 + 6*t^8 - 6*t^7 +
    9*t^6 - 12*t^5 + 9*t^4 - 6*t^3 + 6*t^2 - 4*t + 1)
> HilbertSeries(I) eq HilbertSeries(R);
true
```

110.16 Properties of Invariant Rings

The following functions return non-trivial structural properties of invariant rings of finite groups.

`HilbertSeries(R)`

The Hilbert series of the invariant ring $R = K[V]^G$, returned as an element of the rational function field $\mathbf{Z}(t)$. The Molien series of G will be used if possible; otherwise (the modular matrix group case) secondary invariants for R will be constructed to determine the result.

`HilbertSeriesApproximation(R, n)`

The Hilbert series of the invariant ring $R = K[V]^G$, returned as a Laurent series with n known terms. The conjugacy classes of G will be used to compute the approximation.

`IsCohenMacaulay(R)`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return `true` iff R is Cohen-Macaulay. This is always true in the non-modular case. Otherwise, secondary invariants for R will be constructed to determine the result.

`FreeResolution(R)`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return a free resolution of (the module of) R. This is just the same as the invocation `FreeResolution(Module(R))`. The free resolution is returned as a sequence F such that $F[1]$ is M, $F[i+1]$ is the syzygy module of $F[i]$ for $i < \#F$, and the last element of F is free (its basis has no syzygies).

`MinimalFreeResolution(R)`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return a minimal free resolution of (the module of) R. This is just the same as the invocation `MinimalFreeResolution(Module(R))`.

`HomologicalDimension(R)`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the homological dimension of R. This is just the length of a minimal free resolution of R minus 1 (taking account of the fact that the module M of R is always included in the free resolution).

`Depth(R)`

Given the invariant ring $R = K[V]^G$ of the group G over the field K, return the depth of R. This is $n-d$ by the Auslander-Buchsbaum formula, where n is the rank of R and d is the homological dimension of R.

Example H110E13

We construct a minimal free resolution of the invariant ring of the group generated by the degree-5 Jordan block over $\mathbf{F}_2$ and verify that the depth is 3.

```
> K:=GF(2);
> G := MatrixGroup<5,K | [1,0,0,0,0, 1,1,0,0,0, 0,1,1,0,0,
>                         0,0,1,1,0, 0,0,0,1,1]>;
> R := InvariantRing(G);
> time F := MinimalFreeResolution(R);
Time: 0.690
> F;
Chain complex with terms of degree 3 down to -1
Dimensions of terms: 0 1 7 22 0
> Depth(R);
3
> HomologicalDimension(R);
2
```

Sections 110.17 and 110.18 present functions whose scope is not limited to the context of invariant theory.

110.17 Steenrod Operations

`SteenrodOperation(f, i)`

The i-th Steenrod operation $P^i(f)$ of f, which must be a multivariate polynomial with coefficients in a finite field, and i must be a non-negative integer.

Example H110E14

We demonstrate an elementary use of Steenrod operations.

```
> K:=GF(3);
> F4:=MatrixGroup<4,K |
>   [-1,0,0,0, 1,1,0,0, 0,0,1,0, 0,0,0,1],
>   [1,1,0,0, 0,-1,0,0, 0,1,1,0, 0,0,0,1],
>   [1,0,0,0, 0,1,-1,0, 0,0,-1,0, 0,0,1,1],
>   [1,0,0,0, 0,1,0,0, 0,0,1,1, 0,0,0,-1] >;
> R := InvariantRing(F4);
> f2 := InvariantsOfDegree(R, 2)[1];
> f4 := SteenrodOperation(f2, 1);
> f10 := SteenrodOperation(f4, 3);
> f4;
2*x1^4 + x1^3*x3 + 2*x1^3*x4 + x1*x3^3 + 2*x1*x4^3 + 2*x2^3*x3 + x2^3*x4 +
    2*x2*x3^3 + x2*x4^3 + x4^4
> f10;
2*x1^10 + x1^9*x3 + 2*x1^9*x4 + x1*x3^9 + 2*x1*x4^9 + 2*x2^9*x3 + x2^9*x4 +
```

```
    2*x2*x3^9 + x2*x4^9 + x4^10
> f4 in R;
true
> f10 in R;
true
```

110.18 Minimalization and Homogeneous Module Testing

The following functions work with collections of polynomials which are considered as generators for subalgebras or submodules of a polynomial ring. They are repeated from the chapter on multivariate polynomials since they are used extremely often in invariant theory to express an invariant in terms of the primary and secondary invariants of an invariant ring. Full descriptions of the functions are not given here. See the descriptions in the chapter on multivariate polynomials.

`MinimalAlgebraGenerators(L)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose L is a set or sequence of k polynomials $p_1, \ldots, p_k$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by L. This function returns a minimal generating set of the algebra A as a (sorted) sequence of elements taken from L.

`HomogeneousModuleTest(P, S, F)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose F is an element of R. This function returns whether F is in the module M (considered as a submodule of R). If the result is `true`, the function also returns a sequence $C = [c_1, \ldots, c_r]$ of length r with $c_i \in K[t_1, \ldots, t_r]$ such that $F = \sum_{i=1}^{r} c_i(p_1, \ldots, p_k) \cdot s_i$.

`HomogeneousModuleTest(P, S, L)`

Let $R = K[x_1, \ldots, x_n]$ be a polynomial ring of rank n over the field K. Suppose P is a sequence of k homogeneous polynomials $p_1, \ldots, p_k$ in R and suppose S is a sequence of r homogeneous polynomials $s_1, \ldots, s_r$ in R. Let $A = K[p_1, \ldots, p_k]$ be the subalgebra (*not* ideal) of R generated by P and let $M = A[s_1, \ldots, s_r]$ be the A-module generated by S over A. Finally, suppose L is a sequence of length l of elements of R which are all homogeneous of (weighted) degree d. This function returns parallel sequences B and V with the following properties:

(a) B is sequence of length l of booleans such that for $1 \leq i \leq l$, $B[i]$ is `true` iff $L[i]$ is in the module M.

(b) V is a sequence of length l consisting of sequences of length r and consisting of polynomials in the polynomial ring $T = K[t_1, \ldots, t_r]$. (The polynomial ring $T = K[t_1, \ldots, t_r]$ is constructed separately but automatically with the print names `t1`, `t2`, etc.) If $B[i]$ is `false` (so $L[i]$ is not in M), $V[i]$ is a sequence of r zero polynomials. Otherwise $V[i]$ is a sequence of r polynomials $c_{i,1}, \ldots, c_{i,r}$ in T such that that $L[i] = \sum_{j=1}^{r} c_{i,j}(p_1, \ldots, p_k) \cdot s_j$.

Example H110E15

We demonstrate how the function `MinimalAlgebraGenerators` can be used to compute fundamental invariants (in fact, the MAGMA function `FundamentalInvariants` does just this).

```
> K := RationalField();
> G := PermutationGroup<6 | (1,2,3)(4,5,6), (1,2)(4,5)>;
> R := InvariantRing(G, K);
> P := PrimaryInvariants(R);
> P;
[
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3,
    x4^3 + x5^3 + x6^3
]
> S := SecondaryInvariants(R);
> S;
[
    1,
    x1*x4 + x2*x5 + x3*x6,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x1^2*x4^2 + 2*x1*x2*x4*x5 + 2*x1*x3*x4*x6 + x2^2*x5^2 + 2*x2*x3*x5*x6 +
        x3^2*x6^2,
    x1^3*x4^3 + x1^2*x2*x4*x5^2 + x1^2*x3*x4*x6^2 + x1*x2^2*x4^2*x5 +
        x1*x3^2*x4^2*x6 + x2^3*x5^3 + x2^2*x3*x5*x6^2 + x2*x3^2*x5^2*x6 +
        x3^3*x6^3
]
> MinimalAlgebraGenerators(P cat S);
[
    1,
    x1 + x2 + x3,
    x4 + x5 + x6,
    x1^2 + x2^2 + x3^2,
    x1*x4 + x2*x5 + x3*x6,
    x4^2 + x5^2 + x6^2,
    x1^3 + x2^3 + x3^3 + x4*x5*x6,
    x1^2*x4 + x2^2*x5 + x3^2*x6,
```

```
    x1*x4^2 + x2*x5^2 + x3*x6^2,
    x4^3 + x5^3 + x6^3
]
```

Example H110E16

We demonstrate uses of the function `HomogeneousModuleTest` in invariant theory.

```
> // Create invariant ring R with primaries P, secondaries S
> R := InvariantRing(CyclicGroup(4), GF(2));
> P := PrimaryInvariants(R);
> S := SecondaryInvariants(R);
> #S;
5
> S[5];
x1^3*x3^2 + x1^2*x2^2*x3 + x1^2*x2*x3^2 + x1^2*x2*x4^2 +
    x1^2*x3^3 + x1^2*x3^2*x4 + x1*x2^2*x4^2 + x1*x3^2*x4^2 +
    x2^3*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 + x2^2*x4^3
> // Write S[2] in terms of P and S
> HomogeneousModuleTest(P, S, S[2]^2);
true [
    t1^2*t3^2 + t2^3,
    t1*t2,
    t1^3,
    0,
    0
]
> // Find all invariants I5 of degree 5
> I5 := InvariantsOfDegree(R, 5);
> I5;
[
    x1^5 + x2^5 + x3^5 + x4^5,
    x1^4*x2 + x1*x4^4 + x2^4*x3 + x3^4*x4,
    x1^4*x3 + x1*x3^4 + x2^4*x4 + x2*x4^4,
    x1^4*x4 + x1*x2^4 + x2*x3^4 + x3*x4^4,
    x1^3*x2^2 + x1^2*x4^3 + x2^3*x3^2 + x3^3*x4^2,
    x1^3*x2*x3 + x1*x2*x4^3 + x1*x3^3*x4 + x2^3*x3*x4,
    x1^3*x2*x4 + x1*x2^3*x3 + x1*x3*x4^3 + x2*x3^3*x4,
    x1^3*x3^2 + x1^2*x3^3 + x2^3*x4^2 + x2^2*x4^3,
    x1^3*x3*x4 + x1*x2^3*x4 + x1*x2*x3^3 + x2*x3*x4^3,
    x1^3*x4^2 + x1^2*x2^3 + x2^2*x3^3 + x3^2*x4^3,
    x1^2*x2^2*x3 + x1^2*x2*x4^2 + x1*x3^2*x4^2 + x2^2*x3^2*x4,
    x1^2*x2^2*x4 + x1^2*x3*x4^2 + x1*x2^2*x3^2 + x2*x3^2*x4^2,
    x1^2*x2*x3^2 + x1^2*x3^2*x4 + x1*x2^2*x4^2 + x2^2*x3*x4^2,
    x1^2*x2*x3*x4 + x1*x2^2*x3*x4 + x1*x2*x3^2*x4 + x1*x2*x3*x4^2
]
> // Write all elements of I5 in terms of P and S
> // (the t-variables correspond to elements of P and
```

```
> // the "columns" of the inner sequences to elements of S)
> HomogeneousModuleTest(P, S, I5);
[ true, true, true, true, true, true, true, true, true, true,
true, true, true, true ]
[
    [ t1^5 + t1^3*t2 + t1^3*t3 + t1*t2^2 + t1*t3^2 + t1*t4,
        0, t1^2 + t2 + t3, 0, 0 ],
    [ t1^3*t2 + t1^3*t3 + t1*t4, t1^2 + t2, t2 + t3, 0, 0 ],
    [ t1^3*t3 + t1*t3^2 + t1*t4, 0, t1^2 + t2 + t3, 0, 0 ],
    [ t1^3*t3 + t1*t2^2 + t1*t4, t1^2 + t2, t2 + t3, 0, 0 ],
    [ t1*t2^2 + t1*t3^2, t2, t1^2, 0, 1 ],
    [ t1*t2*t3, t3, t2 + t3, 0, 1 ],
    [ t1*t2*t3 + t1*t4, 0, t1^2 + t2, 0, 0 ],
    [ t1*t3^2 + t1*t4, 0, t3, 0, 0 ],
    [ 0, t3, t3, 0, 1 ],
    [ t1*t3^2, t2, t1^2 + t2, 0, 1 ],
    [ t1*t3^2, 0, 0, 0, 1 ],
    [ t1*t3^2, 0, t2, 0, 1 ],
    [ t1*t4, 0, t3, 0, 0 ],
    [ t1*t4, 0, 0, 0, 0 ]
]
```

110.19 Attributes of Invariant Rings and Fields

In this section we list various attributes of invariant rings which can be examined and set by the user. This allows low-level control of information stored in invariant rings or fields. Note that when the user sets an attribute, only minimal testing can be done on the value so if an incorrect value is set, unpredictable results may occur. Note also that if an attribute is not set, referring to it in an expression (using the ‘ operator) will *not* trigger the calculation of it (while intrinsic functions do); rather an error will ensue. Use the `assigned` operator to test whether an attribute is set.

`R‘PrimaryInvariants`

The attribute for the primary invariants of invariant ring $R = K[V]^G$. If the attribute `R‘PrimaryInvariants` is examined, either the current primary invariants of R are set so they are returned or an error results. If the attribute `R‘PrimaryInvariants` is set by assignment, it must be sequence of n algebraically-independent invariants of G, where n is the rank of R. MAGMA will not necessarily check that this condition is met since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value. Note that this attribute is useful when it is desired to compute secondary invariants of R with respect to some specially constructed primary invariants which would not be constructed by the automatic algorithm in MAGMA.

R‘SecondaryInvariants

The attribute for the secondary invariants of invariant ring $R = K[V]^G$. If the attribute R‘SecondaryInvariants is examined, either the current primary invariants of R are set so they are returned or an error results. If the attribute R‘SecondaryInvariants is set by assignment to Q, primary invariants for R must already be defined, and Q must be sequence of secondary invariants with respect to the primary invariants of R. MAGMA will not necessarily check that this condition is met since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value.

R‘HilbertSeries

The attribute for the Hilbert series of invariant ring $R = K[V]^G$. If the attribute R‘HilbertSeries is examined, either the Hilbert series of R is computed so it is returned or an error results. If the attribute R‘HilbertSeries is set by assignment to H, H must be rational function in the function field $Z(t)$ and equal to the Hilbert series of R. MAGMA will not necessarily check that this condition is met since that may be very time-consuming. If the attribute is already set, the new value must be the same as the old value.

Example H110E17

We demonstrate elementary uses of attributes.

```
> // Create group G and subgroup H of G and invariant rings
> // RG and RH of G and H respectively.
> G := CyclicGroup(4);
> H := sub<G|G.1^2>;
> RG := InvariantRing(G, GF(2));
> RH := InvariantRing(H, GF(2));
>
> // Create Hilbert Series S of RG and set it in RG.
> F<t> := FunctionField(IntegerRing());
> S := (t^3 + t^2 - t + 1)/(t^8 - 2*t^7 + 2*t^5 - 2*t^4 +
>     2*t^3 - 2*t + 1);
> RG‘HilbertSeries := S;
>
> // Note RG has no primary invariants yet so let Magma compute them as PG.
>  RG‘PrimaryInvariants;
>> RG‘PrimaryInvariants;
     ^
Runtime error in ‘: Attribute ’PrimaryInvariants’ for this structure is valid
but not assigned
> PG := PrimaryInvariants(RG);
> PG;
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
```

```
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
>
> // Set primary invariants of RH to PG and compute secondary
> // invariants of RH with respect to PG.
> RH`PrimaryInvariants := PG;
> SecondaryInvariants(RH);
[
    1,
    x2 + x4,
    x2*x4,
    x1*x2 + x1*x3 + x2^2 + x2*x4 + x3*x4 + x4^2,
    x1^2*x2 + x1*x2*x3 + x1*x3*x4 + x2^3 + x3^2*x4 + x4^3,
    x1^2*x2 + x1*x2^2 + x1*x2*x3 + x1*x2*x4 + x1*x3*x4 + x1*x4^2
        + x2^3 + x2^2*x3 + x2*x3*x4 + x3^2*x4 + x3*x4^2 + x4^3,
    x1*x2*x4^2 + x2^2*x3*x4 + x2^2*x4^2,
    x1^2*x2*x4^2 + x1*x2^2*x3*x4 + x1*x2^2*x4^2 + x1*x2*x3*x4^2 +
        x2^3*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 + x2^2*x4^3
]
```

110.20 Invariant Rings of Linear Algebraic Groups

By definition, a linear algebraic group is an affine variety G together with morphisms giving G the structure of a group. In the invariant theory algorithms of MAGMA, the group structure of G is nowhere required. Therefore an algebraic group will be defined by simply giving polynomials defining G as an affine variety. A G-module of an algebraic group is a finite dimensional vector space K^n together with a morphism $G \to \mathrm{GL}_n$ of algebraic groups. In MAGMA, such a morphism is given by specifying an n by n matrix whose entries are polynomials in the same variables as the polynomials specifying G (for more details, see section 110.6). An invariant ring of a linear algebraic group is constructed by giving a linear algebraic group together with a G-module. MAGMA makes no checks that the variety defined by the user has a multiplication making it into an algebraic group, or that the morphism $G \to \mathrm{GL}_n(K)$ really provides an action of G. If they do not, the computations will have unpredictable results. Likewise, MAGMA is unable to decide whether an algebraic group is reductive or linearly reductive. Therefore the user should indicate whether a group has these properties at creation by the options described below. This is important because Derksen's algorithm only works for linearly reductive groups.

110.20.1 Creation

`InvariantRing(I, A)`

`Reductive`	BoolElt	*Default :* `false`
`LinearlyReductive`	BoolElt	*Default :* `false`
`PolynomialRing`	RngMPol	*Default :*

Construct the invariant ring R for the algebraic group G defined by the ideal I and representation matrix A.

If the parameter `Reductive` is set to `true`, then G is assumed to be reductive, while if the parameter `LinearlyReductive` is set to `true`, then G is assumed to be linearly reductive.

If the parameter `PolynomialRing` is set to a value P, then P is used as the polynomial ring in which the invariants of R will lie.

`BinaryForms(N, p)`

`BinaryForms(n, p)`

Let $N = [n_1, \ldots, n_k]$ be a sequence of positive integers and let p be a positive prime or zero. Let $G = \mathrm{SL}_2(K)$ with K an algebraically closed field of characteristic p. This function defines the action on a direct sum of spaces of binary forms with degrees given by the n_i. The function returns three items: the ideal I_G defining G as an algebraic group, the representation matrix A (as a sequence of sequences of polynomials), and a polynomial ring on which G acts with appropriate naming of variables.

The second version of the function is given an integer n, and takes N to be $[n]$.

110.20.2 Access

`GroupIdeal(R)`

Given an invariant ring R defined over an algebraic group G, return the ideal I defining G.

`Representation(R)`

Given an invariant ring R defined over an algebraic group G, return the representation matrix A for G.

110.20.3 Functions

`InvariantsOfDegree(R, d)`

Return a K-basis of the space R_d of the homogeneous invariants of degree d in the invariant ring $R = K[V]^G$ of the algebraic group G over the field K as a sequence of polynomials.

`FundamentalInvariants(R)`

`Optimize`	BoolElt	*Default :* `true`
`Minimize`	BoolElt	*Default :* `true`
`MinimizeHilbert`	BoolElt	*Default :* `true`
`Force`	BoolElt	*Default :* `false`

Given an invariant ring R defined over an algebraic group, return a sequence of fundamental invariants of R, using Derksen's algorithm.

By default, the computation of homogeneous invariants is optimized by extending at each the degree the basis obtained from multiplying lower-degree invariants by appropriate monomials. This method can be suppressed by setting `Optimize` to `false`. By default the generators will be minimal. By setting the parameter `Minimize` to `false`, no minimization will be attempted. By setting the parameter `MinimizeHilbert` to `false`, the basis of the Hilbert ideal will not be minimized.

By default the group must be linearly reductive. Setting the parameter `Force` to `true` will force the application of Derksen's algorithm even though the group may not be linearly reductive.

`DerksenIdeal(R)`

Given an invariant ring R defined over an algebraic group, return a sequence of generators of the Derksen ideal of R. The Derksen ideal is an ideal D of $P[y_1, \ldots, y_n]$, where $P = K[x_1, \ldots, x_n]$ is the ambient polynomial ring of R, and the y_i are new indeterminates. By definition, D is the intersection of all the ideals

$$\langle y_1 - g(x_1), \ldots, y_n - g(x_n) \rangle$$

for all $g \in G$, the group of R. Geometrically, D is the vanishing ideal of the subset

$$\{(x, g(x)) | x \in K^n, g \in G\}$$

of the cartesian product $K^n \times K^n$.

`HilbertIdeal(R)`

`Minimize`	BoolElt	*Default :* `true`
`Force`	BoolElt	*Default :* `false`

Given an invariant ring R defined over a linear algebraic group, return the Hilbert ideal of R. This is the ideal in the polynomial ring generated by all non-constant, homogeneous invariants. The result is a sequence of homogeneous generators (not necessarily invariant).

By default the generators will be minimal. By setting the parameter `Minimize` to `false`, no minimization will be attempted. Also, setting the parameter `Force` to `true` will force the application of Derksen's algorithm even though the group may not be linearly reductive.

Example H110E18

We consider invariant ring of the group $G = SL_2(\mathbf{Q})$, which is is characterised by the equation $\det(A) = 1$.

```
> Q := RationalField();
> P<[a]>:=PolynomialRing(Q, 4);
> A := MatrixRing(P,2)!a;
> IG := ideal<P | Determinant(A) - 1>;
> IG;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: a[1], a[2], a[3], a[4]
Basis:
[
    a[1]*a[4] - a[2]*a[3] - 1
]
```

The simultaneous action of G on three vectors is given by the matrix $I_3 \otimes A$:

```
> T := TensorProduct(MatrixRing(P, 3) ! 1, A);
> T;
[a[1] a[2]    0    0    0    0]
[a[3] a[4]    0    0    0    0]
[   0    0 a[1] a[2]    0    0]
[   0    0 a[3] a[4]    0    0]
[   0    0    0    0 a[1] a[2]]
[   0    0    0    0 a[3] a[4]]
```

We create the invariant ring R of G (which is reductive) with this action and compute fundamental invariants.

```
> IR := InvariantRing(IG, T: Reductive);
> FundamentalInvariants(IR);
[
    x3*x6 - x4*x5,
    x1*x6 - x2*x5,
    x1*x4 - x2*x3
]
```

We see that there are three fundamental invariants. It is well known that the invariant ring of the simultaneous action of SL_n on m vectors is generated by the minors of the $n \times m$ matrix formed by the vectors. We can see this in the present case.

```
> R<x1,x2,x3,x4,x5,x6> := PolynomialRing(Q, 6);
> M := Matrix([[x1,x3,x5], [x2,x4,x6]]);
> M;
[x1 x3 x5]
[x2 x4 x6]
> Minors(M, 2);
[
```

```
    x1*x4 - x2*x3,
    -x1*x6 + x2*x5,
    x3*x6 - x4*x5
]
```

Example H110E19

As a second example, we consider the representation of the group $SL_2(\mathbf{Q}) \times SL_2(\mathbf{Q}) \times SL_2(\mathbf{Q})$ given by the tensor product of the canonical representation:

```
> n:=3;
> P<[x]>:=PolynomialRing(RationalField(), n*4, "grevlex");
> L_A := [MatrixRing(P,2)!x[i..i+3]:i in [1..n*4 by 4]];
> IG := ideal<P|[Determinant(A)-1:A in L_A]>;
> IG;
Ideal of Polynomial ring of rank 12 over Rational Field
Graded Reverse Lexicographical Order
Variables: x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8],
x[9], x[10], x[11], x[12]
Basis:
[
    -x[2]*x[3] + x[1]*x[4] - 1,
    -x[6]*x[7] + x[5]*x[8] - 1,
    -x[10]*x[11] + x[9]*x[12] - 1
]
>
> M:=L_A[1];
> for i:=2 to n do
>     M:=TensorProduct(M,L_A[i]);
> end for;
> M;
[x[1]*x[5]*x[9]   x[1]*x[5]*x[10]   x[1]*x[6]*x[9]   x[1]*x[6]*x[10]
    x[2]*x[5]*x[9]   x[2]*x[5]*x[10]   x[2]*x[6]*x[9]   x[2]*x[6]*x[10]]
[x[1]*x[5]*x[11]   x[1]*x[5]*x[12]   x[1]*x[6]*x[11]   x[1]*x[6]*x[12]
    x[2]*x[5]*x[11]   x[2]*x[5]*x[12]   x[2]*x[6]*x[11]   x[2]*x[6]*x[12]]
[x[1]*x[7]*x[9]   x[1]*x[7]*x[10]   x[1]*x[8]*x[9]   x[1]*x[8]*x[10]
    x[2]*x[7]*x[9]   x[2]*x[7]*x[10]   x[2]*x[8]*x[9]   x[2]*x[8]*x[10]]
[x[1]*x[7]*x[11]   x[1]*x[7]*x[12]   x[1]*x[8]*x[11]   x[1]*x[8]*x[12]
    x[2]*x[7]*x[11]   x[2]*x[7]*x[12]   x[2]*x[8]*x[11]   x[2]*x[8]*x[12]]
[x[3]*x[5]*x[9]   x[3]*x[5]*x[10]   x[3]*x[6]*x[9]   x[3]*x[6]*x[10]
    x[4]*x[5]*x[9]   x[4]*x[5]*x[10]   x[4]*x[6]*x[9]   x[4]*x[6]*x[10]]
[x[3]*x[5]*x[11]   x[3]*x[5]*x[12]   x[3]*x[6]*x[11]   x[3]*x[6]*x[12]
    x[4]*x[5]*x[11]   x[4]*x[5]*x[12]   x[4]*x[6]*x[11]   x[4]*x[6]*x[12]]
[x[3]*x[7]*x[9]   x[3]*x[7]*x[10]   x[3]*x[8]*x[9]   x[3]*x[8]*x[10]
    x[4]*x[7]*x[9]   x[4]*x[7]*x[10]   x[4]*x[8]*x[9]   x[4]*x[8]*x[10]]
[x[3]*x[7]*x[11]   x[3]*x[7]*x[12]   x[3]*x[8]*x[11]   x[3]*x[8]*x[12]
    x[4]*x[7]*x[11]   x[4]*x[7]*x[12]   x[4]*x[8]*x[11]   x[4]*x[8]*x[12]]
> IR:=InvariantRing(IG, M: Reductive);
```

```
> time FundamentalInvariants(IR);
[
    x1^2*x8^2 - 2*x1*x2*x7*x8 - 2*x1*x3*x6*x8 - 2*x1*x4*x5*x8 +
        4*x1*x4*x6*x7 + x2^2*x7^2 + 4*x2*x3*x5*x8 - 2*x2*x3*x6*x7 -
        2*x2*x4*x5*x7 + x3^2*x6^2 - 2*x3*x4*x5*x6 + x4^2*x5^2
]
Time: 0.610
> time DerksenIdeal(IR);
[
    y1^2*y8^2 - 2*y1*y2*y7*y8 - 2*y1*y3*y6*y8 - 2*y1*y4*y5*y8 + 4*y1*y4*y6*y7 +
        y2^2*y7^2 + 4*y2*y3*y5*y8 - 2*y2*y3*y6*y7 - 2*y2*y4*y5*y7 + y3^2*y6^2 -
        2*y3*y4*y5*y6 + y4^2*y5^2 - x1^2*x8^2 + 2*x1*x2*x7*x8 + 2*x1*x3*x6*x8 +
        2*x1*x4*x5*x8 - 4*x1*x4*x6*x7 - x2^2*x7^2 - 4*x2*x3*x5*x8 +
        2*x2*x3*x6*x7 + 2*x2*x4*x5*x7 - x3^2*x6^2 + 2*x3*x4*x5*x6 - x4^2*x5^2
]
Time: 0.010
> time HilbertIdeal(IR);
[
    x1^2*x8^2 - 2*x1*x2*x7*x8 - 2*x1*x3*x6*x8 - 2*x1*x4*x5*x8 + 4*x1*x4*x6*x7 +
        x2^2*x7^2 + 4*x2*x3*x5*x8 - 2*x2*x3*x6*x7 - 2*x2*x4*x5*x7 + x3^2*x6^2 -
        2*x3*x4*x5*x6 + x4^2*x5^2
]
Time: 0.000
```

So in this case, we find that the invariant ring is generated by a single polynomial.

Example H110E20

We compute fundamental invariants for the invariant ring of $G = \mathrm{SL}_2(\mathbf{Q})$ acting on a space of binary forms.

```
> IG, A := BinaryForms([1,1,2,2], 0);
> IG;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: t1, t2, t3, t4
Basis:
[
    t1*t4 - t2*t3 - 1
]
> A;
[t4   -t3   0   0   0   0   0   0   0   0]
[-t2   t1   0   0   0   0   0   0   0   0]
[0   0   t4   -t3   0   0   0   0   0   0]
[0   0   -t2   t1   0   0   0   0   0   0]
[0   0   0   0   t4^2   -t3*t4   t3^2   0   0   0]
[0   0   0   0   -2*t2*t4   t1*t4 + t2*t3   -2*t1*t3   0   0   0]
[0   0   0   0   t2^2   -t1*t2   t1^2   0   0   0]
[0   0   0   0   0   0   0   t4^2   -t3*t4   t3^2]
```

```
[0   0   0   0   0   0   0   -2*t2*t4   t1*t4 + t2*t3   -2*t1*t3]
[0   0   0   0   0   0   0   t2^2   -t1*t2   t1^2]
> R:=InvariantRing(IG,A: LinearlyReductive);
> time FundamentalInvariants(R);
[
    x8*x10 - 1/4*x9^2,
    x5*x10 - 1/2*x6*x9 + x7*x8,
    x5*x7 - 1/4*x6^2,
    x1*x4 - x2*x3,
    x1*x3*x7 - 1/2*x1*x4*x6 - 1/2*x2*x3*x6 + x2*x4*x5,
    x1*x3*x10 - 1/2*x1*x4*x9 - 1/2*x2*x3*x9 + x2*x4*x8,
    x3^2*x10 - x3*x4*x9 + x4^2*x8,
    x3^2*x7 - x3*x4*x6 + x4^2*x5,
    x1^2*x10 - x1*x2*x9 + x2^2*x8,
    x1^2*x7 - x1*x2*x6 + x2^2*x5,
    x1*x2*x5*x10 - x1*x2*x7*x8 - x2^2*x5*x9 + x2^2*x6*x8,
    x1*x4*x5*x10 - 1/2*x1*x4*x6*x9 + x1*x4*x7*x8 - 1/2*x2*x3*x5*x10 +
        1/2*x2*x3*x6*x9 - 3/2*x2*x3*x7*x8 - 1/2*x2*x4*x5*x9 + 1/2*x2*x4*x6*x8,
    x3*x4*x5*x10 - x3*x4*x7*x8 - x4^2*x5*x9 + x4^2*x6*x8
]
Time: 0.650
```

Example H110E21

We do simple computations on an invariant ring of an algebraic group. The group is not reductive, so fundamental invariants cannot be computed, but invariants of specific degrees can be.

```
> K := RationalField();
> Pa<a,b> := PolynomialRing(K, 2);
> IG := ideal<Pa|>;
> A := Matrix(7,
> [1, 0, 0, 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
> 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
> a, 1, 0, 0, 0, 0, 0, b, 0, 1 ]);
> A;
[1 0 0 0 0 0 0]
[a 1 0 0 0 0 0]
[0 0 1 0 0 0 0]
[0 0 a 1 0 0 0]
[0 0 0 0 1 0 0]
[0 0 0 0 a 1 0]
[0 0 0 0 b 0 1]
> R:=InvariantRing(IG, A);
> R;
Invariant Ring of algebraic group
Field of definition:
    Rational Field
> InvariantsOfDegree(R, 1);
```

```
[
    x1,
    x3,
    x5
]
> InvariantsOfDegree(R, 2);
[
    x1^2,
    x1*x3,
    x1*x4 - x2*x3,
    x1*x5,
    x1*x6 - x2*x5,
    x3^2,
    x3*x5,
    x3*x6 - x4*x5,
    x5^2
]
>  FundamentalInvariants(R);
>> FundamentalInvariants(R);
                         ^
Runtime error in 'FundamentalInvariants': Computing fundamental invariants (via
Derksen's algorithm) is only possible for linearly reductive groups
```

110.21 Invariant Fields

If G is a group acting on a polynomial ring $K[x_1, \ldots, x_n]$, it also acts on the rational function field $K(x_1, \ldots, x_n)$ by homomorphic extension. The invariants field $K(x_1, \ldots, x_n)^G$ is the field consisting of all functions which are fixed by G. MAGMA allows the construction of the invariant field by the function `InvariantField`. All that was said above about possible arguments of `InvariantRing` and access functions for invariant rings carries over to invariant fields. The category of invariant fields is `FldInvar`.

110.21.1 Creation

`InvariantField(G, K)`

`InvariantField(G)`

`InvariantField(I, A)`

`Reductive`	BOOLELT	*Default :* `false`
`LinearlyReductive`	BOOLELT	*Default :* `false`
`FunctionField`	FLDFUNRAT	*Default :*

Create the invariant field for the group G over the field K. The arguments and parameters are the same as for the function `InvariantRing`, in the three cases of permutation groups, matrix groups, and algebraic groups.

110.21.2 Access

`FunctionField(F)`

Given an invariant field F, return the underlying function field of F.

`Group(F)`

Given an invariant field F, return the underlying group of F.

`GroupIdeal(F)`

Given an invariant field F defined over an algebraic group G, return the ideal I defining G.

`Representation(F)`

Given an invariant field F defined over an algebraic group G, return the representation matrix A for G.

110.21.3 Functions for Invariant Fields

This section describes functions that apply to invariant fields.

`FundamentalInvariants(F)`

`Al`	MonStgElt	*Default :* "*BethMuellerQuade*"
`Minimize`	BoolElt	*Default :* `true`
`Min`	RngIntElt	*Default :* 0
`BottomUpTo`	RngIntElt	*Default :* 0

Given an invariant field F, return a sequence of fundamental invariants of F which generate F as an algebra over the base field of the ambient rational function field of F.

By default this function uses the algorithm of Beth and Müller-Quade [MQB99]. By setting the parameter `Al` to `"FleischmannKemperWoodcock"`, an alternative algorithm of Fleischmann, Kemper and Woodcock will be used.

By default the returned invariants will be minimal (in the sense of 'non-redundant'). By setting the parameter `Minimize` to `false`, no minimization will be attempted. The other parameters apply to the minimization and are as in the function `MinimizeGenerators` below.

`DerksenIdeal(F)`

Given an invariant field F, return the Derksen ideal of F. This is an ideal D in $K[y_1 \dots y_n]$, where $K = k(x_1 \dots x_n)$ is the ambient rational function field of F, and the y_i are new indeterminates. By definition, D is the intersection of all the ideals

$$< y_1 - g(x_1), \dots, y_n - g(x_n) >$$

for $g \in G$, the group of R. The function returns D as an ideal with a Groebner basis.

`MinimizeGenerators(L)`

`Min`	RngIntElt	*Default :* 0
`BottomUpTo`	RngIntElt	*Default :* 0

Suppose L is a set or sequence of non-constant elements of a rational function field. This function selects a minimal (in the sense of 'irredundant') subset of L which generates the same subfield as L. The function returns a sequence of such minimal generators.

If the parameter `Min` is set to $m > 0$, then the function stops when a generating set with m elements is reached ($m = 0$ is the default and implies no limit).

If the parameter `BottomUpTo` is set to $b > 0$, then the function first tries to eliminate generators by testing if they lie in the subfield generated by a small number of elements from L. This small number is limited by b.

`QuadeIdeal(L)`

`Fy`	BoolElt	*Default :*
`LargeIdeal`	BoolElt	*Default :* `false`

Suppose L is a non-empty set or sequence of non-constant elements from a rational function field $F = k(x_1, \ldots, x_n)$, generating a subfield $K = k(L)$. The Quade ideal, introduced in [MQS99], is the ideal in $F[y_1, \ldots, y_n]$ generated by the kernel of the map $K[y_1, \ldots, y_n] \to F$ given by $y_i \mapsto x_i$. This function returns the Quade ideal (with its basis being a Groebner basis).

The parameter `Fy` may be set to a polynomial ring P of rank n over F, so that the result is an ideal of P. If the parameter `LargeIdeal` is set to `true`, then an ideal in a larger polynomial ring is returned, whose intersection with $F[y_1, \ldots, y_n]$ is the Quade ideal.

Example H110E22

This example works with the invariant field of the finite group C_3 over the rational field.

```
> IF := InvariantField(CyclicGroup(3), RationalField());
> time L := FundamentalInvariants(IF);
Time: 1.780
> L;
[
    x1 + x2 + x3,
    (x1^2*x2 - 3*x1*x2*x3 + x1*x3^2 + x2^2*x3)/(x1^2 - x1*x2 - x1*x3 + x2^2 -
        x2*x3 + x3^2),
    (x1^3 - x1^2*x3 - x1*x2^2 + x2^3 - x2*x3^2 + x3^3)/(x1^2 - x1*x2 - x1*x3 +
        x2^2 - x2*x3 + x3^2)
]
> time DerksenIdeal(IF);
Ideal of Polynomial ring of rank 3 over Multivariate rational function field of
    rank 3 over Rational Field
Graded Reverse Lexicographical Order
```

```
Variables: y1, y2, y3
Dimension 0
Groebner basis:
[
    y2^2 + (-x1^3 + x1^2*x3 + x1*x2^2 - x2^3 + x2*x3^2 - x3^3)/(x1^2 - x1*x2 -
        x1*x3 + x2^2 - x2*x3 + x3^2)*y2 + (-x1^2*x2 + x1^2*x3 + x1*x2^2 -
        x1*x3^2 - x2^2*x3 + x2*x3^2)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 +
        x3^2)*y3 + (x1^3*x2 - x1^2*x2^2 - x1^2*x3^2 + x1*x3^3 + x2^3*x3 -
        x2^2*x3^2)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2),
    y2*y3 + (-x1^2*x2 + 3*x1*x2*x3 - x1*x3^2 - x2^2*x3)/(x1^2 - x1*x2 - x1*x3 +
        x2^2 - x2*x3 + x3^2)*y2 + (-x1^2*x3 - x1*x2^2 + 3*x1*x2*x3 -
        x2*x3^2)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2)*y3 + (x1^2*x2^2 -
        x1^2*x2*x3 + x1^2*x3^2 - x1*x2^2*x3 - x1*x2*x3^2 + x2^2*x3^2)/(x1^2 -
        x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2),
    y3^2 + (x1^2*x2 - x1^2*x3 - x1*x2^2 + x1*x3^2 + x2^2*x3 - x2*x3^2)/(x1^2 -
        x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2)*y2 + (-x1^3 + x1^2*x2 + x1*x3^2 -
        x2^3 + x2^2*x3 - x3^3)/(x1^2 - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2)*y3 +
        (x1^3*x3 - x1^2*x2^2 - x1^2*x3^2 + x1*x2^3 - x2^2*x3^2 + x2*x3^3)/(x1^2
        - x1*x2 - x1*x3 + x2^2 - x2*x3 + x3^2),
    y1 + y2 + y3 - x1 - x2 - x3
]
```

Example H110E23

We can compute with the invariant field of the non-reductive group presented above.

```
> K := RationalField();
> Pa<a,b> := PolynomialRing(K, 2);
> IG := ideal<Pa|>;
> A := Matrix(7,
> [1, 0, 0, 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
> 0, 0, 0, 0, a, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
> a, 1, 0, 0, 0, 0, 0, b, 0, 1 ]);
> A;
[1 0 0 0 0 0 0]
[a 1 0 0 0 0 0]
[0 0 1 0 0 0 0]
[0 0 a 1 0 0 0]
[0 0 0 0 1 0 0]
[0 0 0 0 a 1 0]
[0 0 0 0 b 0 1]
> IF := InvariantField(IG, A);
> IF;
Invariant field of algebraic group
Field of definition: Rational Field
> time FundamentalInvariants(IF);
[
    x5,
```

```
    x3/x5,
    x1,
    (x1*x6 - x2*x5)/x5,
    (x3*x6 - x4*x5)/x5
]
Time: 0.010
> DerksenIdeal(IF);
Ideal of Polynomial ring of rank 7 over Multivariate rational function field of
    rank 7 over Rational Field
Graded Reverse Lexicographical Order
Variables: y1, y2, y3, y4, y5, y6, y7
Groebner basis:
[
    y1 - x1,
    y2 - x1/x5*y6 + (x1*x6 - x2*x5)/x5,
    y3 - x3,
    y4 - x3/x5*y6 + (x3*x6 - x4*x5)/x5,
    y5 - x5
]
```

110.22 Invariants of the Symmetric Group

MAGMA includes basic functions for working with symmetric polynomials, which are invariants of the symmetric group.

`ElementarySymmetricPolynomial(P, k)`

Given a polynomial ring P of rank n, and an integer k with $1 \leq k \leq n$, return the k-th elementary symmetric polynomial of P.

`IsSymmetric(f)`

`IsSymmetric(f, S)`

Given a polynomial f from a polynomial ring P of rank n, return whether f is a symmetric polynomial of P (i.e., is symmetric in all the n variables of P). If the answer is true, a polynomial g from a new polynomial ring of rank n is returned such that $f = g(e_1, \ldots, e_n)$, where e_i is the i-th elementary symmetric polynomial of P. If g is desired to be a member of a particular polynomial ring S of rank n (to obtain predetermined names of variables, for example), then S may also be passed.

Example H110E24

We create a symmetric polynomial from $\mathbf{Q}[a, b, c, d]$ and express it in terms of the elementary symmetric polynomials.

```
> P<a, b, c, d> := PolynomialRing(RationalField(), 4, "grevlex");
> f :=
> a^2*b^2*c*d + a^2*b*c^2*d + a*b^2*c^2*d + a^2*b*c*d^2 + a*b^2*c*d^2 +
>     a*b*c^2*d^2 - a^2*b^2*c - a^2*b*c^2 - a*b^2*c^2 - a^2*b^2*d -
>     3*a^2*b*c*d - 3*a*b^2*c*d - a^2*c^2*d - 3*a*b*c^2*d - b^2*c^2*d -
>     a^2*b*d^2 - a*b^2*d^2 - a^2*c*d^2 - 3*a*b*c*d^2 - b^2*c*d^2 -
>     a*c^2*d^2 - b*c^2*d^2 + a + b + c + d;
> // Check orbit under Sym(4) has size one:
> #(f^Sym(4));
1
> Q<e1, e2, e3, e4> := PolynomialRing(RationalField(), 4);
> l, E := IsSymmetric(f, Q);
> l;
true
> E;
e1 - e2*e3 + e2*e4
```

In the following example, we use a rational function field to define parameters a and b which occur as coefficients of the symmetric polynomial f.

```
> F<a,b> := FunctionField(RationalField(), 2);
> P<x1,x2,x3,x4,x5> := PolynomialRing(F, 5, "grevlex");
> y1 := x1^4 + x1^2*a + x1*b;
> y2 := x2^4 + x2^2*a + x2*b;
> y3 := x3^4 + x3^2*a + x3*b;
> y4 := x4^4 + x4^2*a + x4*b;
> y5 := x5^4 + x5^2*a + x5*b;
> f := y1*y2 + y1*y3 + y1*y4 + y1*y5 + y2*y3 + y2*y4 +
>      y2*y5 + y3*y4 + y3*y5 + y4*y5;
> Q<e1,e2,e3,e4,e5> := PolynomialRing(F,  5);
> l,E := IsSymmetric(f, Q);
> l, E;
true b*e1^3*e2 - 2*a*e1^3*e3 - 4*e1^3*e5 + a*e1^2*e2^2 +
    4*e1^2*e2*e4 + 2*e1^2*e3^2 - b*e1^2*e3 + 2*a*e1^2*e4 -
    4*e1*e2^2*e3 - 3*b*e1*e2^2 + 4*a*e1*e2*e3 + 8*e1*e2*e5 +
    a*b*e1*e2 - 8*e1*e3*e4 - 2*a^2*e1*e3 + b*e1*e4 - 6*a*e1*e5 +
    e2^4 - 2*a*e2^3 - 4*e2^2*e4 + a^2*e2^2 + 4*e2*e3^2 +
    5*b*e2*e3 + 2*a*e2*e4 + b^2*e2 - 3*a*e3^2 - 4*e3*e5 -
    3*a*b*e3 + 6*e4^2 + 2*a^2*e4 - 5*b*e5
```

110.23 Bibliography

[**AM94**] A. Adem and R.J. Milgram. *Cohomology of Finite Groups*. Grundlehren der Mathematischen Wissenschaften. Springer, Berlin-New York-Heidelberg, 1994.

[**Der99**] Harm Derksen. Computation of Invariants for Reductive Groups. *Adv. Math.*, 141:366–384, 1999.

[**Kem96**] Gregor Kemper. Calculating Invariant Rings of Finite Groups over Arbitrary Fields. *J. Symbolic Comp.*, 21(3):351–366, 1996.

[**Kem99**] Gregor Kemper. An Algorithm to Calculate Optimal Homogeneous Systems of Parameters. *J. Symbolic Comp.*, 27(2):171–184, 1999.

[**Kin07**] Simon King. Minimal generating sets of non-modular invariant rings of finite groups. URL:http://arxiv.org/abs/math/0703035, 2007.

[**KS97**] Gregor Kemper and Allan Steel. Some Algorithms in Invariant Theory of Finite Groups. In P. Dräxler, G.O. Michler, and C.M. Ringel, editors, *Computational Methods for Representations of Groups and Algebras, Euroconference in Essen, April 1-5 1997*, number 173 in Progress in Mathematics, Basel, 1997. Birkhäuser.

[**MQB99**] Jörg Müller-Quade and Thomas Beth. Calculating Generators for Invariant Fields of Linear Algebraic Groups. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (Honolulu, HI, 1999)*, number 1719 in LNCS, pages 392–403, Berlin, 1999. Springer.

[**MQS99**] Jörg Müller-Quade and Rainer Steinwandt. Basic algorithms for rational function fields. *J. Symbolic Comp.*, 27(2):143–170, 1999.

111 DIFFERENTIAL RINGS

Chapter 111
DIFFERENTIAL RINGS

111.1 Introduction

The Galois theory of linear differential equations, or differential Galois theory, is the analogue of the classical Galois theory of polynomials for linear differential equations. Generally speaking one studies linear differential equations, that is differential equations of the form

$$L(y) = a_n y^{(n)} + a_{n-1} y^{(n-1)} + \cdots + a_1 y^{(1)} + a_0 y = 0,$$

in which the coefficients a_i are contained in some ring. The natural analogue of a field in the classical case is the notion of a differential field, that is a specific case of a differential ring. A *differential ring* F is equipped with an additive map $\delta_F : F \to F$ called a *derivation*, satisfying the multiplicative rule

$$\delta_F(a \cdot b) = \delta_F(a) \cdot b + a \cdot \delta_F(b), \quad a, b \in F.$$

A classical derivation is the usual derivative. All differential rings have a ring structure and have a map defined on them. A differential ring that is also a field is called a *differential field*.

The differential rings have type `RngDiff` and their elements have type `RngDiffElt`. All differential rings contain a *differential ring of constants* on which the derivation acts as the zero map. The differential rings and their elements inherit all functionality of the rings from which the differential ring is created. We call the ring from which a differential ring F is created the *underlying ring* of F.

A solution of a differential equation is an element of some differential field. It can happen that a solution is not an element of a given differential field F, but is an element of a differential extension of F. By this we mean a differential field (ring) M with $F \subset M$ such that the derivations satisfy $\delta_M|_F = \delta_F$. This is completely analogous to field extensions induced by solutions of a polynomial.

To clearly describe linear differential equations in Magma we formalize the concept of taking the derivative. To a differential field F with derivation δ_F, one associates a non–commutative ring $F[D]$, *the ring of linear differential operators*. An element of $F[D]$ is called a *differential operator*. A differential operator of *degree* $n \in \mathbf{Z}_{\geq 0}$ in $F[D]$ is of the form

$$L = a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0,$$

with $a_n \neq 0$ and all $a_i \in F$. Addition in $F[D]$ is term–wise and the multiplication of elements in $F[D]$ is determined by the rule

$$D * a = aD + \delta_F(a), \quad a \in F.$$

With these concepts $L(y) = 0$ is the linear differential equation

$$a_n\delta_F^n(y) + a_{n-1}\delta_F^{n-1}(y) + \cdots + a_1\delta_F(y) + a_0 y = 0.$$

For an introduction to the basic concepts in differential Galois theory, one is encouraged to consult [vdPS03]. This book is used as the basis for the implementation of differential rings, fields and operator rings in MAGMA.

111.2 Differential Rings and Fields

111.2.1 Creation

There are two ways to create a differential ring. The first creation is a general creation of a differential ring, for which the user specifies the ring and its derivation. The second creates a differential field which has the structure of a rational function field of transcendence degree 1 over its base field. Its derivation is specified by a differential.

Once a differential ring is created one can ask for its ring or field of fractions.

`DifferentialRing(P, f, C)`

Given a ring P and derivation f acting on P, return the differential ring isomorphic to P, with induced derivation f acting on it, and ring of constants C. The ring C should be a subring of P on which f is zero.

Example H111E1

Here we illustrate the creation and printing of a general differential ring.

```
> P := PolynomialRing(Rationals());
> f := map<P->P | a:->5*Derivative(a)>;
> R := DifferentialRing(P, f, Rationals());
> R;
Differential Ring of Univariate Polynomial Ring over
Rational Field with derivation given by Mapping
from: RngUPol: P to RngUPol: P given by a rule [no inverse]
```

`RationalDifferentialField(C)`

The differential field in one variable over the constant field C. If this field is called F, say, then the derivation on F is given by $d/(1)d(F.1)$, where $F.1$ is the variable of F, and $(1)d(F.1)$ is its differential in the differential space of F. Any exact field with polynomial GCD is valid input for C.

Example H111E2

Here we illustrate the creation and printing of the differential field obtained from the command `RationalDifferentialField`.

```
> F<z> := RationalDifferentialField(Rationals());
> F;
Differential Ring of Algebraic function field defined over
Rational Field by $.2 - 4711 with
derivation given by (1) d(z)
```

`DifferentialLaurentSeriesRing(C)`

The differential Laurent series ring (in one variable) over the constant field C. If this field is called F, say, then the derivation on F is given by $F.1 \cdot d/d(F.1)$, where $F.1$ is the variable of F.

Example H111E3

This example illustrates the creation and printing of the differential Laurent series ring obtained from the command `DifferentialLaurentSeriesRing`.

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> S;
Differential Ring of Laurent series field in t over Rational Field
with derivation given by Mapping from: Laurent series field in t over Rational
Field to Laurent series field in t over Rational Field given by a rule [no
inverse]
```

`RingOfFractions(R)`

Returns the differential ring $R[r^{-1} : r \in R$ not a zero divisor$]$ of fractions of the differential ring R, together with the inclusion map from R to the newly created ring.

`FieldOfFractions(R)`

Returns the differential field of fractions of the differential ring R, together with the inclusion map from R to the newly created field.

`AssignNames(~R, S)`

Given a differential ring R with n indeterminates and a sequence S of n strings, assign the elements of S to the names of the variables of R.

This procedure only changes the names used in the printing of the elements of R.

111.2.2 Creation of Differential Ring Elements

The easiest way to create an element in a given ring is to use the angle bracket construction to attach names to the indeterminates of the ring. Others are given below.

`Name(R, i)`

`R . i`

The i-th indeterminate of the differential ring R, where i is between 1 and the number of generators of R.

`R ! s`

Coerce the element s in the differential ring R. Elements that are coercible are elements that are coercible in the underlying ring of the differential ring R.

`Zero(R)`

The zero element of the differential ring R.

`One(R)`

`Identity(R)`

The identity element of the differential ring R.

`SeparatingElement(F)`

Returns the separating element of the algebraic differential field F.

Example H111E4

We construct the differential field $F = \mathbf{Q}(z)$ with derivation d/dz and show some of the elements that can be created.

```
> F<z> := RationalDifferentialField(Rationals());
> F.1;
z
> two := F!2;
> two;
2
> Parent(two) eq F;
true
> Zero(F); One(F);
0
1
> Parent(Zero(F)) eq F and Parent(Identity(F)) eq F;
true
> elt := SeparatingElement(F);
> elt;
z
> ISA(Type(elt),RngDiffElt);
true
> Parent(elt) eq F;
```

```
true
> elt eq F!SeparatingElement(UnderlyingRing(F));
true
```

111.3 Structure Operations on Differential Rings

111.3.1 Category and Parent

Differential Rings form the MAGMA category `RngDiff`. The notional power structures exist as parents of differential rings.

`Category(R)`

`Type(R)`

The category, or type, of the differential ring R.

`Parent(R)`

The power structure of the differential ring R.

111.3.2 Related Structures

The underlying ring and constant ring from which the differential ring was created can each be retrieved as described below. There is also the concept of a base ring. If one has created a differential extension M/F in MAGMA, then F is the *base ring* of M.

`UnderlyingRing(R)`

The underlying ring of the differential ring R. The type of the underlying ring indicates what ring R inherits from.

`UnderlyingField(R)`

The underlying ring of the differential ring R, provided it is a field.

`BaseRing(R)`

The base ring of the differential ring R.

`BaseField(R)`

The base ring of the differential ring R, provided it is a field.

`ConstantRing(R)`

The constant ring of the differential ring R. The derivation of R acts trivially on the constant ring. It is therefore contained in the differential ring of constants of R.

`ConstantField(R)`

The constant ring of the differential ring R, provided it is a field.

`ExactConstantField(F)`

The exact constant field of F, i.e. the algebraic closure in F of the constant field of F, together with the inclusion map to F. The field F must be a function field. The differential field F must have been created with respect to a differential. If the derivation of F has been constructed with respect to a differential, then the exact constant field coincides with the differential field of constants of F.

`Generators(R)`

The list of generators of the differential ring R. If there is no list assigned to R, one is constructed by default from the underlying ring of R.

Example H111E5

First we construct the differential field $F = \mathbf{Q}(z)$ with derivation d/dz and show what some of the related structures are. Then we construct the field extension $M = \mathbf{Q}(z, \alpha)$, where α is a root of the polynomial $X^2 - 2$. We do this with the usual `ext< >` constructor. For M we again derive some related structures.

```
> F<z> := RationalDifferentialField(Rationals());
> ConstantRing(F);
Rational Field
> UnderlyingRing(F);
Algebraic function field defined over Rational Field by
$.2 - 4711
>
> _<X> := PolynomialRing(F);
> M<alpha> := ext< F | X^2-2 >;
> BaseRing(M);
Differential Ring of Algebraic function field defined over Rational Field by
$.2 - 4711
with derivation given by (1) d(z)
> BaseRing(M) eq F;
true
> ConstantRing(M);
Rational Field
> E := ExactConstantField(M);
> E;
Number Field with defining polynomial $.1^2 - 2 over the Rational Field
> Generators(M);
[ alpha ]
```

Example H111E6

Related structures also exist for differential Laurent series rings.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> UnderlyingRing(S);
Laurent series field in t over Rational Field
```

```
> ConstantRing(S);
Rational Field
> Generators(S);
[ t ]
```

111.3.3 Derivation and Differential

The derivation of a differential ring and its differential, whenever applicable, can be retrieved as indicated below.

`Derivation(R)`

The derivation of the differential ring R.

`Differential(F)`

The differential belonging to the derivation of the differential field F. The field F must have been constructed in such a way that its derivation is defined by a differential.

Example H111E7

```
> F<z> := RationalDifferentialField(Rationals());
> Derivation(F);
Mapping from: RngDiff: F to RngDiff: F given by a rule [no inverse]
> Differential(F);
(1) d(z)
```

111.3.4 Numerical Invariants

`Ngens(R)`

The number of indeterminates associated with the differential ring R.

111.3.5 Predicates and Booleans

`R eq F`

Returns `true` if and only if the differential rings R and F are the same.

`IsIdentical(R, F)`

Returns `true` if and only if the differential rings R and F are identical.

`IsDomain(R)`

Returns `true` if and only if the differential ring R is a domain.

`IsField(R)`

Returns `true` if and only if the differential ring R is field.

`IsDifferentialField(R)`

Returns `true` if and only if the ring R is a differential field.

`IsAlgebraicDifferentialField(R)`

Returns `true` if and only if the field structure of the differential ring R is an algebraic function field.

`IsDifferentialSeriesRing(R)`

Returns `true` if and only if the underlying ring of the differential ring R is a series ring.

`IsDifferentialLaurentSeriesRing(R)`

Returns `true` if and only if the underlying ring of the differential ring R is a Laurent series ring and R has been created with a known constant ring.

Example H111E8

This example shows some booleans for various differential rings.

```
> F<z>:=RationalDifferentialField(Rationals());
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> IsAlgebraicDifferentialField(F);
true
> IsDifferentialSeriesRing(F);
false
> IsAlgebraicDifferentialField(S);
false
> IsDifferentialSeriesRing(S);
true
> IsDifferentialLaurentSeriesRing(S);
true
```

`HasProjectiveDerivation(F)`

Returns `true` if and only if F is a differential ring with derivation weakly of the form $(F.1) \cdot d/d(F.1)$.

`HasZeroDerivation(F)`

Returns `true` if and only if the algebraic differential field or differential series ring F has zero derivation. When F is a series ring we relax being zero to being weakly zero.

Example H111E9

```
> F<z>:=RationalDifferentialField(Rationals());
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> HasProjectiveDerivation(F);
false
> HasProjectiveDerivation(ChangeDerivation(F,z));
true
> HasZeroDerivation(F);
false
> HasProjectiveDerivation(S);
true
> HasProjectiveDerivation(ChangeDerivation(S,S!3));
false
> HasZeroDerivation(S);
false
```

111.3.6 Precision

`RelativePrecision(F)`

Returns the relative precision of the underlying series ring of F.

`RelativePrecisionOfDerivation(F)`

Given a differential Laurent series ring F, returns the relative precision of the ring derivative of $F.1$.

Example H111E10

This example illustrate the relative precision of differential rings.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> Derivative(t);
t
> IsDifferentialLaurentSeriesRing(S);
true
> RelativePrecision(S);
20
> RelativePrecision(UnderlyingRing(S));
20;
> V<w>:=DifferentialLaurentSeriesRing(Rationals():Precision:=30);
> RelativePrecision(V);
30
> RelativePrecision(V) eq RelativePrecision(UnderlyingRing(V));
true
```

Example H111E11

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> RelativePrecisionOfDerivation(S);
Infinity
> V<w> := ChangeDerivation(S,t+O(t^6));
> Derivation(V)(w);
w^2 + O(w^7)
> RelativePrecisionOfDerivation(V);
5
```

`ChangePrecision(F, p)`

Returns the differential series ring isomorphic to F with relative precision p. The map returned is the induced map of F to the new field.

Example H111E12

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RelativePrecision(S);
20
> V<w>,mp :=  ChangePrecision(S,10);
> Type(V);
RngDiff
> IsDifferentialLaurentSeriesRing(V);
true
> RelativePrecision(V);
10
```

```
> RelativePrecision(1/(w-1)) eq 10;
true
> mp(t) eq w;
true
> w@@mp eq t;
true
> derivt := Derivation(S)(t);
> derivt;
t
> derivw := Derivation(V)(w);
> derivw;
w
> mp(derivt) eq Derivation(V)(w);
true
```

111.4 Element Operations on Differential Ring Elements

111.4.1 Category and Parent

`Category(s)`

`Type(s)`

The category, or type, of the differential ring element s.

`Parent(s)`

The parent of the differential ring element s.

111.4.2 Arithmetic

All the usual arithmetic operations are possible for differential ring elements.

`s + t`

The sum of the two differential ring elements s and t.

`-s`

The negation of the differential ring element s.

`s - t`

The difference between the differential ring elements s and t.

`s * t`

The product of the differential ring elements s and t.

`s ^ n`

Given a differential ring element s and an integer n, return the n-th power of s. If s is invertible, n may be negative.

`s div t`

Given the differential ring elements s and t, return the exact division of s by t, if s is divisible by t.

`s / t`

Given the differential field elements s and t, return s divided by t.

111.4.3 Predicates and Booleans

`s eq t`

Return `true` iff the differential ring elements s and t are exactly the same.

`IsZero(s)`

Return `true` iff the differential ring element s is the zero element of its parent.

`IsOne(s)`

Return `true` iff the differential ring element s is the unity element of its parent.

`IsWeaklyEqual(s, t)`

Return `true` if and only if the differential ring element s is weakly equal to the differential ring element t.

`IsWeaklyZero(s)`

Return `true` if and only if the differential ring element s is weakly equal to the zero element of its parent.

`IsOrderTerm(s)`

Return `true` if and only if the differential ring element s is purely an order term of a differential series ring.

Example H111E13

This examples shows the booleans for various differential rings.

```
> F<z> := RationalDifferentialField(Rationals());
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> IsOne(F!1);
true
> t eq t+O(t^2);
false
> IsWeaklyEqual(t, t+O(t^2));
true
> IsWeaklyZero(t^(-1));
false
> IsWeaklyZero(O(t));
true
> IsOrderTerm(t+O(t^2));
false
> IsOrderTerm(O(t));
true
```

111.4.4 Coefficients and Terms

`O(s)`

Creates the order term of the differential series s.

`Truncate(s)`

The known part of the differential series s.

`Eltseq(s)`

Returns the coefficients of the differential ring element s.

`Exponents(s)`

Returns the interval from the valuation of s to (including) the degree of s.

Example H111E14

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> K<x>, mp := ext<F|X^2+X+1>;
> seq := Eltseq(x^2);
> seq;
[ -1, -1 ]
> Universe(seq) eq F;
true
```

Example H111E15

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> O(t+t^2);
O(t)
> Parent(O(t)) eq S;
true
> trunc := Truncate(t^(-1)+5*t^2 +O(t^4));
> trunc;
t^-1 + 5*t^2
> Parent(trunc) eq S;
true
> seq := Eltseq(trunc);
> seq;
[ 1, 0, 0, 5 ]
> Universe(seq) eq Rationals();
true
> Exponents(trunc);
[ -1 .. 2 ]
```

111.4.5 Conjugates, Norm and Trace

`MinimalPolynomial(s)`

The minimal polynomial of the differential field element s over the base field.

Example H111E16

```
> F<z> := RationalDifferentialField(Rationals());
> P<X> := PolynomialRing(F);
> K<x>, mp := ext<F|X^2+X+1>;
> f := MinimalPolynomial(x^2);
> f;
X^2 + X + 1
> Parent(f) eq P;
true
> g := MinimalPolynomial(x+3/2);
> g;
X^2 + -2*X + 7/4
```

111.4.6 Derivatives and Differentials

`Derivative(s)`

The image of s under the derivation of the parent of s. Notice that it can be different to the "usual" derivative, as it relies on the defined derivation.

`Differential(s)`

Returns the differential of s in the algebraic differential field F, as a differential in the differential space of the underlying ring of F.

Example H111E17

```
> F<z> := RationalDifferentialField(Rationals());
> Derivative(z^2 + 7/z);
(2*z^3 - 7)/z^2
> Differential(z);
(1) d(z)
> Differential(1/z+6+5*z);
((5*z^2 - 1)/z^2) d(z)
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> Derivative(5 + 2*t + 3*t^2);
2*t + 6*t^2
```

111.5 Changing Related Structures

Sometimes whilst working with a differential ring R, one might wish to consider the same ring, but with a different derivation or with a larger constant ring. It is a consequence of the creation of a differential ring, that its constant ring may actually be smaller than its differential ring of constants.

To alter the settings defined by the creation of a differential ring or field the following functions are available.

`ChangeDerivation(R, f)`

Returns a differential ring isomorphic to R, but whose derivation is the map $f\cdot$ `Derivation(`R`)` induced by the isomorphism. The ring element f must be non–zero. The isomorphism of R to the new differential ring is also returned. The new differential ring has the same underlying ring as R.

Example H111E18

```
> F<z> := RationalDifferentialField(Rationals());
> Derivative(z^2);
2*z
> K, toK := ChangeDerivation(F, z);
> K;
Differential Ring of Algebraic function field defined over Rational Field by
$.2 - 4711
with derivation given by (1/z) d(z)
> toK;
Mapping from: RngDiff: F to RngDiff: K given by a rule
> Derivative(toK(z^2));
2*z^2
> UnderlyingRing(F) eq UnderlyingRing(K);
true
```

Notice that the differential of K is $(1/z)d(z)$, so that the derivation of K is $z \cdot d/dz$, as requested.

`ChangeDifferential(F, df)`

Returns the algebraic differential field, whose underlying ring is the one of F, but with derivation with respect to the differential df. The map returned is the bijective map from F into the new algebraic differential field.

Example H111E19

```
> F<z> := RationalDifferentialField(Rationals());
> df := Differential(1/z);
> df in DifferentialSpace(UnderlyingRing(F));
true
> M<u>, mp := ChangeDifferential(F,df);
> IsAlgebraicDifferentialField(M);
true
> Domain(mp) eq F and Codomain(mp) eq M;
true
> Differential(M);
(-1/u^2) d(u)
> mp(z);
u
> Derivation(M)(u);
u^2
> Derivation(F)(z);
1
> dg := Differential(z^3+5);
> N<v>, mp := ChangeDifferential(F,dg);
> Differential(M);
```

```
(3*v^2) d(v)
> mp(z);
v
> Derivation(N)(mp(z));
1/3/v^2
```

`ConstantFieldExtension(F, C)`

Returns the differential field isomorphic to the differential field F, but whose constant field is the extension C, and the isomorphism from F to the new field. The differential field F must be an algebraic function field.

Example H111E20

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> M := ext< F | X^2-2 >;
> ConstantField(M);
Rational Field
> _<x>:=PolynomialRing(Rationals());
> C := NumberField(x^2-2);
> Mext, toMext := ConstantFieldExtension(M, C);
> ConstantField(Mext);
Number Field with defining polynomial x^2 - 2 over the Rational Field
> toMext;
Mapping from: RngDiff: M to RngDiff: Mext given by a rule
```

Example H111E21

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> P<T> := PolynomialRing(Rationals());
> Cext := ext<Rationals()|T^2+1>;
> Sext<text>, mp := ConstantFieldExtension(S,Cext);
> IsDifferentialLaurentSeriesRing(Sext);
true
> ConstantRing(Sext) eq Cext;
true
> Derivative(text^(-2)+7+2*text^3+O(text^6));
-2*text^-2 + 6*text^3 + O(text^6);
> mp;
Mapping from: RngDiff: S to RngDiff: Sext given by a rule
> mp(t);
text
```

`Completion(F, p)`

`Precision` RngIntElt *Default :* ∞

The completion of the differential field F with respect to the place p. The place p should be an element of the set of places of F. The derivation of the completion is the one naturally induced by the derivation of F. The map returned is the embedding of F into the completion. Upon creation one can set the precision by using `Precision`. If no precision is given, then a default value is taken.

Example H111E22

This example illustrates the creation of the differential Laurent series ring by using the command `Completion`.

```
> F<z> := RationalDifferentialField(Rationals());
> pl := Zeros(z)[1];
> S<t>, mp := Completion(F,pl: Precision := 5);
> IsDifferentialLaurentSeriesRing(S);
true
> mp;
Mapping from: RngDiff: F to RngDiff: S given by a rule
> Domain(mp) eq F, Codomain(mp) eq S;
true true
> Derivation(S)(t);
1
> 1/(1-t);
1 + t + t^2 + t^3 + t^4 + O(t^5)
```

Example H111E23

This example shows that one does not have to restrict to differential fields of genus 0 to use `Completion`.

```
> F<z> := RationalDifferentialField(Rationals());
> P<Y> := PolynomialRing(F);
> K<y> := ext<F|Y^2-z^3+z+1>;
> Genus(UnderlyingRing(K));
1
> pl:=Zeros(K!z)[1];
> Degree(pl);
2
> S<t>, mp := Completion(K,pl);
> IsDifferentialLaurentSeriesRing(S);
true
> C<c> := ConstantRing(S);
> C;
Number Field with defining polynomial $.1^2 + 1 over the Rational Field
> mp(y) + O(t^4);
```

```
c - t - 4*t^3 + O(t^4)
```

111.6 Ring and Field Extensions

The first differential ring and field extensions we consider are the ones induced by a differential operator. Given a differential operator

$$L = a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0, \quad a_n \neq 0$$

in a differential operator ring $F[D]$ with coefficients in a differential field F, we construct a ring or field extension of degree n over F, whose indeterminates play the role of a formal solution of $L(y) = 0$ and its derivatives.

Given a differential field F, it is also possible to construct differential extensions of the form $F[X]/f(X)$, where $f(X)$ is an irreducible polynomial over F.

`DifferentialRingExtension(L)`

Constructs a differential ring extension of the base ring of the differential operator L, by adding a formal solution of L and its formal derivatives as indeterminates.

Let P denote the new differential ring, and F the coefficient ring of L. The ring F is a differential field. If n is the degree of L, the underlying ring of P is a multivariate polynomial ring of degree n over F. We thus have $P = F[Y_1, Y_2, \ldots, Y_n]$, with indeterminates $Y_1, Y_2, \ldots, Y_n$. If L is written as $a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0 \in F[D]$, then the derivation of P is induced by the differential operator L as follows: $\delta_P(Y_i) = Y_{i+1}$, for $i < n$ and $a_n \delta_P(Y_n) = -a_{n-1} Y_{n-1} - \cdots - a_2 Y_2 - a_1 Y_1$. With this construction Y_1 mimics a solution of $L(y) = 0$, and all the others are its derivatives.

`DifferentialFieldExtension(L)`

Constructs a differential field extension of the base ring of the differential operator L, by adding a formal solution of L and its formal derivatives as indeterminates.

The construction of the new differential field is completely analogous to the differential ring created by `DifferentialRingExtension(L)`. The only difference is that now a differential field $M = F(Y_1, Y_2, \ldots, Y_n)$, with n indeterminates $Y_1, Y_2, \ldots, Y_n$ is created. The action of the derivation of M on $Y_1, Y_2, \ldots, Y_n$ is as described in `DifferentialRingExtension(L)`.

Example H111E24

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := z^2*D^2-z*D+1;
> P<Y1,Y2> := DifferentialRingExtension(L);
> P;
Differential Ring Extension over F
with derivation given by Mapping from: Polynomial ring of rank 2 over F to
Polynomial ring of rank 2 over F given by a rule [no inverse]
> Derivative(Y1);
Y2
> Derivative(Y2);
-1/z^2*Y1 + 1/z*Y2
```

Example H111E25

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := z^2*D^2-1;
> M<Y,DY> := DifferentialFieldExtension(L);
> IsDifferentialField(M);
true
> Derivative(Y);
DY
> Derivative(DY);
-1/z^2*Y
```

`ext< F | f >`

The differential field extension $F(\alpha)$ of the differential field F, where α is a root of the irreducible polynomial f over F. The angle bracket notation may be used to assign the root α to an identifier.

Example H111E26

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> M<alpha> := ext< F | X^2-z >;
> M;
Differential Ring Extension over F by $.1^2 - z
with derivation given by (1) d(z)
> alpha^2;
z
```

The differential of M is the differential dz of the differential space of F lifted to the space of differentials of M.

`ExponentialFieldExtension(F, f)`

Returns the differential field $F(E)$ as an extension of F, such that the derivation of E is $f \cdot E$. The parent of f must be F.

`LogarithmicFieldExtension(F, f)`

Returns the differential field $F(L)$ as an extension of F, such that the derivation of L is $F(L)!f$. The parent of f must be F.

Example H111E27

```
> F<z> := RationalDifferentialField(Rationals());
> K<E> := ExponentialFieldExtension(F, z);
> K;
Differential Ring Extension over F
with derivation given by Mapping from: Multivariate Rational function field of
rank 1 over F to Multivariate Rational function field of rank 1 over F given by
a rule [no inverse]
> Derivative(E);
z*E
> _<L> := LogarithmicFieldExtension(F, 1/z);
> Derivative(L);
1/z
> Parent($1) eq Parent(L);
true
```

`PurelyRamifiedExtension(f)`

Creates a purely ramified field extension M of the differential field F with respect to the purely ramified polynomial $f \in F[X]$. By definition, such a polynomial f is of the form $X^n - a \cdot (F.1)$ for some constant element a in F and positive integer n. The returned extension field M is of the same type as F. The allowed differential fields are algebraic differential fields and differential Laurent series rings. When F is a differential Laurent series ring, its derivation is required to be weakly of the form $c * (F.1) * d/d(F.1)$ for some constant c. The relative precision of M is then n times the relative precision of F. The second argument returned is the embedding map of F into M. The inverse map acts on elements for which it is defined. Otherwise it returns 0.

Example H111E28

A purely ramified extension of an algebraic differential field is constructed in this example.

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> Fext<v>, mp := PurelyRamifiedExtension(X^2-5*z);
> IsAlgebraicDifferentialField(Fext);
true
> mp(z) eq 1/5*v^2;
true
> Parent(mp(z)) eq Fext;
true
> Derivation(Fext)(mp(z));
1
> Derivation(Fext)(v);
1/2/z*v
> Derivation(Fext)(v^2) eq Fext!5;
true
> Inverse(mp)(v^2);
5*z;
```

Example H111E29

A differential Laurent series ring with a derivation without an order term is considered in this example.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> _<T>:=PolynomialRing(S);
> pol := T^4-5*t;
> Sext<r>,mp := PurelyRamifiedExtension(pol);
> IsDifferentialLaurentSeriesRing(Sext);
true
> BaseRing(Sext) eq S and ConstantField(Sext) eq ConstantField(S);
true
> RelativePrecision(Sext);
80
> RelativePrecisionOfDerivation(Sext);
Infinity
> Derivation(S)(t);
t
> mp(t);
1/5*r^4
> Derivation(Sext)(mp(t));
1/5*r^4
> mp(Derivation(S)(t));
1/5*r^4
> x := 4+6*t+O(t^6);
> mp(x);
```

```
4 + 6/5*r^4 + O(r^24)
> Derivation(Sext)(mp(x));
6/5*r^4 + O(r^24)
> mp(Derivation(S)(x));
6/5*r^4 + O(r^24)
> Inverse(mp)(r^4-r^8);
5*t - 25*t^2
> Inverse(mp)(r^4+O(r^5));
5*t + O(t^2)
> Derivation(Sext)(r);
1/4*r
```

Example H111E30

The ring in this example has an order term in its derivation. Therefore, taking a derivative of an element x is of influence on the relative precision of the image of x.

```
> F<z> := RationalDifferentialField(Rationals());
> FF<z>:=ChangeDerivation(RationalDifferentialField(Rationals()),z);
> RR<DD>:=DifferentialOperatorRing(FF);
> RS<DS>, mpRRtoRS :=Completion(RR,Zeros(z)[1]);
> S<t>:=BaseRing(RS);
> IsDifferentialLaurentSeriesRing(S);
true
> _<T> := PolynomialRing(S);
> E<r>, mp := PurelyRamifiedExtension(T^3-5*t);
> IsDifferentialLaurentSeriesRing(E);
true
> RelativePrecision(E);
60
> RelativePrecisionOfDerivation(E);
60
> Derivation(E)(r);
1/3*r + O(r^61);
> mp(t);
1/5*r^3
> Derivation(S)(t);
t + O(t^21)
> Derivation(E)(mp(t));
1/5*r^3 + O(r^63)
> mp(Derivation(S)(t));
1/5*r^3 + O(r^63)
> x:=t^(-2) +7+t^3 +O(t^15);
> Derivation(S)(x);
-2*t^-2 + 3*t^3 + O(t^15)
> Derivation(E)(mp(x));
-50*r^-6 + 3/125*r^9 + O(r^45)
> mp(Derivation(S)(x));
```

```
-50*r^-6 + 3/125*r^9 + O(r^45)
> y := 2*t+O(t^25);
> Derivation(S)(y);
2*t + O(t^21)
> Derivation(E)(mp(y)) eq mp(Derivation(S)(y));
true
> Derivation(E)(mp(y));
2/5*r^3 + O(r^63)
```

111.7 Ideals and Quotient Rings

A differential ideal $I \subset R$ of a differential ring R is an ideal of R that is closed under the derivation of R. However, we consider a differential ideal as an ideal of the underlying ring of R. More specifically, ideals of differential rings are restricted to those rings whose underlying rings are multivariate polynomial rings.

111.7.1 Defining Ideals and Quotient Rings

`DifferentialIdeal(L)`

Given a sequence L with entries in a differential ring R, return the differential ideal generated by the entries of L as an ideal of the underlying ring of R. The underlying ring of R must be of type `RngMPol`. At first the elements of L may generate an ideal which is not closed under the derivation of R. By adding as many derivatives of the elements to the set of generators of the ideal as needed, one obtains a full set of generators for the calculated differential ideal.

`QuotientRing(R, I)`

Given a differential ring R and a differential ideal I, return the differential quotient ring $Q = R/I$. The derivation of Q is induced by the derivation of R. It maps $Q.i$ to $Q!\delta_R(R.i)$, for $i = 1, 2, \ldots, m$ where m is the number of generators of Q (or R). The induced quotient map from R to Q is also returned.

Example H111E31

```
> P := PolynomialRing(Rationals(),1);
> f := map<P->P | a:->a*Derivative(a,1)>;
> R<T> := DifferentialRing(P, f, Rationals());
> L := [T^2+T-1];
> I := DifferentialIdeal(L);
> I;
Ideal of Polynomial ring of rank 1 over Rational Field
Lexicographical Order
Variables: T
Basis:
```

```
[
    T^2 + T - 1,
]
> Q<X>, toQ := QuotientRing(R,I);
> Q;
Differential Ring of Affine Algebra of rank 1 over Rational Field
Lexicographical Order
Variables: X
Quotient relations:
[
    X^2 + X - 1
]
with derivation given by Mapping from: Affine Algebra of rank 1 over Rational
    Field to Affine Algebra of rank 1 over Rational Field given by a rule [no
inverse]
> toQ(T);
X
> Derivative(T^2);
2*T^3
> Derivative(X^2);
X
```

111.7.2 Boolean Operations on Ideals

`IsDifferentialIdeal(R, I)`

Returns `true` if and only if I is a differential ideal of the differential ring R.

111.8 Wronskian Matrix

Let R be a differential ring and let $y_1, y_2, \ldots, y_n$ be elements of R. The *wronskian matrix* of $y_1, y_2, \ldots, y_n$ is defined as the $n \times n$ matrix

$$W(y_1, y_2, \ldots, y_n) = \begin{pmatrix} y_1 & y_2 & \cdots & y_n \\ \delta_R(y_1) & \delta_R(y_2) & \cdots & \delta_R(y_n) \\ \vdots & \vdots & \ddots & \vdots \\ \delta_R^{n-1}(y_1) & \delta_R^{n-1}(y_2) & \cdots & \delta_R^{n-1}(y_n) \end{pmatrix}$$

The *wronskian determinant*, or simply the *wronskian*, of $y_1, y_2, \ldots, y_n$ is the determinant of the wronskian matrix $W(y_1, y_2, \ldots, y_n)$.

`WronskianMatrix(L)`

Given a sequence of differential ring elements L, return the Wronskian matrix of L whose entries are elements of the universe of L.

`WronskianDeterminant(L)`

Given a sequence of differential ring elements L, return the determinant of the Wronskian matrix of L as well as the matrix itself.

Example H111E32

```
> F<z> := RationalDifferentialField(Rationals());
> WronskianMatrix([1,z,z^2]);
[1 z z^2]
[0 1 2*z]
[0 0 2]
> WronskianDeterminant([1,z^2,1/z]);
6/z
[z z^2 1/z]
[1 2*z -1/z^2]
[0 2 2/z^3]
```

111.9 Differential Operator Rings

111.9.1 Creation

`DifferentialOperatorRing(F)`

Returns the differential operator ring over the differential field F.

Example H111E33

```
> F<z> := RationalDifferentialField(Rationals());
> R := DifferentialOperatorRing(F);
> R;
Differential operator ring over Differential Ring of Algebraic function field
defined over Rational Field by
$.2 - 4711
with derivation given by (1) d(z)
```

`AssignNames(~R, S)`

Given a differential operator ring R with n indeterminates and a sequence S of n strings, assign the elements of S to the names of the variables of R.

This procedure only changes the names used in the printing of the elements of R.

111.9.2 Creation of Differential Operators

The easiest way to create an element in a given ring is to use the angle bracket construction to attach a name to the indeterminate of the differential operator ring. Other constructions are given below.

`Name(R, i)`

`R . i`

The i-th indeterminate of the differential ring R, where i must be 1.

`R ! s`

Coerce the element s into the differential operator ring R. Elements that are coercible into R are elements coercible into its underlying ring, sequences, and differential operators defined over the base ring of the coefficient ring of R.

When the base ring of R is an algebraic differential field, elements of other differential operator rings over algebraic differential fields can be coerced into R so long as the underlying rings of the differential fields are the same.

`Zero(R)`

The zero element of the differential operator ring R.

`One(R)`

The identity element of the differential operator ring R.

Example H111E34

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> R.1;
D
> R!(1/z);
1/z;
>R![1/2,0,5,z];
z*D^3 + 5*D^2 + 1/2
> S<T> := DifferentialOperatorRing(ChangeDerivation(F,z));
> R!T;
z*D
> S!D;
1/z*T
```

111.10 Structure Operations on Differential Operator Rings

111.10.1 Category and Parent

Differential Operator Rings form the MAGMA category `RngDiffOp`. The notional power structures exist as parents of differential operator rings.

`Category(R)`

`Type(R)`

The category, or type, of the differential operator ring R.

`Parent(R)`

The power structure of the differential operator ring R.

111.10.2 Related Structures

As outlined in the introduction, a differential operator ring R is of the form $F[D]$, for a differential ring F. The ring F is called the *base ring* or *coefficient ring* of R.

`BaseRing(R)`

`CoefficientRing(R)`

The base ring, or coefficient ring, of the differential operator ring R.

`ConstantRing(R)`

The constant ring of the differential ring operator R.

111.10.3 Derivation and Differential

By construction the variable D of a differential operator ring $F[D]$ is related to the derivation δ_F. That is why δ_F is also considered to be the derivation of R.

`Derivation(R)`

The derivation of the differential operator ring R.

`Differential(R)`

The differential belonging to the derivation of the differential operator ring R. The derivation must have been constructed in such a way that it is defined by a differential.

Example H111E35

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> BaseRing(R) eq F;
true
> Derivation(R);
Mapping from: RngDiff: F to RngDiff: F given by a rule [no inverse]
> Differential(R);
(1) d(z)
```

111.10.4 Predicates and Booleans

`R eq F`

Returns `true` if and only if the differential operator rings R and F are the same.

`IsIdentical(R, F)`

Returns `true` if and only if the differential operator rings R and F are identical.

`IsDifferentialOperatorRing(R)`

Returns `true` if and only if the given argument is a differential operator ring.

`HasProjectiveDerivation(R)`

Returns `true` iff R is defined over a ring F with derivation weakly of the form $(F.1) \cdot d/d(F.1)$.

`HasZeroDerivation(R)`

Returns `true` iff the base ring of R is an algebraic differential field or a differential series ring F such that the derivation of R acts as a (weak) zero derivation on $F.1$.

Example H111E36

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> IsDifferentialOperatorRing(F);
false
> IsDifferentialOperatorRing(R);
true
> Derivation(R)(z);
1
> HasProjectiveDerivation(R);
false
> HasProjectiveDerivation(ChangeDerivation(R,z));
true
> HasZeroDerivation(R);
false
```

Example H111E37

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> V<W> := DifferentialOperatorRing(S);
> IsDifferentialOperatorRing(V);
true
> Derivation(V)(t);
t
> HasProjectiveDerivation(V);
true
> HasZeroDerivation(V);
false
> P<Q>, mp := ChangeDerivation(V,3/t);
> IsDifferentialOperatorRing(P);
true
> HasProjectiveDerivation(P);
false
> X<y> := BaseRing(P);
> Q*y;
y*Q + 3
```

111.10.5 Precision

`RelativePrecisionOfDerivation(R)`

The relative precision of the derivation of an operator ring over a Laurent series ring.

Example H111E38

This example illustrates the relative precision of derivations of differential operatorrings.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> RelativePrecisionOfDerivation(RS);
Infinity
> RV<DV> := ChangeDerivation(RS, t^2+O(t^8));
> relprec := RelativePrecisionOfDerivation(RV);
> relprec;
6
> RelativePrecisionOfDerivation(BaseRing(RV)) eq relprec;
true
```

111.11 Element Operations on Differential Operators

111.11.1 Category and Parent

`Category(L)`

`Type(L)`

The category, or type, of the differential operator L.

`Parent(L)`

The parent of the differential operator L.

111.11.2 Arithmetic

All the usual arithmetic operations are possible for differential operators. It follows from the multiplication rule for differential operators that the multiplication of differential operators is non–commutative.

`s + t`

The sum of the two differential operators s and t.

`-s`

The negation of the differential operator s.

`s - t`

The difference between the differential operators s and t.

`s * t`

The product of the differential operators s and t.

`s ^ n`

Given a differential operator s and an integer $n \geq 0$, return the n-th power of s.

Example H111E39

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> (z*D-1)*(D+1);
z*D^2 + (z - 1)*D + -1
> (D+1)*(z*D-1);
z*D^2 + z*D + -1
> (D-1/z)^2;
D^2 + -2/z*D + 2/z^2
```

111.11.3 Predicates and Booleans

`s eq t`

Return `true` iff the differential operators s and t are exactly the same.

`IsZero(L)`

Return `true` iff the differential operator L is the zero element of its parent.

`IsOne(L)`

Return `true` iff the differential operator L is the unity element of its parent.

`IsMonic(L)`

Return `true` iff the differential operator L is monic.

`IsWeaklyEqual(L, P)`

Returns `true` if and only if the differential operator L is weakly equal to the operator P. This means that the i-th coefficients of L and P should be weakly equal to each other for every $i \in [0..\max(\deg(L), \deg(P))]$.

`IsWeaklyZero(L)`

Returns `true` if and only if the differential operator $L \in R$ is weakly equal to $R!0$.

`IsWeaklyMonic(L)`

Returns `true` if and only if the leading coefficient of the differential operator L is weakly equal to 1.

111.11.4 Coefficients and Terms

Differential operators look like univariate polynomials with coefficients in a differential ring. Some of the terminology used for polynomial rings is mimicked for differential operators.

`Eltseq(L)`

`Coefficients(L)`

Given an operator L with coefficients in R, this function returns the sequence of elements in R, that are the coefficients of L. The sequence is ordered from the constant coefficient to the coefficient of the highest order term of L.

`Coefficient(L, i)`

Given an operator L with coefficients in R, this function returns the coefficient of the monomial of degree i in L, as an element of R.

`LeadingCoefficient(L)`

Given an operator L with coefficients in R, this function returns the coefficient of the highest order term of L.

`LeadingTerm(L)`

The leading term of the differential operator L.

`Terms(L)`

Given an operator L with coefficients in R, this function returns the sequence of non–zero coefficients of L as elements of R. The sequence is ordered from the lowest order term to the highest order term in L.

Example H111E40

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^3  + (-4*z + 5)*D + (3*z - 4);
> L;
D^3 + (-4*z + 5)*D + 3*z - 4
> Eltseq(L);
[ 3*z - 4, -4*z + 5, 0, 1 ]
> LeadingTerm(L);
D^3
> Terms(L);
[
    3*z - 4,
    (-4*z + 5)*D,
    D^3
]
```

111.11.5 Order and Degree

`Order(L)`
`Degree(L)`

Returns the order of the differential operator L. In the case that L is identically 0, the order is defined to be -1.

`WeakOrder(L)`
`WeakDegree(L)`

If the differential operator L is defined over a differential series ring, then the exponent of the highest coefficient of L that is not weakly 0 is returned.

Example H111E41

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> L := D^2 + 2*t;
> P := O(t)*D^3 + (1+O(t))*D^2  + 2*t;
> Order(L);
2
> Degree(P);
3
> L eq P;
false
> IsWeaklyEqual(L,P);
true
> WeakOrder(P);
2
```

111.11.6 Related Differential Operators

`MonicDifferentialOperator(L)`

Given the differential operator L, this function returns the monic differential operator $1/c \cdot L$, where c is the leading coefficient of L.

`Adjoint(L)`

Returns the formal adjoint of the differential operator L. The *formal adjoint* of $L = \sum_{i=0}^{n} a_i D^i$ in the differential operator ring $R = F[D]$ over F, is the differential operator $L^* := \sum_{i=0}^{n} (-1)^i D^i * a_i \in R$. It follows from the definition that the orders of L and L^* are the same and that the leading coefficient of L^* is $(-1)^n a_n$.

`Translation(L, e)`

If R is the parent of the differential operator L and e is a suitable ring element, then the operator in R obtained by replacing $R.1$ by $R.1 + e$ in L is returned. The second argument returned is the translation map on R by e.

`TruncateCoefficients(L)`

If L is defined over a differential series ring, then returned is the operator whose coefficients are the truncations of the coefficients of L.

Example H111E42

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := z*D^3  + (-4*z + 5)*D + (3*z - 4);
> Order(L);
3
> MonicDifferentialOperator(L);
D^3 + (-4*z + 5)/z*D + (3*z - 4)/z
> Adjoint(L);
-z*D^3 + -3*D^2 + (4*z - 5)*D + 3*z
> trans, mp := Translation(L, 2);
> trans;
z*D^3 + 6*z*D^2 + (8*z + 5)*D + 3*z + 6
```

Example H111E43

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L := (5-O(t))*DS^3+(2*t^-1+t^2+O(t^4))*DS - t^-2+t+O(t^3);
> L;
(5 + O(t))*DS^3 + (2*t^-1 + t^2 + O(t^4))*DS + -t^-2 + t + O(t^3)
> TruncateCoefficients(L);
5*DS^3 + (2*t^-1 + t^2)*DS + -t^-2 + t
> L -TruncateCoefficients(L);
O(t)*DS^3 + O(t^4)*DS + O(t^3)
```

111.11.7 Application of Operators

As pointed out in the introduction a differential operator $L = a_n D^n + a_{n-1} D^{n-1} + \cdots + a_1 D + a_0$ in $F[D]$ leads to the differential equation $L(y) = 0$ given by

$$L(y) = a_n \delta_F^n(y) + a_{n-1} \delta_F^{n-1}(y) + \cdots + a_1 \delta_F(y) + a_0 y$$

This notation is formal, but also defines an action of L on any element $y \in F$. The function `Apply` returns the ring element obtained by this action.

`Apply(L, f)`
`L(f)`
`f @ L`

Given a differential operator L and a ring element f, return the ring element obtained after applying L to f, as an element of the base ring of L. The element f must be coercible into the base ring of L.

Example H111E44

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^2-2/z^2;
> Apply(L, z);
-2/z
> L(z);
-2/z
> Apply(L, z^2);
0
```

111.12 Related Maps

This section is devoted to maps between differential operator rings.

`TranslationMap(R, e)`

Returns a map on the differential operator ring R that replaces $R.1$ by $R.1 + e$ when applied to a differential operator for some suitable ring element e.

`LiftMap(m, R)`

Let $m : F \to M$ be a differential map on differential fields and R a differential operator ring over F. Then this routine lifts the given map to a map on the differential operator rings $R \to S$, where the basefield of S is M.

Example H111E45

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> transmap := TranslationMap(R, 2 + z);
> Codomain(transmap) eq R;
> transmap(D);
D + z + 2
> transmap(D^2);
D^2 + (2*z + 4)*D + z^2 + 4*z + 5
> P<T> := PolynomialRing(F);
> M<u>, mp := ext<F|T^2+z>;
> liftmap := LiftMap(mp, R);
> Rprime<Dprime> := Codomain(liftmap);
> IsDifferentialOperatorRing(Rprime);
true
> BaseRing(Rprime) eq M;
true
> liftmap(D);
Dprime
```

```
> liftmap(R!z);
z
> Derivation(Rprime)(liftmap(z));
1
> Derivation(Rprime)(u);
1/2/z*u
```

111.13 Changing Related Structures

It may happen that certain intrinsics only work for differential operator rings whose derivations are of a specific form, or whose constant fields have to be large enough. Some of the functions available for changing settings of the differential rings or fields can be used to change the desired related structure on the operator ring directly. To alter some of the settings of a differential operator ring, the following functions are available.

`ChangeDerivation(R, f)`

Returns a differential operator ring isomorphic to R, but whose derivation is given by $f*$ `Derivation(`R`)`. The ring element f must be non–zero. The isomorphism of R to the new differential ring is also returned. The base ring of the new differential operator ring is isomorphic to the one of R, but it has derivation `ChangeDerivation(BaseRing(`R`))`.

`ChangeDifferential(R, df)`

Returns the differential operator ring with differential df, and whose underlying ring of its basefield coincides with the one of R. The map returned is the bijective map of R into the new operator ring. The base ring of the new differential operator ring is isomorphic to the one of R. However, the returned inclusion map and taking derivatives may not be commutative.

Example H111E46

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> df := Differential(z^3+5);
> RM<DM>, mp := ChangeDifferential(R,df);
> Domain(mp) eq R and Codomain(mp) eq RM;
true
> M<u> := BaseRing(RM);
> IsDifferentialOperatorRing(RM) and IsAlgebraicDifferentialField(M);
true
> mp(RM!z);
u
> mp(D);
3*u^2*DM
```

```
> D*z, mp(D*z);
u*D + 1
3*u^3*DM + 1
> DM*u;
u*DM + 1/3/u^2
> Differential(RM);
(3*u^2) d(u)
```

`ConstantFieldExtension(R, C)`

Returns the ring of differential operators with base ring isomorphic to that of the differential operator ring R, but whose constant field is C. The derivation is extended over the new constant field. The second argument returned is the map from R to the new operator ring.

`PurelyRamifiedExtension(R,f)`

When R is a differential operator ring over a differential ring F, this function returns an operator ring over the purely ramified extension of F, as induced by the polynomial f. The polynomial f is of the form $X^n - a \cdot (F.1)$ for some constant element a in F and positive integer n.

Example H111E47

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> _<T> := PolynomialRing(S);
> Rext<Dext>, mp := PurelyRamifiedExtension(R, T^7-3*t);
> Sext<text> := BaseRing(Rext);
> Domain(mp) eq R and Codomain(mp) eq Rext;
true
> IsDifferentialLaurentSeriesRing(Sext);
true
> BaseRing(Sext) eq S;
true
> RelativePrecision(Sext) eq 7*RelativePrecision(S);
true
> D*t;
t*D + t
> mp(D);
Dext
> mp(R!t) eq Rext!(1/3*text^7);
true
> Dext*text;
text*Dext + 1/7*text
```

`Completion(R, p)`

`Precision` RngIntElt *Default :* ∞

Returns the operator ring $\tilde{R}$, whose base ring is the completion of the base ring of the operator ring R w.r.t. the place p. The second return value is the natural embedding of R into $\tilde{R}$. The precision of the base ring of $\tilde{R}$ can be set by setting `Precision` upon creation. If no precision is set, a default value for the precision is taken.

`Localization(R, p)`

Returns the operator ring whose differential has valuation -1 at p, with derivation $t \cdot d/dt$, where t is the uniformizing element at the place p. The natural map between the operator rings, and the induced image of p are also returned.

`Localization(L, p)`

Given the differential operator L over an algebraic differential field, returns the localized operator of L at the place p. The embedding map between the parents as well as the induced image of the place are also returned.

`Localization(R)`

Given a differential operator ring R over the differential Laurent series ring $C((t))$, returns the operator ring whose derivation is of the form $t \cdot d/dt$, and the natural map between the operator rings.

`Localization(L)`

Given the differential operator L over a differential series ring, returns the localized operator of L and the embedding map between the parents.

Example H111E48

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> D*t;
t*D + t
> Rprime<Dprime>, mp := ChangeDerivation(R,t^2);
> Fprime<tprime> := BaseRing(Rprime);
> mp;
Mapping from: RngDiffOp: R to RngDiffOp: Rprime given by a rule
> Dprime*tprime;
tprime*Dprime + tprime^3
> P<T> := PolynomialRing(Rationals());
> Cext := ext<Rationals()|T^2+1>;
> Rext<Dext>, mp := ConstantFieldExtension(R,Cext);
> Cext eq ConstantRing(BaseRing(Rext));
true
> mp(D);
```

Dext

Example H111E49

This examples illustrates how to use Completion on operator rings.

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> pl := Zeros(z)[1];
> Rcompl<Dcompl>, mp := Completion(R,pl);
> IsDifferentialOperatorRing(Rcompl);
true
> S<t> := BaseRing(Rcompl);
> IsDifferentialLaurentSeriesRing(S);
true
> mp(D);
Dcompl
> Dcompl*t;
t*Dcompl + 1
```

Example H111E50

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> pl := Zeros(z-1)[1];
> Rloc<Dloc>, mp, place:=  Localization(R,pl);
> Domain(mp) eq R, Codomain(mp) eq Rloc;
true true
> place;
(z - 1)
> Differential(BaseRing(Rloc));
(1/(z - 1)) d(z)
> mp(D);
1/(z - 1)*Dloc
> Dloc*(z-1);
(z - 1)*Dloc + z - 1
> L := D + z;
> Lloc, mp, place := Localization(L,Zeros(z)[1]);
> Lloc;
1/z*$.1 + z
```

111.14 Euclidean Algorithms, GCDs and LCMs

A ring of differential operators shares many properties with a univariate polynomial ring. Two of them are GCD and LCM algorithms. However, a consequence of the non–commutative multiplication of a differential operator ring is that the GCD and LCM algorithms cannot be used directly. For instance, in the euclidean algorithm multiplication of the quotient can be done on the left or the right. Therefore one needs to specify the direction of the multiplication in the GCD and LCM algorithms for differential operator rings.

111.14.1 Euclidean Right and Left Division

`EuclideanRightDivision(N, D)`

Given differential operators N and D, return two differential operators Q and R, such that $N = Q \cdot D + R$, with `Degree(`R`)` $<$ `Degree(`D`)`. An error occurs if D is 0.

`EuclideanLeftDivision(D, N)`

Given differential operators D and N, return two differential operators Q and R, such that $N = D \cdot Q + R$, with `Degree(`R`)` $<$ `Degree(`D`)`. An error occurs if D is 0.

Example H111E51

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L1 := D;
> L2 := (D-3)*(D+z);
> EuclideanRightDivision(L1, L2);
0
D
> Q, R := EuclideanRightDivision(L2, L1);
> Q, R;
D + z - 3
-3*z + 1
> L2 eq Q*L1+R;
true
> EuclideanLeftDivision(L2, L1);
0
D
> S, T := EuclideanLeftDivision(L1, L2);
> S, T;
D + z - 3
-3*z
> L2 eq L1*S+T;
true
```

111.14.2 Greatest Common Right and Left Divisors

`GreatestCommonRightDivisor(A, B)`

`GCRD(A, B)`

Given two differential operators $A, B \in R$, return the unique monic differential operator $G \in R$ that generates the left ideal $RA + RB$.

`ExtendedGreatestCommonRightDivisor(A, B)`

Given two differential operators $A, B \in R$, this function returns three operators $G, U, V \in R$, that satisfy $U \cdot A + V \cdot B = G$. The differential operator G is the unique monic right GCD of A and B.

`GreatestCommonLeftDivisor(A, B)`

`GCLD(A, B)`

Given two differential operators $A, B \in R$, return the unique monic differential operator $G \in R$ that generates the right ideal $AR + BR$.

`ExtendedGreatestCommonLeftDivisor(A, B)`

Given two differential operators $A, B \in R$, this function returns three operators $G, U, V \in R$, that satisfy $A \cdot U + B \cdot V = G$. The differential operator G is the unique monic left GCD of A and B.

Example H111E52

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L1 := D^3+z*D^2+D-z;
> L2 := D^2+(z-3)*D-3*z+1;
> GreatestCommonRightDivisor(L1, L2);
D + z
> GreatestCommonRightDivisor(L1, L2) eq GCRD(L1, L2);
true
> G, U, V :=ExtendedGreatestCommonRightDivisor(L1, L2);
> G, U, V;
D + z
1/8
-1/8*D + -3/8
> G eq U*L1+V*L2;
true
> GreatestCommonLeftDivisor(L1, L2);
1
> GCLD(L2,L2*L1) eq L2;
true
```

111.14.3 Least Common Left Multiples

`LeastCommonLeftMultiple(L)`

Let $L = D - r$ be a monic operator of degree 1 in $R = F[D]$. Return the least common left multiple of L and all its conjugates over the base ring of F, with respect to the coercion of this base ring into F.

`LeastCommonLeftMultiple(A, B)`

`LCLM(A, B)`

Given two differential operators $A, B \in R$, return the unique monic differential operator $L \in R$, that generates the left ideal $RA \cap RB$. The order of the least common multiple of A and B is at most `Order(`A`)+ Order(`B`)`.

`ExtendedLeastCommonLeftMultiple(A, B)`

Given two differential operators $A, B \in R$, return three operators $L, U, V \in R$, that satisfy $L = U \cdot A = V \cdot B$. The differential operator L is the unique monic left LCM of A and B.

`ExtendedLeastCommonLeftMultiple(S)`

Given the non–empty sequence of differential operators S, this function returns the unique monic left LCM L of the entries of S, as well as a sequence Q of length $\#S$, satisfying $L = Q[i] \cdot S[i]$ for $i = 1, 2, \ldots, \#S$.

Example H111E53

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> LCLM(D, D-z);
D^2 + (-z^2 - 1)/z*D
> L1 := D^3+z*D^2+D-z;
> L2 := D^2+(z-3)*D-3*z+1;
> LeastCommonLeftMultiple(L1, L2);
D^4 + (z - 3)*D^3 + (-3*z + 2)*D^2 + (-z - 3)*D + 3*z - 1
> L, U, V := ExtendedLeastCommonLeftMultiple(L1, L2);
> L, U, V;
D^4 + (z - 3)*D^3 + (-3*z + 2)*D^2 + (-z - 3)*D + 3*z - 1
D + -3
D^2 + -1
> L eq U*L1;
true
> L eq V*L2;
true
> L, Q := ExtendedLeastCommonLeftMultiple([D,D+1,z*D+1]);
> L;
D^3 + (z^2 - 6)/(z^2 - 2*z)*D^2 + (2*z - 6)/(z^2 - 2*z)*D
> Q[3]*(z*D+1) eq L;
true
```

Example H111E54

```
> F<z> := RationalDifferentialField(Rationals());
> P<T> := PolynomialRing(F);
> M<u> := ext<F|T^2+T+1>;
> RM<DM> := DifferentialOperatorRing(M);
> LeastCommonLeftMultiple(DM-u^2);
DM^2 + DM + 1
> lclm := LeastCommonLeftMultiple(DM-u+1);
DM^2 + 3*DM + 3
> EuclideanRightDivision(lclm, DM-u+1);
DM + u + 2
0
> N<v>, mp := ext<F|T^2-z>;
> RN<DN> := DifferentialOperatorRing(N);
> lclm := LeastCommonLeftMultiple(DN-v);
> lclm;
DN^2 + -1/2/z*DN + -z
> LeastCommonLeftMultiple(DN-v, DN+v) eq lclm;
true
> EuclideanRightDivision(lclm,DN-v);
DN + v - 1/2/z
0
> EuclideanRightDivision(lclm,DN+v);
DN + -v - 1/2/z
0
```

111.15 Related Matrices

The *companion matrix* of a monic linear differential operator

$$D^n + a_{n-1}D^{n-1} + \cdots + a_0 \in F[D]$$

is defined as the $n \times n$ matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \\ -a_0 & -a_1 & -a_2 & -a_3 & \dots & -a_{n-1} \end{pmatrix}$$

`CompanionMatrix(L)`

Returns the companion matrix of the monic differential operator L.

Example H111E55

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^3-z*D^2+2*D+5;
> CompanionMatrix(L);
[0 1 0]
[0 0 1]
[-5 -2 z]
```

111.16 Singular Places and Indicial Polynomials

All functions treated in this section concern differential operator rings defined over a function field of transcendence degree one. They all require that the derivation of such an operator ring $F[D]$ over F is defined with respect to a differential.

Any solution of a differential operator $L \in F[D]$ is also a solution of the operator $c \cdot L$, for any non–zero $c \in F$. For considering solutions we may therefore consider L to be monic of the form

$$L = D^n + a_{n-1}D^{n-1} + \cdots + a_0,$$

with coefficients $a_i \in F$ for $i = 1, 2, \ldots n-1$. Each of the coefficients is a rational function in F. They play an important role in the rational solutions of $L(y) = 0$ in F that come to expression in so-called regular and singular places.

111.16.1 Singular Places

Given a place (q) in the set of places of F, a putative rational solution $y \in F$ of $L(y) = 0$ has a q-adic expansion

$$y = y_\alpha \, q^\alpha + y_{\alpha+1} \, q^{\alpha+1} + \ldots$$

It has a pole at the place (q) if the valuation α is negative. After substituting this solution in $L(y)$ it becomes clear that there is only a finite number of places which can occur as poles of an arbitrary solution of $Ly = 0$. Such a place is either a pole of one of the coefficients a_i or a zero or a pole of the differential ω of $F[D]$. There exists a classification for the poles of solutions of $L(y) = 0$.

Given a place (q) and local parameter t at (q), the differential operator can be rewritten as a differential operator $\tilde{L} \in \tilde{F}[\tilde{D}]$, with $\tilde{F} \cong F$, the valuation of whose differential at (q) is 0. The place (q) is defined to be a *singular place* of L, if one of the coefficients of $\tilde{L}$ has negative valuation at (q). Places that are not singular are called *regular*.

There are two kinds of singular places of a differential operator; the *regular singular* places and the *irregular singular* places. With the notation as above, a singular place (q) of L is regular singular if the valuation of the coefficient of $(\tilde{D})^i$ in $\tilde{L}$ is at most $i - n$ for every $i \in \{0, 1, \ldots, n-1\}$. Otherwise it is irregular singular.

`IsRegularPlace(L, p)`

Returns `true` iff the place p is a regular place of the differential operator L. If p is not a regular place of L `false` is returned. This function only works for operators whose derivation is defined by a differential.

`IsRegularSingularPlace(L, p)`

Returns `true` iff the place p is a regular singular place of the differential operator L. If p is not a regular singular place of L `false` is returned. This function only works for operators whose derivation is defined by a differential.

`IsIrregularSingularPlace(L, p)`

Returns `true` iff the place p is an irregular singular place of the differential operator L. If p is not an irregular singular place of L `false` is returned. This function only works for operators whose derivation is defined by a differential.

`SetsOfSingularPlaces(L)`

Two sets are returned. The first set contains precisely all regular singular places of the differential operator L. The second set consists of all irregular singular places of L. This function only works for operators whose derivation is defined by a differential.

`IsFuchsianOperator(L)`

Returns `true` iff the differential operator L is Fuchsian (i.e. if all singular places of L are regular singular). If L is not Fuchsian, `false` is returned. Secondly, the set of all singular places of L, is returned only if L is a Fuchsian differential operator. This function only works for operators whose derivation is defined by a differential.

`IsRegularSingularOperator(L)`

Returns `true` if and only if the differential operator L is regular singular. The operator may be defined over a differential Laurent series ring, F say. In this case being regular singular means that the operator must be regular singular at $F.1$. Then also there is no second argument returned. In the case that the derivation is defined with respect to a differential, then the values from `IsFuchsianOperator(`L`)` are returned.

Example H111E56

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> H := (z^2-z)*D^2+(3*z-6)*D+1;
> IsRegularPlace(H, Zeros(z)[1]);
false
> IsRegularSingularPlace(H, Zeros(z)[1]);
true
> SetsOfSingularPlaces(H);
```

```
{ (1/z), (z - 1), (z) }
{}
> IsFuchsianOperator(H);
true { (z), (1/z), (z - 1) }
> IsFuchsianOperator(D^2-1/z^3);
false
```

Example H111E57

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> IsRegularSingularOperator(D^2 -t*D+2);
true
> IsRegularSingularOperator(D^2 +3);
true
> IsRegularSingularOperator(D^2 +3 +O(t));
true
> IsRegularSingularOperator(D^2 +3*t^(-1));
false
```

111.16.2 Indicial Polynomials

For the definition of a indicial polynomial at a place, we refer to Section 4.1 in [vdPS03].

`IndicialPolynomial(L, p)`

Returns the monic indicial polynomial of the differential operator L at the place p. This function only works for operators whose derivation is defined by a differential and whose base ring has one generator.

Example H111E58

```
> F<z> := RationalDifferentialField(Rationals());
> _<T> := PolynomialRing(Rationals());
> R<D> := DifferentialOperatorRing(F);
> H := (z^2-z)*D^2+(3*z-6)*D+1;
> IndicialPolynomial(H, Zeros(z)[1]);
T^2 + 5*T
> IndicialPolynomial(H, Zeros(z-1)[1]);
T^2 - 4*T
> IndicialPolynomial(H, Zeros(1/z)[1]);
T^2 - 2*T + 1
> Apply(H, (z-1)^4/z^5);
0
```

111.17 Rational Solutions

`RationalSolutions(L)`

Given a differential operator L, a basis of the nullspace of rational solutions of $L(y) = 0$ in F is returned as a sequence of basis elements. This function only works for operators whose derivation is defined by a differential. The algorithm that is used is described in Section 4.1 of [vdPS03].

`HasRationalSolutions(L, g)`

Given a differential operator L with coefficients in F and an element g of F, return `true` if there is an element $y \in F$ satisfying $L(y) = g$. If such a solution exists a particular solution in F and the basis of the nullspace of rational solutions in F are also returned. If there is no solution, only `false` is returned. This function only works for operators whose derivation is defined by a differential.

Example H111E59

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> H := (z^2-z)*D^2+(3*z-6)*D+1;
> RationalSolutions(H);
[ (z^4 - 4*z^3 + 6*z^2 - 4*z + 1)/z^5 ]
> L := D^2-6/z^2;
> RationalSolutions(L);
[ z^3, 1/z^2 ]
> Apply(L, z^3+1/z^2);
0
> HasRationalSolutions(L, 6/z);
true -z [ z^3, 1/z^2 ]
> L(-z);
6/z
```

111.18 Newton Polygons

The Newton polygon of a differential operator L and its Newton polynomials can be used to factorize L. A classical example of the Newton polygon uses the derivation $z \cdot d/dz$, where z is a generator of the basefield of L. This Newton polygon is known as *the* Newton polygon of L. Its definition, as used in MAGMA, is given in §3 of [vH97b] and is applicable to operators over a Laurent series ring with generator z, as well as to operators over fields for which a set of places exist. For fields in the latter category it is however not necessary to restrict to the derivation $z \cdot d/dz$ based at $z = 0$ (in other words: at the place (z)). More generally, the Newton polygon of L at the place (p) is the Newton polygon at $t = 0$ after rewriting L as a differential operator $\tilde{L}$ in a local parameter t of (p), such that derivation of $\tilde{L}$ is of the form $t \cdot d/dt$.

`NewtonPolygon(L)`

Returns the Newton Polygon of the differential operator L over a differential Laurent series ring. This means that for the computation of the Newton polygon L may have had to be rewritten as a differential operator $\tilde{L}$ over a differential Laurent series ring $C((t))$, say, such that $\tilde{L}$ has derivation $t \cdot d/dt$. The second argument returned is the operator $\tilde{L}$.

`NewtonPolygon(L, p)`

Returns the Newton polygon of the differential operator L at the place p. The derivation of L must be defined with respect to a differential and the base ring of L should have one generator. For the computation of the Newton polygon another differential operator $\tilde{L}$, say, may have had to be calculated. The differential of the derivation of $\tilde{L}$ has valuation -1 at the place p. The differential operator $\tilde{L}$ is also returned.

`NewtonPolynomial(F)`

Returns the Newton polynomial of the face F of a Newton polygon. The Newton polygon must have been created with respect to a differential operator. The Newton polynomial depends on a uniformizing element, therefore, its variable is well–defined up to scalar multiplication by a non–zero element. The definition of the Newton polynomial of a face that is used by MAGMA, is given in Section 3 of [vH97b].

`NewtonPolynomials(L)`

Returns all Newton polynomials of L with respect to the faces of its Newton polygon. The second argument returned is the corresponding slopes.

Example H111E60

```
> K := RationalDifferentialField(Rationals());
> F<z> := ChangeDerivation(K, K.1);
> Differential(F);
(1/z) d(z)
```

```
> R<D> := DifferentialOperatorRing(F);
> L := 10*z*D^2+3*D-1;
> npgon, op := NewtonPolygon(L, Zeros(z)[1]);
> npgon;
Newton Polygon of 10*z*$.1^2 + 3*$.1 - 1 over Algebraic function field
defined over Rational Field by
$.2 - 4711 at (z)
> op;
10*z*D^2 + 3*D + -1
> faces:= Faces(npgon);
> faces;
[ <0, 1, 0>, <-1, 1, -1> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
3*T - 1
> NewtonPolynomial(faces[2]);
10*T + 3
```

Example H111E61

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> L := D^2+z*D-3*z^2;
> npgon, op := NewtonPolygon(L, Zeros(1/z)[1]);
> op;
1/z^2*$.1^2 + (-z^2 + 1)/z^2*$.1 + -3*z^2
> Differential(Parent(op));
(-1/z) d(z)
> Valuation($1,Zeros(1/z)[1]);
-1
> faces:= Faces(npgon);
> faces;
[ <-2, 1, -2> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
T^2 - T - 3
```

Example H111E62

This example corresponds to Examples 3.46 and 3.49.2 from [vdPS03].

```
> S<t> := DifferentialLaurentSeriesRing(Rationals());
> R<D> := DifferentialOperatorRing(S);
> L := t*D^2+D-1;
> npgon, op := NewtonPolygon(L);
> L eq op;
true
```

```
> Faces(npgon);
[ <0, 1, 0>, <-1, 1, -1> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomials(L);
[
    T - 1,
    T + 1
]
[ 0, 1 ]
> L := D^2+(1/t^2+1/t)*D+(1/t^3-2/t^2);
> npgon, op := NewtonPolygon(L);
> L eq op;
true
> NewtonPolynomials(L);
[
    T + 1,
    T + 1
]
[ 1, 2 ]
```

111.19 Symmetric Powers

`SymmetricPower(L, m)`

Returns the m-th symmetric power of the differential operator L as an element of the parent of L. The symmetric power is monic where possible. If n denotes the order of L, then the degree of the m-th symmetric power of L is at most $\binom{n+m-1}{n-1}$. The algorithm that is used is based on algorithms given in [BMW97].

Example H111E63

```
> F<z> := RationalDifferentialField(Rationals());
> R<D> := DifferentialOperatorRing(F);
> SymmetricPower(D^2, 3);
D^4
> SymmetricPower(D^3-1, 2);
D^6 + -7*D^3 + -8
```

111.20 Differential Operators of Algebraic Functions

An algebraic function g in a differential field extension M/F satisfies a linear differential equation $L(y) = 0$ with coefficients in $F \subset M$. If the minimal polynomial of g over F is of degree n, then the order of L is at most n.

`DifferentialOperator(f)`

Given the irreducible polynomial $f(X) \in F[X]$, return the monic differential operator over F of minimal degree to which a formal root of f is a solution. The field F must be a differential field. The base ring of the created differential operator is F.

The algorithm used in this function is straightforward. If g is a root of an irreducible polynomial $f(X) \in F[X]$, where F is a differential field, then $f(g) = 0$ induces a unique derivation on g. The field $M = F(g)$ is an algebraic differential field extension of F containing all derivatives of g. If n is the degree of the polynomial f, then M/F is a field extension of degree n. This implies that there must be at least one non–trivial linear relation between $g, \delta_M(g), \ldots, \delta_M^n(g)$. The linear relation between these elements involving the lowest powers of δ_M^i gives exactly the desired monic differential operator after a suitable normalization.

Example H111E64

```
> F<z> := RationalDifferentialField(Rationals());
> _<X> := PolynomialRing(F);
> f := X^3-z;
> L := DifferentialOperator(f);
> L;
$.1 + -1/3/z
> M<alpha> := ext<F|f>;
> R<D> := DifferentialOperatorRing(M);
> Apply(R!L,alpha);
0
```

111.21 Factorisation of Operators over Differential Laurent Series Rings

When factoring a non-trivial linear differential operator in

$$P[\delta] := k((t))[\delta],$$

with constant field k, differential Laurent Series ring $k((t))$ in t, and derivation δ, one at least wants to compute two operators L and R in P such that $f = L \cdot R$. It is important to distinguish the left and right hand operators (L and R) as such as multiplication generally is non-commutative. When considering differential operators in the operator ring P, it is common to consider δ to be $t \cdot d/dt$. This specific form has the advantage that the degree

of $\delta \cdot t$ in t remains one, since $\delta \cdot t = t \cdot \delta + t$. This specific derivation is called the *projective derivation* and we consider this derivation in the rest of this section. In this sense powers of t are eigenvectors under the application of the derivation.

A possible approach for factoring a linear operator would be to compute all non-constant irreducible right hand factors and then use recursion on the appropriate left hand factors. However, this causes a problem as there may be infinitely many factorisations. For instance $\delta^2-\delta$ has infinitely many factorisations (parametrisations) $(\delta-c/(t+c))(\delta-t/(t+c))$ for any $c \in P^1(k)$. The approach we take to find right hand factors follows [vH97b], and chooses a canonical representative from a class of right hand factors. The obtained representatives can be used in the factorisation of linear differential operators over a rational function field, see [vH97a].

A non-trivial linear differential operator f in P acts on the solution space V of all differential operators in the differential closure of $k((t))$. This action is $\bar{k}$-linear and surjective on V. The kernel $V(f)$ of this map has dimension equal to the order of the operator. The general solution space V can be split into a direct sum of smaller $\bar{k}$-vector spaces V_e. These are minimal such that $f(V_e) \subset V_e$ for every operator in P. Its kernel $V_e(f)$ consists of all solutions of $f(y) = 0$ in V_e. As a consequence the solution space of the operator then is $V(f) = \oplus_e V_e(f)$. For each of the e with non-trivial solution space of f, the idea now is to find an irreducible right hand factor of f in $k((t))[\delta]$ that annihilates $V_e(f)$.

111.21.1 Slope Valuation of an Operator

The Newton Polynomial of an operator f and its slopes contain useful information regarding its factorisation possibilities. It was proved by Malgrange that an operator over a Laurent series ring is reducible if its Newton polygon has at least two slopes. When on the other hand the Newton polygon has one positive slope, and the accompanying Newton polynomial has two relatively prime factors, then the operator is irreducible as well. It can be shown that an irreducible right hand factor has an irreducible Newton polynomial that divides the Newton polynomial of f. When an appropriate irreducible factor of the polynomial of f is taken, One can start building a right hand factor operator with coefficients of a certain precision that has exactly the irreducible factor as its Newton polynomial. Subsequently one can try to lift the coefficients to a better, possibly predescribed precision.

Various measures for the precision are possible, for instance the absolute precision of the coefficients is common. Another valuation metric is or one related to the slope of the Newton polynomial. It is defined as follows. Assume that $P := k((t))[\delta]$ has projective derivation $t \cdot d/dt$, Let s be a rational with numerator n and denominator d such that $\gcd(n, d) = 1$ and $d > 0$ hold. Then the *slope valuation* of a monomial $ct^i\delta^j$, $c \in k\backslash\{0\}$, with respect to s is defined as $V_s(ct^i\delta^j) := id - jn$. The *slope valuation* of the operator $L \in P$ with respect to s subsequently is defined as the minimum of the slope valuation of each non-zero monomial in L w.r.t s. Commonly s is a slope of the Newton polynomial of L. If the operator is the zero operator, the slope valuation is defined to be infinite.

```
SlopeValuation(L,s)
```

Returns the valuation of L, with respect to the rational slope s, when the derivation of L is projective.

Example H111E65

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> P<D>:=DifferentialOperatorRing(S);
> L:=t^(-2)*D^3+t^7;
> SlopeValuation(L,0);
-2
> SlopeValuation(L,1/2);
-7
> SlopeValuation(L,5);
-17
> L:=(0+O(t^6))*D;
> SlopeValuation(L,0);
Infinity
> Valuation(0+O(t^6));
6
> SlopeValuation(P!0,3);
Infinity
```

111.21.2 Coprime Index 1 and LCLM Factorisation

Coprime index 1 factorisation and LCLM factorisation are two factorisation methods that use similar factorisation machinery, but may result in different factorisations, see [vH97b]. Given an operator $f \in k((t))[\delta]$, both compute a right hand factor with respect to each distinct irreducible factor of the Newton polynomial of f. A local lifting procedure with respect to the slope valuation metric is performed to obtain the coefficients of the right hand factors up to a certain accuracy. In addition, no intermediate differential field extensions of $k((t))$ are used.

The coprime index 1 algorithm does not factor an operator f that has Newton polynomial of the form p^n, $n \geq 2$, with slope $s > 0$. The sum of the degrees of the obtained right hand factors of f may be less than the degree of f itself. Their least common left divisor M divides f on the right (i.e. $f = N * M$) with a kernel of dimension less or equal to $\deg(f)$. The LCLM algorithm, on the other hand, produces a set of right hand factors of a monic operator f whose LCLM is exactly f up to some precision.

`Factorisation(L)`

`Factorization(L)`

`Precision`	RngIntElt	*Default :* -1
`Algorithm`	MonStgElt	*Default :* "*Default*"

Returns a sequence M of operator sequences $[A, B]$ such that $L = A \cdot B$ holds, and B does not have a non-trivial coprime index 1 or LCLM factorisation. As the algorithm

sometimes cannot conclude if a right hand factor is irreducible, a second sequence entry $N[i]$ states `True` if the right hand factor $M[i][2]$ is undisputedly irreducible. The optional argument for the precision is the accuracy up to which the lifting procedure would be performed. The default accuracy is the relative precision of the basering of L. The algorithms used can be eiher "LCLM" or "CoprimeIndexOne". The algorithms used are based on various algorithms in [vH97b].

Example H111E66

The operator $t^2 \cdot d/dt^2 - t \cdot d/dt$ with coefficients in Q has infinitely many factorisations, since it can be written as $(t \cdot d/dt - c/(t+c)) \cdot (t \cdot d/dt - t/(t+c))$ for any rational number c. The factorisation code chooses a canonical right hand factor.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L := D^2 -D;
> factsL,blsL:=Factorisation(L:Algorithm:="LCLM");
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
The number of slopes of the Newton polynomial: 1
> (#factsL eq #blsL) and (#factsL eq 1);
true
> blsL;
[ false ]
> factsL[1];
[
    1,
    D^2 + -1*D
]
> factsL,blsL:=Factorization(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> (#factsL eq #blsL) and (#factsL eq 1);
true
> blsL;
[ true ]
> factsL[1];
[
    D,
    D + -1
]
```

Example H111E67

This example corresponds to Example 3.46 in [vdPS03].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
```

```
> L:=D*(D+1/t);
> L;
D^2 + t^-1*D + -t^-1
> factsL,blsL:=Factorisation(L:Precision:=4);
Performing coprime index 1 LCLM factorisation.
> blsL;
[ true, true ]
> #factsL eq 2;
true
> factsL[1];
[
    D + t^-1 + 1 - t + 3*t^2 + O(t^3),
    D + -1 + t - 3*t^2 + 13*t^3 + O(t^4)
]
> factsL[2];
[
    D,
    D + t^-1
]
> factsL:=Factorisation(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> #factsL eq 2;
true
> blsL;
[ true, true ]
> [v[1]*v[2]:v in factsL];
[
    D^2 + (t^-1 + O(t^19))*D + -t^-1 + O(t^19),
    D^2 + t^-1*D + -t^-1
]
```

Example H111E68

This example corresponds to Example 3.49 in [vdPS03].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L:=(D+1/t)*(D+1/t^2);
> L;
D^2 + (t^-2 + t^-1)*D + t^-3 - 2*t^-2
> factsL, blsL:=Factorisation(L:Algorithm:="CoprimeIndexOne",Precision:=6);
Performing coprime index 1 factorisation.
> blsL;
[ true, true ]
> #factsL;
2
> factsL[1];
```

```
[
    D + t^-2 + 2 + t - 3*t^2 - 8*t^3 + O(t^4),
    D + t^-1 - 2 - t + 3*t^2 + 8*t^3 - 9*t^4 + O(t^5)
]
> factsL[2];
[
    D + t^-1,
    D + t^-2
]
> [v[1]*v[2] :v in factsL];
[
    D^2 + (t^-2 + t^-1 + O(t^4))*D + t^-3 - 2*t^-2 + O(t^3),
    D^2 + (t^-2 + t^-1)*D + t^-3 - 2*t^-2
]
```

Example H111E69

This example shows that not all operators may be factored by the factorisation routine. Notice that the Newton polynomial of the operator is a square of a polynomial.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L:=(D+2/t)^2;
> np:=NewtonPolygon(L);
> faces:=Faces(np);
> #faces eq 1;
true
> NewtonPolynomial(faces[1]);
$.1^2 + 4*$.1 + 4
> factsL_lclm,blsL_lclm:=Factorisation(L);
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> factsL_lclm;
[
    [
        1,
        D^2 + 4*t^-1*D + 4*t^-2 - 2*t^-1
    ]
]
> blsL_lclm;
[ false ]
> factsL_c1,blsL_c1:=Factorisation(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> factsL_c1 eq factsL_lclm;
true
> blsL_c1 eq blsL_lclm;
```

```
true
```

Example H111E70

This example shows that one may not retrieve the factorisation as expected.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> R<D>:=DifferentialOperatorRing(S);
> L := (D+1/(t+1))*D;
> factsL_c1,blsL_c1:=Factorisation(L:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> (#factsL_c1 eq 1) and (#blsL_c1 eq 1);
true
> factsL_c1[1][2];
D + O(t^20)
> factsL_lclm, blsL_lclm:=Factorisation(L:Algorithm:="LCLM");
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
The number of slopes of the Newton polynomial: 1
> (#factsL_lclm eq 1) and (#blsL_lclm eq 1);
true
> factsL_c1[1][2], blsL_lclm[1];
D + O(t^20)
false
> M:=(D+1/(t-1))*D;
> factsM:=Factorisation(M:Algorithm:="CoprimeIndexOne");
Ring precision as default precision taken.
Performing coprime index 1 factorisation.
> # factsM eq 1;
> factsM[1][2]+O(t^4);
D + -1 - 1/2*t - 5/12*t^2 - 3/8*t^3 + O(t^4)
```

Example H111E71

One may wish to adjust the default precision to retrieve enough terms in the coefficients in the factors. This example shows the effect on a rational slope 1/5 on the number of terms in the series coefficients of the factors obtained.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals():Precision:=15);
> R<D>:=DifferentialOperatorRing(S);
> L:=(D^5+t^(-1))*D;
> np:=NewtonPolygon(L);
> Slopes(np);
[ 0 , 1/5 ]
> factsL,blsL:=Factorisation(L:Algorithm:="LCLM");
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
```

```
> blsL;
[ true, true ]
> [v[2]:v in factsL];
[
    D,
    D^5 + (-1 - t - 63*t^2 + O(t^3))*D^4 + (1 + 3*t + 253*t^2 +
    O(t^3))*D^3 + (-1 - 7*t - 825*t^2 + O(t^3))*D^2 + (1 + 15*t + 2545*t^2
    + O(t^3))*D + t^-1 - 1 - 31*t - 7713*t^2 + O(t^3)
]
> [v[1]*v[2]:v in factsL];
[
    D^6 + t^-1*D,
    D^6 + O(t^3)*D^5 + O(t^3)*D^4 + O(t^3)*D^3 + O(t^3)*D^2 + (t^-1 +
    O(t^3))*D + O(t^2)
]
> factsL:=Factorisation(L:Algorithm:="LCLM",Precision:=75);
Performing coprime index 1 LCLM factorisation.
> [v[1]*v[2]:v in factsL];
[
    D^6 + t^-1*D,
    D^6 + O(t^15)*D^5 + O(t^15)*D^4 + O(t^15)*D^3 + O(t^15)*D^2 + (t^-1 +
    O(t^15))*D + O(t^14)
]
```

111.21.3 Right Hand Factors of Operators

While resorting to field extensions can result in more complex and time consuming computations, they can be used for obtaining irreducible right hand factors over the original base field.

An effective algorithm to obtain a monic irreducible right hand factor of degree one $\tilde{L}$ of $L \in k((t))[\delta]$ in some field extension $\tilde{k}((\tilde{t}))[\tilde{\delta}]$ is presented in [vH97b, §5.1]. Such a factor $\tilde{L} = \tilde{\delta} - r(\tilde{t})$ of L is called a *Ricatti factor* of L. The operator $\mathrm{LCLM}(\tilde{\delta} - \sigma_1(r), \tilde{\delta} - \sigma_2(r), \ldots, \tilde{\delta} - \sigma_m(r))$ where the σ_i are the Galois group elements of $\tilde{k}((\tilde{t}))/k((t))$, then is a monic and irreducible operator invariant under the Galois action. Hence, it naturally reduces to a monic irreducible right hand factor of L.

Other right hand factors of L, that may be defined over a finite field extensions of $k((t))$ are the so-called *semi-regular* parts of L. Such an operator $R_e(L)$ is the monic right hand semi-regular factor of the translation $S_e(L)$ of L by e. Its degree is equal to the dimension of a non-trivial $\bar{k}$-linear vector space $V_e(L)$, and acts as zero on it. In other words $S_{-e}(R_e)$ is a monic right hand factor of L, possibly defined over a field extension of $k((t))$. It can be shown that L is the least common left multiple of all such right hand factors.

The routine `RightHandFactors` returns right hand factors of a given operator possibly by using temporary field extensions. Each of these are canonical representatives of all right hand factors belonging the slopes of the Newton polynomial of the operator. Currently

one of the internal routines may fail when performing calculations for some specific right hand factor. In this case we cannot conclude it to be irreducible.

`RightHandFactors(L)`

`Precision` RngIntElt *Default :* -1

The canonical list of monic right hand factors of L, one per slope of the Newton polynomial of L. The i^{th} entry in the second sequence returned is true if the i^{th} right hand factor is undisputedly irreducible. The precision attribute relates to the absolute precision a coefficient in a right hand factor should minimally have.

Example H111E72

This example is Example 3.49 from [vdPS03]. The same right hand factors as in Example H111E68 are obtained.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^2+(1/t^2+1/t)*DS +(1/t^3-2/t^2);
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
> #rhf eq 2;
true
> bl;
[ true, true ]
> (Parent(rhf[1]) eq RS) and (Parent(rhf[2]) eq RS);
true
> lhf,rem := EuclideanRightDivision(L, rhf[1]);
> rem;
O(t^20)
> lhf*rhf[1];
DS^2 + (t^-2 + t^-1 + O(t^22))*DS + t^-3 - 2*t^-2 + O(t^20)
> EuclideanRightDivision(L, rhf[2]);
DS + t^-1
0
```

Example H111E73

This example corresponds to Example 3.52 in [vdPS03]. The answer in the book is erroneous.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^2-3/2*DS+(2*t-1)/(4*t);
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
> bl;
[ true ]
> #rhf eq 1;
true
```

```
> rhf[1] eq L;
true
```

Example H111E74

The operator in this example is the same as in Example H111E69 where the routine `Factorisation` was used. The operator did not factor there, but does when using `RightHandFactors`.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=(DS+2/t)^2;
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Computing a first order Ricatti factor.
Performing LCLM calculation on the Ricatti factor.
> rhf;
[
    DS + 2*t^-1
]
> bl;
[ true ]
```

Example H111E75

This example corresponds to Example 3.53 in [vdPS03].

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^2 +(4+2*t-t^2-3*t^3)/(t^2)*DS+ (4+4*t-5*t^2-8*t^3-3*t^4+2*t^6)/(t^4);
> np:=NewtonPolygon(L);
> faces:=Faces(np);
> #faces eq 1;
true
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
T^2 + 4*T + 4
> factsL, blsL := Factorisation(L);
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> blsL;
[ false ]
> (#factsL eq 1) and (factsL[1][2] eq L);
true
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
```

```
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Computing a first order Ricatti factor.
Performing LCLM calculation on the Ricatti factor.
> bl;
true
> Degree(rhf[1]);
1
> bl;
true
> Parent (rhf[1]) eq RS;
true
> L - EuclideanRightDivision(L, rhf[1])*rhf[1];
O(t^22)*DS + O(t^20)
```

Example H111E76

A collection of operators having the same Newton polynomial $(T^2+1)(T-1)(T+1)$.

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L := DS^4-1/t^4;
> faces:=Faces(NewtonPolygon(L));
> faces;
[ <-1, 1, -4> ]
> _<T> := PolynomialRing(Rationals());
> NewtonPolynomial(faces[1]);
T^4 - 1
> rhf, bl := RightHandFactors(L);
Performing coprime index 1 LCLM factorisation.
> bl;
[ true, true, true ]
> [Degree(v) : v in rhf];
[ 1, 1, 2 ]
>  L - EuclideanRightDivision(L, rhf[1])*rhf[1];
O(t^23)*DS^3 + O(t^22)*DS^2 + O(t^21)*DS + O(t^20)
>  L - EuclideanRightDivision(L, rhf[2])*rhf[2];
O(t^23)*DS^3 + O(t^22)*DS^2 + O(t^21)*DS + O(t^20)
>  L - EuclideanRightDivision(L, rhf[3])*rhf[3];
O(t^23)*DS^3 + O(t^22)*DS^2 + O(t^21)*DS + O(t^20)
> M := DS^4-1;
> faces:=Faces(NewtonPolygon(M));
> faces;
[ <0, 1, 0> ]
> NewtonPolynomial(faces[1]);
T^4 - 1
```

```
> rhf, bl := RightHandFactors(M);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing LCLM calculation on a semi-regular part.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Computing a first order Ricatti factor.
Performing LCLM calculation on the Ricatti factor.
> rhf;
[
    DS^2 + 1,
    DS + -1
]
> bl;
[ true, true ]
```

Example H111E77

This is the main example of [vH97b]

```
> S<t>:=DifferentialLaurentSeriesRing(Rationals());
> RS<DS> := DifferentialOperatorRing(S);
> L:=DS^9 + 2*t^-1*DS^8 + 3*t^-2*DS^7 + 2*t^-3*DS^6 + (t^-4 + 2*t^-2)*DS^5 +
> (-3*t^-5 + 5*t^-4)*DS^3 + 3*t^-5*DS^2 + (2*t^-6 + 2*t^-5)*DS + 7*t^-5;
> facts := Factorisation(L);
Ring precision as default precision taken.
Performing coprime index 1 LCLM factorisation.
> [Degree(v[2]): v in facts];
[ 1 2 2 4 ]
> isweaklyzero := [];
> vals :=[];
> for i in [1..4] do
>   _,rem := EuclideanRightDivision(L, facts[i][2]);
>   isweaklyzero[i] := IsWeaklyZero(rem);
>   vals[i] := [Valuation(v) : v in Eltseq(rem)];
> end for;
> isweaklyzero;
[ true, true, true, true ]
> [Minimum(v) : v in vals];
[ 14 , 4 , 4 , 11]
> rhf, bl := RightHandFactors(L:Precision:=30);
Performing coprime index 1 LCLM factorisation.
Calculating semi-regular parts.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Performing coprime index 1 LCLM factorisation.
Performing LCLM calculation on a semi-regular part.
```

```
Computation of the LCLM failed.
> bl;
[ true, true, true, false ]
> [Degree(v): v in rhf];
[ 1 2 2 4 ]
> isweaklyzero := [];
> vals := [];
> for i in [1..4] do
>   [Degree(v): v in Eltseq(rhf[i])];
>   _,rem := EuclideanRightDivision(L, rhf[i]);
>   isweaklyzero[i] := IsWeaklyZero(rem);
>   vals[i] := [Valuation(v) : v in Eltseq(rem)];
> end for;
[ 35, 0 ]
[ 35, 35, 0 ]
[ 35, 35, 0 ]
[ 31, 32, 33, 34, 0 ]
> isweaklyzero;
[ true, true, true, true ]
> [ Minimum(v) : v in vals];
[ 30, 30, 30, 27 ]
```

111.22 Bibliography

[**BMW97**] Manuel Bronstein, Thom Mulders, and Jacques-Arthur Weil. On symmetric powers of differential operators. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (Kihei, HI)*, pages 156–163 (electronic), New York, 1997. ACM.

[**vdPS03**] Marius van der Put and Michael F. Singer. *Galois theory of linear differential equations*, volume 328 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, Berlin, 2003.

[**vH97a**] Mark van Hoeij. Factorization of differential operators with rational functions coefficients. *J. Symbolic Comput.*, 24(5):537–561, 1997.

[**vH97b**] Mark van Hoeij. Formal solutions and factorization of differential operators with power series coefficients. *J. Symbolic Comput.*, 24(1):1–30, 1997.

PART XV
ALGEBRAIC GEOMETRY

112 SCHEMES

Chapter 112
SCHEMES

112.1 Introduction and First Examples

Schemes are rather general objects of algebraic geometry. A standard reference is Hartshorne's introductory text [Har77]. Included among all schemes are many familiar geometric objects such as plane curves. In Magma, one can work with many of these familiar objects but not with entirely general schemes. Roughly speaking, a scheme in MAGMA is any geometric object defined by the vanishing of polynomial equations in affine or projective space. In particular, there is no facility for defining a scheme *a priori* in terms of a collection of affine patches. Schemes are not automatically normalized. Maps between schemes can be defined by polynomials or quotients of polynomials.

The sections in this introduction contain examples covering the basic idioms understood by MAGMA, especially those for creation of geometric objects, and are intended to be the first place of reference for newcomers to this module.

The design of the general scheme module is not particularly subtle or difficult but the philosophy behind it does require a small amount of understanding. There are two things in particular. Firstly, while constructing geometric objects is easy, many of the constructors take an ambient scheme as an argument as well as some polynomials. Thus one's initial step is often to create some large ambient space which is not of primary interest but in which many schemes will lie. In doing so, one usually assigns names to the coordinate functions of this space, and it is these names which are used when writing the polynomials which define some scheme. Secondly, points are not considered to be elements of schemes, but rather elements of one of a series of *point sets* of schemes indexed by the rings containing the coefficients. (Mathematically speaking, these rings are really algebras over the base ring of the scheme, but in MAGMA the algebra structure is usually implicitly determined by coercion.)

The objects that can be created include

- ambient spaces: affine space, projective space, rational scrolls or weighted projective space over a ring
- schemes as subschemes: the zero locus of polynomials defined on a particular ambient space or on any other scheme
- points of schemes: sequences of ring elements, possibly defined over some extension of the base ring
- maps: sequences of polynomials or rational functions defined on the domain
- linear systems: linear spaces of polynomials defined on ambient spaces

Schemes may be defined quite generally over any ring k, although of course many functions require k to lie in some restricted class. The following restrictions hold: Gröbner basis

calculations may only be performed over an exact field or Euclidean domain; resultant calculations may only be carried out over a unique factorization domain; GCD calculations work over any exact field and the integers and over polynomial rings over either of these; the factorisation of polynomials is possible only over the integers, rationals, finite fields, algebraically closed fields, number fields and function fields. Linear systems are based on the linear algebra module in MAGMA and so are restricted to ambient spaces defined over fields.

The functions described in this chapter are general ones that apply to all schemes. In particular situations additional functions are provided. For example, see Chapter 114 for many specialised functions that apply in the case of curves. In the final subsection of this introduction we say a few words about the various different types of schemes that MAGMA admits.

To some extent we try to emulate Hartshorne's text [Har77] although we remain only a fraction of the way through the material of that book and, of course, we have made innumerable compromises.

In the examples below, > at the start of a line is the MAGMA prompt. It is followed by input which may be typed into a MAGMA session. The remaining lines are output which has been edited slightly in some circumstances, but should nonetheless match closely what appears on screen.

112.1.1 Ambient Spaces

Most schemes are considered to live in an *ambient space*. These include affine and projective spaces, rational scrolls and products. They may be defined over any base ring. The main technical point is that they have an associated coordinate ring (or a homogeneous coordinate ring) that is a polynomial ring, possibly with one or more gradings associated to it.

The syntax for defining an affine space over a ring is similar to that employed when defining a polynomial ring over another ring. The angled bracket notation is used for assigning names to the coordinate functions. It is optional as in the case of polynomial rings. We illustrate by creating an affine 3-space over the finite field of 23 elements with coordinates x, y, z.

Example H112E1

```
> k := FiniteField(23);
> A<x,y,z> := AffineSpace(k,3);
> A;
Affine Space of dimension 3
Variables : x, y, z
```

Various attributes of A are cached and may be subsequently retrieved.

```
> BaseRing(A);
Finite field of size 23
> Dimension(A);
3
```

```
> A.1;
x
> CoordinateRing(A);
Polynomial ring of rank 3 over GF(23)
Lexicographical Order
Variables: x, y, z
```

Projective spaces are normally defined in the same way, but they can also be defined with weights.

```
> P<u,v,w> := ProjectiveSpace(k,[1,2,3]);
> P;
Projective Space of dimension 2
Variables : u, v, w
Gradings :
1       2       3
```

Having defined an ambient space A, polynomials in its coordinate functions can be created. These polynomials are elements of the coordinate ring of A. These polynomials (and sometimes also quotients of them) can be used to define geometric objects related to A. Elements of other polynomial rings have no meaning on A and their use will result in an error.

112.1.2 Schemes

Subschemes of ambient spaces prescribed by the vanishing of finitely many polynomials may be defined. Just as a polynomial ideal in MAGMA belongs to some polynomial ring, so any scheme defined by polynomials is contained in some ambient space. In the case of ideals of polynomial rings the function `Generic()` recovers the polynomial overring. For schemes the analogous function is `AmbientSpace()` (or `Ambient`) which recovers the ambient space.

At the time a scheme is created the system only checks that the defining equations make sense — that they are defined on the nominated ambient space, and are homogeneous if necessary — and does not check other properties such as whether or not the scheme is empty.

In this example, the twisted cubic curve in projective 3-space is defined in terms of equations. The constructor takes these equations in a sequence as one of its arguments.

Example H112E2

```
> k := Rationals();
> P<u,v,w,t> := ProjectiveSpace(k,3);
> M := Matrix(CoordinateRing(P),2,3,[u,v,w,v,w,t]);
> eqns := Minors(M,2);
> C := Scheme(P,eqns);
> C;
Scheme over Rational Field defined by
u*w - v^2
```

```
-u*t + v*w
v*t - w^2
> AmbientSpace(C);
Projective Space of dimension 3
Variables : u, v, w, t
> Dimension(C);
1
> IsNonsingular(C);
true
```

In fact, it is possible to create schemes without reference to an ambient space. For instance, the intrinsic `Spec` may be applied to an affine algebra. But even so, the ambient space defined by the polynomial overring of that algebra is created in the background and may be recovered using the intrinsic `Ambient`. Many constructors require a reference to some overscheme to be clear about *which* scheme the new object is meant to live in.

We note here that there is an important difference between affine schemes and projective ones.

For affine schemes, the defining ideal (generated by the defining equations) is *unique*. That is, there is a $1-1$ correspondence between subschemes of an affine ambient space and the ideals of the coordinate ring of that ambient.

On the other hand, for projective ambients, a given subscheme is defined by multiple homogeneous ideals of the ambient coordinate ring. In this case however, there is a *largest* defining ideal for each subscheme, which we refer to as the *saturated* one.

The practical effect of this is that MAGMA may have to replace the original defining ideal of a projective scheme with the saturated ideal to guarantee the correct result of certain functions (see section 112.3 on page 3494 for more details).

112.1.3 Rational Points

Although closed points of schemes may be defined as schemes in terms of polynomials, there is a far more convenient way to define them: simply coerce the coordinates of the point into the scheme. This is to allow points to be used in a mathematically colloquial way: one understands the statement "the point p lies on the curve C" to mean that the coordinates of p satisfy the equation of C, rather than to mean an inclusion between the two defining ideals. (It also avoids the ideal inclusion test which would be a much more expensive calculation.)

The points $p = (-1, 1)$, $q = (1, 2)$ are created in the affine plane over the finite field of 31 elements. MAGMA's coercion operator, the `!` symbol, provides a concise notation for specifying natural reinterpretations of objects. In this case a sequence of integers is reinterpreted as a point of the finite plane. One of the points created below lies on the standard parabola C.

Example H112E3

```
> k := GF(31);
> A<x,y> := AffineSpace(k,2);
> p := A ! [-1,1];
> q := A ! [1,2];
> p,q;
(30, 1) (1, 2)
> C := Scheme(A,y-x^2);
> p in C;
true (30,1)
> q in C;
false
> [-1,1] in C;
true (30, 1)
```

But this is only the beginning of the story. Objects in MAGMA are always considered to lie in some set or structure called their *parent*. Although it would be natural to take the scheme as the parent of points, instead points have a *point set* as their parent. Point sets are the MAGMA equivalent of "L-valued points" of schemes. If k is the base ring of a scheme X and L is some k-algebra, then the point set of X over L, denoted $X(L)$, comprises points with coordinates in L. In the previous example, the sequences of coordinates were defined over the base ring k and the coercion created elements in the point set $A(k)$. Predicates such as `p in C` were evaluated by testing whether the coordinates of p satisfied the equations of the scheme rather than by consulting their parents. If the point does happen to lie on C, then p is returned as a point of C as second return value. Note the difference between the apparently identical points $(30, 1)$: the first, (p), lies in a point set of A while the second lies in a point set of C. The same effect was achieved using the sequence alone in the last line.
Point sets thus allow one to define points over extensions of k without having to define a new scheme over that extension. In the next fragment, we show how to make a point of C over an extension.

```
> k1<w> := ext< k | 2 >;
> C(k1) ! [w^16,3];
(w^16, 3)
```

An error is signalled if a point set is not nominated.

```
>  C ! [w^16,3];
>>   C ! [w^16,3];
       ^
Runtime error in '!': Illegal coercion
LHS: Sch
RHS: [FldFinElt]
```

The moral is:

A point of a scheme is created by coercing a sequence of coordinates into a point set of the scheme.

While it is true that the "scheme ! coordinates" operation applies when the sequence is defined over the base ring, it might best be thought of as a convenient shorthand in predictable and simple situations. This is analogous to the situation when constructing sets or sequences.

```
> C(k1) ! [w,w^2];
(w, w^2)
>  C(k) ! [w,w^2];
>> C ! [w,w^2];
     ^
Runtime error in '!': Illegal coercion
```

This coercion fails since the coordinates do not belong to a field that embeds into k.

112.1.4 Projective Closure

Affine schemes have projective closures and projective schemes have standard affine patches. For example, the projective plane has three standard affine patches, each of which may be recovered as illustrated in the following example. Here we compute the third affine patch, that is the patch whose points have nonzero first coefficient (in the projective space).

Example H112E4

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> A := AffinePatch(P,3);
> A;
Affine Space of dimension 2
Variables : $.1, $.2
> A<u,v> := A;
> A;
Affine Space of dimension 2
Variables : u, v
> P eq Codomain(ProjectiveClosureMap(A));
true
```

Note that variable names on the patch, and also on closures, are not created automatically. Also, the relationship between patch and closure is cemented by a map.
The projective closures of all schemes contained in a single affine space will lie in a common projective space. Moreover, the closures of schemes lying in distinct affine patches of a single projective space will lie in that same space. In particular, the projective closures of different patches of a projective scheme will be identical.

```
> X := Scheme(A,[u^2-v^3,u^2+v^3]);
> PX := ProjectiveClosure(X);
> PX;
Scheme over Rational Field defined by
z^3
y^2
```

```
> AffinePatch(PX,3) eq X;                    // (1)
true
> aX1:=AffinePatch(PX,1);
> IsEmpty(aX1);                              // (2)
true
> ProjectiveClosure(aX1) eq PX;              // (2)
true
> Y := Scheme(P,[x*y^2-z^3,x*y^2+z^3]);
> AffinePatch(Y,3);
Scheme over Rational Field defined by
u^2 - v^3
u^2 + v^3
> AffinePatch(Y,3) eq X;                     // (3)
true
> Y eq PX;
false
```

This example shows several things. First it shows that taking the appropriate affine patch of a projective closure returns the same scheme again (1). Second, it shows that the projective closure of an affine patch always returns the exactly the same scheme and not a newly created version. This is not always mathematically correct and means that projective closure is dependent on how a scheme was created. Finally, it shows that projective schemes are "saturated at infinity", thereby removing the unnecessary x factor (3).
Points are handled cleanly with respect to projective closure.

```
> p := A ! [0,0];
> PX ! p;
(1 : 0 : 0)
```

Although the coercion of points is very flexible — the affine point p can usually be used in place of the projective point `PX ! p` even when working in PX — code which is explicit about this kind of coercion is probably more clear.

112.1.5 Maps

Maps between schemes are defined in terms of either polynomials or rational polynomials. When a function field exists for the domain, function field elements may also be used. When the domain or codomain is projective, that is, has at least one grading, then compatibility of the defining functions with the gradings will be checked.

Maps to projective spaces are normalised by clearing polynomial denominators.

Example H112E5

```
> k := Rationals();
> P1<s,t> := ProjectiveSpace(k,1);
> P2<x,y,z> := ProjectiveSpace(k,2);
> f := map< P1 -> P2 | [s/t,s^2/(s^2 - t^2),t/s] >;
> f;
```

```
Mapping from: Prj: P1 to Prj: P2
with equations :
s^4 - s^2*t^2
s^3*t
s^2*t^2 - t^4
> IsRegular(f);
true
> Image(f);
Scheme over Rational Field defined by
-x^3*z + x^2*y^2 - 2*x*y^2*z + y^2*z^2
```

The images of points may be computed in the natural way:

```
> p := P1 ! [3,2];
> f(p);
(9/4 : 27/10 : 1)
> f(p) in Image(f);
true (9/4 : 27/10 : 1)
```

And schemes may be pulled back by maps.

```
> S := Scheme(P2,x^2 - y*z);
> Z := Pullback(f,S);
> Z;
Scheme over Rational Field defined by
s^8 - 2*s^6*t^2 - s^5*t^3 + s^4*t^4 + s^3*t^5
> RationalPoints(Z);
{@ (1 : 1), (-1 : 1), (0 : 1) @}
> P := PointsOverSplittingField(Z); P;
{@ (0 : 1), (-1 : 1), (1 : 1), (r1 : 1), (r2 : 1), (r3 : 1) @}
> Ring(Universe($1));
Algebraically closed field with 3 variables
Defining relations:
[
    r3^3 - r3 - 1,
    r2^3 - r2 - 1,
    r1^3 - r1 - 1
]
```

This code computes the intersection of the scheme which is image of f with the conic S (although note that S is defined only as a scheme here and not as a conic). Moving to an algebraic closure one sees a Galois 3-cycle of points among the rational points. Additional functions for analysis of the multiplicities of these intersections are described in Chapter 114.

112.1.6 Linear Systems

Linear systems in projective space are simply collections of hypersurfaces having a common degree which are parametrised linearly by a vector space. The set of all conics in the plane is an example, being parametrised by the 6-dimensional vector space of possible coefficients of a general conic:

$$(a, b, c, d, e, f) \longleftrightarrow (ax^2 + bxy + cy^2 + dxz + eyz + fz^2 = 0).$$

A linear system in MAGMA has associated with it a fixed basis of forms, called its *sections*, of the given degree which allows vectors of coefficients to be interpreted as hypersurfaces. In the example above, the basis is the set of monomials of degree 2 and the bijection between vectors of coefficients and hypersurfaces is explicit. Linear systems are closely related to maps since their sections may be used to define a map from the space on which they are defined to some other projective space.

Let P be a projective plane over some field k with coordinates x, y, z. The linear system of conics on P is created.

Example H112E6

```
> Q := RationalField();
> P<x,y,z> := ProjectiveSpace(Q,2);
> L := LinearSystem(P,2);
> L;
Linear system on Projective Space of dimension 2
Variables : x, y, z
with 6 sections: x^2 x*y x*z y^2 y*z z^2
> Sections(L);
[ x^2, x*y, x*z, y^2, y*z, z^2 ]
```

These sections are now used to make a map from the plane P to 5-space. Its image is the *Veronese surface*.

```
> P5<u0,u1,u2,u3,u4,u5> := ProjectiveSpace(Q,5);
> phi := map< P -> P5 | Sections(L) >;
> Image(phi);
Scheme over Rational Field defined by
-u0*u3 + u1^2
-u0*u4 + u1*u2
-u0*u5 + u2^2
-u1*u4 + u2*u3
-u1*u5 + u2*u4
-u3*u5 + u4^2
```

Geometrical conditions may also be imposed on a linear system. For example, given a linear system L and a point p lying in the projective space on which L is defined, a subsystem of L consisting of those hypersurfaces of L which contain p may be defined implicitly. It may be checked (choosing more convenient coordinates) that the image of this new system is a projection

of the Veronese surface from the point $f(3,2,1) = (9,6,4,3,2,1)$ lying on it. This is an embedding of the blowup of the plane P at the point $(3,2,1)$ also known as the *cubic scroll*.

```
> p := P ! [3,2,1];
> L1 := LinearSystem(L,p);
> L1;
Linear system on Projective Space of dimension 2
Variables : x, y, z
with 5 sections:
x^2 - 9*z^2
x*y - 6*z^2
x*z - 3*z^2
y^2 - 4*z^2
y*z - 2*z^2
> P4<v0,v1,v2,v3,v4> := ProjectiveSpace(Q,4);
> phi := map< P -> P4 | Sections(L1) >;
> Image(phi);
Scheme over Rational Field defined by
-v0*v3 + v1^2 - 4*v2^2 + 12*v2*v4 - 9*v4^2
-v0*v4 + v1*v2 - 2*v2^2 + 3*v2*v4
-v1*v4 + v2*v3 - 2*v2*v4 + 3*v4^2
```

It may also be checked that this image can be described as the vanishing of the two by two minors of a certain matrix.

```
> M := Matrix(3,[v0, v1 + (2*v2 - 3*v4), v2, v1 - (2*v2 - 3*v4), v3, v4]);
> Minors(M,2);
[
    v0*v4 - v1*v2 + 2*v2^2 - 3*v2*v4,
    v1*v4 - v2*v3 + 2*v2*v4 - 3*v4^2,
    v0*v3 - v1^2 + 4*v2^2 - 12*v2*v4 + 9*v4^2
]
> ideal< CoordinateRing(P4) | $1 > eq Ideal(Image(phi));
true
```

112.1.7 Aside: Types of Schemes

This section gives a general overview of the types of scheme that MAGMA admits. While it is not necessary to know this for most applications, it is a useful guide to our point of view and gives some indication of the different data structures used.

The main type is `Sch`. Very general functions such as `BaseRing()` apply at this level. Gröbner basis calculations also apply. One level below this there are

`Aff` — a type for affine spaces

`Prj` — a type for projective spaces

`Crv` — a type for curves

`Clstr` — a type for zero-dimensional schemes, or clusters

These inherit all the operations which apply at the level of `Sch`. The affine type is straightforward. The projective type contains spaces with graded polynomial coordinate rings. There are a number of subtypes of this which identify spaces with more than one grading.

Objects of the curve type `Crv` currently only include schemes defined by a single non-trivial equation in a two-dimensional ambient space. Functions which apply to the curve type are detailed in Chapter 114. The most powerful of those require certain irreducibility and separability conditions to be satisfied by the equation.

More specialised curve types are derived from `Crv`. They include `CrvCon` for plane conics especially those defined over the rationals where fast point-finding algorithms exist; `CrvRat` for rational curves for which a parametrisation algorithm exists; `CrvEll` for elliptic curves, probably the most sophisticated part of the entire geometry module; and `CrvHyp` for hyperelliptic curves where there are fast algorithms for computing on the jacobian. Each of these has a handbook chapter which presents their specialised functions.

112.2 Ambients

For the purposes of this chapter, any scheme is contained in some ambient space, either an affine space or one of a small number of standard projective spaces: these are projective space itself, possibly weighted, and rational scrolls. The basic property of these spaces is that they have some kind of coordinate ring that is a polynomial ring. It happens again and again that we lift polynomials to these polynomial rings before working with them. It is possible to define schemes without reference to such an ambient space, but one will be created in the background in any case.

Listed in this section are the basic creation methods for ambient spaces. Names for the coordinates will usually be required for creating polynomials later on. Names of coordinate functions may be defined using the diamond bracket notation in the same way as for polynomial rings. Coordinate names defined using this will be *globally* defined and retained even outside the context in which they were set. Alternatively, explicit naming functions may be used after creation.

112.2.1 Affine and Projective Spaces

These are the basic ambient spaces. They are used in many situations and are usually sufficient, although there are more in the next section.

`AffineSpace(k,n)`

Create an n-dimensional affine space over the ring k. The integer n must be positive. Names can be assigned using the angle bracket notation, e.g. `A<x, y, z> := AffineSpace(Rationals(), 3)`, which will assign the names to the coordinate ring, usually a multivariate polynomial ring, in the same way as the angle bracket notation works for the multivariate polynomial rings.

`ProjectiveSpace(k,n)`

`ProjectiveSpace(k,W)`

Create an n-dimensional projective space over the ring k. The integer n must be positive. The second argument to this intrinsic can be a sequence of positive integer weights. These weights will be assigned to the coordinate functions of the space. The dimension of the space is one less than the length of this sequence. (At present there are very few functions to perform analysis on weighted projective spaces, but maps between them are treated correctly.) Names can be assigned using the angle bracket notation, e.g. `P<x, y, z> := ProjectiveSpace(Rationals(), 2)`, which will assign the names to the coordinate ring, usually a multivariate polynomial ring, in the same way as the angle bracket notation works for multivariate polynomial rings.

`AffineSpace(R)`

`Spec(R)`

Create the affine space whose coordinate ring is the multivariate polynomial ring R. The coordinate names for the affine space will be inherited from R.

`ProjectiveSpace(R)`

`Proj(R)`

Create the projective plane whose homogeneous coordinate ring is the multivariate polynomial ring R. If R has been assigned a grading then that grading will be used otherwise it will be considered to have the standard grading by degree.

`AssignNames(~A,N)`

A procedure to change the print names of the coordinate functions of the ambient space A. It leaves A unchanged except that the visible names of the first $\#N$ coordinate functions are replaced by the strings of N and the rest return to their default. It does *not* assign the coordinate functions themselves to any identifiers. That must be done by hand, for instance by the command `x := A.1;`. Note that this will change the variable names of the coordinate (polynomial) ring.

`A . i`

`Name(A,i)`

The ith coordinate function of A as an element of the coordinate ring of A.

Example H112E7

An affine 3-space over the finite field of 11^2 elements is created. Initially only the first two coordinate functions are named. The third adopts the default name `$.3`. The identifier which refers to it is `A.3`. Then new names are assigned to all coordinate functions and new identifiers set to refer to them. (Check what happens if the identifiers are not assigned.) Notice that the previous identifiers, x, y, are not erased, although their print values have been updated.

```
> A<x,y> := AffineSpace(FiniteField(11,2),3);
> A;
Affine Space of dimension 3
Variables : x, y, $.3
> AssignNames(~A,["u","v","w"]);
> u := A.1; v := A.2; w := A.3;
> A;
Affine Space of dimension 3
Variables : u, v, w
> x;
u
> u eq x;
true
```

Print values are global in MAGMA, meaning that even if they are changed in the local environment of a function or procedure the new names will persist.

`A eq B`

Returns `true` if and only if the ambient spaces A and B are identical. This will only be the case if both A and B refer to the same instance of creation of the space.

112.2.2 Scrolls and Products

These spaces are created using multiple gradings. They are not as fundamental as the affine and projective spaces of the previous section and may be passed over on first reading.

As we have said, the important thing about ambient spaces in this system is that their coordinate rings are essentially polynomial rings. For affine spaces, this is literally true. For projective spaces, one talks about the *homogeneous* coordinate ring and restricts attention to homogeneous polynomials, that is, polynomials whose terms all have the same weight with respect to a single grading, but nonetheless one is working inside a polynomial ring.

This trick can be pushed further by admitting more than one grading on a polynomial ring. The standard example of this is the family of rational ruled surfaces, or rational surface scrolls, which have a bihomogeneous coordinate ring with four variables u, v, x, y and two gradings which are often chosen to be

$$[1, 1, -n, 0] \quad \text{and} \quad [0, 0, 1, 1]$$

for some nonnegative integer n. Now we restrict attention to polynomials of some homogeneous *bidegree* (or more generally, *multidegree*) of the form $[a, b]$. As with ordinary

projective space, polynomials of a given bidegree form a vector space with a favourite basis of monomials. For example, if $n = 3$, then monomials of bidegree $[2, 1]$ have as their basis

$$u^5x, u^4vx, \ldots, v^5x, uy, vy.$$

As with ordinary projective space, one can ask which polynomials have negative degree. The difference is that this time there is just a little room for manoeuvre: with the same n the monomials of degree $[-4, 2]$ are

$$u^2x^2, uvx^2, v^2x^2.$$

We consider the geometric object having this coordinate ring to be a primitive object of the system. It is called a ruled surface and denoted $\mathbf{F}_n$. This seems to be a rather implicit approach at first, but is very similar to the way one works with projective space itself: at first one learns about homogeneous polynomials, degree 0 rational polynomials, affine patches and so on, but once used to that one gets on and uses it.

There are affine patches on $\mathbf{F}_n$ — a standard 'first' patch is the affine plane where $v \neq 0$ and $y \neq 0$ — and one can study them to get an idea of what this surface looks like. In this case it is a $\mathbf{P}^1$ bundle over $\mathbf{P}^1$, the map to $\mathbf{P}^1$ given by the bidegree $[0, 0]$ ratio u/v and the fibre coordinate being the ratio x/y, not of trivial bidegree on the whole surface, but well-defined for any fixed value of u/v.

We think of the gradings as forming a matrix and allow more general scrolls arising from matrices which are partially row reduced to have rows of the form

$$\begin{pmatrix} 1 & \ldots & 1 & * & \ldots & * & \ldots & & & & \\ 0 & \ldots & 0 & 1 & \ldots & 1 & * & \ldots & & & \\ \vdots & & & & & & & & & & \\ 0 & \ldots & 0 & 0 & \ldots & 0 & 0 & \ldots & 0 & 1 & \ldots & 1 \end{pmatrix}$$

where the $*$ entries may be any integers. To get products of projective spaces we simply set all the entries to be 0. Thinking about which multihomogeneous polynomials arise in this ring and comparing with functions on products of projective spaces is a first exercise.

A reference for the point of view we take on these scrolls is [Rei97] Chapter 2.

`DirectProduct(A,B)`

The product of the ambient spaces A and B together with a sequence containing the two projection maps.

`RuledSurface(k,a,b)`

This returns the ruled surface defined over the ring k whose negative section has selfintersection $\pm(a - b)$, where a, b are non-negative integers.

`RuledSurface(k,n)`

If n is a nonnegative integer, this returns the ruled surface defined over the ring k whose negative section has selfintersection $-n$. The integer n must be non-negative. In terms of the gradings, this means using the standard gradings as described in the introduction with the top-right-hand entries being $-n, 0$.

`AbsoluteRationalScroll(k,N)`

If N is a sequence nonnegative integers this returns the rational scroll with base ring k and gradings with entries being $-N$.

`ProductProjectiveSpace(k,N)`

If $N = [n_1, \ldots, n_r]$ is a sequence of positive integers this returns the product of ordinary projective spaces

$$\mathbf{P}^{n_1} \times \ldots \times \mathbf{P}^{n_r}$$

of dimensions of N over the ring k. This does not create independent copies of the projective factors and in particular does not return projection maps to the factors.

`SegreProduct(Xs)`

Computes the product X of a finite sequence of schemes Xs lying in ordinary projective space. X is constructed in *ordinary* projective space. It is embedded there via an iterated Segre embedding (see Ex. 2.14, Section 2, Chapter 1 of [Har77]). This intrinsic is provided because it is often easier to work in ordinary projective space. However, the user should be warned that the dimension of the ambient increases markedly with Segre embeddings. If the r schemes $Xs[i]$ lie in $\mathbf{P}^{n_i}$, then X will lie in $\mathbf{P}^N$ where N is $(n_1 + 1) * (n_2 + 1) * \ldots (n_r + 1) - 1$.

A sequence containing the r projection maps from X to the $Xs[i]$ is also returned.

`SegreEmbedding(X)`

X should be a scheme lying in a product projective ambient (or the ambient itself!). Computes and returns the image Y of X in ordinary projective space under the iterated Segre embedding (see the preceding intrinsic) along with a scheme isomorphism from X to Y. This is a specialised intrinsic that is generally much faster than using the general machinery to construct the Segre map on the ambient of X and ask for the image of X.

Example H112E8

In the following example, we Segre embed the product of an elliptic curve E with itself into ordinary projective space using the two intrinsics.

```
> Q := RationalField();
> P2<x,y,z> := ProjectiveSpace(Q,2);
> E := Curve(P2,y^2*z-x^3-x^2*z-z^3);
> ExE := SegreProduct([E,E]);
```

```
> P8<a,b,c,d,e,f,g,h,i> := Ambient(ExE);
> ExE;
Scheme over Rational Field defined by
-f*h + e*i,
-c*h + b*i,
-f*g + d*i,
-e*g + d*h,
-c*g + a*i,
-b*g + a*h,
-c*e + b*f,
-c*d + a*f,
-b*d + a*e,
-g^3 - g^2*i + h^2*i - i^3,
-d^3 - d^2*f + e^2*f - f^3,
-c^3 - c^2*i + f^2*i - i^3,
-b^3 - b^2*h + e^2*h - h^3,
-a^3 - a^2*g + d^2*g - g^3,
-a^3 - a^2*c + b^2*c - c^3
> // or we could have started with ExE in product projective space
> P22<x,y,z,s,t,u> := ProductProjectiveSpace(Q,[2,2]);
> EE := Scheme(P22,[y^2*z-x^3-x^2*z-z^3, t^2*u-s^3-s^2*u-u^3]);
> EE;
Scheme over Rational Field defined by
-x^3 - x^2*z + y^2*z - z^3,
-s^3 - s^2*u + t^2*u - u^3
> ExE_1 := SegreEmbedding(EE);
> // transfer ExE_1 to the Ambient of ExE to compare
> ExE_1 := Scheme(P8,[Evaluate(pol,[a,b,c,d,e,f,g,h,i]) : pol in
>             DefiningPolynomials(ExE_1)]);
> ExE eq ExE_1;
true
```

112.2.3 Functions and Homogeneity on Ambient Spaces

`CoordinateRing(A)`

The coordinate ring of the ambient space A. This is some polynomial ring of appropriate rank over the base ring. Gradings on this ring are usually independent of those of the scheme. Note that if the coordinate ring has zero rank then it will be the base ring.

`FunctionField(A)`

The function field of the ambient space A. This is a field isomorphic to the field of fractions of the coordinate ring of A.

`HasFunctionField(A)`

`Gradings(X)`

A sequence containing all the gradings on the projective space X. Each such grading is a sequence of integers whose length is the same as the number of coordinate functions of X. The same sequence is returned when this function is applied to any scheme contained in X.

`NumberOfGradings(X)`

`NGrad(X)`

The number of independent gradings on the projective space X. The same number is returned when this function is applied to any scheme contained in X.

`NumberOfCoordinates(X)`

`Length(X)`

The number of coordinate functions of the ambient space of the scheme X. This is equal to the number of coordinates of any point of X.

`Lengths(X)`

The lengths of the groups of ones in the gradings of a scroll X.

`IsHomogeneous(X,f)`

Returns `true` if and only if the polynomial f is homogeneous with respect to all of the gradings on the scheme X.

`Multidegree(X,f)`

The sequence of homogeneous degrees of the polynomial f with respect to the gradings on the scheme X.

112.2.4 Prelude to Points

Points of schemes are handled in an extremely flexible way: their coordinates need not be elements of the base ring, for instance. We don't discuss the details here but simply show how to create points in ambient spaces and illustrate with an example. This is already enough for non-specialised purposes: intrinsics which take point arguments for computing, say, the tangent space to a curve at a point, can take the underlying point of the ambient space or the point of the curve equally well. Having said that, the later section on points, Section 112.7, should be taken as the definitive reference.

`A ! [a,b,...]`

`A(L) ! [a,b,...]`

For elements $a, b, \ldots$ in the base ring of the scheme A this creates the set-theoretic point $(a, b, \ldots)$ in the affine case, or $(a : b : \ldots)$ in the projective case. Over a field, the projective point will be normalised so that its final nonzero coordinate entry is 1 (and further analogous normalisations when there are at least two gradings as in the case of surface scrolls). The first constructor can only be used if the sequence contains elements of the base ring of A. The second version of the constructor is the standard one. Using it rather than the first allows the user to specify the ring in which the coefficients are to be considered. See the discussion of point sets in the section below on points.

Example H112E9

We create some points with identical coordinates. They are deemed to be unequal by the equality test, but if what you really care about is their coordinates you can check that those are equal.

```
> k<w> := FiniteField(3^2);
> A := AffineSpace(k,2);
> p := A ! [1,2];
> K := ext< k | 2 >;
> q := A(K) ! [1,2];
> m := hom< k -> k | w^3 >;  // Frobenius
> r := A(m) ! [1,2];
> p eq q;
true
>  p eq r;
>>   p eq r;
     ^
Runtime error in 'eq': Arguments are not compatible
Argument types given: Pt, Pt
>  q eq r;
>> q eq r;
     ^
Runtime error in 'eq': Arguments are not compatible
Argument types given: Pt, Pt
```

In the example above, the first method used for the creation of a point is sufficient if you only want to create a point with coefficients in the base ring. The second and third point creations are more precise: they decree exactly the k-algebra in which the coefficients will lie. One should think of the expression `A ! [1,2]` as merely being a convenient shorthand, analogous to defining a sequence without being explicit about its universe.

`Origin(A)`

The origin of the affine space A.

`Simplex(A)`

The sequence of points of the ambient space A having coordinates $(1, 0, \ldots, 0)$, $\ldots (0, \ldots, 0, 1)$ and $(1, \ldots, 1)$ whether A is affine or projective space.

`Coordinates(p)`

The sequence of ring elements corresponding to the coordinates of the point p.

`p[i]`

`Coordinate(p,i)`

The ith coordinate of the point p.

`p @ f`

`f(p)`

`Evaluate(f, p)`

Evaluate the function f of the function field of the scheme X or its ambient at point p which lies on X.

Example H112E10

Although there are many rings which may appear as function fields in various contexts, their elements can all be evaluated at points. (Section 114.8 discusses the function field of a curve which is being used here only as an example.)

```
> A<x,y> := AffineSpace(Rationals(),2);
> FA<X,Y> := FunctionField(A);
> C := Curve(A,x^3 - y^2 + 3*x);
> FC<u,v> := FunctionField(C);
> p := A ! [1,2];
> q := C ! [1,2];
> f := x/y;
> g := X/Y;
> h := u/v;
> Evaluate(f,p), Evaluate(f,q);
1/2 1/2
> Evaluate(g,p), Evaluate(g,q);
1/2 1/2
> Evaluate(h,q);
1/2
> Evaluate(h,p);
1/2
> Evaluate(h,C!p);
1/2
```

112.3 Constructing Schemes

As shown in the examples in the introduction to this chapter, schemes are defined inside some ambient space, either affine or projective space, by a collection of polynomials from the coordinate ring associated with that space. Schemes may also be defined inside other schemes using polynomials from the coordinate ring of the bigger scheme or polynomials from the ambient space.

There is very little difference between creation methods for affine and projective schemes. Of course, in the projective case, the defining polynomials are checked for homogeneity or if an ideal is used, a check is made that its basis contains only homogeneous elements. Otherwise, the only check made at the time of creation is that the polynomials used to define the scheme really do lie in, or are coerced automatically into, the coordinate ring of the chosen ambient space.

Saturation:

As mentioned in the introduction, for schemes in projective spaces, there is a largest ideal which defines that scheme. Technically speaking, if I is any homogeneous defining ideal, this maximal one can be obtained from I by removing the primary components whose radical contains a certain *redundant ideal* of the ambient coordinate ring. This redundant ideal defines sets of points that are illegal projectively. For example, in ordinary projective space, there is one illegal point with all coordinates 0 and this is defined by the redundant ideal $(x_1, \ldots, x_n)$. The operation to remove these primary components is ideal saturation of I by the redundant ideal, so we refer to the maximal defining ideal as *saturated* and a scheme X as saturated if its current ideal (as returned by `DefiningIdeal(X)`) is known to be the maximal one.

There are several basic functions that rely on the defining ideal of a scheme X being saturated. The most important are `IsReduced, IsIrreducible, Prime/Primary Components, eq` for any projective schemes and `Dimension, f(X)` (map images) for multi-graded projective schemes.

As the process of saturation may be quite expensive in higher dimensional ambient spaces, the ideal of X is not saturated until the saturation property is required and once saturation has been performed, this is recorded internally. Additionally, scheme constructions like `Union` will automatically produce a result marked as saturated if that can be deduced from the construction method and the saturation state of the argument schemes. In particular, any ambient or scheme defined by a single equation in an ambient is marked as saturated on construction. The `ProjectiveClosure` of an affine scheme is also saturated by construction.

Furthermore, for all of the basic `Scheme` and `Curve` constructors where saturation of the ideal generated by the defining equations is not automatic, there is a `Saturated` parameter that the user can set to be `true` to mark the initial defining ideal as saturated without further checking.

`Scheme(X,f)`
`Scheme(X,F)`
`Scheme(X,I)`

`Scheme(X,Q)`

`Nonsingular`	BoolElt	*Default :* `false`
`Reduced`	BoolElt	*Default :* `false`
`Irreducible`	BoolElt	*Default :* `false`
`GeometricallyIrreducible`	BoolElt	*Default :* `false`
`Saturated`	BoolElt	*Default :* `false`

Create the scheme inside the scheme X defined by the vanishing of the polynomial f, or the sequence of polynomials F, or the ideal of polynomials I, or the ideal in the denominator of the quotient ring $Q = R/I$. In each case, the polynomials must be elements of the coordinate ring of A or automatically coercible into it.

If any of the optional parameters are set to `true`, MAGMA will assume without checking that the scheme has the corresponding property. This may enable subsequent calculations to be done faster; note that if the assumption is not correct, *arbitrary misbehaviour may result.* The option `Saturated` only makes sense when the ambient is projective, and refers to the defining ideal rather than the scheme.

`Cluster(X,f)`
`Cluster(X,F)`
`Cluster(X,I)`
`Cluster(X,Q)`

`Saturated`	BoolElt	*Default :* `false`

Create the 0-dimensional scheme inside the scheme X defined by the vanishing of the given polynomials. These can be given as the single polynomial f, or the sequence of polynomials F, or the ideal of polynomials I, or the ideal in the denominator of the quotient ring $Q = R/I$. In each case, the polynomials must be elements of the coordinate ring of X or automatically coercible into it.

Example H112E11

In this example we simply create three schemes. The first is an ambient space A, the affine plane, while the others are subschemes of A.

```
> A<x,y,z> := AffineSpace(Rationals(),3);
> X := Scheme(A,x-y);
> X;
Scheme over Rational Field defined by
x - y
> Y := Scheme(X,[x^2 - z^3,y^3 - z^4]);
> Y;
Scheme over Rational Field defined by
x^2 - z^3
y^3 - z^4
```

```
x - y
> Ambient(Y) eq A;
true
```

Note that since Y was created as a subscheme of X it inherits the equations of X. The ambient space of Y is still considered to be A.

`Spec(R)`

The scheme Spec(R) associated to the affine algebra R. A new affine space `Spec(Generic(R))` will be created as the ambient space of this scheme.

`Proj(R)`

The scheme Proj(R) associated to the affine algebra R which will be interpreted with its grading (which will be the standard grading by degree if no other has been assigned). A new projective space `Proj(Generic(R))` will be created as the ambient space of this scheme.

`EmptyScheme(X)`

`EmptySubscheme(X)`

The subscheme of X defined, for an affine scheme X by the trivial polynomial 1, or by maximal ideal $(x_1, \ldots, x_n)$ for a projective scheme X. The returned scheme is marked as saturated.

`X meet Y`

`Intersection(X,Y)`

The intersection of schemes X and Y in their common ambient space. This simply concatenates their defining equations without testing for emptiness.

`X join Y`

`Union(X,Y)`

The union of schemes X and Y in their common ambient space. This is formed by creating the intersection of their defining ideals which is done using a Gröbner basis computation. If both X and Y are saturated then the result is as well and is marked as such.

`&join S`

The union of the schemes in the sequence S in their common ambient space.

`Difference(X, Y)`

Returns the scheme that is obtained by taking the closure of the result of removing (`X meet Y`) from the scheme X, counting multiplicity. The ideal of the result will be the colon ideal of the ideal of X and the ideal of the scheme Y. If X is saturated then the result is as well and is marked as such.

`RemoveLinearRelations(X)`

Convenience function that takes linear relations between variables on X and uses them to eliminate variables. The intrinsic is currently only avaialable for X in ordinary projective space. The result is scheme Y that lies in a lower dimensional projective space that can be identified with the smallest linear subspace of the ambient of X that contains X. Y is returned along with the (linear) scheme isomorphism from X to Y.

`BlowUp(X,Y)`

`BlowUp(X,p)`

`Ordinary` BoolElt *Default :* `true`

The first intrinsic constructs the scheme obtained from blowing up subscheme Y of scheme X (see Section 7, Chapter II of [Har77]). The second is a user convenience special case that blows up the subscheme consisting of a point p (and all of its conjugates if it is not defined over the base field) in a pointset $X(K)$.

Currently X must lie in an ambient that is affine, ordinary projective or product projective. If parameter `Ordinary` is `true` (the default), and X is projective, then the result of the blow-up is embedded in ordinary projective space via the Segre embedding. Otherwise, the result will lie in an ambient that is the direct product of the ambient of X and an ordinary projective space.

For an example, see the first part of the chapter on algebraic surfaces.

The implementation makes use of the ReesIdeal intrinsic.

Example H112E12

The behaviour of `Difference` is shown.

```
> A2<x,y>:=AffineSpace(Rationals(),2);
> C:=Scheme(A2,(x*y)); //union of the x- and y-axis
> X2:=Scheme(A2,x^2);  //y-axis with double multiplicity
> Difference(X2,C); //y-axis with mult. 1.
Scheme over Rational Field defined by
x
> O:=Scheme(A2,[x,y]);
> Difference(C,O);
Scheme over Rational Field defined by
x*y
```

Removing "ambient" spaces is tricky: Everything is removed.

```
> Difference(C,A2);
Scheme over Rational Field defined by
1
> A3<x,y,z>:= AffineSpace(Rationals(),3);
> C:=Scheme(A3,Ideal([x,z])*Ideal([y,z])); //again, union of x- and y-axis
> Z:=Scheme(A3,[z]); //the x,y plane
> Difference(C,Z);
```

```
Scheme over Rational Field defined by
x,
y,
z
```

As one can see, the Z-plane is removed with multiplicities: all that's left is the origin, which has multiplicity 2 in C and only multiplicity 1 in Z.

`Saturate(~X)`

If the scheme X is projective and is not already saturated, saturate its defining ideal.

`AssignNames(~X,N)`

Assign the strings in the sequence N to the ambient coordinate functions of the scheme X.

`X . i`

`Name(X,i)`

The ith coordinate function of the ambient space of the scheme X. The dot notation `X.i` may also be used.

112.4 Different Types of Scheme

As discussed briefly in Section 112.1.7, there are a number of different increasingly specialised data types for schemes. It is often useful to check whether a given scheme can be thought of as belonging to one of these more specialised classes, and if so and appropriate then actually making the type change. In this section we document a number of such type-checking and type-change intrinsics, most of which are of the form `IsSpecialisedType`. These intrinsics always return a boolean value. If that value is `true` then they may also return a new scheme of the given specialised type, although in some trivial cases this does not happen. Of course, each of the different types of scheme has its own methods of construction independently of these intrinsics.

`IsAffine(X)`

Returns `true` if and only if the scheme X is an affine space.

`IsProjective(X)`

Returns `true` if and only if the scheme X is a projective space. Projective space here includes the case of scrolls.

`IsOrdinaryProjectiveSpace(X)`

Returns `true` if and only if the scheme X is a projective space in the usual sense: its coordinate ring has a single grading in which all the variables have weight one.

`IsAmbient(X)`

Return `true` if the scheme X is an ambient space.

`IsCluster(X)`

Returns `true` if and only if the scheme X is a zero-dimensional scheme (but not the empty scheme). See Section 112.8 for intrinsics which apply to clusters.

`IsCurve(X)`

Returns `true` if and only if X is a one-dimensional scheme. See Chapter 114 for intrinsics which apply to curves.

`IsPlaneCurve(X)`

Returns `true` if and only if X is a one-dimensional scheme defined by a single equation in a two-dimensional ambient space.

`IsConic(X)`

Returns `true` if and only if the scheme X is a curve (in the sense of `IsCurve(X)`) which is nonsingular and defined by an equation of degree 2. See Chapter 119 for intrinsics which apply to such conics.

`IsRationalCurve(X)`

Returns `true` if and only if the scheme X is a curve (in the sense of `IsCurve(X)`) which has genus 0. See Chapter 119 for intrinsics which apply to rational curves.

`IsHyperellipticCurve(X)`

Return `true` and a hyperelliptic curve if the scheme X is *trivially* a hyperelliptic curve. This occurs if and only if X is already of `CrvHyp` type or is defined by a non-singular Weierstrass equation in correctly weighted two-dimensional projective space. For a general scheme of type `Crv`, the intrinsic `IsHyperelliptic` in Chapter 114 will determine whether X is isomorphic to a hyperelliptic curve and return an isomorphism to a `CrvHyp` if so. This takes a lot more work in general. See Chapter 125 for more information on hyperelliptic curves.

`IsModularCurve(X)`

Return `true` if and only if the scheme X is a curve of type `CrvMod`. See Chapter 128 for more information on modular curves.

112.5 Basic Attributes of Schemes

These intrinsics report on basic features of the ambient space of a scheme or the equations defining a scheme. In many cases they simply call the corresponding function of the ambient space; the intrinsic `BaseRing()` is an example. The first set of these functions consists of those that only make reference to the ambient space, while the second set is concerned with the defining equations of the scheme.

112.5.1 Functions of the Ambient Space

`AmbientSpace(X)`

`Ambient(X)`

The ambient space containing the scheme X.

`SuperScheme(X)`

The scheme X was created as a subscheme of.

`BaseRing(X)`

`CoefficientRing(X)`

The base ring of the scheme X.

`BaseField(X)`

`CoefficientField(X)`

The base ring of the scheme X if it is a field, otherwise an error.

`IsAffine(X)`

Returns `true` if and only if the ambient space of the scheme X is affine.

`IsProjective(X)`

Returns `true` if and only if the ambient space of the scheme X is projective.

`IsOrdinaryProjective(X)`

Returns `true` if and only if the ambient space of the scheme X is an ordinary projective space, that is, its coordinate ring is generated in degree 1 with respect to the grading on the space.

`IsPlanar(X)`

Return `true` if the ambient of the scheme X is 2-dimensional.

`IsSaturated(X)`

Returns `true` if and only if the current defining ideal of the scheme X, as returned by `DefiningIdeal(X)` is saturated (see section 112.3 on page 3494).

112.5.2 Functions of the Equations

There are many ways to recover the equations which define a scheme. The standard method is to use the `DefiningPolynomials` function (or its singular versions) since it doesn't involve ideal theory overheads and certainly won't call any Gröbner basis functions.

`DefiningPolynomials(X)`

The defining polynomials for the ideal of the scheme X.

`DefiningPolynomial(X)`

The defining polynomial of the scheme X if it is a hypersurface. If X is not a hypersurface, an error is reported.

`DefiningIdeal(X)`

The ideal of a multivariate polynomial ring defining the scheme X.

`CoordinateRing(X)`

The quotient of the coordinate ring of the ambient space of the scheme X by the ideal of X.

`Curve(X)`

The smallest scheme in the inclusion chain above the scheme X which is a curve.

`GroebnerBasis(X)`

Return a sequence containing the polynomials of a Gröbner basis of the defining ideal of the scheme X. Note that the defining polynomials of X will not be changed, but that the basis of the ideal of X will be updated with the Gröbner basis as is the standard in the multivariate polynomial ring module.

`MinimalBasis(X)`

Return a minimal basis of the defining ideal of the scheme X, that is, a sequence of polynomials, no subsequence of which forms a basis of the ideal of X. Note that the defining polynomials of X will not be changed. This is the best human readable basis that MAGMA can supply.

`IsHypersurface(X)`

Returns `true` if and only if the scheme X is definable by a single polynomial. This function will perform a GCD calculation to simplify multiple defining polynomial if possible. The polynomial is returned as a second value.

`JacobianIdeal(X)`

The ideal of partial derivatives of the polynomials which define the scheme X.

`JacobianMatrix(X)`

The matrix $(\partial f_i/\partial x_j)$ of partial derivatives of the defining polynomials of the scheme X.

`HessianMatrix(X)`

The hessian matrix $(\partial^2 f/\partial x_i \partial x_j)$ of the hypersurface X where f is the polynomial which defines X.

`X eq Y`

Returns `true` if the schemes X and Y have the same types, ambients and ideals. If Gröbner basis calculations are not available this question may not be able to be decided. If X and Y are projective then they are saturated before ideal equality is tested for.

`IsSubscheme(X, Y)`

Returns `true` if and only if the scheme X is contained, scheme-theoretically, in the scheme Y. A Gröbner basis calculation checks the reverse inclusion of the corresponding ideals. If X and Y are projective, then X is saturated before the test for inclusion.

`IsLinear(X)`

Return `true` if the scheme X is defined by linear equations, possibly after taking a Gröbner basis.

Example H112E13

In this example we first create some schemes and then test them for inclusions and equality.

```
> P<u,v,w> := ProjectiveSpace(GF(11),2);
> C := Scheme(P,u^2 + u*w + 6*v^2);
> Z := Scheme(C,[u,v]);
> IsSubscheme(Z,C);
true
```

Now we will make another scheme which has the same polynomials as C but which is written in disguise. While the disguise in this case is simply to multiply the polynomial by 2 — the rather-too-obvious false nose and eyebrows among polynomials — the point is to note that the equality test in MAGMA is not fooled. The equality test identifies that the underlying defining ideals are the same and returns `true`.

```
> D := Scheme(P,2*u^2 + 2*u*w + v^2);
> D eq C;
true
> IsSubscheme(C,D) and IsSubscheme(D,C);
true
> DefiningIdeal(D) eq DefiningIdeal(C);
true
> DefiningPolynomial(D) eq DefiningPolynomial(C);
```

```
false
```

As we see in the final line above, checking the equality of ideals corresponds to the natural interpretation of equality.
There are a couple of caveats to this lesson, however. For instance, it is necessary, that the ideals to be comparable, i.e. the schemes must be embedded in the same ambient space.

```
> X<r,s,t> := ProjectiveSpace(GF(11),2);
> E := Scheme(P,r^2 + r*s + 6*t^2);
> E eq C;
false
```

112.6 Function Fields and their Elements

Since the function field of an irreducible variety is a birational invariant, function fields in MAGMA are associated with the projectively closed varieties. For an affine scheme X, the fields returned by `FunctionField(X)` and `FunctionField(ProjectiveClosure(X))` are identical. Currently, only function fields of projective spaces (and therefore of affine spaces) and curves are supported.

The following functions are provided for working with function fields and their elements. Some of these functions are concerned with converting function field elements into elements of the field of fractions of the coordinate ring of (an affine patch of) the scheme of the function field.

Additionally, function field elements may be used in the definition of scheme maps (see Section 112.14) from the projective or affine schemes on which they are defined to other schemes and may be evaluated at points as described earlier.

Function fields of schemes have type `FldFunFracSch` and their elements have type `FldFunFracSchElt`. These types inherit from `RngFunFracSch`, `RngFunFrac`, `RngMPolRes` and `RngFunFracSchElt`, `RngFunFracElt`, `RngMPolResElt` respectively.

`Scheme(F)`

Return the (projective) scheme F is the function field of.

`IntegerRing(F)`
`Integers(F)`

The integer ring of the function field F. This will be the coordinate ring of one of the patches of the scheme of F.

`AssignNames(~F, S)`

Assign the strings in S to be the names of the integer ring of F.

`F ! g`

Coerce the element g into the function field F of a scheme where g is some function on the scheme of F, for example, g may be an element of the field of fractions of the coordinate ring of a scheme having F as its function field.

`F . i`

Return the ith indeterminate of the coordinate ring of the scheme of F as an element of the function field F.

`ProjectiveFunction(f)`

Given an element f of a function field of a scheme, return f as an element of the field of fractions of the coordinate ring of the scheme f is a function on.

`ProjectiveRationalFunction(f)`

Given an element f of a function field of a (projective) scheme X, returns an element of the field of fractions of the coordinate ring of the ambient of X whose restriction to X as a rational function is f.

`RestrictionToPatch(f, Xi)`

Given an element f of a function field of a (projective) scheme X return f as an element of the field of fractions of the coordinate ring of the scheme Xi which must be a patch of X.

`Numerator(f)`
`Denominator(f)`

Given an element f of a function field of a scheme, return the numerator or denominator of f.

`f * g` `f + g` `f - g` `- f` `f / g` `f ^ n`
`f eq g` `IsZero(f)` `IsOne(f)` `IsMinusOne(f)` `IsUnit(f)`

`IntegralSplit(f, X)`

Given a function f on the (projective) scheme X return the numerator and the denominator of g, where g is some rational function on the ambient P of X restricting to f and considered as an element of the field of fractions of the coordinate ring of P.

`Numerator(f, X)`

The first return value of `IntegralSplit(f, X)`.

`Denominator(f, X)`

The second return value of `IntegralSplit(f, X)`.

Example H112E14

Some conversion of function field elements are shown.

```
> P2<X, Y, Z>:=ProjectiveSpace(Rationals(), 2);
> C := Curve(P2, X^2+Y^2-Z^2);
> K<xK, yK> := FunctionField(C);
> aC1<x1,y1> := AffinePatch(C, 1);
> aC2<x2,z2> := AffinePatch(C, 2);
> aC3<y3,z3> := AffinePatch(C, 3);
>
> f := (xK + yK)/(yK);
> K!f;
(xK + yK)/yK
> ProjectiveFunction($1);
(X + Y)/Y
> IntegralSplit(f, C);
X + Y
Y
> RestrictionToPatch(f, aC1);
($.1 + $.2)/$.2
> IntegralSplit(f, aC1);
x1 + y1
y1
> IntegralSplit(f, P2);
x2 + 1
1
```

`Restriction(f, Y)`

Given f in the function field of the scheme X and Y a subscheme of X with a function field, returns g in the function field of Y obtained by restricting f to Y. If f has a pole along Y, then `Infinity` is returned. An error occurs if f is not defined along Y. Presently, the only nontrivial application of this routine is when Y is a curve and X is the ambient of Y.

`GenericPoint(X)`

Returns a point in the pointset `X(FunctionField(X))` of the scheme X, whose coordinates generate `FunctionField(X)`.

112.7 Rational Points and Point Sets

There are two ways to think of points. If X is a scheme defined over a ring k and L is a k-algebra, then there is a set, called a *point set* and denoted $X(L)$ which is the set of points of X having coordinates in L or, in MAGMA terminology, the *parent* of such points. Note that in MAGMA a k-algebra is interpreted to mean any ring which admits coercion from k or which is the codomain of a ring homomorphism whose domain is k. When thinking of points as a sequence of coordinates on some scheme this type of point should be used. It is created by coercing the sequence of coordinates into the required point set using a statement such as

```
> X(L) ! [1,2,3];
```

Alternatively, if the universe of the sequence is equal to the base ring of the scheme, one may simply coerce the sequence into the scheme.

```
> X ! [1,2,3];
```

When the universe of the sequence is the integers, MAGMA will coerce them into the base ring of the scheme and again this shorthand will work.

The word point always refers to an object whose parent is some point set.

When a scheme is defined over a finite field, there are intrinsics which list all of its points defined over that field or over any finite extension of it.

An alternative approach is to consider points, or sets of points, as schemes in their own right. They can be defined by equations, after all. We call such zero-dimensional schemes *clusters*. They are more general than simply collections of points since their ideals could be nonradical. They are discussed in the Section 112.8 together with intrinsics which translate between points and clusters.

If p is a point, there are two ways of accessing its coordinates. The intrinsic `Coordinates` returns the sequence of all coordinates of p while `p[i]` returns the i-th coordinate alone. For example,

```
> p := X ! [1,2,3];
> Coordinates(p);
[ 1, 2, 3 ]
> p[1];
1
```

See Section 112.2.4 for descriptions of these and some other basic functions.

`X(L)`
`PointSet(X,L)`
`X(m)`
`PointSet(X,m)`

The point set of the scheme X of points whose coordinates lie in the ring L or in the codomain of the map m. The map m is a ring homomorphism from the base ring

of X to some other ring. Coercion from the base ring of X to L must be possible if m is not given.

`P eq Q`

Returns `true` if and only if the point sets P and Q where created on the same scheme and with the same map from the base ring of that scheme.

`Scheme(P)`

The scheme X associated to the point set P where P is of the form $X(L)$ for some extension L of the base ring of X.

`Curve(P)`

The smallest scheme in the inclusion chain above the scheme associated to the point set P which is a curve.

`Ring(P)`

The ring L associated to the point set P where P is of the form $X(L)$ for some scheme X.

`RingMap(P)`

The map from the base ring of the scheme of P to the ring of the pointset P.

`X ! Q`

`X(L) ! Q`

The point of the scheme X or the point set $X(L)$ (where X is a scheme and L is some extension ring of its base ring) determined by the sequence of coordinates Q. The universe of the sequence Q must be the base ring of X, or the ring L or some ring from which coercion into one of these is possible.

`p eq q`

Returns `true` if and only if the points p and q lie in some common scheme (possibly after coercion) and their coordinates are equal.

`p in X`

Returns `true` if and only if the point p lies in the scheme X or is coercible into it.

`Scheme(p)`

The scheme on which the point p lies.

`Curve(p)`

The smallest scheme in the inclusion chain above the scheme on which the point p lies which is a curve.

`Q in X`

Returns `true` if and only if all of the points of the set or sequence Q lie in the scheme X or are coercible into it.

`S subset X`

Returns `true` if and only if all points of the set S lie in the scheme X or are coercible into it.

`IsCoercible(X,Q)`

Returns `true` if and only if the sequence Q is the sequence of coordinates of some point of the scheme X. In that case, also return the point.

`RationalPoints(X)`

`RationalPoints(X,L)`

`Points(X)`

`Points(X,L)`

`Bound` RngIntElt *Default :* 1000

An indexed set containing points in the point set $X(L)$, where L is an extension of the base field of X. When not specified, L is taken to be the base field of X. This is implemented in the following situations: (i) L is a finite field, (ii) X has dimension zero, (iii) L is `Rationals()`. In cases (i) and (ii), all the points in $X(L)$ are found. In case (iii), a call to `PointSearch` is made, which searches for points with height up to the specified `Bound` (but note that it does not guarantee finding all of them).

In most cases, the first step is to determine the dimension of X by computing the Groebner basis of its defining ideal. This may be time-consuming; to avoid this one may directly call the relevant search: `Ratpoints` over finite fields, or `PointSearch` (specifying the `Dimension`) over the `Rationals()`.

`RationalPointsByFibration(X)`

`UseHypersurface` BoolElt *Default :* `false`

This is one of the methods used by `RationalPoints` when X is an affine or ordinary projective scheme over a finite field.

The basic idea is to work with a Noether Normalisation of the coordinate ring of X which, in the affine or projective case, gives an everywhere defined map with finite fibres to a linear subspace. We then run over all the points in the subspace adding in the points of X in the finite fibre. Determining the points on a zero-dimensional scheme is relatively fast and so we get a fairly efficient method for listing all points.

There is a variant to the basic algorithm. Rather than running over the linear subspace, it can take fibrations over a hypersurface of dimension one bigger the linear subspace. The points on these as fibred over the subspace may be quicker to find than for a general finite fibration and the finite fibres over the hypersurface are of smaller degree. For example, the general fibre contains only one point when the extra hypersurface equation generates X generically over the subspace.

However it is often slower to use the hypersurface because of the extra computation at the start and the two-stage processing so the default for `UseHypersurface` is `false`. The user may set this parameter to `true` for the variant to be applied.

`Random(S)`

Returns a random point in the pointset $S = X(k)$ where X is a scheme defined over a finite field, and k is a finite field. An error results if the pointset is empty. (Here 'random' simply means all points can occur, but not with uniform distribution.)

This is implemented using the Noether normalisation of X (similarly to `RationalPointsByFibration`).

`HasNonsingularPoint(X)`

`HasNonsingularPoint(X,L)`

Return `true` if and only if the scheme X defined over a finite field contains a non-singular point (defined over the finite field L if it appears as a second argument). In that case, also return such a point.

Example H112E15

In this example we define a scheme over a finite field and compute some points on it. Note that there are two point constructors used here. The first is simply `X ! Q` where Q is a sequence of integers (or base ring elements). In the second, we try to coerce a sequence Q whose elements do not lie in the base ring. The coercion into X cannot be used here. Instead one must be explicit about the intended point set. We have used the `IsCoercible(X,Q)` intrinsic which creates the point as a side-effect. One could also use the `X(L) ! Q` coercion to create the same point.

```
> A<x,y> := AffineSpace(FiniteField(7),2);
> X := Scheme(A,x^2 + y^2 + 1);
> X ! [2,3];
(2, 3)
> L<w> := ext< BaseRing(X) | 2 >;
> IsCoercible(X,[w^4,w^4]);
false
> IsCoercible(X(L),[w^4,w^4]);
true (w^4, w^4)
```

Finding those points was not simply good luck. In fact, we worked backwards and computed all points over the base field or L and chose one from each of those sets.

```
> RationalPoints(X);
{@ (3, 2), (4, 2), (2, 3), (5, 3), (2, 4), (5, 4), (3, 5), (4, 5) @}
> #RationalPoints(X,L);
48
```

We now consider an example of a curve in projective 3-space. In older versions of Magma this ran very slowly indeed. Now however, finding points over a fibration, it only takes a few seconds on a fast machine.

```
> k := GF(7823);
```

```
> R<x,y,z,w> := PolynomialRing(k, 4);
> I := ideal<R | 4*x*z + 2*x*w + y^2 + 4*y*w + 7821*z^2 + 7820*w^2,
>  4*x^2 + 4*x*y + 7821*x*w + 7822*y^2 + 7821*y*w +
>  7821*z^2 + 7819*z*w + 7820*w^2>;
> C := Curve(Proj(R), I);
> // a genus 0 curve with 1 cusp as singularities => 7823+1 points
> pts := RationalPointsByFibration(C); // could also just use RationalPoints
> #pts;
7824
```

Note that MAGMA has very fast machinery for computations like this for elliptic and hyperelliptic curves.

112.8 Zero-dimensional Schemes

This section describes intrinsics for creating zero dimensional schemes or *clusters*. It also discusses those functions which converts a finite set of points into the reduced zero dimensional having this support. Throughout this subsection, a lowercase p denotes a point of a scheme.

The word cluster refers to schemes that are known to be zero dimensional. In general, the intrinsic `Cluster` converts points to clusters while the function `RationalPoints` finds the points on a cluster which are rational over its base field.

Note that there are four constructors of the form `Cluster(X,data)` analogous to the four `Scheme(X,data)` constructors but which make an additional dimension test and type change before returning a cluster determined as a subscheme of X by the data of the second argument.

`Cluster(p)`

`Cluster(X, p)`

`Cluster(S)`

`Cluster(X, S)`

The reduced scheme supported at the point p, or supported at the set of points S, as a subscheme of the scheme X if given.

`RationalPoints(Z)`

`RationalPoints(Z,L)`

The set of rational points of the cluster Z. If an extension of the base field L is given as a second argument, the set of points of $Z(L)$, those points whose coordinates lie in L, is returned.

`PointsOverSplittingField(Z)`

If Z is a cluster this will determine some (not necessarily optimal) point set $Z(L)$ in which all points of Z having coordinates in an algebraic closure of the base field lie and will return all points of $Z(L)$.

`HasPointsOverExtension(X)`

`HasPointsOverExtension(X,L)`

Returns `false` if and only if all points in the support of the scheme X over an algebraic closure of its base field are already defined over its current base field, or all lie in the point set $X(L)$ if the second argument L is given. This intrinsic is most useful when trying to decide whether or not to make an extension of the base field of X to reveal non-rational points. The base field of X does not need to be a finite field.

`Degree(Z)`

The degree of the cluster Z. If Z is reduced, this is equal to the maximum number of points in the support over Z over some extension of its base ring.

Example H112E16

In this example we intersect a pair of plane curves. (Note that much more specialised machinery for working with curves is available in Chapter 114.) First we define two curves and find their points of intersection over the base field. The degree of the cluster Z is the usual numerical *intersection number* of the curves C and D. Here we are more interested in finding exactly those points that lie in the intersection.

```
> k := FiniteField(5);
> P<x,y,z> := ProjectiveSpace(k,2);
> C := Scheme(P,x^3 + y^3 - z^3);
> D := Scheme(P,x^2 + y^2 - z^2);
> Z := Intersection(C,D);
> IsCluster(Z);
true
> Degree(Z);
6
> RationalPoints(Z);
{ (1 : 0 : 1), (0 : 1 : 1) }
> HasPointsOverExtension(Z);
true
```

If C and D were rather general, that is, if Z was reduced, then we would expect 6 points in their intersection. We can't expect that here, but the final line above does confirm that we haven't yet seen all the points of intersection. We allow MAGMA to compute directly over a splitting field.

```
> PointsOverSplittingField(Z);
{ (0 : 1 : 1), ($.1^14 : $.1^22 : 1), ($.1^22 : $.1^14 : 1), (1 : 0 : 1) }
> L<w> := Ring(Universe($1));
> L;
```

```
Finite field of size 5^2
> PointsOverSplittingField(Z);
{ (0 : 1 : 1), (w^14 : w^22 : 1), (w^22 : w^14 : 1), (1 : 0 : 1) }
```

In this case we see that the support is not six points but only four.

112.9 Local Geometry of Schemes

We now discuss intrinsics which apply to a scheme at a single point. At the expense of increasing the number of intrinsics, we try to follow the convention that an intrinsic may simply take a point as its argument or it may take both a point and a scheme as its arguments. In the former case, the implicit scheme argument is taken to be the scheme associated to the point set of the point. In the latter case, it is first checked that the point can be coerced into some point set of the given scheme argument. There are reasons for allowing both methods. Of course, if one is confident about which scheme, X say, a point p lies on then there is no ambiguity about writing, say, `IsNonsingular(p)` rather than `IsNonsingular(X,p)`. On the other hand, the second expression is easier to read, and also guards against the possibility of accidentally referring to the wrong scheme; that is a particular risk here since the answer makes sense even if p lies on some other scheme—imagine the confusion that could arise given a point of a nonsingular curve lying on a singular surface inside a nonsingular ambient space, for instance. But also there are trivial cases when scheme arguments are necessary, `IntersectionNumber(C,D,p)` for example. In fact, that particular example exemplifies the value of points being highly coercible—it is very convenient that the point p could lie in a point set of either C or D or indeed neither of these as long as it could be coerced to them if necessary.

Sometimes a function will require that the point argument is rational, that is, has coordinates in the base ring.

112.9.1 Point Conditions

`IsSingular(X,p)`

Returns `true` if and only if the point p is a singular point of the scheme X.

`IsNonsingular(X,p)`

Returns `true` if and only if the point p is a nonsingular point of the scheme X.

`IsOrdinarySingularity(X,p)`

Returns `true` if and only if the tangent cone to the scheme X at the point p is reduced and X is singular at p. Currently, the scheme X must be a hypersurface.

112.9.2 Point Computations

`Multiplicity(p)`

`Multiplicity(X,p)`

The multiplicity of the point p as a point of the scheme X. If X is not a hypersurface, computed using local Groebner bases.

`TangentSpace(p)`

`TangentSpace(X,p)`

The tangent space to the scheme X at the point p. This linear space is embedded as a scheme in the same ambient space as X. An error will be signalled if p is a singular point of X or is not a rational point of X.

`TangentCone(p)`

`TangentCone(X,p)`

The tangent cone to X at the point p embedded as a scheme in the same ambient space. If the scheme X is not a hypersurface, the computation uses local Groebner bases.

112.9.3 Analytically Hypersurface Singularities

We will say that an isolated singular point p in a pointset $X(k)$ over a field k is a *hypersurface singularity* if the completion of its local ring is isomorphic to the quotient of a power series ring $k[[x_1, \ldots, x_d]]$ by a single power series $F(x_1, \ldots, x_d)$. That is, it is analytically equivalent to the singularity of an analytic hypersurface defined by F at the origin. Clearly d is equal to the dimension of the tangent space at p here. This type of singularity occurs quite commonly (e.g., singular points on actual hypersurfaces, A-D-E singularities on surfaces).

This section contains intrinsics to test whether such a p is a hypersurface singularity and compute the equivalent analytic equation F to given precision, to increase the precision of F at a later stage and to expand a rational function on X to the corresponding element in the field of fractions of $k[[x_1, \ldots, x_d]]$ to any required precision.

`IsHypersurfaceSingularity(p,prec)`

The point p is a singular point in $X(k)$ for a field k. It should be an isolated singularity on X and should only lie on irreducible components of X whose dimension d is maximal, i.e. the dimension of X (these conditions are not checked). The integer *prec* should be positive.

The function returns whether p is a hypersurface singularity on X as defined above. This is also equivalent to the conditions that the tangent space of p has dimension $d + 1$ and that the local ring at p is a local complete intersection ring.

If this is true, the function also returns extra values. The second return value is a multivariate polynomial F_1 in a polynomial ring $k[x_1, \ldots, x_{d+1}]$ such that the

singularity p is analytically equivalent to that of the analytic hypersurface $F \in k[[x_1, \ldots, x_{d+1}]]$ at the origin and F_1 is equal to F for terms of degree less than or equal to *prec*.

The third is a sequence of simple rational functions on X/k (i.e., quotients of polynomials in the coordinate ring of the ambient of X that, if X is projective, have the same degree for all gradings) such that x_i corresponds to the ith rational function of the sequence. If X is affine, these rational functions will all be k-linear forms in the coordinate variables and if X is ordinary projective, linear forms in the coordinate variables divided by a particular variable that is non-zero at p.

The fourth return value is a data record that is needed if the user wants to later expand F to higher precision or to expand arbitrary rational functions on X/k at p.

The implementation makes use of MAGMA's local Groebner bases, after localising p to an affine patch and translating it to the origin.

`HypersurfaceSingularityExpandFurther(dat,prec,R)`

The record *dat* should be the data record returned for a hypersurface singularity p in $X(k)$ by the intrinsic above, *prec* a positive integer and R a polynomial ring over k of rank $d+1$ (d is the dimension of X).

Returns a polynomial that expands the equation of the analytic hypersurface F to include all terms of degree less than or equal to *prec*. The result will be returned as an element of R (with the ith variable of R corresponding to the analytic variable denoted x_i in the exposition of the previous intrinsic).

This intrinsic is very useful to expand F to higher precision after the original call that determined that p was a hypersurface singularity.

`HypersurfaceSingularityExpandFunction(dat,f,prec,R)`

The record *dat* should be the data record returned for a hypersurface singularity p in $X(k)$ by the first intrinsic of this subsection, *prec* a positive integer and R a polynomial ring over k of rank $d+1$ (d is the dimension of X). f should be a rational function on X/k given as an element of the field of fractions of the coordinate ring of the ambient of X or the base change of X to k, if k isn't the base ring of X. f can in fact be given as an element of any rational function field over k whose rank is equal to the rank of the coordinate ring of X. In any case, when X is projective, the numerator and denominator of f have to be homogeneous and have the same degree with repect to all gradings of X.

Returns f pulled back to the analytic coordinate ring at p (identified with $k[[x_1, \ldots, x_{d+1}]]/(F)$ in the notation introduced above) and expanded to required precision. In fact, the return value is given as two polynomials a and b in R, whose variables are identified with the x_i, such that a/b is the finite approximation to the value of the pullback. a and b are actually just the pullbacks of the numerator and denominator of f expanded to include all terms of degree less than or equal to *prec*. The value is returned as two polynomials rather than a quotient as b may be zero if *prec* is sufficiently small even when the denominator of f doesn't vanish on X.

Example H112E17

We consider a singular degree 4 Del Pezzo surface in $\mathbf{P}^4$ over $\mathbf{Q}$ with two conjugate singular points defined over a quadratic extension.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> X := Scheme(P4,[x^2+y^2-2*z^2, x*t+t^2-y*u+2*u^2]);
> IsIrreducible(X);
true
> sngs := SingularSubscheme(X);
> Support(sngs);
> pts := PointsOverSplittingField(sngs);
> pts;
{@ (0 : 0 : 0 : r1 : 1), (0 : 0 : 0 : r2 : 1) @}
> pt := pts[1];
> k := Ring(Parent(pt));
> k;
Algebraically closed field with 2 variables over Rational Field
Defining relations:
[
    r2^2 + 2,
    r1^2 + 2
]
> p := X(k)!Eltseq(pt);
> boo,F,seq,dat := IsHypersurfaceSingularity(p,3);
> boo;
true
> R<a,b,c> := Parent(F);
> F;
2*r1*a*b^2 - 2*a^2*c - 3/2*r1*a*c^2 + 8*a^2 - 4*b^2 + 4*r1*a*c + c^2
> HypersurfaceSingularityExpandFurther(dat,4,R);
-a^4 - r1*a^3*c + 1/2*a^2*c^2 + 2*r1*a*b^2 - 2*a^2*c - 3/2*r1*a*c^2 +
Ψ8*a^2 - 4*b^2 + 4*r1*a*c + c^2
> Rk<p,q,r,s,w> := PolynomialRing(k,5);
>  //can use Rk as base-changed coordinate ring here
> HypersurfaceSingularityExpandFunction(dat,(p^2+q*s)/w^2,3,R);
2*r1*a^3 - 3*a^2*c - 1/2*r1*a*c^2 + 4*a^2 + (2*r1 + 1)*a*c - 1/2*c^2 + r1*c
1
```

112.10 Global Geometry of Schemes

Many of the names of intrinsics in this section come from the usual terminology of algebraic geometry. A reference for them is Hartshorne's book [Har77], especially Chapter II, Section 3.

`Dimension(X)`

The dimension of the scheme X. If X is irreducible then the meaning of this is clear, but in general it returns only the dimension of the highest dimensional component of X. The dimension of an empty scheme will be returned as -1. If the dimension is not already known, a Gröbner basis calculation is employed. If X is projective in a multi-graded ambient then it is saturated before this calculation takes place. The computation method involves computing the `Dimension` of the `Ideal` of the scheme, and then (for projective schemes) subtracting the number of gradings.

`Codimension(X)`

The codimension of the scheme X in its ambient space. In fact, this number is calculated as the difference of `Dimension(A)` and `Dimension(X)` where A is the ambient space, so if X is not irreducible this number is the codimension of a highest dimensional component of X.

`Degree(X)`

The degree of the scheme X.

`ArithmeticGenus(X)`

The arithmetic genus of a scheme X. The ambient space of X must be ordinary projective space.

`IsEmpty(X)`

Returns `true` if and only if the scheme X has no points over any algebraic closure of its base field. This intrinsic tests if the ideal of X is trivial (in a sense to be interpreted separately according to whether X is affine or projective) and then applies the Nullstellensatz.

`IsNonsingular(X)`

Returns `true` if and only if the scheme X is nonsingular and equidimensional over an algebraic closure of its base field. The test `IsEmpty` for the emptiness of the scheme is applied to the scheme defined by the vanishing of appropriately sized minors of the jacobian matrix of X.

`IsSingular(X)`

Returns `true` if and only if the scheme X either has a singular point or fails to be equidimensional over an algebraic closure of its base field.

`SingularSubscheme(X)`

The subscheme of the scheme X defined by the vanishing of the appropriately sized minors of the jacobian matrix of X. If X is not equidimensional, its lower dimensional components will be contained in this scheme whether they are singular or not.

`PrimeComponents(X)`

A sequence containing the irredundant prime components of the scheme X.

`PrimaryComponents(X)`

A sequence containing the irredundant primary components of the scheme X.

`ReducedSubscheme(X)`

The subscheme of X with reduced scheme structure, followed by the map of schemes to X. This function uses a Gröbner basis to compute the radical of the defining ideal of X.

`IsIrreducible(X)`

Returns `true` if and only if the scheme X has a unique prime component. If X is not a hypersurface, a Gröbner basis calculation is necessary and X is saturated before this occurs if it is projective.

`IsReduced(X)`

Returns `true` if and only if the defining ideal of the scheme X equals its radical. If X is a hypersurface the evaluation of this intrinsic uses only derivatives so works more generally than the situations where a Gröbner basis calculation is necessary. In the latter case, X is saturated before the calculation if it is projective.

`IsCohenMacaulay(X)`

`IsGorenstein(X)`

`IsArithmeticallyCohenMacaulay(X)`

`IsArithmeticallyGorenstein(X)`

`CheckEqui` BoolElt *Default :* `false`

These intrinsics currently only apply to schemes in ordinary projective space. The first two intrinsics return whether X is (locally) Cohen-Macaulay/Gorenstein, meaning that the local ring of every the scheme-theoretic point on X satisfies the property. The second two intrinsics return whether the coordinate ring of X (the polynomial coordinate ring of the projective ambient quotiented by the maximal defining ideal of X) satisfies the corresponding property. The arithmetic version implies the local version. The results are stored internally with X for future reference. Also, if X is known to be non-singular, we can immediately deduce the local version of the properties is true and this check is also performed internally.

There is a further slight restriction in that X has to be equidimensional (each irreducible component having the same dimension and there being no other scheme-theoretic "associated points" beside the generic points of the irreducible components : true if X is also reduced). This is *not* checked by default in order to save some computation time. If the user is unsure whether X is equidimensional, the `CheckEqui` parameter should be set to `true` which forces a check.

The implementations use the minimal free polynomial resolution of the maximal defining ideal of X. The arithmetic versions are actually faster than the plain versions. `IsGorenstein` may be particularly heavy computationally as it has to check whether the canonical sheaf is locally free of rank 1 after the Cohen-Macaulay property has been verified.

Example H112E18

In this example we write down a rather unpleasant scheme and analyse the basic properties of its components.

```
> A<x,y,z> := AffineSpace(Rationals(),3);
> X := Scheme(A,[x*y^3,x^3*z]);
> Dimension(X);
2
> IsReduced(X);
false
> PrimaryComponents(X);
[
    Scheme over Rational Field defined by
    x,
    Scheme over Rational Field defined by
    x^3
    y^3,
    Scheme over Rational Field defined by
    y^3
    z
]
> ReducedSubscheme(X);
Scheme over Rational Field defined by
x*y
x*z
```

The reduced scheme of X is clearly the union of a line and a plane. The scheme X itself is more complicated, having another line embedded in the plane component.

112.11 Base Change for Schemes

Let A be some ambient space in MAGMA. For example, think of A as being the affine plane. Let k be its base ring and R_A its coordinate ring. If $m : k \to L$ is a map of rings (a coercion map, for instance) then there is a new ambient space denoted A_L and called the *base change of A to L* which has coordinate ring R_{A_L} with coefficient ring L instead of k. (Mathematically, one simply tensors R_A with L over k. In MAGMA the equivalent function at the level of polynomial rings is `ChangeRing`.) There is a base change function described below which takes A and L (or the map $k \to L$) as arguments and creates this new space A_L. Note that there is a map from the coordinate ring of A to that of A_L determined by the map m.

This operation is called *base extension* since one often thinks of the map m as being an extension of fields. Of course, the map m could be many other things. One key example where the name *extension* is a little unusual would be when m is the map from the integers to some finite field.

Now let X be a scheme in MAGMA. Thus X is defined by some polynomials $f_1, \ldots, f_r$ on some ambient space A. Given a ring map $k \to L$ there is a base change operation for X which returns the *base change of X to L*, denoted X_L. This is done by first making the base change of A to L and then using the map from the coordinate ring of A to that of A_L to translate the polynomials f_i into polynomials defined on A_L. These polynomials can then be used to define a scheme in A_L. It is this resulting scheme which is the base change of X to L.

If one has a number of schemes in the same ambient space and wants to base change them all at the same time, a little care is required. The function which takes a scheme and a map of rings as argument will create a new ambient space each time so is unsuitable. Better would be to base change the ambient space and then use the base change function which takes the scheme and the desired new ambient space as argument. (This latter base change function appears to be different from the other ones. In fact it is not. We described base change above as a function of maps of rings. Of course, there is a natural extension to maps of schemes. With that extension, this final base change intrinsic really is base change with respect to map of ambient spaces.)

`BaseChange(A,K)`

`BaseExtend(A,K)`

If A is a scheme defined over a field k and K is an extension into which elements of k can be automatically coerced then this returns a new scheme A_K defined over K. No cached data about A will be transferred to A_K and coordinate names will have to be defined again on A_K if needed.

`BaseChange(A,m)`

`BaseExtend(A,m)`

If m is a map of rings whose domain is the base ring of the scheme A, this returns the base change of A to the codomain of m. The equations of A, if any, are mapped to the new ambient coordinate ring using m.

BaseChange(F,K)

BaseExtend(F,K)

BaseChange(F,m)

BaseExtend(F,m)

If F is a sequence of schemes lying in a common ambient space whose base ring admits automatic coercion to K or is the domain of a ring map m then this returns the base change of the elements of F as a new sequence.

BaseChange(X,A)

BaseExtend(X,A)

BaseChange(X,A,m)

BaseExtend(X,A,m)

If X is any scheme whose ambient space B is of the same type (affine or projective) and dimension as the ambient space A but either has a base ring which admits coercion to that of A or the map m is a ring map from the base ring of B to that of A then this returns a scheme with the equations of X as a subscheme of A. The equations are transferred to A using coercion or the map m.

BaseChange(X, n)

BaseExtend(X, n)

The base change of the scheme X, where the base ring of X is a finite field to the finite field which is a degree n extension of the base field of X.

Example H112E19

Here are two curves whose intersection points are not defined over the rationals and one of which only splits after a field extension. The basic function to calculate intersection points only searches for them over the current field of definition so misses them at first. But with an extra argument it is able to search over an extension of the base without actually changing base of the schemes. This contrasts with finding higher dimensional components of schemes which always requires the base change.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,x^2 + y^2);
> IsIrreducible(C);
true
> D := Curve(A,x - 1);
> IntersectionPoints(C,D);
{}
> Qi<i> := QuadraticField(-1);
> IntersectionPoints(C,D,Qi);
{ (0, i), (0, -i) }
```

So we have found the intersection points (although we haven't explained how we chose the right field extension). Now we do the same calculation again but by making the base change of all

schemes to the field Qi. Over this field the intersection points are immediately visible, but also the curve C splits into two components.

```
> B<u,v> := BaseChange(A,Qi);
> C1 := BaseChange(C,B);
> D1 := BaseChange(D,B);
> IsIrreducible(C1);
false
> IntersectionPoints(C1,D1);
{ (0, i), (0, -i) }
> PrimeComponents(C1);
[
    Scheme over Qi defined by u + i*v,
    Scheme over Qi defined by u - i*v
]
```

112.12 Affine Patches and Projective Closure

In MAGMA, any affine ambient space A has a unique projective closure. This may be assigned different variable names just like any projective space. The projective closure intrinsics applied to affine schemes in A will return projective schemes in the projective closure of A. Conversely, a projective space has a number of standard affine patches. These will be the ambient spaces of the standard affine patches of a projective scheme. In this way, the closures of any two schemes lying in the same space will also lie in the same space. The same goes for standard affine patches. These relationships between affine and projective objects are very tightly fixed: asking for the projective closure of an affine scheme will always return the identical object, for instance.

`ProjectiveClosure(X)`

The projective closure of the scheme X. If the projective closure has already been computed, this scheme will be returned. If X is an affine space for which no projective closure has been computed, the projective closure will be a projective space with this space as its first standard patch. Otherwise, the result will lie in the projective closure of the ambient space of X. If X has been computed as an affine patch the projective closure will be the scheme it is an affine patch of even if this is not mathematically correct (see Example H112E21).

`AffinePatch(X,i)`

The ith affine patch of the scheme X. The number of affine patches is dependent on the type of projective ambient space in which X lies, but for instance, the standard projective space of dimension n has $n+1$ affine patches. In that case, i can be any integer in the range $1, \ldots, n+1$. The order for affine patches is the natural one once you decide that the first patch is that with final coordinate entry nonzero (in the projective closure).

`AffinePatch(X,p)`

A standard affine patch of the scheme X containing the point p. The second return value is the point corresponding to p in that patch.

`IsStandardAffinePatch(A)`

Return whether the affine space A is a standard affine patch of its projective closure and if so which patch it is. For A to be a non–standard patch means that its projective closure must have been set using `MakePCMap`. Returns `false` if A does not have a projective closure to be a patch of.

`NumberOfAffinePatches(X)`

Return the number of standard affine patches of the scheme X (O if X is an affine scheme).

`HasAffinePatch(X, i)`

Return whether the ith patch of the scheme X can be created.

Example H112E20

This example shows that taking patches and closures several times really does return identical schemes.

```
> A1<u,v> := AffineSpace(GF(5),2);
> X := Scheme(A1,u^2 - v^5);
> PX<U,V,W> := ProjectiveClosure(X);
> PX;
Scheme over GF(5) defined by
U^2*W^3 + 4*V^5
> AffinePatch(PX,1) eq X;
true
> X2<u2,w2> := AffinePatch(PX,2);
> X2;
Scheme over GF(5) defined by
u2^2*w2^3 + 4
> ProjectiveClosure(X2) eq ProjectiveClosure(X);
true
```

Example H112E21

Even if those schemes are not mathematically correct.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> L := Curve(P2,Z);
> Laff := AffinePatch(L,1);
> Dimension(Laff);
-1
> Laff;
```

```
Scheme over Rational Field defined by
1
> ProjectiveClosure(Laff) eq L;
true
> ProjectiveClosure(EmptyScheme(Ambient(Laff)));
Scheme over Rational Field defined by
1
```

`HyperplaneAtInfinity(X)`

The hyperplane complement of the scheme X in its projective closure.

`ProjectiveClosureMap(A)`

`PCMap(A)`

The map from the affine space A to its projective closure.

`AffineDecomposition(P)`

Projective spaces have a standard disjoint decomposition into affine pieces—not the same thing as the affine patches—of the form

$$\mathbf{P}^n = \mathbf{A}^n \cup \mathbf{A}^{n-1} \cup \cdots \cup \mathbf{A}^1 \cup p$$

where $\mathbf{A}^n$ is the first affine patch, $\mathbf{A}^{n-1}$ is the first affine patch on the hyperplane at infinity and so on. Finally, p is the point $(1 : 0 : \cdots : 0)$. This intrinsic returns a sequence of maps from affine spaces to the projective space P whose images are these affine pieces of a decomposition. The point p is returned as a second value.

`CentredAffinePatch(S, p)`

An affine patch of S centred at the point p and the embedding into S, achieved by translation of a standard affine patch.

112.13 Arithmetic Properties of Schemes and Points

This section contains several functions of arithmetic interest, that apply to general schemes defined over the rationals, number fields, or function fields.

112.13.1 Height

`HeightOnAmbient(P)`

`Absolute`	BoolElt	*Default :* `false`
`Precision`	RngIntElt	*Default :* 30

The height of the given point, as a point in the ambient projective space (or affine space). This is the *exponential* height (for the logarithmic height, take `Log`!). By default it is the *relative* height, unless the optional parameter `Absolute` is given. The function works when the ambient space (of the scheme containing P) is any affine or projective space (possibly weighted), and the ring of definition of P is contained in the rationals, a number field or a function field.

Note that `Height` is also defined for certain special schemes, such as elliptic and hyperelliptic curves. In these cases the `Height` is a *canonical, logarithmic, absolute* height.

112.13.2 Restriction of Scalars

`RestrictionOfScalars(S, F)`

`WeilRestriction(S, F)`

`SubfieldMap`	Map	*Default :*
`ExtensionBasis`	[FldElt]	*Default :* []

Given an affine scheme S whose base ring is a field K, the function returns its restriction of scalars from K to F. The scheme S must be contained in an affine space. The field F should be either a subfield of K, or else isomorphic to a subfield of K; in this case the inclusion map may be specified as the optional argument `SubfieldMap`, or otherwise is the same as the map returned by `IsSubfield(F,K)`.

The restriction of scalars S_{res} is a scheme over F satisfying the following functorial property (for point sets): $S_{res}(R) = S(R \otimes_F K)$ for all rings $R \supseteq F$.

Four objects are returned; the first is the scheme S_{res}, and the other three are maps of various kinds:

- the natural map of schemes from the base extension $(S_{res})_{\otimes K}$ to S,
- a function which takes a point in a point set $S_{res}(R)$ and computes its image under the map $S_{res}(R) \to S(RK)$ (if there is no known relationship between R and K an error results), and
- a map of point sets $S(K) \to S_{res}(F)$.

The result is obtained by direct substitution using the standard basis of K/F, or the basis given in ExtensionBasis if this is specified.

112.13.3 Local Solubility

Let X be a scheme over a number field K. One of the fundamental problems in arithmetic geometry is to decide if $X(K)$ is empty. In general, this is a very hard question. One way to arrive at an affirmative answer is to prove that $X(L)$ is empty for some larger field L. An important class of such field is formed by *complete local fields*. These are obtained by taking the p-adic topology on K associated to a prime ideal p of the ring of integers of K and to consider the topological completion L of K (see `Completion` on page 891).

`IsEmpty(Xm)`

`Smooth`	BoolElt	*Default :* `false`
`AssumeIrreducible`	BoolElt	*Default :* `false`
`AssumeNonsingular`	BoolElt	*Default :* `false`
Verbose	`LocSol`	*Maximum :* 2

Let X be a scheme over a number field K and let L be a completion of K at a finite prime. If Xm is the point set of X taking values in L, this function returns if the point set is empty. If `false` is returned, then (a minimal approximation to) a point is returned.

Setting `AssumeIrreducible := true` tells the system to assume that X is irreducible. This leads to unpredictable results if the components of X have distinct dimensions.

Setting `AssumeNonsingular := true` tells the system to assume that X has no L-rational singular points. If there are such points and this flag is set, then an infinite loop can result.

Setting `Smooth := true` tells the system to only consider nonsingular points on X. This is only implemented for plane curves. The system will blow up any L-rational singular points when encountered and test the desingularized model for solubility. If a point is found, then an approximation may be returned that is indistinguishable from a singular point.

The following classes of schemes are recognized and treated separately :

(i) Hyperelliptic curves over fields with large residue field of odd characteristic. For these fields a generalization by Nils Bruin of an algorithm presented in [MSS96] is used. The complexity of this algorithm is independent of the size of the residue field.

(ii) Hyperelliptic curves over fields with small residue field or residue field of even characteristic. For these fields a depth-first backtracking algorithm is used to construct a solution.

(iii) Nonsingular curves represented by a possibly singular planar model. In this case, a depth-first backtracking algorithm is used to construct a solution. Whenever a tentative solution approximates a rational singularity, that singularity is blown up and the construction is continued on the desingularized model with Hensel's lemma as a stopping criterion. Use `Smooth := true` to access this option.

(iv) An intersection in $\mathbf{P}^3$ of a quadratic cone with a singularity at $(0:0:0:1)$ and another quadric. This case (often needed in computations concerning elliptic curves) is handled by testing hyperelliptic curves for local solubility.

(v) General schemes. The system decomposes X into primary components over K, which are equidimensional. For each of the components, it tests the singular subscheme for solubility. If a solution is found, this is returned. Otherwise, it tests the scheme itself for solubility using a depth-first backtracking algorithm with Hensel's lemma as stopping criterion.

Note that to construct a point set over a completion of a number field, one should use `PointSet(X,phi)` where `Kp, phi := Completion(K,p)`. This rather cryptic syntax is due to the fact that, although a completion strictly speaking is an *extension* of K, MAGMA does not recognise this fact and therefore does not allow for automatic coercion into K_p. The system has to be presented explicitly with the map ϕ from K into K_p.

For a description of the algorithms used, see [Bru04].

Example H112E22

Some usage of `IsEmpty` is illustrated below.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,X^2+Y^2);
> IsEmpty(C(pAdicField(2,20)));
false (O(2^2) : O(2^2) : 1 + O(2^20))
> IsEmpty(C(pAdicField(2,20)):Smooth);
true
> K<i> := NumberField(PolynomialRing(Rationals())![1,0,1]);
> CK := BaseChange(C,K);
> p := (1+i)*IntegerRing(K);
> Kp,toKp := Completion(K,p);
> CKp := PointSet(CK,toKp);
> IsEmpty(CKp:Smooth);
false (O(Kp.1^3) : O(Kp.1^3) : 1 + O(Kp.1^100))
```

Example H112E23

And a simpler example.

```
> p := 32003;
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(p*(x^10+p*x^3-p^2*4710));
> Qp := pAdicField(p);
> time IsEmpty(C(Qp));
true
Time: 0.090
```

`IsLocallySolvable(X, p)`

`Smooth`	BoolElt	*Default :* `false`
`AssumeIrreducible`	BoolElt	*Default :* `false`
`AssumeNonsingular`	BoolElt	*Default :* `false`

Given a projective scheme X defined over a number field or over the rationals, test if the scheme is locally solvable at the prime ideal p (for number fields) or prime number p (for rationals) indicated. If the scheme is found to have a local point, then `true` is returned together with an approximation to a point. Otherwise, `false` is returned.

The optional parameters have the same meaning as for `IsEmpty` (see above). For a description of the algorithms used, see [Bru04].

Example H112E24

A locally solvable scheme and a non–locally solvable scheme (when considering only non–singular points) are both shown below.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,X^2+Y^2);
> IsLocallySolvable(C,2);
true (O(2^2) : O(2^2) : 1 + O(2^50))
> IsLocallySolvable(C,2:Smooth);
false
> K<i>:=NumberField(PolynomialRing(Rationals())![1,0,1]);
> CK:=BaseChange(C,K);
> p:=(1+i)*IntegerRing(K);
> IsLocallySolvable(BaseChange(C,K),p:Smooth);
true (O($.1^3) : O($.1^3) : 1 + O($.1^100))
```

`LiftPoint(P, n)`

`Strict`	BoolElt	*Default :* `true`

`LiftPoint(F, d, P, n)`

`Strict`	BoolElt	*Default :* `false`

Let P be in $X(L)$, where X is a scheme over the rationals or over a number field and L is a completion of the base field at a finite prime. This routine attempts to lift P to the desired precision n using quadratic newton iteration. This routine only works if P is distinguishable from all singular points on X. Note that if X is of positive dimension, then the lift is inherently arbitrary.

Due to limited precision used in the computation, it may be impossible to attain the desired precision exactly. By default this leads to an error. If the user specifies `Strict := false` the system silently returns a lift with maximum attainable precision if the desired precision cannot be reached.

The second form provides the same functionality, but requires all input data to be supplied over L. The sequence F should be the defining equations of a scheme over L of dimension d. The sequence P should be coordinates of an approximation of a point on that scheme. The returned sequence is a lift of the point described by P to precision n, if possible.

In principle, a point returned by `IsEmpty` has sufficient precision for `LiftPoint` to work. However, `IsEmpty` may perform nontrivial operations on the scheme. A nonsingular point on a component of X may be singular on X itself.

Example H112E25

A lift of a point in a non empty pointset is given.

```
> P2<X,Y,Z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P2,X^2+Y^2-17*Z^2);
> Qp:=pAdicField(2,20);
> bl,P:=IsEmpty(C(Qp));
> LiftPoint(P,15);
(9961 + O(2^15) : O(2^15) : 1 + O(2^20))
```

112.13.4 Searching for Points

The following intrinsic implements a nontrivial method to search for points on any scheme defined over the rationals.

`PointSearch(S,H : parameters)`

`Dimension`	RngIntElt	*Default :* 0
`Primes`	SeqEnum	*Default :* [ ]
`OnlyOne`	BoolElt	*Default :* `false`

Searches for points on the scheme S up to roughly height H. The scheme must be in either an affine space or a non-weighted projective space.

This uses a p-adic algorithm: first find points locally modulo a small prime (or two small primes), then lift these p-adically, and then see if these give global solutions. Lattice reduction is used at this stage, and this makes the method far more efficient than a naive search, for most schemes. Note that points which reduce to singular points modulo p are not necessarily found.

If `OnlyOne` is `true`, then the computation will terminate as soon as one point is found. The algorithm computes the dimension of the scheme unless `Dimension` is set to a nonzero value.

The algorithm chooses its own primes unless `Primes` is non-empty; either one prime or two can be specified. The algorithm slows down considerably in higher dimension — for threefolds, it can take a few hours to search for points up to height 500 or so. For special types of curves (ternary cubics, intersection of quadrics), efficient search methods are implemented under other names (for instance `Points` and `PointsQI`).

Example H112E26

```
> P<a,b,c,d> := ProjectiveSpace(Rationals(),3);
> S := Scheme (P, a^2*c^2 - b*d^3 + 2*a^2*b*c + a*b^3 - a*b^2*c +
>                 7*a*c^2*d + 4*a*b*d^2);
> Dimension(S);
2
> time PS := PointSearch(S,100);
Time: 9.050
> #PS; // not necessarily exhaustive
67571
```

112.14 Maps between Schemes

Given schemes X and Y one can define a map $f : X \to Y$ in a number of ways. The basic method is to give a sequence of polynomials or quotients of polynomials defined on X. If X has an associated function field then function field elements may also be used. Alternative sets of defining polynomials/rational functions may also be given, as long as these represent the same rational map as the original defining set.

Scheme maps in MAGMA represent *rational* maps between schemes. That is, a map $f : X \to Y$ between schemes X and Y actually corresponds to a morphism from a dense Zariski-open subset of X to Y and two maps f and g from X to Y are considered to be equal (and will be so deemed by `eq`, for example) if they are equal as morphisms when restricted to a dense open subset of X that lies in a domain of definition of f and a domain of definition of g. The precise open domain of definition U of a map f is unspecified but in most functional contexts, it is equal to the complement of the base scheme f as returned by `BaseScheme(f)`. This is the set of scheme-theoretic points at which none of the maps given by the defining polynomials/rational functions or any set of alternative defining polynomials are 'naively' defined. `Extend` computes alternative defining equations that reduces the base scheme so that its open complement is equal to the maximal domain of definition of the rational map represented. However, this is a fairly generic implementation that can be computationally very heavy in many cases. When the domain X is a curve with function field and the codomain Y is ordinary projective, the implicit domain of definition for computing the image of points is the maximal domain of definition of f rather than the base scheme. The function field machinery is used here.

Similarly, isomorphisms between schemes can be defined using inverse defining polynomials/rational functions and these represent birational maps rather than actual scheme isomorphisms.

There are some natural functions associated to a map f and some other more complicated functions. The most natural things are, for a point p of X, computing the image point $f(p)$ and, for a subscheme $S \subset Y$ computing the preimage scheme `Pullback(f,S)`. More complicated functions include the standard Gröbner basis algorithm for computing images $f(T) \subset Y$ of subschemes T of X. This is defined when T doesn't lie in the

base scheme of f and the image is actually the scheme theoretic closure of the image of f restricted to the open subset of T on which it is defined.

For some functions such as the image computation just mentioned, it is a requirement that the schemes X and Y be defined over a common base ring and that the map f be defined over the identity map (or coercion map) of this common base ring.

Maps respect point set structures. Indeed, given a map f as above and an extension L of k, the base ring of X, there is a map $f(L) : X(L) \to Y(L)$. The point set of Y is determined by the composition of the map of base rings with the extension map $k \to L$. This map $f(L)$ cannot be created without creating the scheme map f first, and in any case it is not usually explicitly required: the evaluation of $f(p)$ will invoke it in the background, for example. But when the main purpose of f is to transfer a large number of points from $X(L)$ to $Y(L)$ then it is best to assign $g := f(L)$ explicitly and use the map g. In this way the small additional overheads involved in repetitively constructing $f(L)$ are avoided. Perhaps more importantly, it also ensures that the image points all lie in the same point set.

Maps respect projective closures of schemes. That is, given f as above one can compute the projective closure of f which is a map from the projective closure of X to that of Y and which agrees with f where they are both defined.

From V2.16, there is a slight variant on the basic scheme map type (`MapSch`), which we refer to as scheme graph maps and are defined by the graph of the map in the product of the domain and codomain. These currently only have very basic constructors and somewhat less functionality than scheme maps. They will be described in a separate subsection.

Much of the functionality only works for scheme maps f between schemes whose base rings are fields. We usually do not explicitly mention in this documentation when this is a requirement, but the functions will return an error if the base rings are not fields in cases where they must be.

112.14.1 Creation of Maps

The most basic map constructors are described here. It is possible to create maps defined over a map of the base rings of the schemes.

```
map< X -> Y | F >
```

```
map< X -> Y | F, G >
```

```
map< X -> Y | u, F >
```

`Check`	BoolElt	*Default :* `true`
`CheckInverse`	BoolElt	*Default :* `true`

Create the map $X \to Y$ of schemes determined by the sequence F of polynomials or rational functions defined on X. The two schemes X and Y must be defined over compatible base rings, that is, there must exist a ring map u from the base ring of Y to that of X. The polynomials can be elements of the coordinate ring of the ambient space containing X or elements of the coordinate ring of X itself. In any case, the polynomials will be lifted to elements of the coordinate ring of the

ambient space and any ring elements which admit this operation could be used. If the function field of X exists then elements of this may also be used.

The argument u is a map from the base ring of Y to that of X. If it is omitted then natural coercion map is assumed. In practice, the base rings are often identical in which case coercion will simply be the identity map.

A birational inverse specified by polynomials or rational functions on Y can be given as the sequence G.

It is also allowed for F and G to be a sequence of sequences of polynomials or rational functions, giving alternate sets of defining equations (inverse defining equations) that must define the same rational map on X (rational inverse on Y). The base scheme of the map (the inverse) is the intersection of the base schemes of the individual sets of defining equations (inverse defining equations) as described in `BaseScheme`.

There are two parameters `Check` and `CheckInverse` which control exactly what validation checks are carried out.

If the parameter `Check` is set to `false` then no checking occurs at all.

If `Check` is `true` but CheckInverse is `false`, then the map is checked to be well defined from domain to codomain and any inverse to be well defined in the opposite direction. However if there are inverse functions specified, it is not checked that the forward and inverse functions actually define (birational) inverse maps.

If both parameters are `true` then all checks mentioned above are done.

`iso< X -> Y | F, G >`

Create the map $X \to Y$ of schemes determined by the sequence F for which the map $Y \to X$ determined by the sequence G is a birational inverse. The sequences F and G should contain polynomials or rational functions defined on X and Y respectively. The two schemes X and Y must be defined over the same base ring.

The two check parameters for the main map constructor also apply here with the same meaning.

Example H112E27

Map creation is very similar to that for maps between polynomial rings, although here one must put the polynomial arguments into a single sequence.

```
> k := Rationals();
> A<t> := AffineSpace(k,1);
> B<x,y> := AffineSpace(k,2);
> f := map< A -> B | [t^3 + t, t^2 - 3] >;
> f;
Mapping from: Aff: A to Aff: B
with equations :
t^3 + t
```

```
t^2 - 3
```

Features of the map can be computed. Of course, the domain and codomain are trivial attributes of the map while its image requires a Gröbner basis computation.

```
> Domain(f) eq A;
true
Variables : t
> Codomain(f);
Affine Space of dimension 2
Variables : x, y
> Image(f);
Scheme over Rational Field defined by
-x^2 + y^3 + 11*y^2 + 40*y + 48
```

Example H112E28

Defining maps using function field elements directly can be particularly convenient for curves. For example, the user may have a set of such functions coming from some divisor computations. Here is a simple (artificial!) example where a rational map from a projective curve or any of it's affine patchs to the projective line corresponding to a rational function is defined.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^2+y^2-z^2);
> C1 := AffinePatch(C,1);
> C2 := AffinePatch(C,2);
> F := FunctionField(C);
> f := F!(x/y);
> P1 := ProjectiveSpace(Rationals(),1);
> mp1 := map<C->P1 | [f,1]>;
> mp2 := map<C1->P1 | [f,1]>;
> mp3 := map<C2->P1 | [f,1]>;
```

Example H112E29

Maps of the base ring can be included in scheme maps as is the case for maps of polynomial rings. Here we make a Frobenius map.

```
> k<w> := FiniteField(3^2);
> u := hom< k -> k | w^3 >;
> A<t> := AffineSpace(k,1);
> f := map< A -> A | u, [t^3] >;
> f;
Mapping from: Aff: A to Aff: A
with equations :
t^3
and map between base rings
```

```
Mapping from: FldFin: k to FldFin: k given by a rule [no inverse]
```

Notice next how the map f fixes points defined over the prime subfield of k but moves those points with coordinates having nontrivial w component.

```
> p := A ! [w];
> f(p);
(w^3)
> f(A ! [2]);
(2)
```

`IdentityMap(X)`

Create the identity map of the scheme X.

`ConstantMap(X,Y,p)`

`map< X -> Y | Q >`

The map taking all points of the scheme X to the point p of scheme Y where Q is the sequence of coordinates of p.

`Projection(X,Y)`

The linear projection from projective space X to projective space Y that omits the first $\dim X - \dim Y$ coordinates.

`Projection(X, Q)`

`Projection(X)`

`Projection(X, p)`

The projection of the scheme X away from the point p into the projective ambient space Q (if given). If p is not given it is taken to be $(1 : 0 : \ldots : 0)$.

`ProjectionFromNonsingularPoint(X,p)`

The projection of the scheme X from the nonsingular (and rational) point p of X. The projection map is returned as a second value. The image of the blowup of p as a point of X is returned as a third value. If this is a point, it is returned as a point type.

`ProjectiveMap(L, Y)`

`ProjectiveMap(L)`

Given a list L of functions in the function field of X, where X is a projective scheme, return the projective map $X \to Y$ defined by taking those functions as projective coordinates. The scheme Y should be a projective space of dimension one less than the length of L. If Y is not supplied, a new projective space of appropriate dimension is created.

`ProjectiveMap(f, Y)`

`ProjectiveMap(f)`

A short form for `ProjectiveMap([f,1],X,Y)`.

Example H112E30

The above function is illustrated.

```
> P2<X,Y,Z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P2,
>      X^4-2*X^3*Y-X^2*Y^2-2*X^2*Y*Z+2*X*Y^3+2*X*Y^2*Z+Y^4-7*Y^3*Z+Y^2*Z^2);
> omega:=CanonicalDivisor(C);
> Degree(omega); //genus 0 curve
-2
> L:=Basis(-omega);
> L;
[
    ($.2 - 4/25) * ($.1^3 - $.1*$.2 + 5/56*$.2^3 + 20/63*$.2^2) * ($.2 - 4) *
    ($.1^3*$.2 - 1/4*$.1^2*$.2^2 - 26/5*$.1^2*$.2 + 4/5*$.1^2 - 7/4*$.1*$.2^3 -
        2*$.1*$.2^2 - 3/4*$.2^4 + 151/20*$.2^3 + 4*$.2^2 - 4/5*$.2)^-1,
    ($.2) * ($.1^2 - 3/14*$.2^2 + 1/63*$.2) * ($.2 - 4) * ($.2 - 4/25) *
    ($.1^3*$.2 - 1/4*$.1^2*$.2^2 - 26/5*$.1^2*$.2 + 4/5*$.1^2 - 7/4*$.1*$.2^3 -
        2*$.1*$.2^2 - 3/4*$.2^4 + 151/20*$.2^3 + 4*$.2^2 - 4/5*$.2)^-1,
    ($.1 + 13/28*$.2 - 8/63) * ($.2 - 4) * ($.2 - 4/25) * ($.2)^2 * ($.1^3*$.2 -
        1/4*$.1^2*$.2^2 - 26/5*$.1^2*$.2 + 4/5*$.1^2 - 7/4*$.1*$.2^3 -
        2*$.1*$.2^2 - 3/4*$.2^4 + 151/20*$.2^3 + 4*$.2^2 - 4/5*$.2)^-1
]
> mp:=ProjectiveMap(L,P2); //anticanonical embedding
> mp;
Mapping from: Crv: C to Prj: P2
with equations :
23/16*X^3 - 113/64*X^2*Y + 5/16*X^2*Z - 71/64*X*Y^2 + 93/16*X*Y*Z + 21/64*Y^3 -
    61/64*Y^2*Z - 5/16*Y*Z^2
-1/8*X^3 + 23/16*X^2*Y + X^2*Z - 19/16*X*Y^2 - 15/8*X*Y*Z - 21/16*Y^3 +
    115/16*Y^2*Z - Y*Z^2
X^3 - 53/32*X^2*Y - 1/8*X^2*Z - 11/32*X*Y^2 - 3/4*X*Y*Z + 21/32*Y^3 -
    13/32*Y^2*Z + 1/8*Y*Z^2
> mp(C);
Curve over Rational Field defined by
X^2 - 15/8*X*Y + 23/4*X*Z - 229/32*Y^2 - 17/16*Y*Z - Z^2
```

`Elimination(X,V)`

The affine scheme obtained by eliminating the ambient variables of the affine scheme X whose indices appear in V from the equations of X. Thus if $V = [2, 5]$ then the result will be a scheme in the affine subspace $u = v = 0$ where u and v are the second and fifth variables of the ambient space of X.

`Inverse(f)`

The inverse of the map of schemes f if inverse defining equations have been included in the definition of f, otherwise an error.

`IsInvertible(f)`

Tests whether the map f between schemes is birational. If so, returns a birational inverse. This function works fairly generically, using Groebner basis computations over standard affine patches of the domain and codomain to compute the closure of the graph of f and retrieve inverse equations from that. It can be very expensive computationally.

`HasKnownInverse(f)`

Returns `true` if the map f has an inverse stored.

Example H112E31

This example shows how `IsInvertible` is more powerful then `Inverse`.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2, X^3*Y^2 + X^3*Z^2 - Z^5);
> Genus(C);
1
> pt := C![1,0,1];
> E,toE := EllipticCurve(C,pt);
> IsInvertible(toE);
true Mapping from: CrvEll: E to Crv: C
with equations :
$.1^3
-3*$.1*$.2*$.3 - 9*$.2*$.3^2
$.1^3 + 3*$.1^2*$.3
and inverse
-3*X^2*Z + 3*X*Z^2
-3*X^2*Y
X^2*Z - 2*X*Z^2 + Z^3
>  Inverse(toE);
>> Inverse(toE);
          ^
Runtime error in 'Inverse': Map has no inverse
```

`g * f`

The composition $g \circ f$, but note the convention for order of composition: the order of mapping is that g acts first and is followed by f. Strictly speaking, one might want to see evaluation of points done on the left to make sense of this, (this can be done using `p @ f` instead of `f(p)`). Since one would usually assign a new identifier to this composition, this is not a large problem. Only simple error checking is done — domain–codomain matching and that the composition doesn't have so many zero components that it is projectively illegal.

Where the expansion of such compositions could be expensive, the resulting map will be stored as a composition. The equivalent expanded map can be created by

```
> gf := g*f;
> dp := DefiningPolynomials(gf);
> dpi := InverseDefiningPolynomials(gf);
> m := map<D -> C | dp, dpi>;
```

where D is the domain of g and C is the codomain of f. The composition will act differently to the expanded map - it will be undefined at all the places each factor is undefined.

`Components(f)`

The maps composed to form f.

Example H112E32

As part of its generic map machinery, MAGMA has a structure for the set of all maps between two given schemes. There is also a structure for the group of all automorphisms of a scheme which is discussed in Section 112.14.6. Using this space one can realise the effect of a map on Hom spaces. We will make two Hom spaces having a common codomain.

```
> k := Rationals();
> P<x,y,z,t> := ProjectiveSpace(k,3);
> A := Scheme(P,Minors(M,2))
>           where M is Matrix(CoordinateRing(P),2,3,[x,y,z,y,z,t]);
> B := Scheme(P,x*t - y*z);
> F<r,s,u,v> := RuledSurface(k,0,0);
> HomAF := Maps(A,F);
> HomBF := Maps(B,F);
> HomAF;
Set of all maps from A to F
```

Given a map $A \to B$ we make the map from `HomBF` to `HomAF` given by composition. Although A lies inside B, we choose a map $A \to B$ which isn't this inclusion.

```
> i := map< A -> B | [y,x,t,z] >;
> ii := map< HomBF -> HomAF | g :-> i * g >;
```

The map ii of Hom spaces realises the composition of maps with i. We test this on a single map $f : B \to F$.

```
> f := map< B -> F | [x,y,z,t] >;
```

```
> Expand(ii(f)) eq Expand(i*f);
true
```

`Restriction(f,X,Y)`

`Check` BoolElt *Default :* `true`

The restriction of the map of schemes f to the scheme X in its domain. The codomain of the new map is considered to be the scheme Y which must either contain the codomain of f, or lie in that codomain but contain the scheme $f(X)$.

By default the program checks these subscheme relationships; this may be time-consuming, and can be skipped by setting the optional parameter `Check` to `false`.

`Expand(phi)`

Given a map ϕ between schemes stored in factored form, return the map in expanded form. Note that if ϕ is a composite of maps, each with many alternative defining polynomials, then computing the expansion can be very expensive. In other situations computing the expansion of ϕ can cause huge (intermediate) expressions and therefore be very expensive as well.

Due to the automatic simplification in map creation, the base scheme of the returned map might be smaller than the base scheme of ϕ.

`Extend(phi)`

Given a map ϕ between schemes, returns an expanded map with extra alternative equations in order to reduce the base scheme as far as possible, i.e. so that the open complement of the base scheme is the maximal domain of definition of the rational map represented by ϕ. This routine is potentially very expensive because it requires Groebner basis computations on several affine graph ideals of ϕ.

`Prune(phi)`

Given a map ϕ between schemes in expanded form, removes alternative equations that do not reduce the base scheme of ϕ. If ϕ has a known inverse, this is returned unaltered.

`Normalization(phi)`

`Normalisation(phi)`

The map created from ϕ by removing common factors from the defining polynomial.

Example H112E33

It is shown below how to convert a map between schemes into a proper morphism.

```
> P2<x,y,z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P2,x^3+y^3-2*z^3);
> E,phicomp:=EllipticCurve(C,Place(C![1,1,1]));
> Puvw<u,v,w>:=Ambient(E);
```

We get ϕ as `phicomp` in expanded form.

```
> phi:=Expand(phicomp);
> phi;
Mapping from: Crv: C to CrvEll: E
with equations :
3*x^2*y - 3*x^2*z - 3*x*y^2 + 9*x*y*z - 6*x*z^2 - 6*y^2*z + 15*y*z^2 - 9*z^3
-9*x^2*y + 27*x*y^2 - 54*x*y*z + 18*x*z^2 + 45*y^2*z - 81*y*z^2 + 27*z^3
y^3 - 3*y^2*z + 3*y*z^2 - z^3
```

But ϕ still has a base scheme.

```
> Degree(BaseScheme(phi));
6
```

So we extend ϕ to `phiext` defined on C entirely.

```
> phiext:=Extend(phi);
> phiext;
Mapping from: Crv: C to CrvEll: E
with equations :
3*x^2*y - 3*x^2*z - 3*x*y^2 + 9*x*y*z - 6*x*z^2 - 6*y^2*z + 15*y*z^2 - 9*z^3
-9*x^2*y + 27*x*y^2 - 54*x*y*z + 18*x*z^2 + 45*y^2*z - 81*y*z^2 + 27*z^3
y^3 - 3*y^2*z + 3*y*z^2 - z^3
and alternative equations :
3*x^3 + 6*x^2*z - 6*x*y*z + 9*x*z^2 + 3*y^3 - 15*y*z^2
-18*x^2*z + 54*x*y*z - 27*x*z^2 + 99*y*z^2
x*z^2 + 2*y^2*z - 3*y*z^2
```

And there is no base scheme anymore!

```
> Degree(BaseScheme(phiext));
0
```

This map is invertible.

```
> bl,phii:=IsInvertible(phiext);
> assert bl;
> phii;
Mapping from: CrvEll: E to Crv: C
with equations :
1/6*u^3 - 13/2*u^2*w - 2*u*v*w + 21/2*u*w^2 - 1/6*v^2*w + 3*v*w^2 + 27*w^3
u^2*w - 30*u*w^2 - 3*v*w^2 + 36*w^3
u^2*w - 3*u*w^2 - 18*w^3
```

```
and inverse
3*x^2*y - 3*x^2*z - 3*x*y^2 + 9*x*y*z - 6*x*z^2 - 6*y^2*z + 15*y*z^2 - 9*z^3
-9*x^2*y + 27*x*y^2 - 54*x*y*z + 18*x*z^2 + 45*y^2*z - 81*y*z^2 + 27*z^3
y^3 - 3*y^2*z + 3*y*z^2 - z^3
and alternative inverse equations :
3*x^3 + 6*x^2*z - 6*x*y*z + 9*x*z^2 + 3*y^3 - 15*y*z^2
-18*x^2*z + 54*x*y*z - 27*x*z^2 + 99*y*z^2
x*z^2 + 2*y^2*z - 3*y*z^2
```

But the inverse still has a base scheme.

```
> Degree(BaseScheme(phii));
6
```

So extend `phii`.

```
> phiiext:=Extend(phii);
```

No base scheme left!

```
> Degree(BaseScheme(phiiext));
0
```

Note that the inverse – `phiext` – is still retained. So `phiiext` now really is a morphism.

```
> phiiext;
Mapping from: CrvEll: E to Crv: C
with equations :
1/6*u^3 - 13/2*u^2*w - 2*u*v*w + 21/2*u*w^2 - 1/6*v^2*w + 3*v*w^2 + 27*w^3
u^2*w - 30*u*w^2 - 3*v*w^2 + 36*w^3
u^2*w - 3*u*w^2 - 18*w^3
and inverse
3*x^2*y - 3*x^2*z - 3*x*y^2 + 9*x*y*z - 6*x*z^2 - 6*y^2*z + 15*y*z^2 - 9*z^3
-9*x^2*y + 27*x*y^2 - 54*x*y*z + 18*x*z^2 + 45*y^2*z - 81*y*z^2 + 27*z^3
y^3 - 3*y^2*z + 3*y*z^2 - z^3
and alternative equations :
-1/234*u^3 + 7/26*u^2*w + 5/117*u*v*w - 57/26*u*w^2 + 1/702*v^2*w - 6/13*v*w^2 -
    27/13*w^3
-1/39*u^2*w + 1/117*u*v*w + 21/13*u*w^2 - 1/351*v^2*w + 2/39*v*w^2 - 54/13*w^3
u*w^2 - 1/351*v^2*w + 4/39*v*w^2 + 18/13*w^3
-729/26*u^3 + 3/13*u^2*v + 32805/26*u^2*w + 17/13*u*v^2 + 4293/13*u*v*w -
    19683/26*u*w^2 + 4/39*v^3 + 537/26*v^2*w - 1377/13*v*w^2 + 19683/13*w^3
-27/13*u^2*v + 9/13*u*v^2 + 1233/13*u*v*w + 4/39*v^3 + 132/13*v^2*w -
    1215/13*v*w^2
u*v^2 + 4/39*v^3 - 9/13*v^2*w + 756/13*v*w^2
and alternative inverse equations :
3*x^3 + 6*x^2*z - 6*x*y*z + 9*x*z^2 + 3*y^3 - 15*y*z^2
-18*x^2*z + 54*x*y*z - 27*x*z^2 + 99*y*z^2
x*z^2 + 2*y^2*z - 3*y*z^2
```

112.14.2 Basic Attributes

112.14.2.1 Trivial Attributes

`Domain(f)`

The domain of the map of schemes f.

`Codomain(f)`

The codomain of the map of schemes f.

`DefiningPolynomials(f)`

`DefiningEquations(f)`

The sequence of functions used to define the map of schemes f.

If f is stored as an unexpanded composition then it will be expanded and the defining equations of the expansion returned.

`FactoredDefiningPolynomials(f)`

If the map of schemes f was created by composition (and not expanded) return the sequence of sequences of the defining equations of the maps which were composed to form f otherwise return `DefiningPolynomials` of f.

`InverseDefiningPolynomials(f)`

The sequence of functions used to define the inverse of the map of schemes f.

If f is stored as an unexpanded composition then it will be expanded and the inverse defining equations of the expansion returned.

`FactoredInverseDefiningPolynomials(f)`

If the map of schemes f was created by composition (and not expanded) and has an inverse return the sequence of sequences of inverse defining equations of the maps which were composed to form f otherwise return `InverseDefiningPolynomials` of f.

`AllDefiningPolynomials(f)`

The polynomials of all definitions of the map of schemes f.

`AllInverseDefiningPolynomials(f)`

The polynomials of all definitions of the inverse of the map of schemes f if f has a known inverse.

`AlgebraMap(f)`

The underlying map of polynomial rings determining the map of schemes f. Thus if F is the sequence of defining equations of f and x is the first variable of the codomain then `F[1]` will be the image of x under `AlgebraMap(f)`.

`FunctionDegree(f)`

The degree of the homogeneous polynomials which define the map f of projective schemes. If there are alternative defining polynomials, returns the minimum value over the different sets of defining polynomials.

112.14.2.2 Basic Tests

`f eq g`

Returns `true` if and only if the maps of schemes f and g have the same domain and codomain and define the same rational map.

`IsRegular(f)`

`IsPolynomial(f)`

Returns `true` if and only if the map of schemes f is defined at all points of its domain.

`IsIsomorphism(f)`

Returns `true` if and only if the map of schemes $f : X \to Y$ has inverse defining equations or if they may be easily computed (e.g. the projective closure of the map has inverse defining equations). If so, return a map $g : X \to Y$ which is of the recognised isomorphism type as a second value.

`IsDominant(f)`

Returns `true` if and only if the closure of the image of the map of schemes f is the whole of its codomain.

`IsLinear(f)`

Returns `true` if and only if the map of schemes f is a regular map defined by linear polynomials.

`IsAffineLinear(f)`

Returns `true` if and only if the map of schemes f is a map between affine spaces defined by polynomials of degree at most 1.

112.14.3 Maps and Points

Given a map $f : X \to Y$ of schemes and point p of X outside of the base scheme of f, then the image $f(p)$ is a point of Y. Moreover, given an extension K of the base rings of X and Y, there is a map of point sets $f(K) : X(K) \to Y(K)$. This isn't often needed, but should be used as in the example below, when very many point images are required. Note that it will ensure that all points are returned in the same determined point set. Maps also behave well with respect to sets and sequences of points.

`f(p)`

The point $f(p)$ if the point p is in a point set of the domain of the map of schemes f and doesn't lie in the base scheme of f. An error results if p is in the base scheme (except in the curve case described below). Sets and sequences of points are handled in the same way. If the domain of the map is a curve C with a function field (see Algebraic Curves chapter) and p is in the base scheme of f, then now (from V2.17), MAGMA tries to compute the image of p by working with the function field places over p without having to extend f via `Extend`. If p is non-singular then there is only one place above it and the image will exist if the codomain is projective. If there are several places over p, the image will be computed and returned when the image of all of the places is the same and is defined over p's point set base ring (that must be a field).

`Pullback(f, p)`

The preimage of a point p under the map of schemes f. When f is an isomorphism of schemes with an inverse g, the returned result is the point $g(p)$. Otherwise the pullback is returned as a subscheme of the domain of f. This is identical to `Pullback(f,S)`, where S is the one-point scheme containing p.

When a scheme is returned, it will contain the base scheme of f (which won't map to p under f), but this can be remedied using the (potentially expensive) function call `Difference(Pullback(f,p), BaseScheme(f))`.

`p @@ f`

This is the same as `Pullback(f,p)` *except* when f is an isogeny between elliptic curves. in which case one rational point in the preimage is returned (if none exist, an error results).

`f(K)`

`f(m)`

The map induced by the map of schemes $f : X \to Y$ on point sets $X(K) \to Y(K)$. If m is a ring map from the base ring of X and u is the map of base rings from Y to X then $f(m)$ will be the map of point sets $X(m) \to Y(m(u))$.

Example H112E34

Mapping a single point is easy.

```
> P1<s,t> := ProjectiveSpace(Rationals(),1);
> P3<w,x,y,z> := ProjectiveSpace(Rationals(),3);
> f := map< P1 -> P3 | [s^4,s^3*t,s*t^3,t^4] >;
> p := P1 ! [2,1];
> f(p);
(16 : 8 : 2 : 1)
```

If many points need to be mapped from a fixed point set, a small overhead can be avoided by working with the map of point sets directly.

```
> K := QuadraticField(5);
> g := f(K);
> ims := [];
> for i in [1..100] do
> Append(~ims, g(P1 ! [i,1]));
> end for;
```

This example could also have been handled in one step using a sequence constructor.

```
> pts := [ P1 ! [i,1] : i in [1..100] ];
> f(pts) eq ims;
true
```

An example where the projection from an elliptic curve to the projective line contains the origin in its base scheme but the image under the extension of the projection is computed by using the place machinery without having to globally extend the map.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> E := Curve(P2,Y^2*Z-X^3-X*Z^2);
> p := E![0,0,1];
> f := map<E->P1|[X,Y]>;
> p in BaseScheme(f);
true (0 : 0 : 1)
> f(p);
(0 : 1)
```

112.14.4 Maps and Schemes

The natural operation for maps on schemes is *pullback*, that is, compute the preimage. This is rather trivial. On the other hand, computing the image of schemes under a map requires a Gröbner basis calculation so is much harder. If the map has an inverse this image calculation is automatically replaced by the more simple pullback using the inverse map.

Note that, strictly speaking, the image algorithm computes the closure of the image of the map. We still call it the image, though, and don't worry that there may be some points of the image that are not the set-theoretic image of any point of the domain.

Over a field, the equations of the image of a map in a particular degree can be computed using linear algebra, so a distinct intrinsic is provided for this.

Other schemes related to a map are also discussed here.

`Pullback(f, X)`

The scheme in the domain of the map of schemes f given by the pullback of the equations defining the subscheme X of the codomain of f. The result will contain the base scheme of f (which won't map onto X under f) but this can be remedied using the (potentially expensive) function call `Difference(Pullback(f,X), BaseScheme(f))`.

`Image(f)`

`f(X)`

Let X be a subscheme of the domain of the map of schemes f such that, if U is the open complement of the base scheme of f, $X \cap U$ is Zariski-dense in X. This intrinsic returns the scheme-theoretic closure of $f(X \cap U)$ in the codomain of f, which we refer to simple as the image $f(X)$. For the first signature the image of the entire domain is returned. Moreover, it is stored with the map f so can be called again later without any recomputation. Note that if the domain of f is projective multi-graded, then X is saturated before the computation to ensure the correct result. For computaional efficiency, we do not check that $X \cap U$ is indeed Zariski-dense in X.

`Image(f,X,d)`

A basis of the polynomials of degree d in the codomain of the map of schemes f which contain the image $f(X)$. The scheme X must be a subscheme of the domain of f and d must be a positive integer.

For best results for projective schemes, remove the contribution of the irrelevant ideal corresponding to the zero point from X (normalize the equations of the scheme).

Example H112E35

Consider the embedding of the projective line in 3-space as a quartic. It can be defined as the image of a map determined by a 4-dimensional subspace of the degree 4 monomials on the line, for instance: see [Har77] Chapter II, Example 7.8.6 or the later section here on linear systems.

```
> P1<s,t> := ProjectiveSpace(Rationals(),1);
> P3<w,x,y,z> := ProjectiveSpace(Rationals(),3);
> f := map< P1 -> P3 | [s^4,s^3*t,s*t^3,t^4] >;
> Image(f);
Scheme over Rational Field defined by
-w^2*y + x^3
w*y^2 - x^2*z
-x*z^2 + y^3
-w*z + x*y
> IsNonsingular(Image(f));
true
> f(p) in Image(f) where p is P1 ! [2,1];
true (16 : 8 : 2 : 1)
```

If the Gröbner basis computation is too expensive, or if a partial solution for the image computation would be acceptable, the function `Image(f,C,d)` described above and illustrated in the next example calculates those hypersurfaces of degree d containing $f(C)$. Given a bound on d, the equations of the image could also be calculated using this function.

Example H112E36

A situation where one is really interested in the equations of the image in a particular degree occurs in the case of canonical curves. Usually the ideal is generated in degree 2, but for trigonal curves the degree 2 generators only cut out a surface scroll on which the curve is cut out by a relative equation of degree 3.
Here we simply assert that the curve C has genus 5 and that the map f is the canonical map of the curve C. Chapter 114 describes functions that determine both invariants.

```
> k := Rationals();
> P2<X,Y,Z> := ProjectiveSpace(k,2);
> P4<a,b,c,d,e> := ProjectiveSpace(k,4);
> C := Curve(P2, X^5 + X*Y^3*Z + Z^5);
> f := map< P2 -> P4 | [Y*Z, X*Y, Z^2, X*Z, X^2] >;
> S := Image(f,C,2);
> S;
Scheme over Rational Field defined by
a*d - b*c
a*e - b*d
c*e - d^2
> Dimension(S);
2
> f(C);
Curve over Rational Field defined by
-d^2 + c*e,
-b*d + a*e,
```

```
-b*c + a*d,
a*b^2 + c^2*d + e^3,
a^2*b + c^3 + d*e^2
```

In this case both image computations are fast so the timing difference between them is tiny. But pushing the genus a little higher soon makes the point.
It is easy to see that S is a scroll: its equations are the rank 2 minors of the matrix

$$\begin{pmatrix} a & c & d \\ b & d & e \end{pmatrix}.$$

`BaseScheme(f)`

The subscheme of the domain X of the map of schemes f where the map is 'naively' not defined. When f is expanded, this is equal to the intersection of the base schemes of each of the alternate sets of defining polynomials/rational functions. For a single set of defining polynomials/rational functions, the base scheme is defined by the union of subschemes of X where a denominator of a defining rational function vanishes when the codomain is affine, and by the subset of points of X which are mapped by the given defining polynomials into a null point (with coordinates on which all polynomials in an irrelevant ideal vanish) when the codomain is projective.

`BasePoints(f)`
`BasePoints(f,L)`

If the base scheme of the map of schemes f is finite, this returns a sequence containing those points defined over the base ring which lie in it. Otherwise an error is reported. If a second argument L is included which is an extension field of the base field then base points defined over L are returned.

Example H112E37

We find the base points of a map, although we have to extend the field before we find them all.

```
> k := GF(7);
> P<x,y,z> := ProjectiveSpace(k,2);
> p := x^2 + y^2;
> f := map< P -> P | [p*x,p*y,z^2*(z-x)] >;
> BasePoints(f);
{}
> Degree(BaseScheme(f));
6
> HasPointsOverExtension(BaseScheme(f));
true
```

Clearly we are not seeing all the points of indeterminacy of f. We clumsily extend the base field until we do see enough points. Of course, it is clear that the problem is the polynomial p, so a degree 2 extension will be enough.

```
> BasePoints(f,ext<k|2>);
```

```
{ (1 : $.1^36 : 1), ($.1^12 : 1 : 0), ($.1^36 : 1 : 0), (1 : $.1^12 : 1) }
> HasPointsOverExtension(BaseScheme(f),ext<k|2>);
false
```

Example H112E38

In this example we make an elementary transformation of scrolls.

```
> Q := Rationals();
> F<u,v,x,y> := RuledSurface(Q,2);
> G<a,b,r,s> := RuledSurface(Q,3);
> F;
Rational Scroll of dimension 2
Variables : u, v, x, y
Gradings :
1       1       -2      0
0       0       1       1
> phi := map< F -> G | [u,v,x,y*u] >;
```

Next we find the base points of the map ϕ by hand.

```
> Scheme(F,[u,v]) join Scheme(F,[x,u*y]);
Scheme over Rational Field defined by
u*x
v*x
u*y
> RationalPoints($1);
{ (0 : 1 : 0 : 1) }
```

The map ϕ is the elementary transformation in the point $(0 : 1 : 0 : 1)$ of F. That is, it is the blowup of this point followed by the contraction of the birational transform of the fibre through this point.

112.14.5 Maps and Closure

`ProjectiveClosure(f)`

The map induced by the map of schemes f between the projective closure of its domain and codomain. If either domain or codomain is already projective, then it remains unchanged in the new map. In particular, if both domain and codomain are already projective, then the returned map is simply f itself.

`MakeProjectiveClosureMap(A, P, S)`

`MakePCMap(A, P, S)`

`MakeProjectiveClosureMap(m)`

`MakePCMap(m)`

If A is an affine space and P a projective space and if S is a sequence of polynomials on A defining a map from A to P (or if m is such a map) then this map is set as the projective closure map of A. There is very little functionality for projective closure maps which are not the standard ones, so this intrinsic is usually used in cases where no relationship between A and P yet exists but the user would like A to behave as a standard patch on P so that the closure of a scheme in A is a scheme in P.

`RestrictionToPatch(f,j)`

The restriction of the map f, a map of schemes from an affine scheme to a projective scheme, to a rational map from its domain to the jth standard affine patch of its codomain.

`RestrictionToPatch(f,i,j)`

The restriction of the map f, a map between two projective schemes, to a rational map from the ith standard affine patch of its domain to the jth patch of its codomain.

Example H112E39

The application of closure and patching functions is straightforward. To compute the restriction of a map f to the ith patch of the domain and jth patch of the codomain, essentially set the $\dim+2-i$th coordinate function to 1 and divide by the $\dim+2-j$th defining equation of f.

```
> P<w,x,y,z> := ProjectiveSpace(Rationals(),3);
> f := map< P -> P | [1/w,1/x,1/y,1/z] >;
> f12 := RestrictionToPatch(f,1,2);
> f12;
Map of affine spaces defined by [ $.3/$.1, $.3/$.2, $.3 ]
```

The functions are inevitably rational so cannot be expressed in any coordinates that might already exist on the affine patches. Instead they are expressed in terms of the generators of the function fields.

```
> ProjectiveClosure(f12);
Map of projective spaces defined by [ x*y*z, w*y*z, w*x*z, w*x*y ]
> ProjectiveClosure(f12) eq f;
true
```

However, as seen in the final line above, the relationship between a map and its closure is maintained.

112.14.6 Automorphisms

Automorphisms of schemes defined over a field may be constructed. The main cases where there is significant functionality is for automorphisms of affine and projective spaces and curves. Recall that for projective spaces the only regular automorphisms are the linear maps. However there are many more rational automorphisms, often called *Cremona transformations*. In the case of the projective plane, these form a group generated by linear automorphisms together with a single quadratic transformation. In higher dimensions, the structure of this group is unknown.

Affine spaces have much more complicated automorphism groups. Decomposition results are known in the case of the affine plane over certain fields (the complex numbers for instance), but otherwise no general statements are known. More information and references can be found in [vdE00], especially in the opening essay.

Although automorphisms can be computed, groups of automorphisms cannot be computed except in a very few cases. For the case of curves, see the `Algebraic Curves` chapter. For ambients, there is currently a function `AutomorphismGroup` which returns a group together with a map matching group elements with the automorphism they represent only in the case of linear automorphisms of projective spaces defined over a finite field.

`Automorphism(X,F)`

The automorphism of the scheme X determined by the sequence of polynomials F defined on X. This function uses a Gröbner basis calculation. If the inverse functions are already known then one can use the `map< | >` constructor and then a type change or the `iso< | >` constructor. This is illustrated in Example H112E40.

`IdentityAutomorphism(X)`

`IdentityMap(X)`

The identity map $X \to X$ on the scheme X.

`IsEndomorphism(f)`

Returns `true` if and only if the domain and range of the maps of schemes f are equal.

`IsAutomorphism(f)`

Returns `true` if and only if the maps of schemes f is an automorphism of its domain. In this case f is returned as an automorphism as the second value.

Example H112E40

In this example we show how to make an automorphism from equations, and also how to include the equations of the inverse map if they are already known. The automorphism is the hyperelliptic involution on a hyperelliptic curve, although we don't use the machinery of hyperelliptic curves here.

```
> A<u,v> := AffineSpace(Rationals(),2);
> f := v^5 + 2*v^3 + 5;
> C := Curve(A,u^2 - f);
> phi := Automorphism(C,[-u,v]);
> Type(phi);
MapAutSch
> phi;
Mapping from: Crv: C to Crv: C
with equations :
-u
v
and inverse
-u
v
```

In this case we clearly know the inverse map in advance. We can make an automorphism of C as follows.

```
> psi := map< C -> C | [-u,v],[-u,v] >;
> psi eq phi;
true
> Type(psi);
MapSch
```

The map ψ is fine, but it is not of the same type as ϕ. We make the type change, if desired, as follows.

```
> bool,psi1 := IsAutomorphism(psi);
> bool;
true
> Type(psi1);
MapAutSch
```

Example H112E41

A standard Gröbner basis exercise is to test particular examples of the Jacobian conjecture. This states that a polynomial map of the plane is invertible if (and only if) its jacobian determinant is everywhere nonzero. The problem here is to calculate the conjectured inverse polynomial map.

```
> A<u,v> := AffineSpace(Rationals(),2);
> f := u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3;
> g := u + v^2 + 1;
> J := JacobianMatrix([f,g]);
> Determinant(J);
```

```
-1
> m := map< A -> A | [f,g] >;
> m;
Mapping from: Aff: A to Aff: A
with equations :
u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3
u + v^2 + 1
> IsAutomorphism(m);
true
> m;
Mapping from: Aff: A to Aff: A
with equations :
u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3
u + v^2 + 1
and inverse
-u^2 + 2*u*v^3 - 6*u*v + 10*u - v^6 + 6*v^4 - 10*v^3 - 9*v^2 + 31*v - 26
u - v^3 + 3*v - 5
> Type(m);
MapSch
> Inverse(m);
Mapping from: Aff: A to Aff: A
with equations :
-u^2 + 2*u*v^3 - 6*u*v + 10*u - v^6 + 6*v^4 - 10*v^3 - 9*v^2 + 31*v - 26
u - v^3 + 3*v - 5
and inverse
u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3
u + v^2 + 1
```

The automorphism test returns two values. The first, `true`, confirms that the map m is an automorphism. In doing this test, MAGMA computes the inverse and stores it with m. The second is the same map m, now with its inverse computed, but with the type of an automorphism.

```
> _,maut := IsAutomorphism(m);
> maut;
Mapping from: Aff: A to Aff: A
with equations :
u^3 + 3*u^2*v^2 + 3*u^2 + 3*u*v^4 + 6*u*v^2 + v^6 + 3*v^4 + v + 3
u + v^2 + 1
and inverse
-u^2 + 2*u*v^3 - 6*u*v + 10*u - v^6 + 6*v^4 - 10*v^3 - 9*v^2 + 31*v - 26
u - v^3 + 3*v - 5
> Type(maut);
MapAutSch
```

112.14.6.1 Affine Automorphisms

The first constructor below checks that the proposed map is indeed an automorphism by computing an inverse for the map determined by the arguments. It is a potentially expensive test. The other constructors are all either clearly automorphisms or else require only very simple tests.

`Automorphism(A,F)`

The automorphism of the affine space A determined by the sequence of functions F defined on A.

`Automorphism(A,M)`

The linear automorphism of the affine space A determined by the entries of the matrix of base ring elements M acting from the right on points.

`Translation(A,p)`

The translation map of the affine space A taking the rational point p to the origin.

`PermutationAutomorphism(A, g)`

`Automorphism(A, g)`

The automorphism of the affine space A that permutes its coordinates according to the permutation g.

Example H112E42

Permutations are easy to create as elements of the symmetric group. The symmetric group used must act on the set of n points, where n is the dimension of the affine space, even if it is only intended to permute a few of the coordinates.

```
> A := AffineSpace(Rationals(),5);
> g := SymmetricGroup(5) ! (1,2,3);
> f := PermutationAutomorphism(A,g);
> p := A ! [1,2,3,4,5];
> f(p);
(2, 3, 1, 4, 5)
```

`Automorphism(A,p)`

The automorphism which takes the first coordinate x of the affine space A to $x + p$. The polynomial p must be a function on A which does not involve x.

`AffineDecomposition(f)`

If f is an affine linear endomorphism, that is, an automorphism of some affine space defined by polynomials of degree at most 1, this returns a linear endomorphism ℓ and a translation t such that, in MAGMA notation, `f eq l * t`.

Example H112E43

In this example we make an affine linear map by composing a linear map and a translation. Then we promptly decompose it into these components.

```
> A<x,y,z> := AffineSpace(Rationals(),3);
> f := Automorphism(A,2*y+3*z) * Translation(A,A ! [2,3,5]);
> l,t := AffineDecomposition(f);
> l,t;
Mapping from: Aff: A to Aff: A
with equations :
x + 2*y + 3*z
y
z
and inverse
x - 2*y - 3*z
y
z
Mapping from: Aff: A to Aff: A
with equations :
x - 2
y - 3
z - 5
and inverse
x + 2
y + 3
z + 5
> f eq l * t;
true
```

Note that the composition `l * t` in MAGMA means that the map ℓ is applied first followed by t.

```
> p := A ! [1,2,3];
> f(p);
(12, -1, -2)
> t(l(p));
(12, -1, -2)
```

`NagataAutomorphism(A)`

This intrinsic returns the Nagata automorphism in a standard form:

$$(u, v, w) \mapsto (-u^2w^3 - 2uv^2w^2 - 2uvw + u - v^4w - 2v^3, uw^2 + v^2w + v, w).$$

Recall that this is an automorphism of affine 3-space A which is not known to be tame, that is, admits no known factorisation into automorphisms of the types listed above.

`Projectivity(A,M)`

The restriction to the affine space A of the linear automorphism of its projective closure determined by the matrix M. Note that this map is not usually regular on A but it is an isomorphism where it is defined.

Example H112E44

Most projectivities on an affine space are not regular maps. By definition, the equations which define them are naturally rational polynomials. That is the reason for naming the variables in the field of fractions of the coordinate ring of A in the following code.

```
> k := FiniteField(23);
> A<x,y,z> := AffineSpace(k,3);
> M := Matrix(k,4,4,[1,2,3,-4,2,3,5,6,3,4,5,9,4,5,6,0]);
> phi := Projectivity(A,M);
> KA<u,v,w> := Parent(x/y);
> phi;
Mapping from: Aff: A to Aff: A
with equations :
(6*u + 12*v + 18*w + 22)/(u + 7*v + 13*w)
(12*u + 18*v + 7*w + 13)/(u + 7*v + 13*w)
(18*u + v + 7*w + 8)/(u + 7*v + 13*w)
and inverse
(11*u + 15*v + 5)/(u + 20*w + 2)
(9*u + 16*v + w + 12)/(u + 20*w + 2)
(12*u + 15*v + 3*w + 16)/(u + 20*w + 2)
```

Notice that the inverse of ϕ has also been computed. In fact, ϕ has been returned as an automorphism even though it is not regular.

```
> Type(phi);
MapAutSch
> IsRegular(phi);
false
```

112.14.6.2 Projective Automorphisms

As in the case of affine spaces, a version of the automorphism constructor is provided which takes a sequence of polynomials as second argument. It is included mainly for completeness whereas the other constructors are more appropriate for constructing automorphisms in the contexts in which they often arise.

Projective automorphisms (which are regular) are always linear so they have an associated matrix with respect to the basis of monomials. A function is provided to retrieve this matrix, and conversely automorphisms may be created using matrices. In fact, if the projective space is defined over a finite field, then the automorphism group can be computed (as a group in MAGMA) and its elements can be realised as matrices.

Also included are functions for creating a nonregular birational automorphism of projective space, the standard *quadratic transformation*. When working in the plane, this together with linear automorphisms generates the group of birational automorphisms. An example of factorising a birational automorphism in this group is given.

`Automorphism(P,F)`

The automorphism of the projective space P determined by the sequence of polynomials F defined on P.

`Matrix(f)`

The matrix corresponding to the linear automorphism f of a projective space.

`Automorphism(P,M)`

The linear automorphism of the projective space P determined by the entries of the matrix of base ring elements M acting on the left of coordinates.

`Aut(P)`

The parent of automorphisms of the projective space P.

`AutomorphismGroup(P)`

The automorphism group of the projective space P together with a map from this group to the set of automorphisms of P, that is, the parent of such automorphisms. The space P must be defined over a finite field for this intrinsic. Note that currently the group returned is a general linear group rather than the projectivised version. This will be changed in the future, but in any case does not create much confusion.

Example H112E45

When a projective space is defined over a finite field, then its automorphism group can be realised as a group of matrices in a natural way. First we see how to use standard intrinsics to realise the correspondence between matrices and linear automorphisms of projective space.

```
> P<x,y,z> := ProjectiveSpace(GF(5),2);
> phi := Automorphism(P,[x+y,y,z]);
> M := Matrix(phi);
> M;
[1 0 0]
[1 1 0]
[0 0 1]
> Automorphism(P,M) eq phi;
true
```

Since P is a projective space defined over a finite field, we can actually work with a group which is isomorphic to its automorphism group. The map m computed below maps matrices to automorphisms and conversely its inverse constructs a matrix from an automorphism.

```
> G,m := AutomorphismGroup(P);
> G;
```

```
GL(3, GF(5))
> m;
Mapping from: GrpMat: G to Parent structure for automorphisms of P
> phi eq m(Transpose(M));
true
> Transpose(phi @@ m);
[1 0 0]
[1 1 0]
[0 0 1]
```

The parent of automorphisms is also an object in MAGMA. It can be created using `Aut(P)`.

```
> Aut(P);
Set of all automorphisms of P
> Aut(P) eq Codomain(m);
true
```

`TranslationOfSimplex(P,Q)`

The unique automorphism of the n-dimensional projective space P taking the $n+2$ standard simplex points $(1:0\ldots:0),\ldots$ and $(1:1\ldots:1)$ to the points of the sequence Q. The sequence Q must comprise $n+2$ linearly independent rational points of P.

`Translation(P,Q)`

This function returns an automorphism which translates the standard coordinate points to the points of the sequence Q. The sequence Q must comprise $n+1$ linearly independent rational points of the projective space P where n is the dimension of P. This intrinsic puts no condition on the usual final point $(1:1\ldots:1)$ of the standard simplex. In other words, it creates the automorphism of P by the matrix having the coordinates of the $n+1$ points of Q as its columns. This automorphism is not uniquely determined since $\mathrm{PGL}(n,k)$ is $n+2$ transitive.

`Translation(P,p,q)`

A choice of linear automorphism of the projective space P which takes the rational point p to the rational point q.

`Translation(X,p)`

A choice of linear automorphism of the projective space P taking the point $(0:\ldots:0:1)$ to the rational point p.

Example H112E46

The intrinsic `Translation(X,p)` works in both the affine and the projective context. For an affine scheme, it makes the translation which moves the point p to the origin.

```
> A<u,v> := AffineSpace(Rationals(),2);
> Translation(A,A![1,2]);
Mapping from: Aff: A to Aff: A
with equations :
u - 1
v - 2
and inverse
u + 1
v + 2
```

In the projective case, the resulting translation moves the point p to the image of the origin on the first affine patch. When the point p lies on the first affine patch, then the translation is the obvious one. But when it doesn't a permutation of the coordinates is made first.

```
> P<x,y,z> := ProjectiveSpace(Integers(),2);
> p := P ! [3,2,1];
> f := Translation(P,p);
> f;
Mapping from: Prj: P to Prj: P
with equations :
-x + 3*z
-y + 2*z
z
and inverse
-x + 3*z
-y + 2*z
z
> f(p);
(0 : 0 : 1)
> p := P ! [0,1,0];
> f := Translation(P,p);
> f(p);
(0 : 0 : 1)
> f;
Mapping from: Prj: P to Prj: P
with equations :
-x
-z
y
and inverse
x
-z
y
```

`QuadraticTransformation(P)`

`QuadraticTransformation(P,Q)`

The first function is the standard quadratic transformation of projective space P taking its coordinates to their reciprocals, that is $(x : y : \ldots) \mapsto (1/x : 1/y : \ldots)$. The second conjugates the standard map with a translation of the points of Q to the standard basis vectors. The sequence Q must comprise $n + 1$ linearly independent rational points of P where n is the dimension of P.

`QuadraticTransformation(X)`

`QuadraticTransformation(X,Q)`

The *birational* pullback of the projective scheme X by the quadratic transformation. In the first intrinsic the transformation used is `QuadraticTransformation(P)` while in the second, the transformation is `QuadraticTransformation(P,Q)` where P is the ambient projective space of X. Thus Q must comprise $n+1$ linearly independent rational points of P where n is the dimension of P. Exceptional components in the total pullback of X are removed.

Example H112E47

This example shows how to factorise a simple Cremona transformation. (The reader who knows something about this will note that in this example we take no account of the Nöther–Fano inequalities nor do we analyse infinitely near points. In fact, we are rather lucky to be able to complete the factorisation.)

```
> k := RationalField();
> P<x,y,z> := ProjectiveSpace(k, 2);
> funs := [ 2/3*x^2*y^2 + 2/3*x^2*y*z + x*y^2*z + x*y*z^2,
>     1/3*x^2*y^2 + 4/3*x^2*y*z + x^2*z^2 + 1/2*x*y^2*z + 1/2*x*y*z^2,
>     2/9*x^2*y^2 + 2/3*x^2*y*z + 2/3*x*y^2*z + x*y*z^2 + 1/2*y^2*z^2 ];
> g := map< P -> P | funs >;
> FunctionDegree(g);
4
> RationalPoints(BaseScheme(g));
{@ (3/4 : -1 : 1), (0 : 1 : 0), (0 : 0 : 1), (-3/2 : -1 : 1), (1 : 0 : 0) @}
```

Our aim is to precompose g with quadratic transformations which will reduce the degree of its defining polynomials, currently 4. When the function degree is reduced to 1 the factorisation is complete since the inverse sequence of quadratic transformations will comprise a factorisation of g up to a translation. (Draw the diagram of maps!) That will be done at the end. First a quadratic transformation in the three coordinate points is made since these points are the simplest appearing in the list of base points. (The complete mathematical theory works hard to make the choice of map here the right one: clearly no hard work has been done here in making the choice of coordinate points.)

```
> std_quad := QuadraticTransformation(P);
> g1 := std_quad * g;
> // (Expand and) extend the map to its maximal domain:
```

```
> g1:=Extend(g1);
> FunctionDegree(g1);
2
> S := BaseScheme(g1);
> RationalPoints(S);
{@ (4/3 : -1 : 1), (-2/3 : -1 : 1), (-2/3 : 0 : 1) @}
> HasPointsOverExtension(S);
false
```

Good! With only three non-collinear points of indeterminacy (the final line makes sure there aren't further points defined over some extension of the base ring) we are only one step away from completing the factorisation. We make a quadratic transformation in these three noncollinear points.

```
> tr := Translation(P,[ p : p in $2 ]);
> quad := std_quad * tr;
> g2 := quad * g1;
> g2:=Extend(g2);
> FunctionDegree(g2);
1
```

So g is seen to be the composition of linear translations and standard quadratic transformations: g_2 is itself a linear translation. This sequence of maps is reconstructed in reverse order to get g. All the maps should be inverted, but note that the quadratic transformations are selfinverse so that this is rather easy. Of course, composing the maps in the order they were discovered above produces the birational inverse of g.

```
> f3 := f2 * g2
>     where f2 is f1 * std_quad
>     where f1 is std_quad * Inverse(tr);
> Expand(f3) eq g;
true
```

112.14.7 Scheme Graph Maps

In MAGMA V2.16, we have introduced an alternative to the usual `MapSch` for maps between *ordinary projective* schemes: a new type `MapSchGrph`, which we refer to as graph maps. These are objects whose defining data is the closure of the graph of a rational map, without explicit defining or inverse polynomials. This is a fairly traditional way to consider maps in algebraic geometry and the main motivation for their introduction is for divisor maps of invertible sheaves. These are naturally constructed in graph form and the further derivation of explicit defining polynomials can be quite time-consuming and lead to extremely high degree, ugly results.

As well as being more naturally constructible in certain situations, graph maps have advantages over `MapSch`s in a number of contexts.

1 A graph maps is automatically maximally defined, so `Extend` and alternative equations are unnecessary.

2 Computation of images of subschemes of the domain or of the inverse of a map go, in one way or another, through the graph of the map, so it is more efficient to already have it in graph form.

3 For an invertible graph map, separate inverse equations are not required. It is only necessary to record that it is invertible and consider the "reverse" of the graph.

If $f : X \to Y$ is a rational map, the closure of its graph G naturally lies in $X \times Y$. For computational ease, we take G as lying in the product projective space of the ambients of X and Y, Functionally, it is defined by a bihomogenous ideal in a polynomial ring with $n + m + 2$ variables, where n (resp. m) is the dimension of the ambient of X (resp. Y). There is a primitive basic user constructor described below. Graph maps are more naturally constructed and returned by specialised functions.

Graph maps have most of the functionality of `MapSch` maps including `IsInvertible` and `Expand`. Rather than repeat the list of all of the relevant intrinsics, we note here that the major functionality not currently available for them is

1 No defining or inverse defining polynomials.

2 No pointset map construction: it is not possible to ask for the image or preimage under a graph map of a point in a pointset over a proper extension of the base field. Neither are there images and preimages of rational functions.

3 Restriction to affine patchs of the domain or codomain is not possible (graph maps are only between projective schemes). However, restriction to a closed subscheme of the domain or codomain via `Restriction` is allowable as usual.

4 Any specialised `MapSch` functionality only available for maps between particular scheme types, like maps between curves.

Graph maps can be composed in the usual way for maps, but can not be mixed with `MapSch` maps in composition.

A further minor restriction has been built in for implementational efficiency. It is assumed for the domain X of a graph map that there is no coordinate hyperplane that contains some *but not all* of the irreducible components of X. In particular, there is no problem if X is irreducible.

A graph map f may be converted into normal `MapSch` with an intrinsic given below. If f is known invertible, this also computes inverse defining polynomials. It should be noted that for maps between complicated schemes, this often produces a `MapSch` with extremely high degree defining polynomials and a large base scheme where it is not defined. In such cases, the original `MapSchGrph` can be a functionally much more efficient representation.

`SchemeGraphMap(X, Y, I)`

`Saturated` BoolElt *Default :* `false`

X and Y are ordinary projective schemes with ambients $\mathbf{P}^m$ and $\mathbf{P}^n$ respectively. I is an ideal in an $n + m + 2$ variable polynomial ring R that should have the grevlex ordering. I should define the closure of the graph of a rational map from X to Y if R is identifed with the coordinate ring of the product projective space of $\mathbf{P}^m$ and

$\mathbf{P}^n$ such that the first $m+1$ variables correspond to the variables of the coordinate ring of $\mathbf{P}^m$ in the same order and the last $n+1$ variables correspond to those of $\mathbf{P}^n$ in the same order.

I should thus be bihomogeneous in the first $m+1$ and second $n+1$ variables and be large enough to define the graph of the map, though it doesn't have to be the maximal defining ideal. For example, if a map is explicitly given by defining polynomials $[F_0(x), \ldots, F_n(x)]$ where we use x_i and y_i to denote the variables of R corresponding to the domain and codomain variables, then the ideal generated by

$$y_i F_j(x) - y_j F_i(x) \quad \forall\ 0 \leq i, j \leq n$$

and the defining equations for X will define the graph if the defining polynomials give an everywhere defined map. We give an explicit example below. If there is a non-empty base scheme, it will be necessary to saturate the above ideal by one of the $F_i(x)$ that doesn't vanish on any component of the domain. It is then "domain" saturated.

A defining ideal like the above that *hasn't* been saturated by an $F_i(x)$ is usually not maximal and for functional purposes has to be saturated by an appropriate domain variable using `ColonIdeal`. This is performed internally. If the user already knows that I is domain saturated, he can set the parameter `Saturated` (default `false`) to `true` to avoid this.

This is a simple convenience function that performs practically no checks on the validity of the input data.

`SchemeGraphMapToSchemeMap(f)`

Converts the graph map f into a usual scheme map. As noted in the introduction, if f is a map between fairly complex schemes, this can be quite a computationally heavy procedure and can produce very large degree, non-sparse defining polynomials and the `MapSch` produced can have a large base scheme, even if f is defined everywhere on its domain. If f is known invertible (see below), inverse defining-polynomials are also added to the result.

`IsInvertible(f)`

As for the `MapSch` version, returns whether f is birationally invertible and, if so, also returns the inverse map. This records that f is known invertible internally, as the inverse map just has the graph reversed (also the graph ideal may need to be saturated with respect to a codomain variable, which will be performed here if it is determined that the map is invertible) so no inverse equations are added. The `HasKnownInverse` intrinsic for `MapSch`s, which returns whether the map has already been determined to be invertible, is also available.

Example H112E48

Let X be an elliptic curve, given as an intersection of quadrics in $\mathbf{P}^3$. Here we start with a map from X into $\mathbf{P}^4$, given by cubics, that is birational onto its image. We show how to convert this into a graph map, take restrictions, check for invertibility and various other things.

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Scheme(P3,[x*t-y*z, x^2+y^2-4*z^2+7*t^2]);
> P4<a,b,c,d,e> := ProjectiveSpace(Rationals(),4);
> mp_seq := [x^3,y^3,z^3,t^3,y*z*t]; // polys defining a map to P^4
> mp := map<X->P4|mp_seq>;
> // Will now define the graph map
> R<x1,x2,x3,x4,y1,y2,y3,y4,y5> := PolynomialRing(Rationals(),9,"grevlex");
> hm := hom<CoordinateRing(P3) -> R |[R.i : i in [1..4]]>; // usual map
> grI := ideal<R|[(R.(i+4))*hm(mp_seq[j])-(R.(j+4))*hm(mp_seq[i]):
>   i in [j+1..5] , j in [1..5]] cat [hm(b) : b in Basis(Ideal(X))]>;
> gr_mp := SchemeGraphMap(X,P4,grI); // the graph map
> // check that gr_mp does give mp
> mp eq SchemeGraphMapToSchemeMap(gr_mp);
true
> Y := gr_mp(X);
> Y eq mp(X);
true
> // take restrictions to the image
> gr_mp1 := Restriction(gr_mp,X,Y);
> mp1 := Restriction(mp,X,Y);
> // check that gr_mp1 still gives mp1
> mp1 eq SchemeGraphMapToSchemeMap(gr_mp1);
true
> boo := IsInvertible(gr_mp1);
> boo;
true
> // find the image and preimage of a point
> pt := X![2,0,1,0];
> ipt := gr_mp1(X![2,0,1,0]);
> iminv := ipt @@ gr_mp1;
> Dimension(iminv); Degree(iminv);
0
2
> // the preimage is just pt doubled
> Support(iminv);
{ (2 : 0 : 1 : 0) }
```

112.15 Tangent and Secant Varieties and Isomorphic Projections

The functions in this section relate to the isomorphic projection of schemes in higher-dimensional projective spaces down to lower-dimensional ones. This is achieved through finding points in the ambient projective space which don't lie on either the tangent or secant varieties of the scheme. These varieties are interesting in their own right and we provide functions to compute them as subschemes of the ambient space or to test if a given point lies on them.

112.15.1 Tangent Varieties

For a scheme X in affine or ordinary projective space over a field, the tangent variety TX is a subscheme of the ambient space whose set of closed points is the closure of the union of all tangent spaces of closed points of X. We do not worry if the TX that we construct is necessarily a reduced scheme or not. If X is non-reduced, TX probably won't be and will usually be of larger dimension than expected. If X is projective, the union of tangent spaces is already closed in the ambient space.

`TangentVariety(X)`

`PatchIndex` RngIntElt *Default :* 0

The scheme X must be affine or ordinary projective. If X is projective, the tangent variety can be computed projectively but it is usually quicker to compute the result for an affine patch and then take the projective closure. If the parameter `PatchIndex` is set to $i > 0$ then in the projective case the function will do this, using the ith standard affine patch (see `AffinePatch` on page 3521). This will give the correct result as long as *no component of X lies in the hyperplane complement of the patch.*

`IsInTangentVariety(X,P)`

The computation of the full tangent variety can be quite time consuming except in small dimensional ambient spaces. If the dimension of the ambient space is n, it is effectively calculated as the image of a subscheme in a $2n$-dimensional ambient under projection down to X's ambient space. However, this call gives a much faster way of testing if a particular point P in the ambient space lies in the tangent variety of the scheme X when X is projective.

Example H112E49

```
> P<x,y,z,t> := ProjectiveSpace(RationalField(),3);
> X := Scheme(P,[x*y+z*t,x^2-y^2+2*z^2-4*t^2]);
> Dimension(X);
1
> time TangentVariety(X);
Scheme over Rational Field defined by
x^8 + 4*x^6*y^2 + 25/4*x^6*z^2 - 25/2*x^6*t^2 + 44*x^5*y*z*t + 6*x^4*y^4 +
    25/4*x^4*y^2*z^2 - 25/2*x^4*y^2*t^2 + 27/2*x^4*z^4 - 193/8*x^4*z^2*t^2 +
```

```
    54*x^4*t^4 + 88*x^3*y^3*z*t + 275/2*x^3*y*z^3*t - 275*x^3*y*z*t^3 +
    4*x^2*y^6 - 25/4*x^2*y^4*z^2 + 25/2*x^2*y^4*t^2 - 67/2*x^2*y^2*z^4 +
    2025/4*x^2*y^2*z^2*t^2 - 134*x^2*y^2*t^4 + 11*x^2*z^6 + 22*x^2*z^4*t^2 -
    44*x^2*z^2*t^4 - 88*x^2*t^6 + 44*x*y^5*z*t - 275/2*x*y^3*z^3*t +
    275*x*y^3*z*t^3 + 50*x*y*z^5*t + 200*x*y*z^3*t^3 + 200*x*y*z*t^5 + y^8 -
    25/4*y^6*z^2 + 25/2*y^6*t^2 + 27/2*y^4*z^4 - 193/8*y^4*z^2*t^2 + 54*y^4*t^4
    - 11*y^2*z^6 - 22*y^2*z^4*t^2 + 44*y^2*z^2*t^4 + 88*y^2*t^6 + 2*z^8 +
    16*z^6*t^2 + 48*z^4*t^4 + 64*z^2*t^6 + 32*t^8
Time: 0.440
> time TangentVariety(X: PatchIndex := 4);
Scheme over Rational Field defined by
x^8 + 4*x^6*y^2 + 25/4*x^6*z^2 - 25/2*x^6*t^2 + 44*x^5*y*z*t + 6*x^4*y^4 +
    25/4*x^4*y^2*z^2 - 25/2*x^4*y^2*t^2 + 27/2*x^4*z^4 - 193/8*x^4*z^2*t^2 +
    54*x^4*t^4 + 88*x^3*y^3*z*t + 275/2*x^3*y*z^3*t - 275*x^3*y*z*t^3 +
    4*x^2*y^6 - 25/4*x^2*y^4*z^2 + 25/2*x^2*y^4*t^2 - 67/2*x^2*y^2*z^4 +
    2025/4*x^2*y^2*z^2*t^2 - 134*x^2*y^2*t^4 + 11*x^2*z^6 + 22*x^2*z^4*t^2 -
    44*x^2*z^2*t^4 - 88*x^2*t^6 + 44*x*y^5*z*t - 275/2*x*y^3*z^3*t +
    275*x*y^3*z*t^3 + 50*x*y*z^5*t + 200*x*y*z^3*t^3 + 200*x*y*z*t^5 + y^8 -
    25/4*y^6*z^2 + 25/2*y^6*t^2 + 27/2*y^4*z^4 - 193/8*y^4*z^2*t^2 + 54*y^4*t^4
    - 11*y^2*z^6 - 22*y^2*z^4*t^2 + 44*y^2*z^2*t^4 + 88*y^2*t^6 + 2*z^8 +
    16*z^6*t^2 + 48*z^4*t^4 + 64*z^2*t^6 + 32*t^8
Time: 0.040
> time IsInTangentVariety(X,P![1,2,3,4]);
false
Time: 0.000
```

112.15.2 Secant Varieties

For a scheme X in affine or ordinary projective space over a field, the secant variety SX is a subscheme of the ambient space whose set of closed points is the closure of the union of all lines joining distinct pairs of closed points of X (secants). Again, we do not worry if the SX that we construct is necessarily a reduced scheme or not. Note that the union of all secants is not necessarily a closed subset of the ambient space even when X is projective.

`SecantVariety(X)`

`PatchIndex` RngIntElt *Default :* 0

The scheme X must be affine or ordinary projective. In the projective case we construct SX by taking the projective closure of the result for an appropriate affine patch (intersecting every component of X). For simplicity, we currently only consider standard affine patchs so the function will fail for X projective if *it has components lying in every standard hyperplane.* As for `TangentVariety`, if the parameter `PatchIndex` is set to $i > 0$ then the ith standard affine patch will be the one used and this saves a little time, avoiding a search. Effectively, the computation consists of finding the image of a subscheme in a projection from a $2n + 1$ dimensional

ambient space down to the (n dimensional) ambient space of X and can be quite lengthy.

`IsInSecantVariety(X,P)`

Again as in the tangent variety case, if the scheme X is projective, this call gives a much faster way of testing if a given ambient point P lies in SX than computing the whole of SX with `SecantVariety`. To be precise, this call actually tells you whether or not P lies in the union of secants rather than its closure SX. Additionally, affine patches are not used, so the above restriction on validity doesn't apply.

Example H112E50

```
> P<a,b,c,d,e> := ProjectiveSpace(RationalField(),4);
> X := Scheme(P,[a*d + c*e, a*c + d*e,
>  a^2 - e^2, c^2*e - d^2*e,
>  b^2 + c*d + e^2]); // union of 3 irreducible curves
> Dimension(X);
1
> time SecantVariety(X : PatchIndex := 2);
Scheme over Rational Field defined by
a^4*b^2 + a^4*e^2 - a^3*c^2*e - a^3*d^2*e + a^2*b^4 + 2*a^2*b^2*c*d +
    1/4*a^2*c^4 + 1/2*a^2*c^2*d^2 + 1/4*a^2*d^4 - a^2*e^4 + a*c^2*e^3 +
    a*d^2*e^3 - b^4*e^2 - 2*b^2*c*d*e^2 - b^2*e^4 - 1/4*c^4*e^2 -
    1/2*c^2*d^2*e^2 - 1/4*d^4*e^2
Time: 26.890
> Dimension($1);
3
> time IsInSecantVariety(X,P![0,1,0,-3,0]);
true
Time: 0.000
```

112.15.3 Isomorphic Projection to Subspaces

The aim of the functions here is to try to find embeddings of projective schemes which lie in high-dimensional ambient spaces into lower dimensional spaces via projection.

Let X be a scheme in ordinary projective space that is assumed to be *reduced* but not necessarily irreducible or non-singular, d the dimension of X and n the dimension of its ambient space P. The conditions imply that the tangent variety and secant variety of X are finite unions of subschemes of dimension at most $2d + 1$ (*cf* [Har77] IV.3), so while $n > 2d + 1$, we can find points in P lying in neither of these, unless possibly the base field is a finite field. For such a point, projection down to a hyperplane gives an isomorphism of X to its image (*cf. op. cit.*) and we can continue projecting down one dimension at a time until $n = 2d + 1$.

In fact, the tangent and secant varieties of the image of X will be their images under the projection also, which induces a finite map on them (it's quasi-finite since the inverse

image of a point in the image hyperplane is a line through the projecting point pt, which any (closed) subscheme not containing pt must intersect finitely). Hence their images have the same dimensions. If the maximum of these dimensions is $tsd \leq 2d+1$ we can actually project down to a subspace of dimension tsd. Further, the fact that the tangent and secant images correspond under projection shows that if we find a linear change of variables for the coordinates of P, $(x_1, \ldots, x_n) \to (y_1, \ldots, y_n)$, such that $y_1, \ldots, y_{tsd}$ are a set of Noether normalising variables for the defining ideal of the union of the tangent and secant varieties in the coordinate ring of P, then we can project X isomorphically down to the tsd dimensional linear subspace $y_{tsd+1} = 0, \ldots, y_n = 0$ directly.

This would be the most elegant solution. However, in practise it involves computing the full tangent and secant varieties which can be extremely time-consuming once we are in dimensions above around 3 or 4. By contrast, checking if a random point of P lies in the secant or tangent variety is reasonably fast. Therefore, we choose to follow the method of picking random points to project down one dimension at a time and finish when we reach dimension $2d+1$ (when the secant variety generally fills the ambient space anyway).

`IsomorphicProjectionToSubspace(X)`

Verbose `IsoToSub` *Maximum :* 1

As described above, this function projects the scheme X isomorphically down to a linear subspace of its ambient space P of dimension $2d+1$, if $2d+1 < n$. In the case of finite base fields or infinite ones of small characteristic, the process may stop before reaching this dimension but this should be very rare (difficulties finding random points outside the secant/tangent varieties). The return values consist of the image scheme, with the subspace as its ambient space, and the explicit map taking X to this image.

`EmbedPlaneCurveInP3(C)`

Verbose `EmbCrv` *Maximum :* 1

This function embeds a plane curve C as a non-singular projective curve in ordinary 2- or 3-dimensional projective space over the base field. This is effected by first using the function field machinery to give a non-singular projective embedding and then projecting down to a 3-dimensional subspace if necessary. In rare cases with small finite base fields or in small characteristic, the final embedding may still be into a higher-dimensional ambient space. The image scheme is returned along with the mapping of C to it.

Example H112E51

We will embed a genus 5 plane curve into $P3$.

```
> P<x,y,z> := ProjectiveSpace(RationalField(),2);
> f := x^5+x^2*y^3+y^2*z^3+x^2*z^3-y*z^4-z^5;
> C := Curve(P,f);
> Genus(C);
5
```

```
> SetVerbose("IsoToSub",1);
> SetVerbose("EmbCrv",1);
> C1, mp := EmbedPlaneCurveInP3(C);
Curve of genus 5.
Mapping to 4-space by canonical divisor map
Map was an embedding (non-hyperelliptic curve).
Beginning projection to 3-space.
Projection from dimension 4 to dimension 3:
Finding good projection point...
Performing projection...
Time: 0.002
> P1 := AmbientSpace(C1);
> AssignNames(~P1,["a","b","c","d"]);
> C1;
Scheme over Rational Field defined by
a^2*d^3 + a*b^2*c*d - a*b*c^2*d + a*b*d^3 + a*c^2*d^2 - a*d^4 + b^5 + b^2*c^3 -
    b^2*c*d^2 - b^2*d^3 + 2*b*d^4 - c^2*d^3 - d^5,
a^2*c*d + a*b^3 + a*c^3 - a*c*d^2 + b*c*d^2 - c*d^3,
a*b^2*d - a*b*d^2 + a*c^2*d + b^3*c + c^4 - c^2*d^2,
a^3*d + a^2*c^2 - a^2*d^2 + a*b*d^2 - a*d^3 + b^2*c*d,
a^2*b - c^2*d
> Dimension(C1);
1
> ArithmeticGenus(C1);
5
> IsNonSingular(C1);
true
```

112.16 Linear Systems

Let $f_1, \ldots, f_r$ be homogeneous polynomials of some common degree d on some projective space $\mathbf{P}$ defined over a field. The set of hypersurfaces

$$a_1 f_1 + \ldots + a_r f_r = 0$$

where the a_is are elements of the base field of $\mathbf{P}$ is an example of a linear system. This can be thought of as being the vector space of elements $(a_1, \ldots, a_r)$ or even the projectivisation of that space (since multiplying the equation above by a constant doesn't change the hypersurface it defines and the equation $0 = 0$ doesn't define a hypersurface at all). The same is true if $f_1, \ldots, f_r$ are a finite collection of polynomials defined on some affine space.

All linear systems in MAGMA arise in a similar way to the above example. It doesn't matter whether the linear system is considered to be the collection of hypersurfaces or the collection of homogeneous polynomials or the vector space of coefficients; MAGMA allows each of these interpretations and the distinction is blurred in the text below. One should note that linear systems in MAGMA are being used in a very elementary way: compare

with the discussion on plane conics and cubics in the first two chapters of Reid's Student Text [Rei88].

Immediate applications of linear systems arise because of their close relationship to maps (consider the map to an $r-1$-dimensional projective space defined by the polynomials $f_1, \ldots, f_r$) and their application to the extrapolation a scheme of some particular degree from a set of points lying on it or some subscheme of it.

More ambitious interpretations, as the zero-th coherent cohomology group of an invertible sheaf for instance, cannot be realised explicitly in MAGMA except inasmuch as the user can understand input and output easily in these terms. There is no analysis of linear systems on general schemes and so, in particular, no analysis of exact sequences of cohomology groups.

We give a brief description of the way in which linear systems work in MAGMA, an approach which echoes the more general definition. The *complete linear system on* $\mathbf{P}$ *of degree* d is the collection of all homogeneous polynomials of degree d on $\mathbf{P}$, or equivalently, the degree d hypersurfaces they define. MAGMA does not consider this to be a unique object: each time such a system is created, a completely new object will be created distinct from any previous creation. Its major attributes include a particular basis of degree d polynomials, which is always the standard monomial basis, and a vector space whose vectors correspond to the coefficients of a polynomial with respect to this basis. The vector space is called the *coefficient space* of the linear system. There are comparison maps: one to produce the vector of coefficients of polynomials with respect to the given basis and one to create a polynomial from a vector of coefficients. Most questions involving the analysis of linear systems are translated into the linear algebra setting, solved there and then translated back.

A general linear system corresponds to some vector subspace of the coefficient space of a complete linear system. The correspondence between vectors of coefficients and polynomials are computed at the level of the complete systems so that any two subsystems interpret coefficient vectors with respect to the same basis of polynomials.

112.16.1 Creation of Linear Systems

In practice, linear systems are not often created explicitly by hand. Typically, the complete linear system of all hypersurfaces of a given degree is created and then restricted by the imposition of geometrical conditions. For example, such a condition could require that all hypersurfaces pass through a particular point.

The creation methods below are split into three classes: (i) explicit initial creation methods; (ii) methods of imposing geometrical conditions; and (iii) creation of subsystems by nominating specific technical data calculated in advance by the user.

112.16.1.1 Explicit Creation

Initially, we present three methods by which a linear system can be created. The complete linear system of degree d whose sections are all monomials of that degree has a special creation function. Alternatively, a sequence of monomials of some common degree can be specified to generate the sections of a linear system. The third constructor is useful for

calculating the images of maps and has been seen before: given a scheme S and a map f it calculates the linear system of hypersurfaces which contain $f(S)$.

`LinearSystem(P,d)`

The complete linear system on the affine or projective space P of degree d. In the projective case, this is the space of all homogeneous polynomials of degree d on P, whereas in the affine case it includes all polynomials of degree no bigger than d. The integer d must be strictly positive.

`LinearSystem(P, d)`

The complete linear system on the affine or projective space P of multi-degree d. The length of d must be the number of gradings of P, i.e. one degree for each grading. In the projective case, this is the space of all homogeneous polynomials of degree d on P, whereas in the affine case it includes all polynomials of degree no bigger than d. The integers in d must be strictly positive.

`LinearSystem(P,F)`

If P is a projective space and F is a sequence of homogeneous polynomials all of the same degree defined on P, or if P is an affine space and F is a sequence of polynomials defined on P this returns the linear system generated by these polynomials. If the polynomials in F are linearly independent they will be used as a basis of the sections of the resulting linear system, otherwise a new basis will be computed.

`MonomialsOfWeightedDegree(X, D)`

Return the monomials in the coordinate ring of the ambient of X having degree $D[i]$ with respect to the ith grading of the ambient of X.

Example H112E52

In this example we construct two linear systems on a projective plane. Although they are created in slightly different ways, MAGMA recognises that they are the same. It does the computation as a subspace equality test in the corresponding 'coefficient spaces'.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> L := LinearSystem(P,1);
> K := LinearSystem(P,[x+y,x-y,z+2*z+3*y]);
> L eq K;
true
```

`ImageSystem(f,S,d)`

The linear system on the codomain of the map of schemes f consisting of degree d hypersurfaces which contain $f(S)$. An error is reported if the scheme S does not lie in the domain of f.

Example H112E53

This example demonstrates how one can use an intrinsic based on linear systems, `ImageSystem`, to find the equations of images of maps. The point is that sometimes the usual Gröbner basis can be very difficult, so if one is interested in the equations of low degree then the linear algebra computation might be more convenient.
The curve C has one singularity analytically equivalent to the cusp $u^2 = v^5$ so is well-known to have genus 4.

```
> Q := RationalField();
> P<x,y,z> := ProjectiveSpace(Q,2);
> C := Curve(P,x^5 + y^4*z + y^2*z^3);
```

The canonical embedding of C is therefore given by four conics having a common tangent with the curve at its singularity.

```
> P3<a,b,c,d> := ProjectiveSpace(Q,3);
> phi := map< P -> P3 | [x^2,x*y,y^2,y*z] >;
```

Unless C is hyperelliptic, its canonical image will be the complete intersection of a conic and a cubic in $\mathbf{P}^3$.

```
> IC2 := Image(phi,C,2);
> IC3 := Image(phi,C,3);
> X := Intersection(IC2,IC3);
> Dimension(X);
1
> IsNonsingular(X);
true
> MinimalBasis(X);
[ a*c - b^2, a^2*b + c^2*d + d^3 ]
```

In this case the Gröbner basis of X has six elements so it is not so helpful for human comprehension. (Compare this with [Hartshorne, IV, Example 5.2.2].)

112.16.1.2 Geometrical Restrictions

Consider the following example. Suppose that L is a linear system on the projective plane whose sections are generated by the monomials x^2, xy, yz and let $p = (1 : 0 : 0)$. The phrase '*one imposes the condition on sections of L that they pass through the point p*' refers to the construction of the subsystem of L, all of whose hypersurfaces pass through p. Explicitly, this involves solving the linear equation in a,b,c obtained by evaluating the equation

$$ax^2 + bxy + cyz = 0$$

at the point p. In this example, the equation is $a = 0$ and the required subsystem is the one whose sections are generated by xy and yz.

The functions described in this section all determine a linear subsystem of a given linear system by imposing conditions on the sections of that system.

`LinearSystem(L,p)`

`LinearSystem(L,S)`

Given a point p or a sequence S of points, create the subsystem of the linear system L comprising those hypersurfaces of L which pass through p or the points of S.

`LinearSystem(L,p,m)`

Create the subsystem of the linear system L comprising hypersurfaces which pass through the point p with multiplicity at least m.

Example H112E54

In this example we make some subsystems of linear systems by imposing conditions at points. In the first example, we construct the family of all curves having singularities with prescribed multiplicities at prescribed points. See Chapter 114 for functions which apply to curves.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> L := LinearSystem(P,6);
> p1 := P ! [1,0,0];
> p2 := P ! [0,1,0];
> p3 := P ! [0,0,1];
> p4 := P ! [1,1,1];
> L1 := LinearSystem(L,p1,3);
> L2 := LinearSystem(L1,p2,3);
> L3 := LinearSystem(L2,p3,3);
> L4 := LinearSystem(L3,p4,2);
> #Sections(L4);
7
> C := Curve(P,&+[ Random([1,2,3])*s : s in Sections(L4) ]);
> IsIrreducible(C);
true
> Genus(C);
0
```

In other words, $L4$ parametrises a six-dimensional family of rational plane curves. (At least, the general element of $L4$ is the equation of a rational plane curve — there are certainly degenerate sections which factorise so don't define an irreducible curve at all.) It would be nice to be able to parametrise one of these curves. The problem is that we need to choose a general one in order that it be irreducible, but on the other hand we have very little chance of finding a rational point of a general curve. However, since this family is nice and big, we can simply impose another point condition on it yielding a rational point on each of the restricted elements.

```
> p5 := P ! [2,1,1];
> L5 := LinearSystem(L4,P![2,1,1]);
> C := Curve(P,&+[ Random([1,2,3])*s : s in Sections(L5) ]);
> IsIrreducible(C);
true
> Genus(C);
0
```

```
> L<u,v> := ProjectiveSpace(Rationals(),1);
> phi := Parametrization(C, Place(C!p5), Curve(L));
> Ideal(Image(phi)) eq Ideal(C);
true
```

To illustrate another feature of imposing point conditions on linear systems, we use a point that is not in the base field of the ambient space. Linear systems on an ambient spaces are defined over its base field, so nonrational points impose conditions as the union of their Galois conjugates.

```
> A<x,y> := AffineSpace(FiniteField(2),2);
> L := LinearSystem(A,2);
> L;
Linear system on Affine Space of dimension 2 Variables : x, y
with 6 sections: 1 x y x^2 x*y y^2
> k1<w> := ext< BaseRing(A) | 2> ;
> p := A(k1) ! [1,w];
> p;
(1, w)
> LinearSystem(L,p);
Linear system on Affine Space of dimension 2
Variables : x, y
with 4 sections:
x^2 + 1
x*y + y
x + 1
y^2 + y + 1
> k2<v> := ext< BaseRing(A) | 3> ;
> q := A(k2) ! [1,v];
> LinearSystem(L,q);
Linear system on Affine Space of dimension 2
Variables : x, y
with 3 sections:
x^2 + 1
x*y + y
x + 1
```

Note the minimal polynomial of the y coordinate of the point $(1, w)$ is of degree 2 so is visible in the restricted linear system. On the other hand, v is of order 3 so it imposes more conditions on the linear system.

`LinearSystem(L,X)`

The subsystem of the linear system L comprising elements of L which contain the scheme X. The sections of this linear system is equal to the polynomials of the defining ideal of X whose homogeneous degree is the same as that of L.

`LinearSystemTrace(L,X)`

The trace of the linear system L on the scheme X which lies in the ambient space of L. This merely simulates the restriction of the linear system L on X by taking the sections of L modulo the equations of X. The result is still a linear system on the common ambient space.

Example H112E55

In this example, we restrict the linear system of cubics in space to a scheme, which is in fact a twisted cubic curve. (The intrinsic `Sections` is defined in Section 112.16.2.2.)

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> L := LinearSystem(P,3);
> X := Scheme(P,[x*z-y^2,x*t-y*z,y*t-z^2]);
> #Sections(L);
20
> L1 := LinearSystemTrace(L,X);
> #Sections(L1);
10
```

Taking the sections of L modulo the equations of X reduces the dimension of the space of sections by 10. Of course, there is a choice being made about which particular trace sections to use — in the end, the computation is that of taking a complement in a vector space of some vector subspace.
Since we recognise this scheme X as the image of a projective line, we can confirm that the result is correct. We make a map from the projective line to the space P which has image X. Then we check that the pullback L and $L1$ are equal on the projective line. In fact, since the linear system embedding the line is the complete system of degree 3, and L comprises degree 3 hypersurfaces on P, both pullbacks should give the complete linear system on the line of degree $3 \times 3 = 9$. (The intrinsic `Pullback` is defined in Section 112.16.3.)

```
> P1<u,v> := ProjectiveSpace(BaseRing(P),1);
> phi := map< P1 -> P | [u^3,u^2*v,u*v^2,v^3] >;
> Ideal(phi(P1)) eq Ideal(X);
true
> Pullback(phi,L) eq Pullback(phi,L1);
true
> Pullback(phi,L1);
Linear system on Projective Space of dimension 1
Variables : u, v
with 10 sections:
u^9 u^8*v u^7*v^2 u^6*v^3 u^5*v^4 u^4*v^5 u^3*v^6 u^2*v^7 u*v^8 v^9
```

112.16.1.3 Explicit Restrictions

`LinearSystem(L,F)`

The subsystem of the linear system L generated by the polynomials in the sequence F. An error results if the polynomials of F are not already sections of L. As before, the polynomials in F are used as a basis of the sections of the resulting linear system provided they are linearly independent, otherwise a new basis is computed.

`LinearSystem(L,V)`

The subsystem of the linear system L determined by the subspace V of the complete coefficient space of L. It is an error to call this if V is not a subspace of the coefficient space of L.

112.16.2 Basic Algebra of Linear Systems

This section presents functions for the following tasks: (i) assessing properties of the data type; (ii) geometrical properties of linear systems; (iii) linear algebra operations.

112.16.2.1 Tests for Linear Systems

`Ambient(L)`
`AmbientSpace(L)`

The projective space on which the linear system L is defined.

`L eq K`

Returns `true` if and only if the linear systems L and K are equal if considered as linear subsystems of some complete linear system. An error results if L and K lie in different complete linear systems.

`IsComplete(L)`

Returns `true` if and only if the linear system L is the complete linear system of polynomials of some degree.

`IsBasePointFree(L)`
`IsFree(L)`

Returns `true` if and only if the linear system L has no base points.

112.16.2.2 Geometrical Properties

`Sections(L)`

A sequence whose elements form basis of the sections of the linear system L. By definition, this is a maximal set of linearly independent polynomials which are elements of L.

`Random(LS)`

If the base field of LS admits random elements then this returns a random element of the space of sections in the linear system LS. If the base field is the rational field, then a section having small random rational coefficients is defined. Otherwise, if there is no random element generator for the base field, the zero section is returned.

`Degree(L)`

The degree of the sections of the linear system L.

`Dimension(L)`

The projective dimension of the linear system L. This is the maximal number of linearly independent sections of L minus 1.

`BaseScheme(L)`

The base scheme of the linear system L. This is simply the scheme defined by the sections of L. This function does not perform any tests on this scheme; it might be empty for example.

`BaseComponent(L)`

The hypersurface common to all the elements of the linear system L.

`Reduction(L)`

The linear system L with its codimension 1 base locus removed. In other words, the linear system defined by the sections of L after common factors are removed.

Example H112E56

If one tries to impose too many point conditions on a linear system, the general elements will no longer be irreducible. From a quick genus calculation one might think that it was possible to impose singularities on multiplicities 2, 3, 4 on projective curves of degree 6 to reveal rational curves — indeed $g = 10 - 1 - 3 - 6 = 0$ if the resulting curves are irreducible.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> L := LinearSystem(P,6);
> p1 := P ! [1,0,0];
> p2 := P ! [0,1,0];
> p3 := P ! [0,0,1];
> L1 := LinearSystem(L,p1,4);
> L2 := LinearSystem(L1,p2,3);
```

```
> L3 := LinearSystem(L2,p3,2);
> Sections(L3);
[ x^2*y^3*z, x^2*y^2*z^2, x^2*y*z^3, x^2*z^4, x*y^3*z^2,
    x*y^2*z^3, x*y*z^4, y^3*z^3, y^2*z^4 ]
> BaseComponent(L3);
Scheme over Rational Field defined by z
```

But notice that every section is divisible by z. So the curve $z = 0$ is in the base locus of this linear system, that is, it is contained in every curve in the linear system $L3$. The intrinsic `BaseComponent` identifies this component. The intrinsic `Reduction` creates a new linear system by removing this codimension 1 base locus, as is seen below. First, however, we look at the complete set of prime components of the base scheme and see that, while there is only one codimension 1 component which we already know, there is another component in higher codimension. When we reduce to remove the base component, this other piece of the base scheme remains, and other codimension 2 components also appear.

```
> MinimalPrimeComponents(BaseScheme(L3));
[
    Scheme over Rational Field defined by
    z,
    Scheme over Rational Field defined by
    x
    y
]
> L4 := Reduction(L3);
> Sections(L4);
[ x^2*y^3, x^2*y^2*z, x^2*y*z^2, x^2*z^3, x*y^3*z, x*y^2*z^2,
    x*y*z^3, y^3*z^2, y^2*z^3 ]
> MinimalPrimeComponents(BaseScheme(L4));
[
    Scheme over Rational Field defined by
    x
    y,
    Scheme over Rational Field defined by
    x
    z,
    Scheme over Rational Field defined by
    y
    z
]
> [ RationalPoints(Z) : Z in $1 ];
[
    {@ (0 : 0 : 1) @},
    {@ (0 : 1 : 0) @},
    {@ (1 : 0 : 0) @}
]
```

The linear system $L4$ has sections which are visibly those of $L3$ but with a single factor of z removed. It still has base locus but now that base locus comprises only points. Not surprisingly, it is exactly the three points which we imposed on curves in the first place.

`BasePoints(L)`

A sequence containing the basepoints of the linear system L if the base locus of L is finite dimensional.

`Multiplicity(L,p)`

The generic multiplicity of hypersurfaces of the linear system L at the point p.

112.16.2.3 Linear Algebra

`CoefficientSpace(L)`

The vector space corresponding to the linear system L whose vectors comprise the coefficients of the polynomial sections of L.

`CoefficientMap(L)`

The map from the polynomial ring that is the parent of the sections of the linear system L to the coefficient space of L. When evaluated at a polynomial f, this map will return vector of coefficients of f as a section of L.

`PolynomialMap(L)`

The map from the coefficient space of the linear system L to the polynomial ring that is the parent of the sections of L. When evaluated at a vector v, this map will return the polynomial section of L whose coefficients with respect to the basis of L are v.

`Complement(L,K)`

A maximal subsystem of the linear system L which does not contain any of the hypersurfaces of the linear system K.

`Complement(L,X)`

A maximal subsystem of the linear system L comprising hypersurfaces not containing X.

Example H112E57

In this example we show to define linear systems by referring to subspaces of a coefficient space. The explicit translation intrinsics between the linear algebra language and the linear system language let one 'see' what is happening in the background. We start by defining a linear system whose chosen sections are clearly not linearly independent.

```
> A<x,y> := AffineSpace(FiniteField(2),2);
> L := LinearSystem(A,[x^2-y^2,x^2,y^2]);
> VL := CoefficientSpace(L);
> VL;
KModule VL of dimension 2 over GF(2)
> W := sub< VL | VL.1 >;
> LinearSystem(L,W);
Linear system on Affine Space of dimension 2
Variables : x, y
with 1 section:
x^2
> phi := PolynomialMap(L);
> [ phi(v) : v in Basis(VL) ];
[
    x^2,
    y^2
]
```

Thus we see that MAGMA has chosen the obvious polynomial basis for the sections of L and disregarded the section $x^2 - y^2$.

`L meet K`

`Intersection(L,K)`

The linear system whose coefficient space is the intersection of the coefficient spaces of the linear systems L and K. An error is reported unless L and K lie in the same complete linear system.

`X in L`

Returns `true` if and only if the scheme X occurs among the hypersurfaces comprising the linear system L.

`f in L`

Returns `true` if and only if the polynomial f is a section of the linear system L. That is, `true` if and only if f is in the linear span of the basis of sections defining L.

`K subset L`

`IsSubsystem(L,K)`

Returns `true` if and only if the coefficient space of the linear system K is contained in that of the linear system L. An error is reported if L and K do not lie in a common linear system.

112.16.3 Linear Systems and Maps

The sequence of sections of a linear system may be used to construct a map from the projective space on which the sections are defined to another having the appropriate dimension. This is done directly using the `map` constructor as in Section 112.14. For example, if L is a linear system on some projective space P then the corresponding map can be created as follows.

```
> map< P -> Q | S >
>              where Q is ProjectiveSpace(BaseRing(P),#S-1)
>              where S is Sections(L);
```

There is not a proper inverse to this operation: there is no reason why a map should be determined by linearly independent polynomials. However, the system determined by the polynomials defining a map is still important. It is sometimes called the *homoloidal system* of the map.

`Pullback(f,L)`

The linear system f^*L on the domain of the map of schemes f where L is a linear system on the codomain of f. This requires care when f is not a regular map: it really produces the system of homaloids, that is, the substitution of the map equations into the linear system's sections.

112.17 Divisors

This section contains functionality for working with divisors on varieties (integral schemes defined over a field) of dimension greater than one. Currently, divisors can only be created on projective schemes X and there is also a restriction that X is ordinary projective for many of the less formal intrinsics that rely on the coherent sheaf code. In a number of places it is also required that a divisor D is Cartier (always true if X is non-singular), which we currently cannot check. There are also a number of intrinsics that are specific to surfaces.

Integral divisors are represented as differences of effective divisors, which are represented as subschemes of the scheme they live on. Factorisation into multiples of irreducibles can also be performed and the result is stored once calculated. It is also possible to work with **Q**-divisors. These are represented internally as factorisations with rational multiplicity of components.

The package of divisor functions is at and early stage and a number of the intrinsics are not as general as they could be and/or could be made more efficient. However, it is useful functionality that seems worth exporting now. There is much further work still to be done.

112.17.1 Divisor Groups

As for curves, there is a divisor group object associated to a variety X, which is the parent of all divisors on X. It is of type `DivSch`.

`DivisorGroup(X)`

The divisor group of variety X.

`Variety(G)`

The variety of the divisor group G.

`G1 eq G2`

True if and only if the divisor groups $G1$ and $G2$ are for the same variety.

112.17.2 Creation Of Divisors

Divisors are of type `DivSchElt`. Internally, an integral, effective divisor D on variety X is stored as an ideal which defines D as a subscheme of X. A general divisor is represented internally in partially factored form as a list of pairs of ideals and rational multiplicities $[\langle I_i, m_i\rangle]$ which represents the **Q**-rational (integral, if all m_i are integers) divisor $\sum_i m_i * D(I_i)$, where $D(I_i)$ is the effective divisor on X defined by the ideal I_i. An integral, effective divisor may also have a factorisation stored. The internal factorisation can change over time with the I_i being decomposed into products of larger ideals. When the I_i are all prime ideals, we say that the factorisation is a prime factorisation of D.

This section contains the basic creation functions for divisors.

`Divisor(X,f)`

`Divisor(X,f)`

`Divisor(X,f)`

These create the integral divisor on X defined by a single global element f. In the first two cases, f is an element of the function field (or the field of fractions of the coordinate ring of) the projective ambient of X. The divisor is non-effective (unless it is zero): the divisor of zeroes of f minus its divisor of poles. In the last case f should be a homogeneous polynomial in the coordinate ring of the ambient and the divisor is the effective divisor defined by the subscheme of X whose ideal is generated by the ideal of X and f.

`Divisor(X,Q)`

`Divisor(X,Y)`

`Divisor(X,I)`

`CheckSaturated`	BoolElt	*Default :* `false`
`CheckDimension`	BoolElt	*Default :* `false`
`UseCodimensionOnePart`	BoolElt	*Default :* `false`

These three intrinsics define an integral, effective divisor on projective variety X defined by an ideal or subscheme. For the first, Q is a sequence of elements in the coordinate ring of X and it is equivalent to passing the ideal generated by the ideal of X and Q. For the second, Y is a subscheme of X that should define an effective divisor on X. For the third, I is an ideal in the coordinate ring of the ambient of X, whose saturation J should contain the ideal of X. The effective divisor is given by the closed subscheme of X whose ideal is J.

`CheckSaturated` can be set to `true` in the third case if it is known that I is already saturated or in the second case if it is known that the ideal of Y is already saturated. `CheckDimension` can be set to `true` in any of the cases if it is known that the subscheme defining the divisor is of pure codimension 1 in X. Otherwise this condition is checked with the following exception. In the third case, if parameter `UseCodimensionOnePart` is set to `true` the non-codimension 1 part of the ideal is ignored in creating the divisor.

`HyperplaneSectionDivisor(X)`

Creates a divisor given by a hyperplane section of projective variety X.

`ZeroDivisor(X)`

The zero divisor on variety X.

`CanonicalDivisor(X)`

A canonical divisor on variety X. X must be ordinary projective and should be a Gorenstein scheme for this to give a correct result. It uses the canonical sheaf on X and the next intrinsic.

`SheafToDivisor(S)`

S is a coherent sheaf that should be invertible (locally free, rank 1) on variety X. S is then isomorphic to $L(D)$ for a (Cartier) divisor D (defined up to rational equivalence) on X. Returns such a divisor D which is effective if possible.

`RoundDownDivisor(D)`

For an integral divisor D just returns D. For a non-integral ($\mathbf{Q}$-rational) divisor with a factorisation into sum of rational multiples of prime components, returns the divisor of the integer multiple sum of primes given by rounding down all of the original rational coefficients.

`RoundUpDivisor(D)`

For an integral divisor D just returns D. For a non-integral ($\mathbf{Q}$-rational) divisor with a factorisation into sum of rational multiples of prime components, returns the divisor of the integer multiple sum of primes given by rounding up all of the original rational coefficients.

`FractionalPart(D)`

Returns $D-$ `RoundDownDivisor`(D).

`IntegralMultiple(D)`

Finds a positive integer N such that $E = N * D$ is an integral divisor. Returns E and N. Doesn't attempt to find the smallest possible N by analysing the full prime factorisation of D.

112.17.3 Ideals and Factorisations

Divisors are stored in ideal or factored form as described in the introduction to the last section. This section contains functions related to these representing structures.

`Ideal(D)`

Returns the defining ideal for an effective, integral divisor D.

`Support(D)`

The subscheme of the variety of effective **Q**-divisor D that gives its support.

`IdealOfSupport(D)`

The ideal in the coordinate ring of the ambient of the variety of the effective **Q**-divisor D that defines it's support.

`SignDecomposition(D)`

The decomposition of D into two effective divisors A and B such that $D = A - B$. A and B are returned. Note that they are not guaranteed to be relatively prime for this intrinsic.

`IdealFactorisation(D)`

Returns the current stored factorisation of D as a sequence of pairs of ideals and rational multiplicities.

`CombineIdealFactorisation(~D)`

Simplify the current ideal factorisation of D by combining terms with the same ideal.

`ComputeReducedFactorisation(~D)`

`ReducedFactorisation(D)`

Replace the ideal factorisation of D with an equivalent reduced factorisation where all ideals occurring are primary. The first intrinsic just does the replacement internally. The second intrinsic also returns the result.

`ComputePrimeFactorisation(~D)`

`PrimeFactorisation(D)`

Replace the ideal factorisation of D with an equivalent prime factorisation where all ideals occurring are prime. The first intrinsic just does the replacement internally. The second intrinsic also returns the result.

`Multiplicity(D,E)`

The multiplicity of prime divisor E in divisor D.

112.17.4 Basic Divisor Predicates

`IsZeroDivisor(D)`

Returns whether D is the zero divisor.

`IsIntegral(D)`

Returns whether D is an integral divisor. If this isn't immediately obvious from the current factorisation, will convert to a prime factorisation and try to combine terms.

`IsEffective(D)`

Returns whether D is an effective divisor. If this isn't immediately obvious from the current factorisation, will convert to a prime factorisation and try to combine terms.

`IsPrime(D)`

Returns whether D is a prime divisor.

`IsFactorisationPrime(D)`

Returns whether the current factorisation of D is a prime factorisation (i.e. all ideals occurring are prime).

`IsDivisible(D)`

Returns whether D is integral and divisible as an integral divisor by an integer $n > 1$. If so, also returns the maximum such n.

112.17.5 Arithmetic of Divisors

```
D1 + D2
D1 + D2
D1 + D2
D1 - D2
D1 - D2
D1 - D2
-D
```

Addition, subtraction and unitary minus on divisors. For addition and subtraction, one argument may be a toric divisor whose toric variety is the variety of the scheme divisor.

```
n * D
r * D
```

Multiplication of a divisor D by an integer n or rational number r. Note that the multiplication by r is the only current primitive method for constructing non-integral divisors.

```
D1 eq D2
```

Returns whether divisors $D1$ and $D2$ lie on the same variety and are equal.

112.17.6 Further Divisor Properties

More complicated predicates on divisors.

```
IsCanonical(D)
```

Returns whether D is a canonical divisor by testing whether its associated sheaf is isomorphic to the canonical sheaf. The variety of D must be ordinary projective here and should be Gorenstein.

```
IsAnticanonical(D)
```

Returns whether D is an anticanonical divisor by testing whether its associated sheaf is isomorphic to the dual of the canonical sheaf. The variety of D must be ordinary projective here and should be Gorenstein.

```
IsCanonicalWithTwist(D)
```

Returns whether D is the sum of a hypersurface divisor of degree d and a canonical divisor by testing whether its associated sheaf is isomorphic to a twist of the canonical sheaf. If so, also returns d. The variety of D must be ordinary projective here and should be Gorenstein.

`IsPrincipal(D)`

Returns whether D with variety X is a principal divisor and, if so, also returns an element f of the function field of the ambient of X such that $D = div(f)$. X should be ordinary projective and D a Cartier divisor here. Uses the Riemann-Roch space of D.

`IsLinearlyEquivalent(D,E)`

Returns whether two divisors D and E on variety X are linearly equivalent and, if so, also returns an element f of the function field of the ambient of X such that $D = E + div(f)$. Uses `IsPrincipal` for the difference between the two divisors, so X must be ordinary projective.

`BaseLocus(D)`

`IsBasePointFree(D)`

`IsMobile(D)`

The first intrinsic computes the base locus of the linear system $|[D]|$ (i.e. the reduced intersection of all effective divisors in the linear system) where $[D]$ is the round down of D . This uses the Riemann-Roch space of $[D]$, which means that this divisor has to be Cartier and the variety X of D has to be ordinary projective.

The second intrinsic returns whether this base locus is empty and the third whether it is of codimension at least two in X (i.e. there are no common divisor components to the full linear system). X and D obviously have to satisfy the same conditions.

`IntersectionNumber(D1,D2)`

$D1$ and $D2$ are divisors on a variety X which must be of dimension 2. One of the two divisors is assumed to be Cartier. Computes the intersection pairing number $D1.D2$.

`SelfIntersection(D)`

The intersection number of D with itself. D and X must satisfy the conditions for the last intrinsic.

`Degree(D)`

`Degree(D,H)`

D (resp. D and H) lies on a variety X of dimension 2. The first computes the intersection number of D with respect to a hyperplane divisor. The second computes the intersection number of D with H (so is just equivalent to `IntersectionNumber(D,H)`).

`IsNef(D)`

D should be a $\mathbf{Q}$-Cartier divisor on a projective surface X. Currently, it also has to be effective. Returns whether D is a nef divisor: i.e. whether it has non-negative intersection with all effective divisors on X.

`IsNefAndBig(D)`

D and X are as above. D must be effective. Returns whether D is a nef divisor AND has positive self-intersection.

`NegativePrimeDivisors(D)`

D and X again should satisfy the same conditions as in `IsNef`. Returns a sequence of prime divisor components of D which have negative intersection with D.

`ZariskiDecomposition(D)`

D and X again should satisfy the same conditions as in `IsNef`. Returns a pair of $\mathbf{Q}$-divisors P and N such that $D = P + N$, P is nef and N has negative-definite support (i.e. the intersection pairing on its prime components is negative definite).

112.17.7 Riemann-Roch Spaces

This section contains functions to compute and work with Riemann-Roch spaces for a divisor D. It is always assumed that X, the variety of D, is an ordinary projective space here. The Riemann-Roch space is the finite-dimensional subspace of the vector space (over the basefield) of rational functions on X consisting of zero and all $f \neq 0$ such that $D + div(f)$ is an effective divisor. Thus, the Riemann-Roch space of D is the same as the Riemann-Roch space of $[D]$, `RoundDownDivisor(D)`. Because the sheaf code is used (or a slight variant for non-effective divisors), it is also required that $[D]$ is Cartier for all relevant intrinsics.

`Sheaf(D)`

The invertible sheaf corresponding to the divisor class of divisor D. If D is not integral, its round down $[D]$ is used. This divisor must be Cartier and its variety X must be ordinary projective. Is effectively the same function as in the Sheaf package when D is effective and uses a slight variant otherwise.

`RiemannRochBasis(D)`
`RiemannRochSpace(D)`

The first returns a basis of the Riemann-Roch space of the round down $[D]$ of divisor D (as a sequence of elements in the function field F of the ambient of D's variety, X). The second returns the Riemann-Roch space as an abstract vector space V over the base field along with a map from V to F. X must be ordinary projective and $[D]$ Cartier. If D is effective, this is the same as using the coherent sheaf function that computes the associated invertible sheaf and Riemann-Roch basis. If D is non-effective a slight variant is used.

`RiemannRochCoordinates(f,D)`

Returns whether f can be coerced into the function field of the ambient of D's variety X and whether f then lies in the Riemann-Roch space of D. If so also returns the coordinates of f with respect to the basis of the Riemann-Roch space returned by `RiemannRochBasis(D)`. As usual, X must be ordinary projective and $[D]$, the round down of D, must be Cartier.

`IsLinearSystemNonEmpty(D)`

Returns whether there is an effective divisor linearly equivalent to D and, if so, returns such a divisor. Uses the Riemann-Roch space of D. The conditions on D and X, its variety, are as for the preceding intrinsics.

112.18 Isolated Points on Schemes

There is now experimental code to try to find isolated points of schemes in $\mathbf{A}(Q)^n$ that are defined by n or more equations. There is no restriction that the scheme itself must be of dimension 0, and in many applications, there is an uninteresting (or degenerate) variety of positive dimension, with the isolated points being the ones of interest.

The first technique to do this is to find points locally modulo some prime, at which the Jacobian matrix is of maximal rank. A separate procedure then lifts such points via a Newton method, and tries to identify them (via LLL) in a number field. If the degree is known (or suspected), this information can be passed to the lifting function, and otherwise it will try a generic method that applies LLL for degrees $3 \cdot 2^n$ and $4 \cdot 2^n$ as n increases, expecting to catch other solutions via a factorization into a smaller field (for instance, a degree 6 solution will show up in degree 8 as the expected minimal polynomial multiplied by a more-or-less random quadratic one).

The running time of such a process is often dominating by the searching for local points, and so various methods can be used to pre-condition the system. Firstly, variables which appear as a linear monomial in one of the equations can be (iteratively) eliminated. Already it is not clear what the best order is, so the user can specify it. In given examples, it is not untypical for later steps to take 10 times as long due to a different choice of eliminations, so some experimenting with this parameter can be useful. Secondly, resultants can be used to try to eliminate more variables, perhaps concentrating on those which appear to a small degree in the equations. By default, if a variable appears to degree 2 or less in all the equations, a resultant step will be applied. The resulting equations can be quite complicated (taking many megabytes to represent them), but it is still often faster to loop over p^{V-1} possible local points compared to p^V. The resultants can introduce extraneous solutions, which are checked when undoing the modifications to the system of equations. An error can occur if the variable elimination reduces the number of equations below the dimension.

Another concern for running time is in the order of the above operations. For instance, computing the resultants can be time-consuming in some examples, and perhaps doing so modulo p for the primes of interest might be a superior method in some cases. A second example is that recognising points over the number fields in question might be faster with

a reduced system and then undoing the resultants and linear eliminations in the number field, though the lifting might be slower in that case. The algorithm as implemented tries to handle the general case well.

Note that the scheme given to the IsolatedPoints routines might also have components of positive dimension, but the techniques here will only work to find the points that do not lie on them.

`LinearElimination(S)`

`EliminationOrder`	SeqEnum	*Default :* [ ]

Given a scheme, iteratively eliminate variables that appear strictly linearly (that is, as a monomial times a constant) in some equation. The `EliminationOrder` vararg allows an ordering to be specified. The results can vary drastically when the order is changed. The returned value is a map from the resulting scheme to the input scheme, with an inverse.

`IsolatedPointsFinder(S,P)`

`LinearElimination`	SeqEnum	*Default :* [ ]
`ResultantElimination`	SeqEnum	*Default :* [ ]
`FactorizationInResultant`	BoolElt	*Default :* `true`

Given a affine n-dimensional scheme defined over the rationals by at least n equations, and a sequence of primes, try to find liftable points of the scheme modulo the primes. The variables given in the `LinearElimination` vararg will be eliminated in that order. If this is non-empty, an automatic procedure will be applied. Similarly with `ResultantElimination`. Finally, by default, computed resultants have `SquarefreeFactorization` applied to them (and repeated factors removed), and this can be turned off.

`IsolatedPointsLifter(S,P)`

`LiftingBound`	RngIntElt	*Default :* 10
`DegreeBound`	RngIntElt	*Default :* 32
`OptimizeFieldRep`	BoolElt	*Default :* `false`
`DegreeList`	SeqEnum	*Default :* [ ]

Given a affine n-dimensional scheme defined over the rationals by at least n equations, and a sequence of finite field elements giving a point (on the scheme) whose Jacobian matrix is of maximal rank, attempt to lift the point via a Newton method and recognise it over a number field. The function returns `false`if a point was not found, and else returns both `true`and the point that was found.

The `LiftingBound` vararg determines how many lifting steps to use, with the default being 10, so that approximately precision $p^{2^{10}}$ is obtained. The `DegreeBound` is a limit on how high of a degree of field extension to check for solutions. The method used only checks for some of the degrees via LLL, as usually smaller degrees will

show up in factors. The `DegreeList` vararg can also be used for this, perhaps when the degree of the solution in question is known. The practical limit is likely around 50 in the degree. Finally, there is the `OptimizeFieldRep` boolean vararg, which determines whether the number field obtained will have its optimised representation computed.

`IsolatedPointsLiftToMinimalPolynomials(S,P)`

`LiftingBound`	RngIntElt	*Default :* 10
`DegreeBound`	RngIntElt	*Default :* 32
`DegreeList`	SeqEnum	*Default :* [ ]

This works as with `IsolatedPointsLifter`, but instead of trying to find a common field containing all the coordinates, simply returns a minimal polynomial for each one.

Example H112E58

This example follows an idea of Elkies to construct "large" integral points on elliptic curves. Let X, Y, A, B be polynomials in t, and let $Q(t)$ be quadratic. The idea is that

$$Q(t)Y(t)^2 = X(t)^3 + A(t)X(t) + B(t)$$

will have infinitely many specialisations with $Q(t)$ square (via solving a Pell equation), and thus yield, perhaps after scaling to clear denominators, integral points on the resulting curves. If the degree of A, B is small enough compared to that of X, the resulting specialisation will be quite notable. However, one must avoid the cases where the resulting discriminant $4A(t)^3 - 27B(t)^2$ is zero, as these points do not yield an elliptic curve. It turns out that such points contribute a positive-dimensional component to the solution space, and we simply want to ignore such solutions in general.
The first case of interest is when the degrees of (X, Y, A, B) are $(4, 5, 0, 1)$, solved by Elkies in 1988. One way of parametrising this, taking into account possible rational changes of the t-variable to simplify the system, is:

$$X(t) = t^4 + t^3 + x_2t^2 + x_1t + x_0, Q(t) = t^2 + q_1t + q,0$$

$$Y(t) = t^5 + y_3t^3 + y_2t^2 + y_1t + y_0, A(t) = a_0, B(t) = b_1t + b_0.$$

Then we get 12 equations from requiring that the 0th to 11th degree coefficients of $X^3 + AX + B - QY^2$ all vanish. It turns out that the desired solution is rational in this case, so that (almost) any prime will suffice. The solution can then be lifted – as a final step (particular to this problem), one would have to scale the resulting point so as to ensure that $Q(t)$ represents squares.

```
> K := Rationals();
> R<a0,b0,b1,q0,q1,x0,x1,x2,y0,y1,y2,y3> := PolynomialRing(K,12);
> _<t> := PolynomialRing(R);
> X := t^4+t^3+x2*t^2+x1*t+x0; Y := t^5+y3*t^3+y2*t^2+y1*t+y0;
> Q := t^2+q1*t+q0; A := a0; B := b1*t+b0;
> L := X^3+A*X+B-Q*Y^2;
```

```
> COEFF:=[Coefficient(L,i) : i in [0..11]];
> S := Scheme(AffineSpace(R),COEFF);
```

For this example, we simply call `IsolatedPointsFinder` directly. Alternatively, we could first use `LinearElimination` if desired.

```
> PTS:=IsolatedPointsFinder(S,[13]); PTS; // 13 is a random choice
[* [ 11, 1, 7, 6, 3, 1, 6, 11, 0, 3, 4, 2 ] *]
> b, sol := IsolatedPointsLifter(S,PTS[1]); sol;
(216513/4096, -3720087/131072, 531441/8192,
 11/4, 3, 311/64, 61/8, 9/2, 715/64, 165/16, 77/16, 55/8)
> _<u>:=PolynomialRing(Rationals());
> X := Polynomial([Evaluate(c,Eltseq(sol)) : c in Coefficients(X)]);
> Y := Polynomial([Evaluate(c,Eltseq(sol)) : c in Coefficients(Y)]);
> Q := Polynomial([Evaluate(c,Eltseq(sol)) : c in Coefficients(Q)]);
> A := Evaluate(A,Eltseq(sol));
> B := Polynomial([Evaluate(c,Eltseq(sol)) : c in Coefficients(B)]);
> assert X^3+A*X+B-Q*Y^2 eq 0;
> Q; // note that Q does not represent any squares, but 2*Q(1/2)=9
u^2 + 3*u + 11/4
> B; // also need to clear 2^17 from denominators
531441/8192*u - 3720087/131072
> POLYS := [2^7*X, 2^9*Y, 2^3*Q, 2^14*A, 2^21*B]; // 2^21 in each term
> [Evaluate(f,u/2) : f in POLYS];
[
    8*u^4 + 16*u^3 + 144*u^2 + 488*u + 622,
    16*u^5 + 440*u^3 + 616*u^2 + 2640*u + 5720,
    2*u^2 + 12*u + 22,
    866052,
    68024448*u - 59521392
]
```

As noted by Elkies, one can clean up the final form of the solution if desired, via rational transformations of the u-variable. Since $Q(1) = 2 + 12 + 22 = 36$, the theory of the Pell equation tells us that there are infinitely many integers u such that $Q(u)$ is integral, and these all give integral points on a suitable elliptic curve.
There are only four choices of (X, Y, A, B) degrees that give the "largest" possible integral points via this method. The second case, of degrees $(6, 8, 1, 1)$ has a solution over a quartic number field, and the third case, of degrees $(8, 11, 1, 2)$ has a nonic solution. Both of these were found by the methods of this section, the former taking only a couple of minutes.

Example H112E59

Another application of the isolated points rotine is to compute Belyi maps, one instance of which is in finding solutions to a polynomial version of Hall's conjecture, concerning how small the degree of $X(t)^3 - Y(t)^2$ can be (if the difference is nonzero). The result is that the degree must be at least $1 + \deg(X)/2$, and there are (up to equivalence) finitely many polynomials of that degree, the count of which can be described in terms of some combinatorics, or in terms of simultaneously conjugacy classes of cycle products of given types in a symmetric group.

In this example, we compute the solution for the case where X has degree 12. It turns out that there are 6 solutions in this case, lying over a sextic number field (we of course ignore "solutions" with $X(t)^3 = Y(t)^2$, though similar to the previous example, they contribute a positive-dimensional component of the solution set). The finding of a suitable local point is not particularly easy, so we just note that

$$X(t) \equiv t^{12} + 14t^{10} + 14t^9 + 9t^8 + 6t^7 + 4t^6 + 7t^5 + 6t^4 + 15t^3 + 7t^2 + 3t + 10$$

gives a solution modulo 17, with $Y(t)$ being (of course) the approximate square root of $X(t)^3$. We shall lift this in a way that keeps the t^{11} as zero and the t^9 and t^{10} coefficients as equal (these are from preliminary transformations of the t-parameter that can be applied to the system).
As noted above, it might be easier to remove the y-variables "by hand", and then undo the linear eliminations in the resulting sextic number field. We chose here to work directly. A theorem of Beckmann says that the number field we obtain can only be ramified at primes less than 36.

```
> SetVerbose("IsolatedPoints",1);
> XVARS := ["x"*IntegerToString(n) : n in [0..9]];
> YVARS := ["y"*IntegerToString(n) : n in [0..17]];
> P := PolynomialRing(Rationals(),28);
> AssignNames(~P,XVARS cat YVARS);
> _<t> := PolynomialRing(P);
> Y := &+[P.(i+11)*t^i : i in [0..17]]+t^18;
> X := &+[P.(i+1)*t^i : i in [0..9]]+(P.10)*t^10+t^12;
> Xpt := [GF(17)|10,3,7,15,6,7,4,6,9,14];
> pt := Xpt cat [0 : i in [11..28]];
> FF := GF(17); _<u> := PolynomialRing(FF);
> Xv := Polynomial([FF!Evaluate(c,pt) : c in Coefficients(X)]);
> Xv3 := Xv^3; Yv := u^18;
> for d:=17 to 0 by -1 do // ApproximateSquareRoot
>     Yv:=Yv+Coefficient(Xv3,d+18)/2*u^d; Xv3:=Xv^3-Yv^2; end for;
> Yv^2-Xv^3; // must be degree 7 or less
8*u^7 + 11*u^5 + 10*u^4 + 3*u^3 + 3*u^2 + 11*u + 4
> pt := Xpt cat [Coefficient(Yv,d) : d in [0..17]];
> SYS := [Coefficient(X^3-Y^2,d) : d in [8..35]]; // 28 vars
> S := Scheme(AffineSpace(P),SYS);
> b, sol := IsolatedPointsLifter(S,pt : LiftingBound:=12, DegreeBound:=10);
> K := OptimisedRepresentation(Parent(sol[1]) : PartialFactorisation); K;
Number Field with defining polynomial
  y^6 - y^5 - 60*y^4 - 267*y^3 - 514*y^2 -  480*y - 180 over the Rationals
> Factorization(Discriminant(Integers(K)));
[ <2, 2>, <5, 1>, <13, 1>, <29, 5> ]
```

An alternative of completing the computation is to first use `LinearElimination` before applying `IsolatedPointsLifter`. In any event, the computation of the local point (which we simply assumed to be given) would be the dominant part of the running time.

```
> mp := LinearElimination(S); // a few seconds to evaluate scheme maps
> rmp := // reduced map
>  map<ChangeRing(Domain(mp),GF(17))->ChangeRing(Codomain(mp),GF(17))
```

```
>     | DefiningEquations(mp),DefiningEquations(Inverse(mp)) : Check:=false>;
> PT := Inverse(rmp)(Codomain(rmp)!(pt));
```

The `IsolatedPointLifter` can now be called on `Domain(mp)` and `Eltseq(PT)` (with varargs if desired), and then the result can be mapped back to the original scheme `S` via `mp`. It is a bit hairy to do this directly, as scheme maps do not naturally deal with finite field inputs in all cases. Due to the way that `IsolatedPointsLifter` uses to choose which coordinate to try to recognise first, it could also be slower in the end.

Example H112E60

Here is an example where finding the common field is quite difficult, but finding minimal polynomials for all the coordinates is rather easy. First, a somewhat generic random scheme in $\mathbf{P}^3$ is chosen, such that each variable appears no more than linearly. This has degree less than 4!, and in the example chosen, it has degree 22. Then two local points are found modulo 5. These are then passed to the lifting function, which returns the desired solution.

```
> P<w,x,y,z> := AffineSpace(Rationals(),4);
> f1 := w*x*y - 6*w*x - 7*w*y*z + w*y - 6*w*z - 3*x*y + y + 6*z;
> f2 := 10*w*x*y*z - 4*w*y*z + 2*w*y - 9*w - x*y*z - 10*x*z + y*z - 7*y;
> f3 := 10*w*x*y*z - 6*w*x*y + 8*w*x*z - 4*w*y*z - 6*w*z - x*z + 9*x + 8*y;
> f4 := 6*w*x*y*z + 3*w*x*z + 19*w*y*z - 7*w*z + 8*x*y*z - 2*x*z + 6;
> S := Scheme(P,[f1,f2,f3,f4]);
> SetVerbose("IsolatedPoints",1);
> PTS := IsolatedPointsFinder(S,[5]);
> Degree(S);
22
> b,POLYS := IsolatedPointsLiftToMinimalPolynomials
>                     (S,PTS[1] : DegreeBound:=22,LiftingBound:=10);
> POLYS[1];
1812403568722098960*x^22 + 6297705584492967832*x^21 +
 6527336365144235612 8*x^20 + 8127120407582645599 2*x^19 +
 1307013696001389696 80*x^18 - 2853763840612678416 22*x^17 -
 8021669561188154716 54*x^16 + 2533254447903279968 45*x^15 -
 1266591733002155213 172*x^14 + 2511386184440323009 0*x^13 +
 5065309674068046314 82*x^12 - 1323179973699695447 463*x^11 +
 1605685921502538803112*x^10 - 1318315736155520576802*x^9 +
 9496491295829584599 58*x^8 - 5274413325441713384 90*x^7 +
 2544636840498666075 12*x^6 - 1000391891985775814 40*x^5 +
 2601441129568647585 6*x^4 - 3177984195514332576*x^3 -
 1852946687180290752*x^2 + 971825485320437760*x - 88506566917263360
```

All of the four polynomials in `POLYS` look approximately like this, and all should determine the same field, but it is difficult to find suitable isomorphisms between them, let alone find an `OptimisedRepresentation`.

In fact, the Gröbner basis machinery is superior for this purpose. Writing one of the coordinates in terms of the other (so as to get the field generators in terms of each other) would necessitate quite high precision in the p-adic lifting to recognise the coefficients, likely $5^{2^{14}}$ or more (this takes

about 5 minutes). The Gröbner basis method, which recognises one coordinate and then back-substitutes into the resulting equations, solving them algebraically, takes only about 15 seconds.

```
> time V := Variety(Ideal(S),AlgebraicClosure()); // about 15s
> MinimalPolynomial(V[22][4]); // deg 22, all coeffs about 25 digits
y^22 - 70869414518205839537/142324397561167099 52*y^21 -
    606754258610022348837 3/569297590244668398 08*y^20 + [...]
    [...] + 166661449939161/1779054969514588744
> MinimalPolynomial(V[22][3]); // deg 22, all coeffs about 25 digits
y^22 - 428567519465749893/68067993818308256*y^21 -
    229592953968800596 15/108908790109293209 6*y^20 + [...]
    [...] + 246916540549044143 1/68067993818308256
> V[22][4]; // given simply as r22, all 22 conjugates are found
r22
> V[22][3]; // third coordinate in terms of the fourth
[output takes about 200 lines, involving 750-digit coordinates]
```

Another way to achieve the result is to plug the known coordinate into the system, and use Gröbner bases (or resultants, if possible) to solve it.

```
> K := NumberField(POLYS[1]); // first coordinate
> _<xx,yy,zz> := PolynomialRing(K,3);
> E := [Evaluate(e,[K.1,xx,yy,zz]) : e in DefiningEquations(S)];
> Variety(Ideal(E)); // about 2 seconds
[again a rather bulky output]
```

Both of these uses of Gröbner bases are somewhat specific to the simplicity of the case here, and in more difficult cases would likely be rather onerous. This example does exemplify that for a generic variety the Gröbner basis methods should be superior. The lifting methods are largely for cases where the problem has special structure.

Example H112E61

Here is an example from N. D. Elkies of a polynomial $f(x) \in K(x)$ for which $f(x) - t$ appears to have Galois group M_{23} over $K(t)$ (this is sometimes called the monodromy group of f). This involves trying to find $f = P_3 P_2^2 P_4^4 = P_7 P_8^2 - c$ for some polynomials P_d of degree d, for some constant c. Upon suitable reductions, one gets a system of 8 variables and equations. With a few additional considerations, the search space can be reduced a bit further. As noted by M. Zieve, the equation should have 4 solutions, and thus likely be in a quartic field K that is an extension of $\mathbf{Q}(\sqrt{-23})$. This is indeed the case.

```
> Q := Rationals();
> R<a2,c,b1,b2,c1,c2,c3,c4,d1,d2,d3,d4,d5,d6,d7,
>   e1,e2,e3,e4,e5,e6,e7,e8> := PolynomialRing(Q,(3-1)+2+4+7+8);
> _<t> := PolynomialRing(R);
> P3 := t^3 + a2*t + a2/(26/-27); // normalisation
> P2 := t^2 + b1*t + b2;
> P4 := t^4 + c1*t^3 + c2*t^2 + c3*t + c4;
> P7 := t^7 + d1*t^6 + d2*t^5 + d3*t^4 + d4*t^3 + d5*t^2 + d6*t + d7;
> P8 := t^8 + e1*t^7 + e2*t^6 + e3*t^5 + e4*t^4 +
```

```
>             e5*t^3 + e6*t^2 + e7*t + e8;
> Q := P3 * P2^2 * P4^4 - P7 * P8^2 - c;
> S := Scheme(AffineSpace(R),Coefficients(Q));
> SetVerbose("IsolatedPoints",1);
> v:=[GF(101) | 26, 1,  -26,21,  -19,-27,-22,8, // known point
>              -14,12,26,-3,-37,-43,-22,  44,-11,-13,-21,45,-45,32,46];
> b, pt := IsolatedPointsLifter
>           (S,v : DegreeList:=[4], LiftingBound:=15, OptimizeFieldRep);
> K := Parent(pt[1]);
> DefiningPolynomial(K);
y^4 - 2*y^3 - 10*y^2 + 11*y + 36
> Factorization(Discriminant(Integers(K)));
[ <3, 1>, <23, 3> ]
```

The above code lifts the given point to precision $101^{2^{11}}$, and recognises it in the field K. Next we can compute the polynomial $f(x)$ in question, and see that reductions (modulo 269, say) of $f(x) - i$ do indeed correspond to cycle structures of M_{23}. However, to prove that this really is the Galois group (over the function field) seems to require a more difficult monodromy calculation. A more theoretical (topological) construction was given by P. Müller in 1995, but did not explicitly produce f. It is also still an open question whether there is a polynomial over the *rationals* with Galois group M_{23}.

```
> X := P3 * P2^2 * P4^4;
> f := Polynomial([Evaluate(e,Eltseq(pt)) : e in Coefficients(X)]);
> p := 269;
> P := Factorization(p * Integers(K))[1][1]; assert Norm(P) eq p;
> _, mp := ResidueClassField(P);
> fp := Polynomial([mp(c) : c in Coefficients(f)]);
> D := [[Degree(u[1]) : u in Factorization(fp-i)] : i in [1..p]];
> Sort(SetToSequence(Set([Sort(d) : d in D | &+d eq 23])));
[
    [ 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2 ],
    [ 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3 ],
    [ 1, 1, 1, 2, 2, 4, 4, 4, 4 ],
    [ 1, 1, 1, 5, 5, 5, 5 ],
    [ 1, 1, 7, 7, 7 ],
    [ 1, 2, 2, 3, 3, 6, 6 ],
    [ 1, 2, 4, 8, 8 ],
    [ 1, 11, 11 ],
    [ 2, 7, 14 ],
    [ 3, 5, 15 ],
    [ 23 ]
]
> load m23; // G is M23
> C := [g : g in ConjugacyClasses(G) | Order(g[3]) ne 1];
> S := Set([CycleStructure(c[3]) : c in C]);
> Sort([Sort(&cat[[s[1] : i in [1..s[2]]] : s in T]) : T in S]);
[
    [ 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2 ],
```

```
    [ 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3 ],
    [ 1, 1, 1, 2, 2, 4, 4, 4, 4 ],
    [ 1, 1, 1, 5, 5, 5, 5 ],
    [ 1, 1, 7, 7, 7 ],
    [ 1, 2, 2, 3, 3, 6, 6 ],
    [ 1, 2, 4, 8, 8 ],
    [ 1, 11, 11 ],
    [ 2, 7, 14 ],
    [ 3, 5, 15 ],
    [ 23 ]
]
```

112.19 Advanced Examples

This section contains examples of the use of the scheme machinery that are broader than those brief illustrations of intrinsics in the main text. They show how these functions can be used in collaboration with one another to build computer experiments which back up mathematical intuition.

112.19.1 A Pair of Twisted Cubics

This example constructs a cluster as the intersection of two twisted cubics in space. It uses a pair of curves whose equations are very closely related. Their union admits an automorphism which interchanges the two curves, fixing the cluster.

Example H112E62

We start by making the two twisted cubics, $C1$ and $C2$, as the minors of a pair of 2×3 matrices. It is clear straight away that these curves are closely related; it is a shame that we lose the "format" of the equations, in fact. On the other hand, if one tries to make other interesting examples by such tricks, one does not automatically come up with something so slick (and Gorenstein).

```
> k := Rationals();
> P<x,y,z,t> := ProjectiveSpace(k,3);
> M1 := Matrix(CoordinateRing(P),2,3,[y,t,x,t,x,z]);
> M2 := Matrix(CoordinateRing(P),2,3,[y,x,t,x,t,z]);
> C1 := Scheme(P,Minors(M1,2));
> C2 := Scheme(P,Minors(M2,2));
> Z := Intersection(C1,C2);
> MinimalBasis(Z);
[
    x*t - y*z,
    x*z - t^2,
    x*y - t^2,
    -x^2 + z*t,
    -x^2 + y*t
```

```
]
```

Anyone knowing about Pfaffians can have fun trying to realise these equations as the five maximal Pfaffians of a skew-symmetric 5×5 matrix. Although this example is a bit degenerate, it is reasonable to think of it as a hyperplane section of an elliptic curve of degree 5 (living in $\mathbf{P}^4$) so the ideal of equations will be Gorenstein. Given the Buchsbaum–Eisenbud structure theorem for Gorenstein codimension 3 rings, we are not surprised to see this Pfaffian format. In this example we will settle for confirming that this scheme Z is a cluster of degree 5 and finding its support.

```
> IsCluster(Z);
true
> Degree(Z);
5
> IsReduced(Z);
true
> RationalPoints(Z);
{@ (1 : 1 : 1 : 1), (0 : 0 : 1 : 0), (0 : 1 : 0 : 0) @}
> HasPointsOverExtension(Z);
true
```

As expected, the scheme Z is zero-dimensional and has degree 5. Since it is reduced, its support will comprise five separate points over some extension of the base field. We locate these points by hand by considering the Gröbner basis of the ideal of Z. The last element of a lexicographical Gröbner basis usually suggests a field extension that is relevant to the scheme. So we extend the base field by roots of this polynomial and look for the support over that field.

```
> GB := GroebnerBasis(ChangeOrder(Ideal(Z),"lex"));
> GB[#GB];
z^3*t - t^4
> L<w> := ext< k | U.1^2 + U.1 + 1 > where U is PolynomialRing(k);
> RationalPoints(Z,L);
{@ (w : -w - 1 : -w - 1 : 1), (-w - 1 : w : w : 1),
(1 : 1 : 1 : 1), (0 : 1 : 0 : 0), (0 : 0 : 1 : 0) @}
> HasPointsOverExtension(Z,L);
false
```

The final line confirms that we have found all the points of Z. That was already clear since Z has degree 5 and we see five points, but in other cases, especially when the cluster is not reduced, it might not be so obvious.
Now we look at the union of the two twisted cubics.

```
> C := Union(C1,C2);
> C;
Scheme over Rational Field defined by
x^3 - x*y*t - x*z*t + t^3
-x*t + y*z
```

This curve C is a $2,3$ complete intersection, numerology that is familiar from canonical curves of genus 4. We already know that C is not such a curve since it has two components. Indeed, we

already know that these components are nonsingular and meet in five points. Clearly these points must be singular points of C.

```
> SC := SingularPointsOverSplittingField(C);
> SC;
{ (1 : 1 : 1 : 1), (-r2 - 1 : r2 : r2 : 1), (0 : 0 : 1 : 0),
(-r1 - 1 : r1 : r1 : 1), (0 : 1 : 0 : 0) }
> Ring(Universe(SC));
Algebraically closed field with 2 variables
Defining relations:
[
    r2^2 + r2 + 1,
    r1^2 + r1 + 1
]
```

MAGMA has automatic Gröbner basis based machinery for working in the algebraic closure of the rationals (the so-called *D5 method*). Here we see it in action. The roots that we made explicitly when computing with Z are the new symbols $r1$ and $r2$ — they are the two conjugate roots of the quadratic equation list as the 'Defining relations'. Since $r1 \neq r2$, we see that the singular points really are the points of Z as expected.
From the definition of the matrices $M1$ and $M2$ we can see that the union and intersection of $C1$ and $C2$ should be invariant under the automorphism of P which exchanges x and t. We realise that automorphism here and confirm what we expect by comparing various ideals.

```
> phi := iso< P -> P | [t,y,z,x],[t,y,z,x] >;
> IsAutomorphism(phi);
true
> Ideal(C2) eq Ideal(phi(C1));
true
> Z eq phi(Z);
true
> Ideal(Z) eq Ideal(phi(Z));
true
```

Note that the basic equality test 'eq' for schemes returns `true` in the penultimate line, even though the two arguments were created independently.
The five points of Z obviously have Sym_5 as their permutation group (or $\mathrm{Sym}_2 \times \mathrm{Sym}_3$ over the rationals). How much of that is realised by automorphisms of the union C? We try to realise some elements of this symmetric group.

```
> S5 := SymmetricGroup(5);
> QL := RationalPoints(Z,L);
> rho := S5 ! [ Index(QL,phi(p)) : p in QL ];
> rho;
(1, 2)
```

Of course, this permutation is simply the action of the Galois group of L.

```
> GaloisGroup(L);
Permutation group acting on a set of cardinality 2
```

```
    (1, 2)
```

We make another automorphism: using C explicitly in the constructor ensures that the image of the map is contained in C.

```
> psi := iso< C -> C | [x,z,y,t],[x,z,y,t] >;
> eta := S5 ! [ Index(QL,psi(p)) : p in QL ];
> eta;
(4, 5)
> G := sub< S5 | rho,eta >;
> #G;
4
```

Since these two permutations commute and the small collection of five points is already partitioned by a Galois group action, this example is too simple to use MAGMA's substantial group theory machinery. But one can imagine at this stage finding complicated elements of G and realising them by compositions of the easily recognised automorphisms ρ and η.

112.19.2 Curves in Space

In this example, we construct something that we know is an elliptic curve in space. The point is to realise that within MAGMA by making a new curve of the right type and understanding the translation between the two types, at least to some degree. Something very similar would also works for the canonical models of curves of genus 4, although one has to take care handling the image of the natural projection.

Example H112E63

The first thing to do is to make a curve in space and to choose a nonsingular rational point on that curve. The question of whether or not a rational point is part of the input or part of the algorithm is always tricky since finding good points is often the heart of a problem. That is certainly the case here, so we do not pretend that this is a particularly powerful example.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Scheme(P,[x*y-z*t,x^2 + 2*z^2 - y*t]);
> Dimension(X);
1
> IsNonsingular(X);
true
> p := X ! [0,1,0,0];
```

Next we simply project from this given point p.

```
> Y,pr,q := ProjectionFromNonsingularPoint(X,p);
> bool,C := IsCurve(Y);
> bool;
true
> q := C ! q;
> q;
(0 : 1 : 0)
```

```
> Degree(C);
3
> IsNonsingular(C);
true
> P2<a,b,c> := Ambient(C);
> C;
Curve over Rational Field defined by
a^3 + 2*a*b^2 - b*c^2
```

Since there was a conic (two, in fact) among the equations of the scheme X, the projection from p is necessarily birational to a plane curve. And since p is a nonsingular point, it has a definite rational image point on the projection which is called q above. Since we know that X has genus 1 (as an external fact) and that the projection is birational we already know that the image curve C is the plane elliptic curve we desire. (It is interesting to try this calculation with a curve of higher genus, like the canonical model of a curve of genus 4.)
But we have made no effort to find a good model for C. At this point we can use MAGMA intrinsics to find a better model for C since we have the rational point q lying on C.

```
> EllipticCurve(C,q);
Elliptic Curve defined by y^2 = x^3 + 32*x over Rational Field
Mapping from: Crv: C to Elliptic Curve defined by y^2 = x^3 + 32*x over
Rational Field given by a rule
Mapping from: Elliptic Curve defined by y^2 = x^3 + 32*x over Rational Field to
Crv: C given by a rule
```

The result is a very nice model of an elliptic curve in Weierstraß form. The mapping types returned by this function are not yet fully integrated Scheme maps. But this will be added to MAGMA in due course, after which computations can be done on the good elliptic model and related to the original scheme X.

112.20 Bibliography

[**BC04**] W. Bosma and J. Cannon, editors. *Discovering Mathematics with Magma.* Springer-Verlag, Heidelberg, 2004.

[**Bru04**] Nils Bruin. Some ternary Diophantine equations of signature $(n, n, 2)$. In Bosma and Cannon [BC04].

[**Har77**] Robin Hartshorne. *Algebraic Geometry, GTM 52.* Springer, ASpringer, 1977.

[**MSS96**] J. R. Merriman, S. Siksek, and N. P. Smart. Explicit 4-descents on an elliptic curve. *Acta Arith.*, 77(4):385–404, 1996.

[**Rei88**] Miles Reid. *Undergraduate Algebraic Geometry.* CUP, Cambridge, 1988.

[**Rei97**] Miles Reid. Chapters on Algebraic Surfaces. In J. Kollár, editor, *Complex algebraic varieties, IAS/Park City Mathematics Series 3*, pages 1 – 154. AMS, Providence R.I., 1997.

[**vdE00**] Arno van den Essen. *Polynomial automorphisms and the Jacobian conjecture*, volume 190 of *Progress in Mathematics.* Birkhaeuser, ABirkhaeuser, 2000.

113 COHERENT SHEAVES

Chapter 113
COHERENT SHEAVES

113.1 Introduction

This chapter describes the MAGMA functionality for working with coherent sheaves on ordinary projective schemes. The emphasis in this initial version is on invertible sheaves and on computing associated cohomological invariants and explicit divisor maps. Important examples include canonical and anticanonical maps and adjunction maps on varieties of arbitrary dimension. The tools provided in MAGMA enable the user to compute these in a general and reasonably efficient way. There is also functionality for computing an invertible sheaf corresponding to the class of an effective Cartier divisor given as a closed subscheme as well as a basis for the Riemann-Roch space of that divisor as ambient rational functions. The correspondence between divisors (or their classes) and invertible sheaves will be expanded in later releases. A standard reference for the definition and basic properties of coherent sheaves on Noetherian schemes is Section 5, Chapter II of [Har77].

The package is based on MAGMA's functionality for graded modules over polynomial rings and relies heavily on Gröbner basis computations. A coherent sheaf is represented by a graded module over the coordinate ring of the ambient projective space. The key difference between the category of sheaves and the category of modules is that a sheaf is not represented uniquely. However, there is a unique *maximal* graded module representing it, which is finitely generated (with certain provisos). For certain algorithms – computing cohomology, for example – any module representing the sheaf may be used. However for other calculations, such as explicit Riemann-Roch spaces or divisor maps, the full maximal module, containing the full space of global sections of the sheaf and its small Serre twists, is often required.

One of the basic operations, therefore, is the computation of the maximal module of a sheaf from its initial defining module. We have tried to do this efficiently in reasonable generality. The basic condition is that the support of the sheaf has irreducible components all of the same non-zero dimension. This will be described in more detail in the function descriptions that follow. The user does not have to explicitly make a call to perform the computation, but it may be carried out in the background and the result stored by several other functions.

A coherent sheaf $\mathcal{S}$ is defined by a graded module M over the polynomial ring $R = k[x_0, \ldots, x_n]$ and a subscheme X of $\mathbf{P}^n = Proj(R)$ on which M is supported. That is, the defining ideal $I \subseteq R$ of X annihilates M. In some contexts, X is unimportant and it doesn't matter whether $\mathcal{S}$ is thought of as a sheaf on X or on $\mathbf{P}^n$. In other cases, X plays a role: we can test whether $\mathcal{S}$ is locally free as a sheaf on X or take its dual. The sheaf $\mathcal{S}$ is just the coherent sheaf $\tilde{M}$ on X as described in Prop. 5.11, Section 5, Chapter II of [Har77], with M considered as a graded module over the homogeneous coordinate ring of X.

Sheaves are of type `ShfCoh`. There is also a type `ShfHom` for homomorphisms between sheaves supported on the same scheme X.

The algorithms used in the package are based on a number of computational commutative algebra tricks well-known to the experts.

113.2 Creation Functions

The general creation function for sheaves takes a graded module representing the sheaf and a scheme X on which it is supported. Special constructors are provided in the cases of the structure sheaf of X and the canonical sheaf of X, when X is locally Cohen-Macaulay and equidimensional. The user may also ask for Serre twists of a given sheaf. Other constructions deriving new sheaves from existing sheaves will be described in later sections.

`Sheaf(M, X)`

Given an ordinary projective scheme X and a module M over the coordinate ring of the ambient of X, such that M is annihilated by the defining ideal of X, this function returns the sheaf defined by graded module M on scheme X.

`StructureSheaf(X)`

`StructureSheaf(X, n)`

Given an ordinary projective scheme X, this function returns the structure sheaf $\mathcal{O}_X$ for X, which is the sheaf defined by the coordinate ring R_X of X, as a module. If the intrinsic is called with a second integer-valued argument n, the object returned is a twisted version of the sheaf, that is, Serre's twisting sheaf $\mathcal{O}_X(n)$, which has $R_X(n)$ as its associated graded module (see Section 5, Chapter II of [Har77]). These are all invertible sheaves on X and $\mathcal{O}_X(1)$ is the sheaf $\mathcal{O}_X(H)$ corresponding to the class of a hyperplane divisor H on X.

`CanonicalSheaf(X)`

`CanonicalSheaf(X, n)`

Given an ordinary projective scheme X, this function returns the canonical sheaf K_X for X. The scheme X should be an ordinary projective scheme which is equidimensional and locally Cohen-Macaulay. That is, all of the primary components of X should have the same dimension and its local rings should all be Cohen-Macaulay rings. These conditions aren't checked by MAGMA as the necessary computations can be very expensive in general. A non-singular variety always satisfies these conditions, and many singular normal varieties do also. For example, any curve or normal surface will be equidimensional and locally Cohen-Macaulay. The stronger condition of being *arithmetically* Cohen-Macaulay, can be checked by invoking the intrinsic `IsArithmeticallyCohenMacaulay` with the structure sheaf of X as argument.

Under these conditions, X has a canonical sheaf K_X, defined up to isomorphism, which acts as a dualising sheaf. See Section 7, Chapter III of [Har77] and Chapter

21 of [Eis95] for the module-theoretic background. For non-singular varieties, the canonical sheaf is the usual one: the highest alternating power of the sheaf of Kahler differentials. The function returns the canonical sheaf of X. It is computed from the dual complex to the minimal free resolution of the coordinate ring of X.

If the intrinsic is invoked with an additional integer argument n, it returns the nth Serre twist (see below) of the canonical sheaf $K_X(n)$. For a non-singular variety of dimension d, the map into projective space corresponding to $K_X(d-1)$ is the important *adjunction map*.

`Twist(S, n)`

Given a sheaf S, the function returns the nth Serre twist of $\mathcal{S}$, $\mathcal{S}(n) \cong \mathcal{S} \otimes_{\mathcal{O}_X} \mathcal{O}_X(n)$. If M is a module giving $\mathcal{S}$, then $M(n)$ gives $\mathcal{S}(n)$.

Example H113E1

We construct some sheaves associated with the smooth cubic surface defined by $x^3+y^3+z^3+t^3$ in P^3.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> R := CoordinateRing(P);
> X := Scheme(P,x^3+y^3+z^3+t^3);
> OX := StructureSheaf(X);
```

We first examine the underlying graded module of the structure sheaf.

```
> Module(OX);
Reduced Module R^1/<relations>
Relations:
[x^3 + y^3 + z^3 + t^3]
```

Observe that the canonical sheaf KX of X is isomorphic to the twist $OX(-1)$ of the structure sheaf.

```
> KX := CanonicalSheaf(X);
> Module(KX);
Reduced Module R^1/<relations> with grading [1]
Relations:
[x^3 + y^3 + z^3 + t^3]
> Module(StructureSheaf(X,-1));
Reduced Module R^1/<relations> with grading [1]
Relations:
[x^3 + y^3 + z^3 + t^3]
```

Note that the module column weights are the negations of the Serre twist indices!

```
> Module(Twist(OX,-1));
Reduced Module R^1/<relations> with grading [1]
Relations (Groebner basis):
```

```
[x^3 + y^3 + z^3 + t^3]
```

The equations $x = z, y = t$ define an (exceptional) line in X. We can get its structure sheaf as a sheaf on X using the basic `Sheaf` constructor. The associated invertible sheaf $\mathcal{L}(Y)$ of Y as a divisor on X can be obtained from the `DivisorToSheaf` intrinsic described later in the chapter.

```
> IY := ideal<R|[x+z,y+t]>; // ideal of line
> OY := Sheaf(QuotientModule(IY),X);
> Module(OY);
Graded Module R^1/<relations>
Relations:
[x + z],
[y + t]
> Scheme(OY);
Scheme over Rational Field defined by
x^3 + y^3 + z^3 + t^3
```

`SheafOfDifferentials(X)`

`Maximize` BoolElt *Default :* `false`

Given an ordinary projective scheme X, this function returns the sheaf of 1-differentials on X, $\Omega^1_{X/k}$. The function computes the natural representing module for the sheaf coming from the embedding of X in projective space (see Section 8, Chapter II of [Har77]). If the parameter `Maximize` is `true`, then the maximal module representing this sheaf is computed and used to define it (see next section).

`TangentSheaf(X)`

`Maximize` BoolElt *Default :* `false`

For an ordinary projective scheme X, this function returns the sheaf of tangent vectors for X. The function computes the natural representing module for these sheaves coming from the embedding of X in projective space (see Section 8, Chapter II of [Har77]). If the parameter `Maximize` is `true`, then the maximal module representing this sheaf is computed and used to define it (see next section).

Combining either of the above intrinsics with the `IsLocallyFree` intrinsic, this gives an alternative method for checking non-singularity on varieties that are known to be (locally) Cohen-Macaulay. It is best to use the sheaf of differentials since that is generally easier to compute. This approach can be much faster for varieties having high codimension than the usual Jacobian method.

`HorrocksMumfordBundle(P)`

The projective space P should be ordinary projective 4-space $\mathbf{P}^4$ over a field. The function returns the locally free rank 2 sheaf on P which represents the Horrocks-Mumford bundle (see [HM73]). The scheme of vanishing of a general global section of this sheaf is a two dimensional Abelian variety in P.

113.3 Accessor Functions

The following functions provide an interface to conveniently extract the basic data from a coherent sheaf.

`Module(S)`

Returns the graded module that was used to define sheaf S.

`Scheme(S)`

Returns the ordinary projective scheme X on which the sheaf S is defined.

`FullModule(S)`

Computes and returns the maximal module M_{max} giving sheaf S. The nth graded piece of M_{max} is equal to the global sections of the Serre twist $S(n)$ as a finite dimensional vector space over k, the base field of the scheme X of S. Thus $M_{max} \cong \oplus_{n \in \mathbf{Z}} H^0(X, S(n))$ as in [Har77]. Here, it is implicitly assumed that the exact support of S on X has no irreducible components of dimension 0 and that there are no embedded associated prime places of dimension 0. More concretely, if M is a defining module for S with a possible non-zero finite torsion module for the redundant maximal ideal having been divided out, then no (homogeneous) associated prime of M has dimension 1. This assumption means that the terms in the above direct sum are 0 for $n[[0$ or equivalently that M_{max} is a finitely-generated module.

As mentioned in the introduction, a further assumption, which isn't checked, for the computation of M_{max} is that S is equidimensional, so that M actually has no embedded associated primes and the irreducible components of its exact support have the same non-zero dimension. It may be possible to avoid this assumption with more complex (and computationally heavy) code that works with an equidimensional decomposition of the defining module, but it suffices for many cases of interest (e.g., sheaves with trivial annihilator on a variety or equidimensional scheme).

The method used is basically the computation of the double dual of the defining module over an appropriate polynomial algebra A. A possible approach is to take A as the exact "supporting" algebra $k[x_0, \ldots, x_n]/I$ where the polynomial ring is the coordinate ring of the ambient of X and I is the exact annihilator of M. This would involve stronger assumptions on the support of S and the computation of the dualising module for this A. We choose instead to work with A as a *Noether normalisation* of the above A, which means that A is a simple polynomial ring and is its own dualising module (up to a shift in grading). Then M is re-expressed as a module over this A, M_{max} is computed as a module over A and finally is recovered as a module over $k[x_0, \ldots, x_n]$ by keeping track of the multiplication maps by the x_i variables which don't occur in A.

The module M_{max} is stored so that it is only computed once.

`GlobalSectionSubmodule(S)`

Given a sheaf S, this function returns the submodule of the maximal module M_{max} generated in degrees ≥ 0, that is $\oplus_{n\geq 0} H^0(X, S(n))$.

`SaturateSheaf(~S)`

Procedure to compute and store (but not return) the maximal module M_{max} of the sheaf S.

Example H113E2

The classic example of a natural module which is unsaturated (non-maximal) defining a sheaf is the coordinate ring R of a non-projectively normal non-singular projective variety X. The ring R defines the structure sheaf as usual, but not maximally. By definition, R isn't integrally closed. Its integral closure R_1 is an extension ring, inheriting its natural grading and agreeing with R in all but finitely many graded parts. In fact, R_1 considered as an R-module is precisely the maximal graded module of the structure sheaf!
Such a situation can very commonly arise when a non-singular variety is projected down isomorphically into a subspace of its ambient projective space. The projected down image X is then not even linearly-normal: the degree one graded part of its coordinate ring is missing coordinates that were eliminated in the projection. These must reoccur in the graded R-module that is computed as the maximal module of the structure sheaf.
In the following example, X is taken as the non-singular projection into P^3 of a degree 4 rational normal curve (which naturally lives in P^4). We can see the difference between the maximal module of the structure sheaf and the coordinate ring using Hilbert series. In fact, they just differ by dimension 1 in the 1-graded part, corresponding to that missing coordinate!

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Scheme(P3,[
> y^3 - y*z^2 - 2*y^2*t - 2*x*z*t - 3*y*z*t + z^2*t - y*t^2 + 2*z*t^2 + 2*t^3,
> x^2*z + x*z^2 + y*z^2 + 3*x*z*t + 2*y*z*t - z^2*t + y*t^2 - 2*z*t^2 - 2*t^3,
> y^2*z - y*z^2 + y^2*t - x*z*t - 4*y*z*t + z^2*t - 3*y*t^2 + 2*z*t^2 + 2*t^3,
> x*y - x*z - x*t + y*t]);
> OX := StructureSheaf(X);
> M1 := Module(OX);
> M2 := FullModule(OX);
> h1 := HilbertSeries(M1); h1;
(-t^3 + 2*t^2 + 2*t + 1)/(t^2 - 2*t + 1)
> h2 := HilbertSeries(M2); h2;
(3*t + 1)/(t^2 - 2*t + 1)
> h2-h1;
t
```

113.4 Basic Constructions

The following functions give some basic constructions on sheaves.

`TensorProduct(S, T)`

`Maximize` BoolElt *Default :* `false`

`TensorPower(S, n)`

`Maximize` BoolElt *Default :* `true`

The first intrinsic gives the tensor product (over $\mathcal{O}_X$) of two sheaves on the same scheme X. The second gives the nth tensor power of S if $n > 0$, the $(-n)$th tensor power of the dual (see below) of S if $n < 0$ and the structure sheaf $\mathcal{O}_X$ if $n = 0$.

Defining modules for these constructions are taken as the appropriate tensor products of modules for the constituent sheaves when the parameter `Maximize` is `false`. The user should note that this is the archetypal case where the module constructed to define the resulting sheaf can be far from maximal, even when the defining modules of S and T are maximal. The rank of the presentation of the tensor power of a module rises rapidly with n. Thus, it is usually a good idea to set `Maximize` to `true`, which means that the maximal module of the result is computed and also used as its defining module.

`Dual(S)`

For the sheaf S on a scheme X, the function returns the dual sheaf $\mathrm{Hom}_{\mathcal{O}_X}(S, \mathcal{O}_X)$.

`SheafHoms(S, T)`

For S and T sheaves on the same scheme X, the function returns the sheaf $H = \mathrm{Hom}_{\mathcal{O}_X}(S, T)$. The module defining H is $\mathrm{Hom}(M_{\max}, N_{\max})$, where $M_{\max}$ and $N_{\max}$ are the maximal modules of S and T. This module, M_H, is the maximal module of H.

Also returned is a map that takes a homogeneous element of M_H (which can be recovered with `Module(H)` or `FullModule(H)`) of degree d to the sheaf homomorphism of degree d that it represents (see the next section for information about sheaf homomorphisms). All sheaf homomorphisms can be obtained this way.

`DirectSum(S, T)`

For S and T sheaves on the same scheme X, this function returns the sheaf direct sum $S \oplus T$.

`Restriction(S, Y)`

`Check` BoolElt *Default :* `true`

Given a sheaf S on a scheme X and a subscheme Y of X, the function returns the restriction of S to Y. A check that Y is a subscheme of X will be performed only if the parameter `Check` is `true` (the default).

Example H113E3

We look at the well-known example of a ruling L on a (singular) projective quadric cone X in P^3. We find the associated invertible sheaf $O_X(L)$ using the `DivisorToSheaf` intrinsic. The tensor square of this sheaf is $O_X(2L)$ which is just isomorphic to the $O_X(1)$ Serre twist of the structure sheaf, as $2L$ is a hyperplane section. We verify this by getting the tensor and inspection. Of course we need to saturate the result, illustrating that the basic tensor power of maximal modules usually does not result in a maximal module.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> R := CoordinateRing(P);
> X := Scheme(P,x*y-z^2); // singular projective quadric
> IL := ideal<R|z,y>;     // line y=z=0 on X
> OL := DivisorToSheaf(X,IL); // associated sheaf O(L)
```

We first make sure that OL is saturated.

```
> SaturateSheaf(~OL);
> Module(OL);
Graded Module R^2/<relations>
Relations:
[ y, -z],
[ z, -x]
> O2L := TensorProduct(OL,OL); // or TensorPower(OL,2)
> Module(O2L);
Graded Module R^4/<relations>
Relations:
[ y,  0, -z,  0],
[ 0,  y,  0, -z],
[ z,  0, -x,  0],
[ 0,  z,  0, -x],
[ y, -z,  0,  0],
[ z, -x,  0,  0],
[ 0,  0,  y, -z],
[ 0,  0,  z, -x]
```

Finally, we get the maximum module – just that of OX(1)!

```
> FullModule(O2L);
Reduced Module R^1/<relations> with grading [-1]
Relations:
[x*y - z^2]
```

113.5 Sheaf Homomorphisms

This section describes some basic functionality for homomorphisms between sheaves defined on the same scheme. A sheaf homomorphism is represented by a module homomorphism between representing modules (defining, maximal or global section modules) for the two sheaves. Strictly, only (degree 0) homomorphisms that preserve the module gradings should be allowed but, for flexibility, we allow "homogeneous" homomorphisms that uniformly shift the grading by d and should be thought of as sheaf homomorphisms from the domain sheaf to the dth Serre twist of the codomain sheaf. We then say that the homomorphism is of degree d as for module homomorphisms.

There are some basic constructors and accessor functions, a function to "expand" a chain of homomorphisms to a single homomorphism and image, kernel and cokernel functions. The type of a sheaf homomorphism is `ShfHom`. Note that this is NOT a `Map` subtype, so that a sheaf homomorphism doesn't automatically inherit all of the usual map properties.

`SheafHomomorphism(S, T, h)`

Given sheaves S and T on the same scheme X and a module homomorphism h between M_0 and N_0, this function returns a sheaf homomorphism from S to T. Here M_0 is one of the defining, maximal or global section modules of S and N_0 is a similar module for T. The homomorphism h must be a homogeneous module homomorphism as returned by `IsHomogeneous` and if d is its degree then the homomorphism returned is really one from S to $T(d)$ in the category of $\mathcal{O}_X$-sheaves.

For the construction of sheaf homomorphisms see also `SheafHoms`.

`Domain(f)`

The domain of the sheaf homomorphism f.

`Codomain(f)`

The codomain of the sheaf homomorphism f.

`Degree(f)`

The degree of the sheaf homomorphism f as defined in the introduction to this section.

`ModuleHomomorphism(f)`

The underlying homogeneous graded module homomorphism of the sheaf homomorphism f.

`Kernel(f)`

Given a sheaf homomorphism f, this function returns the kernel of f and its inclusion homomorphism into the domain of f.

`Image(f)`

Given a sheaf f, this function returns the image, I, of f and two sheaf homomorphisms g and h. If f has degree d and S, T are its domain and codomain, then I is a subsheaf of $T(d)$. The second return value g is the restriction of f from S to I and the third return value h is the inclusion of I in $T(d)$, so that g also has degree d and h has degree 0.

`Cokernel(f)`

Given a sheaf homomorphism f, this function returns the cokernel of f and also the quotient homomorphism from the codomain to it.

Strictly speaking, if f has degree d and S, T are its domain and codomain, here we are taking f to be a homomorphism from $S(d) \leftarrow T$ rather than from $S \leftarrow T(d)$.

`Expand(hms)`

If $hms = [h_1, \ldots, h_n]$ is a sequence of sheaf homomorphisms, then this function returns the composition of homomorphisms $h_1 * h_2 * \ldots h_n$. The domain of h_2 must be the codomain of h_1 etc. and the stronger condition that the underlying module homomorphisms must be composable also holds. So the domain of `ModuleHomomorphism(h2)` must be the codomain of `ModuleHomomorphism(h1)` etc.

113.6 Divisor Maps and Riemann-Roch Spaces

As stated at the beginning of the chapter, one of our main initial aims in introducing sheaf machinery has been to provide a way of computing the (rational) maps associated to invertible sheaves in reasonable generality (see Section 7, Chapter 2 of [Har77]) and similarly for effective Cartier divisors (as closed subschemes) in the form of the map or their Riemann-Roch spaces. This section describes the main intrinsics. We hope to add further functionality to capture the correspondence between divisors and invertible sheaves in future releases.

`DivisorMap(S)`

`graphmap` BoolElt *Default :* `false`

Given an invertible sheaf S on the scheme X this function returns the rational map from X into the projective space associated to S. For efficiency, the invertibility of S is not checked, so that if the user is unsure whether a potential S actually is invertible (ie, locally free of rank one) he should apply the `IsLocallyFree` intrinsic.

The rational map that is returned can be thought of as $X \to Proj(R) \to \mathbf{P}^r$ where R is the graded k-subalgebra of the graded ring $\oplus_{n \geq 0} H^0(X, S^{\otimes n})$ generated by the weight 1 subspace $H^0(X, S)$ – the space of global sections of S – and the map to $\mathbf{P}^r$, where $r+1$ is the dimension of the space of global sections, is that induced by choosing a basis for the global sections. The divisor map, as usual, is only unique up to a linear change of coordinates of the codomain. The map is defined on the open subscheme of X where S is generated by global sections. Also returned is the image of the map on X.

In most cases, the map returned is a graph map of type `MapSchGrph` (see Section 112.14.7). This computation naturally computes the graph of the map and, in complicated situations, it is not particularly efficient to convert this to the more usual `MapSch` which will be defined by very nasty, high degree polynomials (often with a large base scheme) without some further specialised reduction routine. The user can however convert to a `MapSch` using `SchemeGraphMapToSchemeMap`. In cases when the sheaf has been constructed from a divisor using the `DivisorToSheaf` intrinsic below with the `GetMax` parameter `true`, so that a Riemann-Roch space has been stored, a traditional `MapSch` is returned. If in doubt, the user can distinguish using the `Type` intrinsic. Sometimes the user may still want a `MapSchGrph` in the latter case (e.g. because it is maximally defined, it is good for getting the genuine inverse image of a point/subscheme without components of the base scheme which appear for a non-maximally defined `MapSch`). This can be forced by setting the parameter `graphmap` to `true`.

The major stage of the computation is the determination of the graph of the map. An ideal defining the graph can be written down directly from the relation matrix of a minimal presentation of the global section submodule M_0 of S, and this ideal then only needs to be saturated with respect to an appropriate domain variable. The submodule M_0 is computed (and stored) as described earlier in the chapter.

`DivisorToSheaf(X, I)`

`GetMax` BoolElt *Default :* `true`

`RiemannRochBasis(X, I)`

Given an ordinary, projective scheme X and an ideal I of the coordinate ring of the ambient of X that defines a subscheme D of X that is an effective Cartier (locally principal) divisor of X, this function returns the invertible sheaf corresponding to the divisor class of D, commonly denoted $\mathcal{L}(D)$ (see Section 6, Chapter 2 of [Har77]). The conditions require D to be purely of codimension 1 in X and that it is everywhere locally defined by a single equation. Again for efficiency, Magma does not perform the computationally expensive checks to verify that D is locally principal within X. If X is a non-singular variety, then a closed subscheme of codimension 1 is automatically Cartier.

If `GetMax` is `true`, then the maximal module of $\mathcal{L}(D)$ is computed and an explicit basis for the Riemann-Roch space $L(D)$ is computed and stored along the way. This basis is of the form $[G_1/G, \ldots, G_n/G]$, where G and the G_i are homogeneous polynomials of some degree d on the ambient of X and the G_i/G are the usual rational functions restricted to X.

If instead of `DivisorToSheaf`, the intrinsic `RiemannRochBasis` is called, then the above procedure is carried out and a basis of the Riemann Roch space is returned as the sequence of numerators $[G_1, \ldots, G_n]$ and the denominator G, along with the sheaf $\mathcal{L}(D)$. If $\mathcal{L}(D)$ has been computed from `DivisorToSheaf` and returned as S, then the RiemannRoch basis can be recovered at a later stage from the *attribute*

of S, `rr_space`. This attribute, if assigned, contains a pair consisting of the above sequence of numerators and the denominator.

The algorithm used is based on the following observation. If we choose $r > 0$ such that I contains a homogeneous polynomial G of degree r that doesn't lie in the ideal of X, I_X (which is a proper subideal of I), then there is a "complementary" divisor E of X such that $rH \sim D + E$, where H is a hyperplane divisor of X. Then $\mathcal{L}(D) \simeq \mathcal{L}(-E)(r)$ and $\mathcal{L}(-E)$ is represented by the module I_E/I_X, where I_E is the ideal of E, a subscheme of X (see Prop 6.18 of the above reference). Once a suitable G is found, I_E is computed by invoking intrinsics `ColonIdeal` and `Saturation` a few times. If the `GetMax` option is on, r is chosen large enough so that $H^1(I_X(m)), m \geq r$ vanishes, which guarantees that we end up with a maximal representing module and can get a full basis of Riemann-Roch numerators with G as the denominator.

Example H113E4

As a simple example, we consider a degree 3 rational scroll in $\mathbf{P}^4$. This is a ruled surface that contains a family of disjoint lines. If l is a line in the family, then the divisor map for $\mathcal{L}(l)$ is a map to the projective line, the fibres of which are the lines of the family. We take such a scroll X and line l and get the Riemann-Roch space $L(l)$ and the divisor map down to $\mathbf{P}^1$.

```
> P4<a,b,c,d,e> := ProjectiveSpace(Rationals(),4);
> X := Scheme(P4,[a*b - c^2, a*d - c*e, c*d - b*e]);
> Il := ideal<CoordinateRing(P4)|[a,c,e]>; // ideal of l
> rr_seq,G, Sl := RiemannRochBasis(X,Il);
> rr_seq; G;
[
    d,
    e
]
e
```

Thus, 1 and d/e are a basis for the rational functions in $L(l)$.

```
> fib_mp := DivisorMap(Sl);
> fib_mp;
Mapping from: Sch: X to Projective Space of dimension 1
Variables: $.1, $.2
with equations :
d
e
```

Here the divisor map is not a graph map and is not maximally defined. So we extend it to make it so. Note that the fibres are lines.

```
> fib_mp := Extend(fib_mp);
> (Codomain(fib_mp)![1,0])@@fib_mp;
Scheme over Rational Field defined by
a,
```

```
c,
e,
a*b - c^2,
a*d - c*e,
c*d - b*e
> (Codomain(fib_mp)![0,1])@@fib_mp;
c,
b,
d,
a*b - c^2,
a*d - c*e,
c*d - b*e
```

Alternatively, we could ask for `fib_mp` as a `MapSchGrph` and not have to extend it.

```
> fib_mp := DivisorMap(Sl : graphmap := true);
> Type(fib_mp);
MapSchGrph
> (Codomain(fib_mp)![1,0])@@fib_mp;
Scheme over Rational Field defined by
a,
c,
e,
a*b - c^2,
a*d - c*e,
c*d - b*e
```

Example H113E5

As a second example, we consider the degree 3 Del Pezzo surface example from the Del Pezzo chapter. There we mapped it to a degree 6 Del Pezzo surface by blowing down 3 disjoint lines in an explicit fashion. We do the same thing here using the sheaf machinery.

First we get the surface X_3 and the union of the 3 lines L_{123}:-

```
> R3<x,y,z,t> := PolynomialRing(Rationals(),4,"grevlex");
> P3 := Proj(R3);
```

We set up the equation defining the degree 3 surface:

```
> F := -x^2*z + x*z^2 - y*z^2 + x^2*t - y^2*t - y*z*t + x*t^2 + y*t^2;
> X3 := Scheme(P3,F);
```

Get the ideal defining the union of the 3 lines:

```
> I1 := ideal<R3|[x,y]>; // line 1 L1
> I2 := ideal<R3|[z,t]>; // line 2 L2
> I3 := ideal<R3|[x-z,y-t]>; //line 3 L3
```

```
> I := I1*I2*I3; // (non-saturated) ideal of L1+L2+L3 = L123
```

Now we blow down to get the degree 6 Del Pezzo in $\mathbf{P}^6$. The divisor we need for the map is $H + L_{123}$ where H is a hyperplane section. We get this simply by twisting the sheaf corresponding to L_{123} once.

```
> S123 := DivisorToSheaf(X3,I);
> H6 := Twist(S123,1); // sheaf of H+L123
> mp, X := DivisorMap(H6);
> X;
Scheme over Rational Field defined by
y[1]*y[2] - y[2]*y[3] - y[4]*y[5] + y[1]*y[6] + 3*y[2]*y[6] - y[4]*y[6] + y[6]^2
    + y[1]*y[7] + 2*y[2]*y[7] - y[4]*y[7] - y[5]*y[7] + 3*y[6]*y[7],
y[2]^2 - y[2]*y[3] + 2*y[2]*y[6] + y[6]^2 + 2*y[2]*y[7] + 3*y[6]*y[7],
y[1]*y[3] - y[2]*y[3] + 2*y[2]*y[6] - y[5]*y[6] + y[6]^2 + y[2]*y[7] - y[4]*y[7]
    + 3*y[6]*y[7],
y[2]*y[4] - y[2]*y[6] + y[4]*y[6] + y[4]*y[7] + y[5]*y[7],
y[3]*y[4] - y[2]*y[6] - y[6]^2 - y[6]*y[7] + y[7]^2,
y[4]^2 - y[4]*y[6] + y[5]*y[6] + y[1]*y[7] - y[2]*y[7] + y[4]*y[7] + y[5]*y[7],
y[2]*y[5] - y[4]*y[6] + y[5]*y[6] - y[2]*y[7] + y[4]*y[7] + y[5]*y[7],
y[3]*y[5] - y[6]^2 - y[2]*y[7] - y[7]^2,
y[5]^2 - y[1]*y[6] + y[2]*y[6] - 2*y[4]*y[6] + y[5]*y[6] + y[1]*y[7] - y[2]*y[7]
> Dimension(X); Degree(X);
2
6
```

113.7 Predicates

This subsection describes tests for several important properties of coherent sheaves. It contains an isomorphism test that, combined with `DivisorToSheaf`, can be used as a test for linear equivalence of Cartier divisors.

`IsLocallyFree(S)`

`UseFitting` BoolElt *Default :* `true`

Given a sheaf S on ordinary projective scheme X, this function returns `true` if and only if S is a locally free sheaf on X of constant rank and, if so, also returns its rank.

Our original implementation was very fast but unfortunately incorrect! We have modified it to the algorithm described below which uses an "étale stratification" of X.

The more straightforward method uses Fitting ideals of the module M (or M_{max}) of S. If the possible rank is d (as determined from Hilbert polynomials), it is required to check that the saturation of the dth Fitting ideal is the full ring and that the $(d-1)$th lies in the saturated ideal of X. This is now the default method, but can be extremely slow and use a large amount of memory. Our alternative method is much

slower than it was but we still find that it can be much faster than the Fitting ideal method for a low dimensional X in a high dimensional ambient and a sheaf S whose (maximal) module has a presentation with a reasonably large minimal number of both generators and relations.

To use the alternative method, the user can set the `UseFitting` parameter to `false`. For this method, it is assumed that X is equidimensional (all of its primary components have the same dimension), (locally) Cohen-Macaulay and connected. MAGMA does not check these conditions. The alternative method is described below.

The equidimensional and locally Cohen-Macaulay assumptions imply that X is faithfully flat over $P_0 = Proj(R_0)$ for a Noether normalisation R_0 of the coordinate ring of X. Standard flatness properties mean that S being locally free over X implies that it is locally free as a sheaf over P_0, which is just a full projective space. Serre's criterion (see [Ser55]) states that the latter is true if and only if all intermediate cohomology rings $H^i(P_0, S(q))$ vanish for $q[[0$. If M_{max} is the maximal graded module of S, this translates to all intermediate $Ext(M_{max}, R_0)$ R_0-modules being finite length, which in turn translates to the dual complex to the minimal free resolution of M_{max} as an R_0-module, having finite-length homology groups at all intermediate places. Rather than actually computing homology modules, we can further translate this condition into a number of equality tests for Hilbert polynomials of cokernels of the maps between free modules in the dual complex. This operation seems to be fairly fast and efficient in practise.

The above gives a necessary condition for local freeness over X (and gives the rank) but it is only sufficient over the Zariski open subset of X over which $X \to P_0$ is unramified. We have adapted it to be applied inductively over a chain of closed subschemes of X, which is what we refer to as the étale stratification of X. At each level, we have a subscheme Y of X and a possibly empty collection of polynomials $\{F_i\}$ which are non-zero divisors on Y and such that a chosen Noether normalisation of Y is unramified (equivalently, étale) outside of the subschemes Y_i defined by F_i adjoined to the equations of Y. These subschemes lie at the next level down. The conditions imply that all of the Y_i are equidimensional and Cohen-Macaulay. For Y_i of positive dimension, we apply the above test to the restriction of S to Y_i, checking that the rank is the same as the rank determined for X at the top level if the test is passed and also that F_i is not a zero-divisor on the module of the restricted sheaf. For Y_i of dimension 0, we perform a more direct test.

The overall algorithm generally takes much longer than just applying the main test to X. The étale stratification can take some time to compute when X lies in higher dimensional ambients. However, the main problem is that the degrees of the Y_i increase as we go down the chain (the subscheme defined by F_i has the degree of Y_i multiplied by the degree of F_i as its degree) and in practice (mainly working with surfaces X), the most time is taken in the bottom level checks on high degree zero-dimensional subschemes. Once computed, the étale stratification is stored with X, so does not need to be recomputed for tests on other sheaves. However, it relies on finding Noether normalisations at each level that are generically unramified, i.e. separable. Currently we do not check the condition and it can fail in small positive

characteristic leading to a crash or wrong results.

`IsIsomorphic(S, T)`

`IsIsomorphicWithTwist(S, T)`

For S and T coherent sheaves on the same base scheme X, this function returns `true` if and only if S is isomorphic to T or (for the second intrinsic) to a Serre twist $T(d)$ of T. In either case, an isomorphism is returned, if one exists and for the second intrinsic, the twist d is also returned as the second return value (so the isomorphism is between S and $T(d)$).

For the implementation, we first do a quick Hilbert polynomial check and then a Betti number check for the maximal modules M_{max} and N_{max} of S and T. This gives necessary conditions for an isomorphism and the possible d in the "with twist" case. Then we look for an isomorphism in the finite dimensional space of homomorphisms between M_{max} and N_{max}. We could have chosen to work with the homomorphisms between the truncated modules with gradings greater than or equal to N, for some N greater than or equal to the regularity of any defining modules for the two sheaves, but these have much larger presentations in general so the computation of homomorphisms is slower.

To look for isomorphisms, we look at the "zero degree" subblocks of the matrices giving a basis to the space of all homomorphisms. This reduces the problem to determining whether there is an invertible matrix in a space of $n \times n$ matrices over the base field. This is known to be a difficult problem in general and currently our implementation is rather weak at this point. We hope to improve it for future releases.

`IsArithmeticallyCohenMacaulay(S)`

Given a sheaf S on an ordinary projective scheme X, this function returns `true` if and only if the maximal graded module M_{max} of S is a Cohen-Macaulay module over the coordinate ring of X.

A scheme X is called arithmetically Cohen-Macaulay if and only if its coordinate ring is a Cohen-Macaulay ring. This is then true if and only if its coordinate ring is equal to the maximal module of $\mathcal{O}_X$ and the intrinsic returns `true`for $\mathcal{O}_X$.

This is a fairly straightforward computation once M_{max} has been determined. If we already know the structure of M_{max} as a module over a Noether normalisation of the coordinate ring of X, it is an immediate freeness check. Otherwise it is a straightforward depth calculation from a minimal free resolution of M_{max} as a graded module over the coordinate ring of the ambient of X.

113.8 Miscellaneous

`CohomologyDimension(S, r, n)`

Given a sheaf S and integers r and n, this function returns the dimension over the base field of the r-th cohomology group of the n-th Serre twist of S, $H^r(X, S(n)$.

This just calls the equivalent function for the maximal module of S or its defining module, if the maximal module has not yet been computed. Note that, in practice, it may often be much faster to use the maximal module so it may be desirable to call `SaturateSheaf` before doing any cohomology computations.

`DimensionOfGlobalSections(S)`

This returns the same dimension as `CohomologyDimension(S,0,0)` – the dimension of the space of global sections of S – but it is computed in a different way that is usually faster. It uses some straightforward linear algebra to compute the dimension of the zero-th graded part of the maximal module of S given as a presentation module.

`IntersectionPairing(S, T)`

If S and T are invertible sheaves on a nonsingular surface X, representing divisor classes D and E, this function returns the surface intersection number $D.E$.

Only minimal checks are made on the validity of the input data. The computation is a standard one using the Hilbert polynomials of S, T and their tensor product.

`ZeroSubscheme(S, s)`

The sheaf S should be a locally free sheaf on a scheme X (local freeness is *not* checked). The element s should be a homogeneous element of the defining, maximal or global section modules of S. Then s represents a global section of the twisted sheaf $S(d)$ if s is homogeneous of degree d. The intrinsic returns the vanishing subscheme of s: the largest subscheme of X on which s restricts to a zero section. If S is invertible of the form $\mathcal{L}(D)$, for example, s represents an effective divisor D_s in the linear system $|D + dH|$ (if it is non-zero) and the vanishing subscheme is D_s as a subscheme of X. Locally, for a Zariski-open set U over which there is an isomorphism $S(d)_U \cong \mathcal{O}_{X\,U}^{\,n}$, $s|_U$ corresponds to an n-tuple of functions $(f_1, \ldots, f_n)$ on U and the vanishing subscheme restricted to U is the closed subscheme of U defined by the ideal $\langle f_1, \ldots, f_n \rangle$.

113.9 Examples

In this section we present some extended examples illustrating various features of the sheaf machinery.

Example H113E6

In this example, we consider a surface X from a special family of rational surfaces of degree 10 in $\mathbf{P}^4$. This family is described by Decker, Ein and Schreyer in Section 2.1 of [DES93]. They have sectional genus 9 and are isomorphic to the plane blown up in 18 points in special position which give 18 exceptional curves in X. The embedding into $\mathbf{P}^4$ is such that four of these exceptional curves are of degree 3, seven curves are of degree 2 and seven curves are of degree 1.

The adjunction map on X is the map corresponding to the divisor $K_X + H$, where K_X is a canonical divisor and H is a hyperplane section, or, equivalently to the sheaf $\mathcal{K}_X(1)$ where $\mathcal{K}_X$ is the canonical sheaf. In our example, the adjunction map maps X to a smooth surface X_1 of degree 13 in $\mathbf{P}^8$ blowing down the seven degree 1 exceptional curves to points and reducing the degrees of the others by 1. The adjunction map on X_1 blows down the seven exceptional curves originally of degree 2 to points and maps X_2 to an anticanonically embedded degree 5 Del Pezzo surface in $\mathbf{P}^5$.

We take a randomly generated surface from this family over a small finite field ($\mathbf{F}_{17}$) and illustrate this process by explicitly computing the adjunction maps and images X_1 and X_2. We show that the intersection pairings of the canonical divisor and hyperplane sections on X, X_1, X_2 are as expected and that X_2 really is an anticanonically embedded Del Pezzo surface. We also expand the composition of the two divisor maps and show that the resulting map is indeed a birational map from X onto X_2.

These surfaces X are defined by one degree 4 and ten degree 5 polynomials in $\mathbf{P}^4$. The embedding is quite a complex one and it is hard to construct one with defining polynomials which are at all sparse. This makes it fairly challenging for explicit computation and also means that an example takes up a lot of page space! An example with relatively small coefficients over $\mathbf{Q}$ can also be processed, though the total running time is a few minutes. Also, the resulting X_2 tends to have very large coefficients. Here we get no coefficient blow-up and X_2 is a much simpler looking surface than X.

```
> P<[x]> := ProjectiveSpace(GF(17),4);
> X := Scheme(P, [
>    10*x[1]^4 + 13*x[1]^3*x[2] + 8*x[1]*x[2]^3 + 4*x[2]^4 + 6*x[1]^3*x[3] +
>        15*x[1]^2*x[2]*x[3] + 14*x[2]^3*x[3] + x[1]^2*x[3]^2 +
>        13*x[1]*x[2]*x[3]^2 + 3*x[2]^2*x[3]^2 + 9*x[1]*x[3]^3 + 2*x[2]*x[3]^3 +
>        10*x[3]^4 + 15*x[1]^3*x[4] + 4*x[1]^2*x[2]*x[4] + 3*x[1]*x[2]^2*x[4] +
>        7*x[2]^3*x[4] + 9*x[1]^2*x[3]*x[4] + 3*x[1]*x[2]*x[3]*x[4] +
>        9*x[2]^2*x[3]*x[4] + 11*x[1]*x[3]^2*x[4] + 6*x[2]*x[3]^2*x[4] +
>        15*x[3]^3*x[4] + x[1]^2*x[4]^2 + 4*x[1]*x[2]*x[4]^2 + 2*x[2]^2*x[4]^2 +
>        12*x[1]*x[3]*x[4]^2 + 8*x[2]*x[3]*x[4]^2 + 9*x[3]^2*x[4]^2 +
>        10*x[1]*x[4]^3 + 5*x[2]*x[4]^3 + 14*x[3]*x[4]^3 + 4*x[1]^3*x[5] +
>        16*x[1]^2*x[2]*x[5] + 15*x[2]^3*x[5] + 13*x[1]^2*x[3]*x[5] +
>        13*x[1]*x[2]*x[3]*x[5] + 10*x[2]^2*x[3]*x[5] + 15*x[1]*x[3]^2*x[5] +
>        7*x[2]*x[3]^2*x[5] + 14*x[3]^3*x[5] + 11*x[1]^2*x[4]*x[5] +
>        10*x[1]*x[2]*x[4]*x[5] + 4*x[2]^2*x[4]*x[5] + x[1]*x[3]*x[4]*x[5] +
```

```
>         12*x[2]*x[3]*x[4]*x[5] + 8*x[3]^2*x[4]*x[5] + 5*x[1]*x[4]^2*x[5] +
>         5*x[2]*x[4]^2*x[5] + 11*x[3]*x[4]^2*x[5] + 10*x[4]^3*x[5] +
>         12*x[1]^2*x[5]^2 + 8*x[1]*x[2]*x[5]^2 + 16*x[2]^2*x[5]^2 +
>         12*x[1]*x[3]*x[5]^2 + x[2]*x[3]*x[5]^2 + 14*x[3]^2*x[5]^2 +
>         8*x[1]*x[4]*x[5]^2 + x[2]*x[4]*x[5]^2 + 3*x[3]*x[4]*x[5]^2 +
>         5*x[4]^2*x[5]^2 + 11*x[1]*x[5]^3 + 13*x[2]*x[5]^3 + 5*x[3]*x[5]^3 +
>         9*x[4]*x[5]^3 + 8*x[5]^4,
>     9*x[1]^4*x[4] + 14*x[1]^3*x[2]*x[4] + 5*x[1]^2*x[2]^2*x[4] +
>         2*x[1]*x[2]^3*x[4] + 2*x[1]^3*x[3]*x[4] + 7*x[1]^2*x[2]*x[3]*x[4] +
>         5*x[1]*x[2]^2*x[3]*x[4] + 7*x[2]^3*x[3]*x[4] + 9*x[1]^2*x[3]^2*x[4] +
>         12*x[1]*x[2]*x[3]^2*x[4] + 2*x[2]^2*x[3]^2*x[4] + 9*x[1]*x[3]^3*x[4] +
>         2*x[2]*x[3]^3*x[4] + x[3]^4*x[4] + 3*x[1]^3*x[4]^2 +
>         5*x[1]^2*x[2]*x[4]^2 + 7*x[1]*x[2]^2*x[4]^2 + 13*x[2]^3*x[4]^2 +
>         11*x[1]^2*x[3]*x[4]^2 + 4*x[1]*x[2]*x[3]*x[4]^2 + 11*x[2]^2*x[3]*x[4]^2
>         + 14*x[1]*x[3]^2*x[4]^2 + 16*x[2]*x[3]^2*x[4]^2 + 15*x[1]^2*x[4]^3 +
>         11*x[1]*x[2]*x[4]^3 + 5*x[2]^2*x[4]^3 + 6*x[1]*x[3]*x[4]^3 +
>         9*x[2]*x[3]*x[4]^3 + 16*x[3]^2*x[4]^3 + 9*x[2]*x[4]^4 + 15*x[3]*x[4]^4 +
>         14*x[4]^5 + 2*x[1]^3*x[2]*x[5] + 6*x[1]^2*x[2]^2*x[5] +
>         3*x[1]*x[2]^3*x[5] + 16*x[2]^4*x[5] + 15*x[1]^3*x[3]*x[5] +
>         6*x[1]*x[2]^2*x[3]*x[5] + 10*x[2]^3*x[3]*x[5] + 14*x[1]^2*x[3]^2*x[5] +
>         13*x[1]*x[2]*x[3]^2*x[5] + 4*x[2]^2*x[3]^2*x[5] + 16*x[1]*x[3]^3*x[5] +
>         13*x[3]^4*x[5] + 14*x[1]^3*x[4]*x[5] + 9*x[1]^2*x[2]*x[4]*x[5] +
>         16*x[1]*x[2]^2*x[4]*x[5] + 14*x[2]^3*x[4]*x[5] +
>         6*x[1]*x[2]*x[3]*x[4]*x[5] + 6*x[2]^2*x[3]*x[4]*x[5] +
>         3*x[1]*x[3]^2*x[4]*x[5] + 7*x[2]*x[3]^2*x[4]*x[5] + 7*x[3]^3*x[4]*x[5] +
>         2*x[1]^2*x[4]^2*x[5] + 15*x[1]*x[2]*x[4]^2*x[5] +
>         9*x[1]*x[3]*x[4]^2*x[5] + 14*x[3]^2*x[4]^2*x[5] + 14*x[1]*x[4]^3*x[5] +
>         6*x[2]*x[4]^3*x[5] + 12*x[3]*x[4]^3*x[5] + 3*x[4]^4*x[5] +
>         9*x[1]^3*x[5]^2 + 12*x[1]^2*x[2]*x[5]^2 + 16*x[1]*x[2]^2*x[5]^2 +
>         x[2]^3*x[5]^2 + 7*x[1]^2*x[3]*x[5]^2 + 5*x[1]*x[2]*x[3]*x[5]^2 +
>         8*x[2]^2*x[3]*x[5]^2 + 2*x[1]*x[3]^2*x[5]^2 + 4*x[2]*x[3]^2*x[5]^2 +
>         13*x[3]^3*x[5]^2 + 7*x[1]^2*x[4]*x[5]^2 + 6*x[2]^2*x[4]*x[5]^2 +
>         16*x[1]*x[3]*x[4]*x[5]^2 + 15*x[2]*x[3]*x[4]*x[5]^2 +
>         7*x[3]^2*x[4]*x[5]^2 + 6*x[1]*x[4]^2*x[5]^2 + 3*x[2]*x[4]^2*x[5]^2 +
>         16*x[3]*x[4]^2*x[5]^2 + 15*x[4]^3*x[5]^2 + x[1]^2*x[5]^3 +
>         13*x[1]*x[2]*x[5]^3 + 6*x[2]^2*x[5]^3 + 8*x[1]*x[3]*x[5]^3 +
>         x[2]*x[3]*x[5]^3 + 9*x[3]^2*x[5]^3 + 3*x[1]*x[4]*x[5]^3 +
>         14*x[2]*x[4]*x[5]^3 + 8*x[3]*x[4]*x[5]^3 + 14*x[4]^2*x[5]^3 +
>         16*x[1]*x[5]^4 + 2*x[2]*x[5]^4 + 7*x[3]*x[5]^4 + 7*x[4]*x[5]^4 +
>         11*x[5]^5,
>     13*x[1]^4*x[4] + 8*x[1]^3*x[2]*x[4] + 14*x[1]^2*x[2]^2*x[4] +
>         3*x[1]*x[2]^3*x[4] + 11*x[2]^4*x[4] + 7*x[1]^3*x[3]*x[4] +
>         3*x[1]^2*x[2]*x[3]*x[4] + 12*x[2]^3*x[3]*x[4] + 3*x[1]^2*x[3]^2*x[4] +
>         13*x[1]*x[2]*x[3]^2*x[4] + 3*x[2]^2*x[3]^2*x[4] + 7*x[1]*x[3]^3*x[4] +
>         2*x[2]*x[3]^3*x[4] + 7*x[3]^4*x[4] + 13*x[1]^3*x[4]^2 +
>         6*x[1]^2*x[2]*x[4]^2 + 6*x[1]*x[2]^2*x[4]^2 + 6*x[2]^3*x[4]^2 +
>         2*x[1]^2*x[3]*x[4]^2 + 15*x[1]*x[2]*x[3]*x[4]^2 + 14*x[2]^2*x[3]*x[4]^2
>         + 3*x[1]*x[3]^2*x[4]^2 + 16*x[2]*x[3]^2*x[4]^2 + 3*x[3]^3*x[4]^2 +
```

```
>        6*x[1]^2*x[4]^3 + 10*x[2]^2*x[4]^3 + 7*x[2]*x[3]*x[4]^3 + 13*x[1]*x[4]^4
>        + 5*x[2]*x[4]^4 + 15*x[3]*x[4]^4 + 13*x[4]^5 + 2*x[1]^4*x[5] +
>        6*x[1]^3*x[2]*x[5] + 12*x[1]^2*x[2]^2*x[5] + 12*x[1]*x[2]^3*x[5] +
>        2*x[2]^4*x[5] + 5*x[1]^3*x[3]*x[5] + 12*x[1]^2*x[2]*x[3]*x[5] +
>        7*x[1]*x[2]^2*x[3]*x[5] + 11*x[2]^3*x[3]*x[5] + 2*x[1]^2*x[3]^2*x[5] +
>        3*x[1]*x[2]*x[3]^2*x[5] + 7*x[2]^2*x[3]^2*x[5] + 16*x[1]*x[3]^3*x[5] +
>        3*x[2]*x[3]^3*x[5] + 13*x[3]^4*x[5] + 2*x[1]^2*x[2]*x[4]*x[5] +
>        12*x[1]*x[2]^2*x[4]*x[5] + 2*x[2]^3*x[4]*x[5] + 10*x[1]^2*x[3]*x[4]*x[5]
>        + 9*x[1]*x[2]*x[3]*x[4]*x[5] + 6*x[2]^2*x[3]*x[4]*x[5] +
>        x[1]*x[3]^2*x[4]*x[5] + 6*x[2]*x[3]^2*x[4]*x[5] + 15*x[3]^3*x[4]*x[5] +
>        2*x[1]^2*x[4]^2*x[5] + 14*x[1]*x[2]*x[4]^2*x[5] +
>        13*x[1]*x[3]*x[4]^2*x[5] + 13*x[2]*x[3]*x[4]^2*x[5] +
>        2*x[3]^2*x[4]^2*x[5] + 12*x[1]*x[4]^3*x[5] + 8*x[2]*x[4]^3*x[5] +
>        8*x[3]*x[4]^3*x[5] + x[4]^4*x[5] + 3*x[1]^3*x[5]^2 +
>        7*x[1]^2*x[2]*x[5]^2 + 4*x[1]^2*x[3]*x[5]^2 + 3*x[1]*x[2]*x[3]*x[5]^2 +
>        9*x[2]^2*x[3]*x[5]^2 + 14*x[1]*x[3]^2*x[5]^2 + 13*x[2]*x[3]^2*x[5]^2 +
>        15*x[3]^3*x[5]^2 + x[1]^2*x[4]*x[5]^2 + 14*x[1]*x[2]*x[4]*x[5]^2 +
>        5*x[2]^2*x[4]*x[5]^2 + 10*x[1]*x[3]*x[4]*x[5]^2 +
>        5*x[2]*x[3]*x[4]*x[5]^2 + 7*x[3]^2*x[4]*x[5]^2 + 13*x[1]*x[4]^2*x[5]^2 +
>        2*x[2]*x[4]^2*x[5]^2 + 9*x[3]*x[4]^2*x[5]^2 + 3*x[4]^3*x[5]^2 +
>        14*x[1]*x[2]*x[5]^3 + 12*x[2]^2*x[5]^3 + 6*x[1]*x[3]*x[5]^3 +
>        16*x[2]*x[3]*x[5]^3 + 8*x[3]^2*x[5]^3 + 3*x[1]*x[4]*x[5]^3 +
>        4*x[2]*x[4]*x[5]^3 + 11*x[3]*x[4]*x[5]^3 + 15*x[4]^2*x[5]^3 +
>        14*x[1]*x[5]^4 + 13*x[2]*x[5]^4 + 4*x[3]*x[5]^4 + 4*x[4]*x[5]^4 +
>        13*x[5]^5,
>    15*x[1]^3*x[2]*x[3] + 11*x[1]^2*x[2]^2*x[3] + 14*x[1]*x[2]^3*x[3] +
>        x[2]^4*x[3] + 2*x[1]^3*x[3]^2 + 11*x[1]*x[2]^2*x[3]^2 + 7*x[2]^3*x[3]^2
>        + 3*x[1]^2*x[3]^3 + 4*x[1]*x[2]*x[3]^3 + 13*x[2]^2*x[3]^3 + x[1]*x[3]^4
>        + 4*x[3]^5 + 2*x[1]^4*x[4] + 11*x[1]^3*x[2]*x[4] + 13*x[1]^2*x[2]^2*x[4]
>        + 4*x[1]*x[2]^3*x[4] + 16*x[2]^4*x[4] + 5*x[1]^3*x[3]*x[4] +
>        4*x[1]^2*x[2]*x[3]*x[4] + 10*x[1]*x[2]^2*x[3]*x[4] + 8*x[2]^3*x[3]*x[4]
>        + 5*x[1]^2*x[3]^2*x[4] + 14*x[1]*x[2]*x[3]^2*x[4] + 2*x[2]^2*x[3]^2*x[4]
>        + 15*x[1]*x[3]^3*x[4] + 13*x[3]^4*x[4] + 9*x[1]^3*x[4]^2 +
>        3*x[1]^2*x[2]*x[4]^2 + 10*x[1]*x[2]^2*x[4]^2 + 12*x[2]^3*x[4]^2 +
>        8*x[1]^2*x[3]*x[4]^2 + 14*x[1]*x[2]*x[3]*x[4]^2 + 3*x[2]^2*x[3]*x[4]^2 +
>        2*x[1]*x[3]^2*x[4]^2 + 5*x[2]*x[3]^2*x[4]^2 + 10*x[3]^3*x[4]^2 +
>        5*x[1]^2*x[4]^3 + x[1]*x[2]*x[4]^3 + 8*x[2]^2*x[4]^3 +
>        7*x[1]*x[3]*x[4]^3 + 10*x[2]*x[3]*x[4]^3 + 13*x[3]^2*x[4]^3 +
>        10*x[1]*x[4]^4 + 7*x[2]*x[4]^4 + 16*x[3]*x[4]^4 + 16*x[4]^5 +
>        8*x[1]^3*x[3]*x[5] + 5*x[1]^2*x[2]*x[3]*x[5] + x[1]*x[2]^2*x[3]*x[5] +
>        16*x[2]^3*x[3]*x[5] + 10*x[1]^2*x[3]^2*x[5] + 12*x[1]*x[2]*x[3]^2*x[5] +
>        9*x[2]^2*x[3]^2*x[5] + 15*x[1]*x[3]^3*x[5] + 13*x[2]*x[3]^3*x[5] +
>        4*x[3]^4*x[5] + 14*x[1]^3*x[4]*x[5] + x[1]^2*x[2]*x[4]*x[5] +
>        10*x[1]*x[2]^2*x[4]*x[5] + 11*x[2]^3*x[4]*x[5] + 5*x[1]^2*x[3]*x[4]*x[5]
>        + 12*x[1]*x[2]*x[3]*x[4]*x[5] + 7*x[2]^2*x[3]*x[4]*x[5] +
>        5*x[1]*x[3]^2*x[4]*x[5] + 3*x[3]^3*x[4]*x[5] + 2*x[1]^2*x[4]^2*x[5] +
>        5*x[2]^2*x[4]^2*x[5] + 2*x[2]*x[3]*x[4]^2*x[5] + 8*x[3]^2*x[4]^2*x[5] +
>        x[1]*x[4]^3*x[5] + 5*x[2]*x[4]^3*x[5] + 3*x[3]*x[4]^3*x[5] +
```

```
>         14*x[4]^4*x[5] + 16*x[1]^2*x[3]*x[5]^2 + 4*x[1]*x[2]*x[3]*x[5]^2 +
>         11*x[2]^2*x[3]*x[5]^2 + 9*x[1]*x[3]^2*x[5]^2 + 16*x[2]*x[3]^2*x[5]^2 +
>         8*x[3]^3*x[5]^2 + 8*x[1]^2*x[4]*x[5]^2 + 11*x[1]*x[2]*x[4]*x[5]^2 +
>         3*x[2]^2*x[4]*x[5]^2 + 6*x[1]*x[3]*x[4]*x[5]^2 + 9*x[2]*x[3]*x[4]*x[5]^2
>         + 5*x[3]^2*x[4]*x[5]^2 + 15*x[1]*x[4]^2*x[5]^2 + 2*x[2]*x[4]^2*x[5]^2 +
>         8*x[3]*x[4]^2*x[5]^2 + 14*x[4]^3*x[5]^2 + x[1]*x[3]*x[5]^3 +
>         15*x[2]*x[3]*x[5]^3 + 10*x[3]^2*x[5]^3 + 11*x[1]*x[4]*x[5]^3 +
>         8*x[2]*x[4]*x[5]^3 + 15*x[3]*x[4]*x[5]^3 + 15*x[4]^2*x[5]^3 +
>         6*x[3]*x[5]^4 + 3*x[4]*x[5]^4,
>     9*x[1]^4*x[3] + 14*x[1]^3*x[2]*x[3] + 5*x[1]^2*x[2]^2*x[3] +
>         2*x[1]*x[2]^3*x[3] + 2*x[1]^3*x[3]^2 + 7*x[1]^2*x[2]*x[3]^2 +
>         5*x[1]*x[2]^2*x[3]^2 + 7*x[2]^3*x[3]^2 + 9*x[1]^2*x[3]^3 +
>         12*x[1]*x[2]*x[3]^3 + 2*x[2]^2*x[3]^3 + 9*x[1]*x[3]^4 + 2*x[2]*x[3]^4 +
>         x[3]^5 + 3*x[1]^3*x[3]*x[4] + 5*x[1]^2*x[2]*x[3]*x[4] +
>         7*x[1]*x[2]^2*x[3]*x[4] + 13*x[2]^3*x[3]*x[4] + 11*x[1]^2*x[3]^2*x[4] +
>         4*x[1]*x[2]*x[3]^2*x[4] + 11*x[2]^2*x[3]^2*x[4] + 14*x[1]*x[3]^3*x[4] +
>         16*x[2]*x[3]^3*x[4] + 15*x[1]^2*x[3]*x[4]^2 + 11*x[1]*x[2]*x[3]*x[4]^2 +
>         5*x[2]^2*x[3]*x[4]^2 + 6*x[1]*x[3]^2*x[4]^2 + 9*x[2]*x[3]^2*x[4]^2 +
>         16*x[3]^3*x[4]^2 + 9*x[2]*x[3]*x[4]^3 + 15*x[3]^2*x[4]^3 +
>         14*x[3]*x[4]^4 + 2*x[1]^4*x[5] + 11*x[1]^3*x[2]*x[5] +
>         13*x[1]^2*x[2]^2*x[5] + 4*x[1]*x[2]^3*x[5] + 16*x[2]^4*x[5] +
>         2*x[1]^3*x[3]*x[5] + 13*x[1]^2*x[2]*x[3]*x[5] + 9*x[1]*x[2]^2*x[3]*x[5]
>         + 5*x[2]^3*x[3]*x[5] + 5*x[1]^2*x[3]^2*x[5] + 3*x[1]*x[2]*x[3]^2*x[5] +
>         8*x[2]^2*x[3]^2*x[5] + x[1]*x[3]^3*x[5] + 7*x[2]*x[3]^3*x[5] +
>         3*x[3]^4*x[5] + 9*x[1]^3*x[4]*x[5] + 3*x[1]^2*x[2]*x[4]*x[5] +
>         10*x[1]*x[2]^2*x[4]*x[5] + 12*x[2]^3*x[4]*x[5] +
>         10*x[1]^2*x[3]*x[4]*x[5] + 12*x[1]*x[2]*x[3]*x[4]*x[5] +
>         3*x[2]^2*x[3]*x[4]*x[5] + 11*x[1]*x[3]^2*x[4]*x[5] +
>         5*x[2]*x[3]^2*x[4]*x[5] + 7*x[3]^3*x[4]*x[5] + 5*x[1]^2*x[4]^2*x[5] +
>         x[1]*x[2]*x[4]^2*x[5] + 8*x[2]^2*x[4]^2*x[5] + 4*x[1]*x[3]*x[4]^2*x[5] +
>         16*x[2]*x[3]*x[4]^2*x[5] + 8*x[3]^2*x[4]^2*x[5] + 10*x[1]*x[4]^3*x[5] +
>         7*x[2]*x[4]^3*x[5] + 2*x[3]*x[4]^3*x[5] + 16*x[4]^4*x[5] +
>         14*x[1]^3*x[5]^2 + x[1]^2*x[2]*x[5]^2 + 10*x[1]*x[2]^2*x[5]^2 +
>         11*x[2]^3*x[5]^2 + 12*x[1]^2*x[3]*x[5]^2 + 12*x[1]*x[2]*x[3]*x[5]^2 +
>         13*x[2]^2*x[3]*x[5]^2 + 4*x[1]*x[3]^2*x[5]^2 + 15*x[2]*x[3]^2*x[5]^2 +
>         10*x[3]^3*x[5]^2 + 2*x[1]^2*x[4]*x[5]^2 + 5*x[2]^2*x[4]*x[5]^2 +
>         6*x[1]*x[3]*x[4]*x[5]^2 + 5*x[2]*x[3]*x[4]*x[5]^2 + 7*x[3]^2*x[4]*x[5]^2
>         + x[1]*x[4]^2*x[5]^2 + 5*x[2]*x[4]^2*x[5]^2 + x[3]*x[4]^2*x[5]^2 +
>         14*x[4]^3*x[5]^2 + 8*x[1]^2*x[5]^3 + 11*x[1]*x[2]*x[5]^3 +
>         3*x[2]^2*x[5]^3 + 9*x[1]*x[3]*x[5]^3 + 6*x[2]*x[3]*x[5]^3 +
>         13*x[3]^2*x[5]^3 + 15*x[1]*x[4]*x[5]^3 + 2*x[2]*x[4]*x[5]^3 +
>         5*x[3]*x[4]*x[5]^3 + 14*x[4]^2*x[5]^3 + 11*x[1]*x[5]^4 + 8*x[2]*x[5]^4 +
>         5*x[3]*x[5]^4 + 15*x[4]*x[5]^4 + 3*x[5]^5,
>     13*x[1]^4*x[3] + 8*x[1]^3*x[2]*x[3] + 14*x[1]^2*x[2]^2*x[3] +
>         3*x[1]*x[2]^3*x[3] + 11*x[2]^4*x[3] + 7*x[1]^3*x[3]^2 +
>         3*x[1]^2*x[2]*x[3]^2 + 12*x[2]^3*x[3]^2 + 3*x[1]^2*x[3]^3 +
>         13*x[1]*x[2]*x[3]^3 + 3*x[2]^2*x[3]^3 + 7*x[1]*x[3]^4 + 2*x[2]*x[3]^4 +
>         7*x[3]^5 + 13*x[1]^3*x[3]*x[4] + 6*x[1]^2*x[2]*x[3]*x[4] +
```

```
>         6*x[1]*x[2]^2*x[3]*x[4] + 6*x[2]^3*x[3]*x[4] + 2*x[1]^2*x[3]^2*x[4] +
>         15*x[1]*x[2]*x[3]^2*x[4] + 14*x[2]^2*x[3]^2*x[4] + 3*x[1]*x[3]^3*x[4] +
>         16*x[2]*x[3]^3*x[4] + 3*x[3]^4*x[4] + 6*x[1]^2*x[3]*x[4]^2 +
>         10*x[2]^2*x[3]*x[4]^2 + 7*x[2]*x[3]^2*x[4]^2 + 13*x[1]*x[3]*x[4]^3 +
>         5*x[2]*x[3]*x[4]^3 + 15*x[3]^2*x[4]^3 + 13*x[3]*x[4]^4 + 15*x[1]^4*x[5]
>         + 15*x[1]^3*x[2]*x[5] + 2*x[1]^2*x[2]^2*x[5] + 16*x[1]*x[2]^3*x[5] +
>         16*x[2]^4*x[5] + 14*x[1]^3*x[3]*x[5] + 4*x[1]^2*x[2]*x[3]*x[5] +
>         10*x[1]*x[2]^2*x[3]*x[5] + 4*x[2]^3*x[3]*x[5] + 8*x[1]^2*x[3]^2*x[5] +
>         5*x[1]*x[2]*x[3]^2*x[5] + 11*x[2]^2*x[3]^2*x[5] + 12*x[2]*x[3]^3*x[5] +
>         2*x[3]^4*x[5] + 15*x[1]^2*x[2]*x[4]*x[5] + 6*x[2]^3*x[4]*x[5] +
>         9*x[1]^2*x[3]*x[4]*x[5] + 9*x[1]*x[2]*x[3]*x[4]*x[5] +
>         15*x[2]^2*x[3]*x[4]*x[5] + 14*x[1]*x[3]^2*x[4]*x[5] +
>         13*x[2]*x[3]^2*x[4]*x[5] + 6*x[3]^3*x[4]*x[5] + 4*x[1]^2*x[4]^2*x[5] +
>         7*x[1]*x[2]*x[4]^2*x[5] + 3*x[2]^2*x[4]^2*x[5] + 8*x[1]*x[3]*x[4]^2*x[5]
>         + 8*x[2]*x[3]*x[4]^2*x[5] + 3*x[3]^2*x[4]^2*x[5] + 15*x[1]*x[4]^3*x[5] +
>         3*x[2]*x[4]^3*x[5] + 8*x[3]*x[4]^3*x[5] + 2*x[4]^4*x[5] +
>         2*x[1]^3*x[5]^2 + 6*x[1]^2*x[2]*x[5]^2 + x[1]*x[2]^2*x[5]^2 +
>         7*x[2]^3*x[5]^2 + 3*x[1]^2*x[3]*x[5]^2 + 16*x[1]*x[2]*x[3]*x[5]^2 +
>         10*x[2]^2*x[3]*x[5]^2 + 10*x[1]*x[3]^2*x[5]^2 + 13*x[2]*x[3]^2*x[5]^2 +
>         2*x[3]^3*x[5]^2 + 4*x[1]^2*x[4]*x[5]^2 + x[1]*x[2]*x[4]*x[5]^2 +
>         9*x[2]^2*x[4]*x[5]^2 + 16*x[1]*x[3]*x[4]*x[5]^2 +
>         8*x[2]*x[3]*x[4]*x[5]^2 + 11*x[1]*x[4]^2*x[5]^2 + 11*x[2]*x[4]^2*x[5]^2
>         + 4*x[3]*x[4]^2*x[5]^2 + 10*x[4]^3*x[5]^2 + 10*x[1]^2*x[5]^3 +
>         14*x[2]^2*x[5]^3 + 16*x[1]*x[3]*x[5]^3 + 13*x[2]*x[3]*x[5]^3 +
>         15*x[3]^2*x[5]^3 + 16*x[1]*x[4]*x[5]^3 + 3*x[2]*x[4]*x[5]^3 +
>         4*x[3]*x[4]*x[5]^3 + 2*x[4]^2*x[5]^3 + x[1]*x[5]^4 + 7*x[2]*x[5]^4 +
>         7*x[4]*x[5]^4 + 2*x[5]^5,
>     15*x[1]^4*x[3] + 11*x[1]^3*x[2]*x[3] + 5*x[1]^2*x[2]^2*x[3] +
>         5*x[1]*x[2]^3*x[3] + 15*x[2]^4*x[3] + 12*x[1]^3*x[3]^2 +
>         5*x[1]^2*x[2]*x[3]^2 + 10*x[1]*x[2]^2*x[3]^2 + 6*x[2]^3*x[3]^2 +
>         15*x[1]^2*x[3]^3 + 14*x[1]*x[2]*x[3]^3 + 10*x[2]^2*x[3]^3 + x[1]*x[3]^4
>         + 14*x[2]*x[3]^4 + 4*x[3]^5 + 15*x[1]^4*x[4] + 15*x[1]^3*x[2]*x[4] +
>         2*x[1]^2*x[2]^2*x[4] + 16*x[1]*x[2]^3*x[4] + 16*x[2]^4*x[4] +
>         14*x[1]^3*x[3]*x[4] + 2*x[1]^2*x[2]*x[3]*x[4] + 15*x[1]*x[2]^2*x[3]*x[4]
>         + 2*x[2]^3*x[3]*x[4] + 15*x[1]^2*x[3]^2*x[4] + 13*x[1]*x[2]*x[3]^2*x[4]
>         + 5*x[2]^2*x[3]^2*x[4] + 16*x[1]*x[3]^3*x[4] + 6*x[2]*x[3]^3*x[4] +
>         4*x[3]^4*x[4] + 15*x[1]^2*x[2]*x[4]^2 + 6*x[2]^3*x[4]^2 +
>         7*x[1]^2*x[3]*x[4]^2 + 12*x[1]*x[2]*x[3]*x[4]^2 + 15*x[2]^2*x[3]*x[4]^2
>         + x[1]*x[3]^2*x[4]^2 + 4*x[3]^3*x[4]^2 + 4*x[1]^2*x[4]^3 +
>         7*x[1]*x[2]*x[4]^3 + 3*x[2]^2*x[4]^3 + 13*x[1]*x[3]*x[4]^3 +
>         12*x[3]^2*x[4]^3 + 15*x[1]*x[4]^4 + 3*x[2]*x[4]^4 + 7*x[3]*x[4]^4 +
>         2*x[4]^5 + 14*x[1]^3*x[3]*x[5] + 10*x[1]^2*x[2]*x[3]*x[5] +
>         13*x[1]^2*x[3]^2*x[5] + 14*x[1]*x[2]*x[3]^2*x[5] + 8*x[2]^2*x[3]^2*x[5]
>         + 3*x[1]*x[3]^3*x[5] + 4*x[2]*x[3]^3*x[5] + 2*x[3]^4*x[5] +
>         2*x[1]^3*x[4]*x[5] + 6*x[1]^2*x[2]*x[4]*x[5] + x[1]*x[2]^2*x[4]*x[5] +
>         7*x[2]^3*x[4]*x[5] + 2*x[1]^2*x[3]*x[4]*x[5] +
>         2*x[1]*x[2]*x[3]*x[4]*x[5] + 5*x[2]^2*x[3]*x[4]*x[5] +
>         8*x[2]*x[3]^2*x[4]*x[5] + 12*x[3]^3*x[4]*x[5] + 4*x[1]^2*x[4]^2*x[5] +
```

```
>         x[1]*x[2]*x[4]^2*x[5] + 9*x[2]^2*x[4]^2*x[5] + 3*x[1]*x[3]*x[4]^2*x[5] +
>         6*x[2]*x[3]*x[4]^2*x[5] + 8*x[3]^2*x[4]^2*x[5] + 11*x[1]*x[4]^3*x[5] +
>         11*x[2]*x[4]^3*x[5] + x[3]*x[4]^3*x[5] + 10*x[4]^4*x[5] +
>         3*x[1]*x[2]*x[3]*x[5]^2 + 5*x[2]^2*x[3]*x[5]^2 + 11*x[1]*x[3]^2*x[5]^2 +
>         x[2]*x[3]^2*x[5]^2 + 9*x[3]^3*x[5]^2 + 10*x[1]^2*x[4]*x[5]^2 +
>         14*x[2]^2*x[4]*x[5]^2 + 13*x[1]*x[3]*x[4]*x[5]^2 +
>         9*x[2]*x[3]*x[4]*x[5]^2 + 4*x[3]^2*x[4]*x[5]^2 + 16*x[1]*x[4]^2*x[5]^2 +
>         3*x[2]*x[4]^2*x[5]^2 + 6*x[3]*x[4]^2*x[5]^2 + 2*x[4]^3*x[5]^2 +
>         3*x[1]*x[3]*x[5]^3 + 4*x[2]*x[3]*x[5]^3 + 13*x[3]^2*x[5]^3 +
>         x[1]*x[4]*x[5]^3 + 7*x[2]*x[4]*x[5]^3 + 13*x[3]*x[4]*x[5]^3 +
>         7*x[4]^2*x[5]^3 + 4*x[3]*x[5]^4 + 2*x[4]*x[5]^4,
>     15*x[1]^3*x[2]^2 + 11*x[1]^2*x[2]^3 + 14*x[1]*x[2]^4 + x[2]^5 +
>         2*x[1]^3*x[2]*x[3] + 11*x[1]*x[2]^3*x[3] + 7*x[2]^4*x[3] +
>         3*x[1]^2*x[2]*x[3]^2 + 4*x[1]*x[2]^2*x[3]^2 + 13*x[2]^3*x[3]^2 +
>         x[1]*x[2]*x[3]^3 + 4*x[2]*x[3]^4 + 16*x[1]^4*x[4] + 2*x[1]^3*x[2]*x[4] +
>         7*x[1]^2*x[2]^2*x[4] + 4*x[1]*x[2]^3*x[4] + 4*x[2]^4*x[4] +
>         9*x[1]^3*x[3]*x[4] + x[1]^2*x[2]*x[3]*x[4] + 9*x[1]*x[2]^2*x[3]*x[4] +
>         4*x[2]^3*x[3]*x[4] + 7*x[1]^2*x[3]^2*x[4] + 5*x[1]*x[2]*x[3]^2*x[4] +
>         11*x[2]^2*x[3]^2*x[4] + 15*x[1]*x[3]^3*x[4] + 15*x[2]*x[3]^3*x[4] +
>         x[3]^4*x[4] + 9*x[1]^2*x[2]*x[4]^2 + 16*x[1]*x[2]^2*x[4]^2 +
>         9*x[2]^3*x[4]^2 + 3*x[1]^2*x[3]*x[4]^2 + 2*x[1]*x[2]*x[3]*x[4]^2 +
>         14*x[2]^2*x[3]*x[4]^2 + 11*x[1]*x[3]^2*x[4]^2 + 16*x[2]*x[3]^2*x[4]^2 +
>         4*x[3]^3*x[4]^2 + x[1]^2*x[4]^3 + 8*x[1]*x[2]*x[4]^3 + 14*x[2]^2*x[4]^3
>         + 3*x[1]*x[3]*x[4]^3 + 16*x[2]*x[3]*x[4]^3 + 12*x[3]^2*x[4]^3 +
>         7*x[1]*x[4]^4 + 5*x[2]*x[4]^4 + 4*x[3]*x[4]^4 + 2*x[4]^5 +
>         8*x[1]^3*x[2]*x[5] + 5*x[1]^2*x[2]^2*x[5] + x[1]*x[2]^3*x[5] +
>         16*x[2]^4*x[5] + 10*x[1]^2*x[2]*x[3]*x[5] + 12*x[1]*x[2]^2*x[3]*x[5] +
>         9*x[2]^3*x[3]*x[5] + 15*x[1]*x[2]*x[3]^2*x[5] + 13*x[2]^2*x[3]^2*x[5] +
>         4*x[2]*x[3]^3*x[5] + 8*x[1]^3*x[4]*x[5] + 16*x[1]^2*x[2]*x[4]*x[5] +
>         11*x[1]*x[2]^2*x[4]*x[5] + 8*x[2]^3*x[4]*x[5] + 10*x[1]^2*x[3]*x[4]*x[5]
>         + 15*x[1]*x[2]*x[3]*x[4]*x[5] + 4*x[2]^2*x[3]*x[4]*x[5] +
>         9*x[1]*x[3]^2*x[4]*x[5] + 16*x[2]*x[3]^2*x[4]*x[5] + 11*x[3]^3*x[4]*x[5]
>         + 4*x[1]^2*x[4]^2*x[5] + 6*x[1]*x[2]*x[4]^2*x[5] + 10*x[2]^2*x[4]^2*x[5]
>         + 11*x[1]*x[3]*x[4]^2*x[5] + 11*x[2]*x[3]*x[4]^2*x[5] +
>         14*x[3]^2*x[4]^2*x[5] + 10*x[1]*x[4]^3*x[5] + 6*x[2]*x[4]^3*x[5] +
>         5*x[3]*x[4]^3*x[5] + 4*x[4]^4*x[5] + 16*x[1]^2*x[2]*x[5]^2 +
>         4*x[1]*x[2]^2*x[5]^2 + 11*x[2]^3*x[5]^2 + 9*x[1]*x[2]*x[3]*x[5]^2 +
>         16*x[2]^2*x[3]*x[5]^2 + 8*x[2]*x[3]^2*x[5]^2 + 3*x[1]^2*x[4]*x[5]^2 +
>         10*x[1]*x[2]*x[4]*x[5]^2 + 9*x[2]^2*x[4]*x[5]^2 +
>         10*x[1]*x[3]*x[4]*x[5]^2 + 11*x[2]*x[3]*x[4]*x[5]^2 +
>         11*x[3]^2*x[4]*x[5]^2 + 3*x[1]*x[4]^2*x[5]^2 + 14*x[2]*x[4]^2*x[5]^2 +
>         7*x[3]*x[4]^2*x[5]^2 + 3*x[4]^3*x[5]^2 + x[1]*x[2]*x[5]^3 +
>         15*x[2]^2*x[5]^3 + 10*x[2]*x[3]*x[5]^3 + x[1]*x[4]*x[5]^3 +
>         2*x[2]*x[4]*x[5]^3 + 5*x[3]*x[4]*x[5]^3 + 6*x[4]^2*x[5]^3 +
>         6*x[2]*x[5]^4 + 16*x[4]*x[5]^4,
>     2*x[1]^4*x[2] + 2*x[1]^3*x[2]^2 + 15*x[1]^2*x[2]^3 + x[1]*x[2]^4 + x[2]^5 +
>         16*x[1]^4*x[3] + 8*x[1]^3*x[2]*x[3] + 7*x[1]^2*x[2]^2*x[3] +
>         10*x[1]*x[2]^3*x[3] + 10*x[2]^4*x[3] + 10*x[1]^3*x[3]^2 +
```

```
>        11*x[1]^2*x[2]*x[3]^2 + 9*x[1]*x[2]^2*x[3]^2 + 10*x[2]^3*x[3]^2 +
>        7*x[1]^2*x[3]^3 + 10*x[1]*x[2]*x[3]^3 + 4*x[2]^2*x[3]^3 + 13*x[1]*x[3]^4
>        + 3*x[3]^5 + 2*x[1]^2*x[2]^2*x[4] + 11*x[2]^4*x[4] + 5*x[1]^3*x[3]*x[4]
>        + 15*x[1]^2*x[2]*x[3]*x[4] + 16*x[1]*x[2]^2*x[3]*x[4] +
>        4*x[2]^3*x[3]*x[4] + 16*x[1]^2*x[3]^2*x[4] + 12*x[1]*x[2]*x[3]^2*x[4] +
>        4*x[2]^2*x[3]^2*x[4] + 15*x[1]*x[3]^3*x[4] + 14*x[2]*x[3]^3*x[4] +
>        5*x[3]^4*x[4] + 13*x[1]^2*x[2]*x[4]^2 + 10*x[1]*x[2]^2*x[4]^2 +
>        14*x[2]^3*x[4]^2 + 10*x[1]^2*x[3]*x[4]^2 + 9*x[1]*x[2]*x[3]*x[4]^2 +
>        2*x[2]^2*x[3]*x[4]^2 + x[1]*x[3]^2*x[4]^2 + 15*x[2]*x[3]^2*x[4]^2 +
>        15*x[3]^3*x[4]^2 + 2*x[1]*x[2]*x[4]^3 + 14*x[2]^2*x[4]^3 +
>        15*x[1]*x[3]*x[4]^3 + 6*x[2]*x[3]*x[4]^3 + 3*x[3]^2*x[4]^3 +
>        15*x[2]*x[4]^4 + 13*x[3]*x[4]^4 + 15*x[1]^3*x[2]*x[5] +
>        11*x[1]^2*x[2]^2*x[5] + 16*x[1]*x[2]^3*x[5] + 10*x[2]^4*x[5] +
>        4*x[1]^3*x[3]*x[5] + 10*x[1]^2*x[2]*x[3]*x[5] + 4*x[2]^3*x[3]*x[5] +
>        3*x[1]^2*x[3]^2*x[5] + 6*x[1]*x[2]*x[3]^2*x[5] + 15*x[2]^2*x[3]^2*x[5] +
>        8*x[1]*x[3]^3*x[5] + 8*x[2]*x[3]^3*x[5] + x[3]^4*x[5] +
>        13*x[1]^2*x[2]*x[4]*x[5] + 16*x[1]*x[2]^2*x[4]*x[5] + 8*x[2]^3*x[4]*x[5]
>        + 5*x[1]^2*x[3]*x[4]*x[5] + 16*x[1]*x[2]*x[3]*x[4]*x[5] +
>        x[2]^2*x[3]*x[4]*x[5] + 10*x[1]*x[3]^2*x[4]*x[5] +
>        9*x[2]*x[3]^2*x[4]*x[5] + 12*x[3]^3*x[4]*x[5] + 6*x[1]*x[2]*x[4]^2*x[5]
>        + 6*x[2]^2*x[4]^2*x[5] + 12*x[1]*x[3]*x[4]^2*x[5] +
>        16*x[2]*x[3]*x[4]^2*x[5] + 6*x[3]^2*x[4]^2*x[5] + 7*x[2]*x[4]^3*x[5] +
>        15*x[3]*x[4]^3*x[5] + 7*x[1]^2*x[2]*x[5]^2 + 3*x[2]^3*x[5]^2 +
>        3*x[1]^2*x[3]*x[5]^2 + 11*x[1]*x[2]*x[3]*x[5]^2 + x[2]^2*x[3]*x[5]^2 +
>        12*x[1]*x[3]^2*x[5]^2 + 8*x[2]*x[3]^2*x[5]^2 + 4*x[3]^3*x[5]^2 +
>        x[1]*x[2]*x[4]*x[5]^2 + 14*x[2]^2*x[4]*x[5]^2 + 8*x[1]*x[3]*x[4]*x[5]^2
>        + 13*x[2]*x[3]*x[4]*x[5]^2 + 4*x[3]^2*x[4]*x[5]^2 +
>        15*x[2]*x[4]^2*x[5]^2 + 16*x[1]*x[2]*x[5]^3 + 10*x[2]^2*x[5]^3 +
>        9*x[1]*x[3]*x[5]^3 + 14*x[2]*x[3]*x[5]^3 + 12*x[3]^2*x[5]^3 +
>        10*x[2]*x[4]*x[5]^3 + x[3]*x[4]*x[5]^3 + 15*x[2]*x[5]^4 + 9*x[3]*x[5]^4,
>    9*x[1]^4*x[2] + 14*x[1]^3*x[2]^2 + 5*x[1]^2*x[2]^3 + 2*x[1]*x[2]^4 +
>        2*x[1]^3*x[2]*x[3] + 7*x[1]^2*x[2]^2*x[3] + 5*x[1]*x[2]^3*x[3] +
>        7*x[2]^4*x[3] + 9*x[1]^2*x[2]*x[3]^2 + 12*x[1]*x[2]^2*x[3]^2 +
>        2*x[2]^3*x[3]^2 + 9*x[1]*x[2]*x[3]^3 + 2*x[2]^2*x[3]^3 + x[2]*x[3]^4 +
>        3*x[1]^3*x[2]*x[4] + 5*x[1]^2*x[2]^2*x[4] + 7*x[1]*x[2]^3*x[4] +
>        13*x[2]^4*x[4] + 11*x[1]^2*x[2]*x[3]*x[4] + 4*x[1]*x[2]^2*x[3]*x[4] +
>        11*x[2]^3*x[3]*x[4] + 14*x[1]*x[2]*x[3]^2*x[4] + 16*x[2]^2*x[3]^2*x[4] +
>        15*x[1]^2*x[2]*x[4]^2 + 11*x[1]*x[2]^2*x[4]^2 + 5*x[2]^3*x[4]^2 +
>        6*x[1]*x[2]*x[3]*x[4]^2 + 9*x[2]^2*x[3]*x[4]^2 + 16*x[2]*x[3]^2*x[4]^2 +
>        9*x[2]^2*x[4]^3 + 15*x[2]*x[3]*x[4]^3 + 14*x[2]*x[4]^4 + 16*x[1]^4*x[5]
>        + 16*x[1]^3*x[2]*x[5] + 16*x[1]^2*x[2]^2*x[5] + 3*x[1]*x[2]^3*x[5] +
>        x[2]^4*x[5] + 9*x[1]^3*x[3]*x[5] + x[1]^2*x[2]*x[3]*x[5] +
>        15*x[1]*x[2]^2*x[3]*x[5] + 10*x[2]^3*x[3]*x[5] + 7*x[1]^2*x[3]^2*x[5] +
>        8*x[1]*x[2]*x[3]^2*x[5] + x[2]^2*x[3]^2*x[5] + 15*x[1]*x[3]^3*x[5] +
>        5*x[2]*x[3]^3*x[5] + x[3]^4*x[5] + 11*x[1]^2*x[2]*x[4]*x[5] +
>        14*x[1]*x[2]^2*x[4]*x[5] + 9*x[2]^3*x[4]*x[5] + 3*x[1]^2*x[3]*x[4]*x[5]
>        + 11*x[1]*x[2]*x[3]*x[4]*x[5] + 14*x[2]^2*x[3]*x[4]*x[5] +
>        11*x[1]*x[3]^2*x[4]*x[5] + 13*x[2]*x[3]^2*x[4]*x[5] + 4*x[3]^3*x[4]*x[5]
```

```
>        + x[1]^2*x[4]^2*x[5] + 5*x[1]*x[2]*x[4]^2*x[5] + 3*x[2]^2*x[4]^2*x[5] +
>        3*x[1]*x[3]*x[4]^2*x[5] + 11*x[2]*x[3]*x[4]^2*x[5] +
>        12*x[3]^2*x[4]^2*x[5] + 7*x[1]*x[4]^3*x[5] + 8*x[2]*x[4]^3*x[5] +
>        4*x[3]*x[4]^3*x[5] + 2*x[4]^4*x[5] + 8*x[1]^3*x[5]^2 +
>        6*x[1]^2*x[2]*x[5]^2 + 11*x[1]*x[2]^2*x[5]^2 + 14*x[2]^3*x[5]^2 +
>        10*x[1]^2*x[3]*x[5]^2 + 14*x[1]*x[2]*x[3]*x[5]^2 + 2*x[2]^2*x[3]*x[5]^2
>        + 9*x[1]*x[3]^2*x[5]^2 + 6*x[2]*x[3]^2*x[5]^2 + 11*x[3]^3*x[5]^2 +
>        4*x[1]^2*x[4]*x[5]^2 + 12*x[1]*x[2]*x[4]*x[5]^2 + 13*x[2]^2*x[4]*x[5]^2
>        + 11*x[1]*x[3]*x[4]*x[5]^2 + 10*x[2]*x[3]*x[4]*x[5]^2 +
>        14*x[3]^2*x[4]*x[5]^2 + 10*x[1]*x[4]^2*x[5]^2 + 4*x[2]*x[4]^2*x[5]^2 +
>        5*x[3]*x[4]^2*x[5]^2 + 4*x[4]^3*x[5]^2 + 3*x[1]^2*x[5]^3 +
>        13*x[1]*x[2]*x[5]^3 + 6*x[2]^2*x[5]^3 + 10*x[1]*x[3]*x[5]^3 +
>        2*x[2]*x[3]*x[5]^3 + 11*x[3]^2*x[5]^3 + 3*x[1]*x[4]*x[5]^3 +
>        11*x[2]*x[4]*x[5]^3 + 7*x[3]*x[4]*x[5]^3 + 3*x[4]^2*x[5]^3 + x[1]*x[5]^4
>        + 9*x[2]*x[5]^4 + 5*x[3]*x[5]^4 + 6*x[4]*x[5]^4 + 16*x[5]^5,
>    13*x[1]^4*x[2] + 8*x[1]^3*x[2]^2 + 14*x[1]^2*x[2]^3 + 3*x[1]*x[2]^4 +
>        11*x[2]^5 + 7*x[1]^3*x[2]*x[3] + 3*x[1]^2*x[2]^2*x[3] + 12*x[2]^4*x[3] +
>        3*x[1]^2*x[2]*x[3]^2 + 13*x[1]*x[2]^2*x[3]^2 + 3*x[2]^3*x[3]^2 +
>        7*x[1]*x[2]*x[3]^3 + 2*x[2]^2*x[3]^3 + 7*x[2]*x[3]^4 +
>        13*x[1]^3*x[2]*x[4] + 6*x[1]^2*x[2]^2*x[4] + 6*x[1]*x[2]^3*x[4] +
>        6*x[2]^4*x[4] + 2*x[1]^2*x[2]*x[3]*x[4] + 15*x[1]*x[2]^2*x[3]*x[4] +
>        14*x[2]^3*x[3]*x[4] + 3*x[1]*x[2]*x[3]^2*x[4] + 16*x[2]^2*x[3]^2*x[4] +
>        3*x[2]*x[3]^3*x[4] + 6*x[1]^2*x[2]*x[4]^2 + 10*x[2]^3*x[4]^2 +
>        7*x[2]^2*x[3]*x[4]^2 + 13*x[1]*x[2]*x[4]^3 + 5*x[2]^2*x[4]^3 +
>        15*x[2]*x[3]*x[4]^3 + 13*x[2]*x[4]^4 + 16*x[1]^4*x[5] +
>        5*x[1]^3*x[2]*x[5] + 11*x[1]^2*x[2]^2*x[5] + 3*x[1]*x[2]^3*x[5] +
>        14*x[2]^4*x[5] + 10*x[1]^3*x[3]*x[5] + 2*x[1]^2*x[2]*x[3]*x[5] +
>        14*x[1]*x[2]^2*x[3]*x[5] + 4*x[2]^3*x[3]*x[5] + 7*x[1]^2*x[3]^2*x[5] +
>        10*x[1]*x[2]*x[3]^2*x[5] + 16*x[2]^2*x[3]^2*x[5] + 13*x[1]*x[3]^3*x[5] +
>        2*x[2]*x[3]^3*x[5] + 3*x[3]^4*x[5] + 5*x[1]^3*x[4]*x[5] +
>        7*x[1]^2*x[2]*x[4]*x[5] + 8*x[1]*x[2]^2*x[4]*x[5] + 2*x[2]^3*x[4]*x[5] +
>        16*x[1]^2*x[3]*x[4]*x[5] + 9*x[1]*x[2]*x[3]*x[4]*x[5] +
>        15*x[1]*x[3]^2*x[4]*x[5] + 3*x[2]*x[3]^2*x[4]*x[5] + 5*x[3]^3*x[4]*x[5]
>        + 10*x[1]^2*x[4]^2*x[5] + 10*x[2]^2*x[4]^2*x[5] + x[1]*x[3]*x[4]^2*x[5]
>        + x[2]*x[3]*x[4]^2*x[5] + 15*x[3]^2*x[4]^2*x[5] + 15*x[1]*x[4]^3*x[5] +
>        14*x[2]*x[4]^3*x[5] + 3*x[3]*x[4]^3*x[5] + 13*x[4]^4*x[5] +
>        4*x[1]^3*x[5]^2 + 13*x[1]^2*x[2]*x[5]^2 + 16*x[1]*x[2]^2*x[5]^2 +
>        14*x[2]^3*x[5]^2 + 3*x[1]^2*x[3]*x[5]^2 + 16*x[1]*x[2]*x[3]*x[5]^2 +
>        11*x[2]^2*x[3]*x[5]^2 + 8*x[1]*x[3]^2*x[5]^2 + 10*x[2]*x[3]^2*x[5]^2 +
>        x[3]^3*x[5]^2 + 5*x[1]^2*x[4]*x[5]^2 + 15*x[1]*x[2]*x[4]*x[5]^2 +
>        9*x[2]^2*x[4]*x[5]^2 + 10*x[1]*x[3]*x[4]*x[5]^2 +
>        9*x[2]*x[3]*x[4]*x[5]^2 + 12*x[3]^2*x[4]*x[5]^2 + 12*x[1]*x[4]^2*x[5]^2
>        + 3*x[2]*x[4]^2*x[5]^2 + 6*x[3]*x[4]^2*x[5]^2 + 15*x[4]^3*x[5]^2 +
>        3*x[1]^2*x[5]^3 + 10*x[1]*x[2]*x[5]^3 + 14*x[2]^2*x[5]^3 +
>        12*x[1]*x[3]*x[5]^3 + 6*x[2]*x[3]*x[5]^3 + 4*x[3]^2*x[5]^3 +
>        8*x[1]*x[4]*x[5]^3 + 4*x[3]*x[4]*x[5]^3 + 9*x[1]*x[5]^4 + 14*x[2]*x[5]^4
```

```
>        + 12*x[3]*x[5]^4 + x[4]*x[5]^4 + 9*x[5]^5]);
```

We check a few of the invariants of X.

```
> Dimension(X);
2
> IsNonsingular(X);
true
> ArithmeticGenus(X);
0
> // Get the sectional genus of X -- ie the genus of a hyperplane section.
> ArithmeticGenus(X meet Scheme(P,P.1));
9
```

Now we construct the canonical sheaf and hyperplane sheaf and check intersection numbers.

```
> KX := CanonicalSheaf(X);
> HX := StructureSheaf(X,1); // hyperplane sheaf
> IntersectionPairing(HX,HX); // should be 10 = Degree(X)
10
> Degree(X);
10
> IntersectionPairing(KX,HX); // should be 6
6
> IntersectionPairing(KX,KX); // should be -9 : lots of exceptional curves!
-9
```

We now get the adjunction map as a divisor map, compute its image X_1 and check some of the invariants of X_1 as well as its corresponding intersection numbers.

```
> mp1,X1 := DivisorMap(Twist(KX,1));
> Dimension(Ambient(X1)); Dimension(X1);
8
2
> KX1 := CanonicalSheaf(X1);
> HX1 := StructureSheaf(X1,1); // hyperplane sheaf of X1
> IntersectionPairing(HX1,HX1); // should be 13 = degree X1
13
> IntersectionPairing(KX1,HX1); // should be -3
-3
> IntersectionPairing(KX1,KX1); // should be -2 : fewer exceptional curves!
-2
```

We construct a second adjunction map to get X_2 and check it as above.

```
> mp2,X2 := DivisorMap(Twist(KX1,1));
> Dimension(Ambient(X2)); Dimension(X2);
5
2
> KX2 := CanonicalSheaf(X2);
> HX2 := StructureSheaf(X2,1); // hyperplane sheaf X2
```

```
> IntersectionPairing(HX2,HX2); // = degree X2 = 5
5
> IntersectionPairing(KX2,HX2); // should be -5
-5
> IntersectionPairing(KX2,KX2); // should be 5
5
```

Now X_2 should be a degree five Del Pezzo surface with $\mathcal{K}_X \simeq \mathcal{O}_X(-1)$. This last isomorphism can be verified by checking that there is a degree -2 isomorphism from $\mathcal{K}_X$ to $\mathcal{O}_X(1)$! The scheme X_2 is much simpler than X: it is defined by five degree 2 polynomials.

```
> boo,d := IsIsomorphicWithTwist(KX2,HX2);
> boo; d;
true
-2
> MinimalBasis(Ideal(X2));
Scheme over GF(17) defined by
y[1]^2 + y[3]^2 + y[1]*y[4] + 15*y[2]*y[4] + 8*y[3]*y[4] + 6*y[4]^2 +
    2*y[1]*y[5] + 12*y[2]*y[5] + y[3]*y[5] + 4*y[4]*y[5] + 4*y[5]^2 +
    6*y[1]*y[6] + 10*y[2]*y[6] + 7*y[3]*y[6] + 7*y[5]*y[6] + 16*y[6]^2,
y[1]*y[2] + 13*y[3]^2 + 3*y[1]*y[4] + 14*y[2]*y[4] + 13*y[3]*y[4] + 5*y[4]^2 +
    14*y[1]*y[5] + 10*y[2]*y[5] + 2*y[3]*y[5] + 9*y[4]*y[5] + 6*y[5]^2 +
    4*y[1]*y[6] + 13*y[2]*y[6] + 10*y[3]*y[6] + 3*y[4]*y[6] + y[5]*y[6] +
    12*y[6]^2,
y[2]^2 + 16*y[3]^2 + 15*y[1]*y[4] + 3*y[3]*y[4] + y[4]^2 + 10*y[1]*y[5] +
    12*y[2]*y[5] + 10*y[3]*y[5] + 11*y[4]*y[5] + 9*y[5]^2 + 5*y[1]*y[6] +
    3*y[2]*y[6] + 2*y[3]*y[6] + 15*y[4]*y[6] + 12*y[5]*y[6] + 5*y[6]^2,
y[1]*y[3] + 13*y[3]^2 + y[1]*y[4] + 11*y[3]*y[4] + y[4]^2 + 16*y[1]*y[5] +
    y[2]*y[5] + 15*y[3]*y[5] + 3*y[4]*y[5] + 7*y[1]*y[6] + 3*y[2]*y[6] +
    9*y[3]*y[6] + 10*y[4]*y[6] + 8*y[5]*y[6] + 6*y[6]^2,
y[2]*y[3] + 16*y[3]^2 + 14*y[1]*y[4] + 3*y[2]*y[4] + y[3]*y[4] + y[4]^2 +
    12*y[1]*y[5] + 9*y[3]*y[5] + 6*y[4]*y[5] + 2*y[5]^2 + 13*y[3]*y[6] +
    9*y[4]*y[6] + 13*y[5]*y[6] + 12*y[6]^2
```

Finally we get the composed map from X to X_2 and check that it is (birationally) invertible.

```
> mp1r := Restriction(mp1,X,X1);
> mp2r := Restriction(mp2,X1,X2);
> mpc := Expand(mp1r*mp2r);
> boo := IsInvertible(mpc);
> boo;
true
```

Example H113E7

In this example, we show how the sheaf machinery can be effectively used as an alternative method to normalise the projective coordinate ring of a normal, but not projectively normal, projective variety. Here the coordinate ring is locally normal at all primes except at the maximal homogeneous ideal.

Our chosen variety is C, an elliptic curve that has been embedded as a degree 8 subvariety of $\mathbf{P}^3$ over $\mathbf{Q}$. The curve C can be thought of as having been embedded in $\mathbf{P}^7$ by a complete linear system of degree 8 and then (isomorphically) projected down into $\mathbf{P}^3$. Such genus one curves embedded as degree 8 curves in $\mathbf{P}^3$ actually arise fairly naturally as models of homogeneous spaces arising in eight-descents.

We wish to recover the full embedding as a projective normal curve in $\mathbf{P}^7$. The coordinate ring of this is isomorphic to the normalisation of the coordinate ring of C in $\mathbf{P}^3$. From a sheaf-theoretic point of view, this is straightforward. The full embedding is the image of the divisor map corresponding to a hyperplane section of C or, equivalently, to the Serre twisting sheaf $\mathcal{O}_X(1)$. The maximal module of $\mathcal{O}_X(1)$ is isomorphic to the normalisation as an R-module, where R is the coordinate ring of C in $\mathbf{P}^3$, and it can be recovered as an algebra by taking the image of its associated divisor map. The global sections of $\mathcal{O}_X(1)$ correspond to the full Riemann-Roch space of the divisor on the abstract curve given by a certain hyperplane divisor on C.

This example also illustrates another interesting point. In situations similar to these, the dimension of the full space of global sections of the Serre twisting sheaf can be computed from cohomology of the coordinate ring R. However, it is faster in this case to explicitly compute the full maximal module of $\mathcal{O}_X(1)$, the zero-th graded part of this corresponding to the space of global sections and having the dimension of the zeroth cohomology group. In fact, though we only need to compute the dimension of this part, it is actually much quicker to compute the maximal module and compute its cohomology than to compute the cohomology of the original defining module, which is R twisted once. This probably reflects to some extent the fact that polynomial ring Groebner basis computations are much more highly tuned currently in MAGMA than the alternating algebra ones used in the cohomology computations. But the maximal module of a sheaf is generally a nicer object than a submodule with bits missing in the lower-graded pieces and has a smaller Castelnuevo-Mumford regularity etc. So, as we see in this example, it is often worth making sure that the maximal module of a sheaf is available before making cohomology calls.

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> C := Curve(P,[ x^2*y^2 - 23/59*x*y^3 + 9/59*y^4 + 27/59*x^3*z - 23/59*x^2*y*
> z - 6/59*x*y^2*z + 6/59*y^3*z - 10/59*x^2*z^2 + 5/59*x*y*z^2 - 3/59*y^2*z^2 +
> 1/59*x*z^3 - 74/59*x^3*t + 115/59*x^2*y*t - 83/59*x*y^2*t + 3/59*y^3*t -
> 105/59*x^2*z*t + 1/59*x*y*z*t - 2/59*y^2*z*t + 36/59*x*z^2*t + 4/59*y*z^2*t -
> 3/59*z^3*t + 297/59*x^2*t^2 - 135/59*x*y*t^2 + 52/59*y^2*t^2 + 68/59*x*z*t^2 -
> 11/59*y*z*t^2 - 18/59*z^2*t^2 - 315/59*x*t^3 + 42/59*y*t^3 + 96/59*t^4,
> x^3*y - 833/354*x*y^3 - 11/236*y^4 - 1633/708*x^3*z - 4675/708*x^2*y*z -
> 2633/708*x*y^2*z - 27/236*y^3*z + 805/354*x^2*z^2 + 223/59*x*y*z^2 -
> 4/59*y^2*z^2 - 38/59*x*z^3 + 3359/708*x^3*t + 3811/354*x^2*y*t +
> 1445/708*x*y^2*t + 303/118*y^3*t - 715/177*x^2*z*t - 527/177*x*y*z*t +
> 211/118*y^2*z*t + 347/354*x*z^2*t - 195/236*y*z^2*t - 4/59*z^3*t -
> 127/236*x^2*t^2 - 8237/708*x*y*t^2 + 65/708*y^2*t^2 + 1973/708*x*z*t^2 +
> 123/59*y*z*t^2 - 24/59*z^2*t^2 - 1753/354*x*t^3 + 873/236*y*t^3 + 128/59*t^4,
> x^4 + 269/354*x*y^3 + 35/236*y^4 + 1849/708*x^3*z + 4255/708*x^2*y*z -
> 247/708*x*y^2*z + 43/236*y^3*z - 727/354*x^2*z^2 - 82/59*x*y*z^2 +
> 2/59*y^2*z^2 + 19/59*x*z^3 - 5603/708*x^3*t - 3469/354*x^2*y*t -
> 1637/708*x*y^2*t - 63/118*y^3*t + 328/177*x^2*z*t - 769/177*x*y*z*t -
> 17/118*y^2*z*t + 151/354*x*z^2*t + 127/236*y*z^2*t + 2/59*z^3*t +
> 1391/236*x^2*t^2 + 7865/708*x*y*t^2 + 823/708*y^2*t^2 - 1901/708*x*z*t^2 +
```

```
> 86/59*y*z*t^2 + 12/59*z^2*t^2 + 493/354*x*t^3 - 761/236*y*t^3 - 64/59*t^4]);
```

Next the hyperplane sheaf of C is constructed and the dimension of the space of global sections is confirmed to be 8 using `DimensionOfGlobalSections` (which also saturates the sheaf).

```
> OC1 := StructureSheaf(C,1);
> DimensionOfGlobalSections(OC1);
8
```

Finally, the projective normal embedding into P^7 is created and we check that the image X is defined by 20 quadrics.

```
> norm_mp, X := DivisorMap(OC1);
> ArithmeticGenus(X);
1
> B := MinimalBasis(Ideal(X));
> #B;
20
> [TotalDegree(f) : f in B];
[ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
```

113.10 Bibliography

[**DES93**] Decker, Ein, and Schreyer. Construction of surfaces in P4. *J. Algebraic Geometry*, 2:185–237, 1993.

[**Eis95**] David Eisenbud. *Commutative Algebra with a View Toward Algebraic Geometry*, volume 150 of *Graduate Texts in Mathematics*. Springer, New York–Berlin–Heidelberg, 1995.

[**Har77**] Robin Hartshorne. *Algebraic Geometry, GTM 52*. Springer, ASpringer, 1977.

[**HM73**] G. Horrocks and D. Mumford. A rank 2 vector bundle on $\mathbf{P}_4$ with 15,000 symmetries. *Topology*, 12:63–81, 1973.

[**Ser55**] J-P. Serre. Faisceaux Algebriques Coherents. *Ann. Maths.*, 61:197–278, 1955.

114 ALGEBRAIC CURVES

Chapter 114
ALGEBRAIC CURVES

114.1 First Examples

This chapter describes functions for constructing and studying algebraic curves. The first section below contains elementary examples to help with getting started. The biggest obstacle is being able to create geometric objects. After that one should be able to consult the later sections for off-the-shelf functions to apply to the curves.

Within MAGMA, curves are realised as a specialised type of scheme, themselves covered in Chapter 112. As schemes they may be defined over any ring, although most functions will require this to be at least a domain and often a field.

In previous versions of MAGMA, curves were restricted to lie in some plane for the dedicated functions below to apply to them. However, we have now generalised the curve definition to apply to any one-dimensional scheme. The general type is `Crv` and the plane curve now has sub-type `CrvPln`. As before, for the vast majority of the specialist functionality to apply to a curve, it has to be integral (reduced and irreducible) and defined over a field.

114.1.1 Ambients

One usually starts by defining an affine or projective space over some base ring in which our curves will live, although it is not absolutely necessary. Normal affine and projective spaces are the commonest, although product or weighted projective spaces may also be used. This space is often referred to as the *ambient space* and these are more fully described in the general chapter on schemes referred to above. A two-dimensional ambient is referred to as a plane. Plane curves - the ones of subtype `CrvPln` - are those whose ambient space is a plane (and these are defined by a single polynomial equation). If you intend later to create several curves and would like them to be taken to lie in the same space, then deliberately creating their common ambient space in advance is certainly the surest way. It is important to be aware that any two ambients that have been created independently will always be distinct. For example, if one wants to intersect two plane curves, the curves are not allowed to lie in distinct planes, even if one might consider the planes to be mathematically equivalent.

The basic creation function takes two arguments. The first is the intended base ring, the second the intended dimension. In the code fragment below we make an affine plane (of dimension 2). The x, y delineated by diamond brackets determine the names of the coordinates. This syntax is just the same as that for analogous structures such as polynomial rings.

```
> k := Rationals();
> A<x,y> := AffineSpace(k,2);
```

```
> A;
Affine space of dimension 2 with coordinates x,y
```

One can retrieve the characteristic data related to a particular plane as in the following examples.

```
> BaseRing(A);
Rational Field
> Dimension(A);
2
> CoordinateRing(A);
Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
```

The variables x and y are named on A, but are really elements of the coordinate ring of A. Thus, `A.1`, the first variable of A which in this case is x, is synonymous with `CoordinateRing(A).1`. Higher dimensional affine ambients are precisely analogous, the difference being that there are more variables.

114.1.2 Curves

Algebraic curves are schemes of dimension one. That is, they are described by the vanishing of a set of polynomials or an ideal in the coordinate ring of their ambient space A and this set of zeros has algebraic dimension 1.

Plane curves are described by the vanishing of a single polynomial equation $f = 0$ in some plane A. In other ambients of dimension d, at least $d-1$ polynomials are needed and usually more (when $d-1$ suffice, we refer to the curve as a global complete intersection).

The polynomials or ideal must belong to the coordinate ring of A and be homogeneous when A is projective. In practical terms, this means that one should usually write the polynomial using the variables of A. When using most of the creation functions below, one must be explicit about the ambient in which the curve lies.

Given some affine plane A with coordinates u, v, define a curve C as the zero locus of a polynomial f in u, v.

```
> A<u,v> := AffineSpace(FiniteField(32003),2);
> f := u^2 - v^5;
> C := Curve(A,f);
> C;
Curve over GF(32003) defined by u^2 + 32002*v^5
```

As we see next, the curve C remains in the ambient space A in which it was constructed. It will always lie in that particular space.

```
> AmbientSpace(C);
Affine space of dimension 2 with coordinates u,v
> AmbientSpace(C) eq A;
true
```

```
> BaseField(C);
Finite field of size 32003
```

MAGMA can solve systems of polynomial equations over a given field (or even over a domain using resultants). In particular, it can find the singular points of C defined over the current base field.

```
> SingularPoints(C);
{ (0,0) }
> HasSingularPointsOverExtension(C);
false
```

The function `HasSingularPointsOverExtension()` above returns `false` if and only if there are no additional singular points defined over some extension of the base field of C. (It calculates the radical of the Jacobian algebra to check whether the singularities over the current base field account for the whole algebra.) So the origin really is the only singularity of C defined over this field. There are functions which help one find the singularities of C over an extension if they exist.

114.1.3 Projective Closure

MAGMA maintains a close connection between a curve and its projective closure. Here we illustrate some of the nice results of this.

In this example we define an affine curve in coordinates x, y and take its projective closure. For clarity, we name the homogeneous coordinates on projective space X, Y, Z. These names are really maintained by the projective space containing D even though they appear to have been created with D. Any other curve in this projective space will be expressed in terms of these variables. Names are not automatically given to the projective space. In this example the choice is made at the first opportunity, but it can be made or changed at any time.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,(y^2 - x^3)^2 - y*x^6);
> D<X,Y,Z> := ProjectiveClosure(C);
> D;
Curve over Rational Field defined by
X^6*Y - X^6*Z + 2*X^3*Y^2*Z^2 - Y^4*Z^3
```

Conversely one can retrieve the affine patches of a projective curve. The standard patches are usually the ones of interest, although others can be recovered. The first line below checks that the first patch really is C. Again, variable names are not automatically determined for curves lying in spaces that have not already been created. As seen below, the only result of not assigning names to variables is that the printing is a little unreadable: the first coordinate function is referred to as `$.1` and so on.

```
> AffinePatch(D,1) eq C;
true
> C2 := AffinePatch(D,2);
```

```
> C2;
Curve defined by -$.1^6*$.2 + $.1^6 + 2*$.1^3*$.2^2 - $.2^3
```

We name the coordinate functions and check that C_2 really is a patch of D.

```
> A2<u,v> := AmbientSpace(C2);
> C2;
Curve defined by -u^6*v + u^6 + 2*u^3*v^2 - v^3
> ProjectiveClosure(C2);
Curve defined by -X^6*Y + X^6*Z - 2*X^3*Y^2*Z^2 + Y^4*Z^3
```

114.1.4 Points

Points have already arisen as the singular points of a curve. In this chapter we always think of points as being a sequence of coordinates that satisfies the equations of a curve rather than as a zero-dimensional scheme. While the coordinates of a point often lie in the base ring of the curve, they may lie in any extension of that base ring. Technically, the parent of points is not considered to be the curve at all, but instead a *point set*. Point sets are characterised by two pieces of information. The first piece of information is some scheme, in this case a curve C. The second is some algebra over the base ring. (Note that while we say *algebra* here to emphasise the mathematical point, in MAGMA one uses any ring which admits a map from the base ring rather than an explicit MAGMA algebra type as the second piece of information.) In other words, given an extension L of the base ring k of a curve C, there is a set, $C(L)$, of points of C with coordinates in L. You can consider the point set $C(L)$ literally to be the set of points of C with coordinates in L, although this set does not actually compute or list all its elements since there are often infinitely many.

Thus points can be thought of as being the closest thing to the notion of point that one uses colloquially: "Is the curve C singular at the point p?", for example, can be translated as `IsSingular(C,p)`.

Creating points is easy. For sequences of coordinates of the base ring of a curve, the expression `C ! [1,2]`, for example, creates the point $(1, 2)$ on the scheme C (assuming that coordinate sequences of length 2 are appropriate for C and that $(1, 2)$ satisfies the equations of C). If the coordinates are in some extension, or if you really want a particular point set as parent, you must be explicit about the point set and write `C(L) ! [1,2]`.

Points may belong to the point sets either of a curve or of the ambient plane of the curve. When a point and a curve are both arguments to a function then this difference isn't visible — in the background the function will check that the point really does lie on the curve if that is what is mathematically required — although being careful about where points lie is good practice. Often one is allowed to omit the curve argument in such functions, in which case one must be clear that the point really does lie exactly where one wants it to lie: `IsNonsingular(p)` is an example of function that is rather susceptible to returning confusing results if one isn't careful about where the point lies.

The `!` operator is MAGMA's usual coercion operator. It can be used in a variety of situations where one might hope to make natural reinterpretations of objects. Here it is

used to reinterpret a sequence of numbers as the coordinates of a point in a plane or a curve.

```
> A<x,y> := AffineSpace(FiniteField(23),2);
> p := A ! [1,2];
> p;
(1, 2)
> q := Origin(A);
> q;
(0, 0)
```

So now it is possible to analyse particular points of a curve.

```
> C := Curve(A,x^2 + 2*x*y^2 - y^5);
> C;
Curve over GF(23) defined by
x^2 + 2*x*y^2 + 22*y^5
> p in C;
true (1, 2)
> IsSingular(C,p);
false
> TangentLine(C,p);
Curve over GF(23) defined by
10*x + 20*y + 19
```

Notice that the statement `p in C` is evaluated in a generous way: p is really in a point set of the plane but it happens to satisfy the equation of the curve and that is what is checked. There are two ways to force the parent of the point to be a point set of C: either use the coercion operator or test it with the standard `IsCoercible()` function. The tangent line to C at p is interpreted as a linear curve embedded in the same plane as C and intersecting it at p. These tangent lines are only defined for plane curves C and are themselves plane curves. In particular, functions which can take a curve as argument can take this tangent line as argument. The tangent line is given the name T in the discussion below and used as an argument in a function which takes two curves and a point. Recall that the tangent line to a curve C at a nonsingular point p is the unique line having intersection number at least 2 with C at p.

```
> T := $1;
> IntersectionNumber(C,T,p);
2
```

114.1.5 Choosing Coordinates

One way to express a curve C after a change of coordinates is to make a map — an automorphism of the ambient space — which realises the change of coordinates and then compute the image of C under this map.

Here is a hands-on analysis of a singularity on a curve which involves changes of coordinates. The example is again a plane curve and involves several functions still only available for such: `TangentCone` and `Blowup`. The singularity is first moved to the origin where coordinates are chosen so that the tangent directions lie along the coordinate axes.

```
> A<x,y> := AffineSpace(GF(11,2),2);
> C := Curve(A,-x^6 + x*y^2 - 2*x*y + x + (y - 1)^3);
> SingularPoints(C);
{ (0, 1) }
> p := Representative(SingularPoints(C));
> f := Translation(A,p);
> C1 := f(C);
> q := Origin(A);
> TangentCone(C1,q);
Curve over GF(11^2) defined by x*y^2 + y^3
> g := Automorphism(A,y);
> C2 := g(C1);
> TangentCone(C2,q);
Curve over GF(11^2) defined by x*y^2
```

The singularity is now in a suitable form for study. The following could be the first steps in an analysis. It can be skipped over by anyone not familiar with the language; otherwise see Chapter 46.

```
> D,E := Blowup(C2);
> IsSingular(D,q),IsSingular(E,q);
true false
> TangentCone(D,q);
Curve over GF(11^2) defined by y^2
> Faces(NewtonPolygon(D));
[ <2, 3, 6> ]
```

So the singularity is almost resolved. One more blowup of the cusp at the origin of D will make a resolution.

114.1.6 Function Fields and Divisors

If C is an integral curve then the field of fractions of its coordinate ring (or the homogeneous part of degree 0 in the projective case), is known as the *function field* of C. The function field allows us to conveniently perform many different computations with the curve. The function fields of an affine curve and its projective closure are isomorphic and are identified in MAGMA (with the projective version).

In fact, there are two MAGMA function fields associated to the curve, both abstractly isomorphic to its field of rational functions. What we refer to as the function field here has type `FunFldFracSch` and is associated with more general schemes. Additional information may be found in Chapter 112. This provides the basic user interface for curve functions when they are explicitly needed. For the most part, functions apply to the curve directly

with the function field being used in the background. The second function field is an algebraic function field (see Chapter 42). This provides much of the deeper functionality associated to the curve but lies even further in the background and most users should never need to access it directly.

Divisors — loosely speaking, formal sums of points of a curve — are an important part of the technology having many substantial applications. Any nonzero rational function determines a divisor: take the formal sum of zeros of the function on the curve (counted with multiplicity) minus the poles of the function. Divisors of this form are called principal divisors. Conversely, divisors arising in this way determine the function which defined them up to a scalar multiple.

There are two important groups in which divisors are often considered: the divisor group in which addition is simply coefficient-wise, and the divisor class group which is the divisor group modulo the subgroup of principal divisors. In the case of elliptic curves, the class group provides a formal setup in which to interpret the group law.

If you want to see it explicitly, the function field of a curve may be accessed.

```
> k := FiniteField(17);
> P<x,y,z> := ProjectiveSpace(k,2);
> E := Curve(P, y^2*z - x^3 - 4*x*z^2);
> E1<u,v> := AffinePatch(E,1);
> F<a,b> := FunctionField(E);
> F;
Function Field of Curve over GF(17) defined by
16*x^3 + 13*x*z^2 + y^2*z
> Genus(E);
1
```

But often you don't need the function field in your hands since the function you want can be called directly with the curve as argument.

Divisors are constructed by referring to the curve on which they should lie together with some characteristic data for them.

```
> Div := DivisorGroup(E);
> Div;
Group of divisors of Curve over GF(17) defined by
16*x^3 + 13*x*z^2 + y^2*z
> p := E ! [0,0,1];
> L := TangentLine(E,p);
> D := Divisor(E,L);
> D;
Divisor of Curve over GF(17) defined by
16*x^3 + 13*x*z^2 + y^2*z
> Decomposition(D);
[
    <Place at (0 : 0 : 1), 2>
```

```
    <Place at (0 : 1 : 0), 1>,
]
```

A little explanation is required. Firstly, the divisor D constructed here is really the divisor of the rational function with zero along L and pole along the line at infinity. Secondly, the basic printing of D is not so helpful: the point is to ensure that potentially lengthy calculations are avoided so it is not immediately printed in 'factorised' form. Next, once factorised, the divisor refers to places of the curve. Since the curve could be singular and divisor computations are done on the nonsingular model, the language of places is used. Note that when printing a place, a point corresponding to the place is shown. Of course, this point does not uniquely characterise the place. If p is a singular point of a curve it is possible to have unequal places that both display p as their support.

In this case, the curve is nonsingular so everything is above board: D is literally the divisor $2p_1 + p_2$ where p_1 is the prime divisor at the point $(0 : 0 : 1)$ and p_2 is the prime divisor at the point $(0 : 1 : 0)$. Now, after defining these prime divisors, we define a new divisor of degree 0.

```
> p1 := Divisor(p);
> p2 := Divisor(E![0,1,0]);
> D2 := D - 3*p1;
> Decomposition(D2);
[
    <Place at (0 : 0 : 1), -1>
    <Place at (0 : 1 : 0), 1>,
]
> Degree(D2);
0
```

The natural question to ask of a divisor of degree 0 is whether or not it is principal.

```
> IsPrincipal(D2);
false
> IsPrincipal(2*D2);
true 1/a
```

So D_2 is not principal but two times D_2 is principal. Moreover, the rational function $1/a$ defines $2 \times D_2$. (This corresponds to the rational function z/x on P.)

Now we look at the class group of E. This function requires that E be defined over a finite field. All the operations above apply to a curve defined over any field.

```
> Cl, _, phi := ClassGroup(E);
> Cl;
Abelian Group isomorphic to Z/4 + Z/4 + Z
Defined on 3 generators
Relations:
    4*Cl.1 = 0
    4*Cl.2 = 0
```

```
> phi;
Mapping from: DivCrv: D to GrpAb: Cl given by a rule
> phi(D2);
2*Cl.2
```

114.2 Ambient Spaces

In this section we show how to create various spaces that can be used as ambient spaces for curves. They can be specified in a number of different ways. Typically, different constructions of such spaces will be taken to be different objects, even if they are defined over the same ring. Names for the coordinates can be defined by using the diamond bracket notation in the same way as for polynomial rings.

The discussion here is rather brief, giving just enough functions to create some basic ambients, their functions and points in them. Consult Chapter 112 for more constructors and functions.

`AffineSpace(k,n)`

`AffinePlane(k)`

Create affine n-dimensional space (resp. the 2-dimensional affine plane) over the ring k.

`ProjectiveSpace(k,n)`

`ProjectivePlane(k)`

Create n-dimensional projective space (resp. the 2-dimensional projective plane) over the ring k.

`DirectProduct(A,B)`

If A and B are both one-dimensional projective spaces (defined using the intrinsic `ProjectiveSpace(k,1)` for example) this forms the product $A \times B$ and also returns a sequence containing the two projection maps.

`RuledSurface(k,n)`

`RuledSurface(k,a,b)`

The rational ruled surface over the ring k which has a curve of self-intersection $-n$ or $\pm(a-b)$. The integer arguments must all be nonnegative. It has four variables, the ratio of the first two defining the structure map to $\mathbf{P}^1$, the second two being homogeneous coordinates on the $\mathbf{P}^1$ fibres of this map.

`CoordinateRing(A)`

The coordinate ring of the ambient space A. This is a multivariate polynomial ring over the base ring of A. The number of variables possessed by the ring will depend upon the space. If A is an affine n-space, the function will return a ring with n variables; an ordinary or weighted projective n-space will result in $n+1$ variables; a ruled surface will result in four variables. The gradings that are implicit on various spaces will not be reflected by the polynomial ring. Indeed, at present, there is no way to impose two different gradings on a polynomial ring in MAGMA.

`FunctionField(A)`

Return the function field of the ambient space A. This is a field isomorphic to the field of fractions of the coordinate ring of A. Its generators can be assigned names in the usual way and, typically, one writes elements of this function field in terms of those generators. Polynomials of the coordinate ring of A can be coerced into this function field.

`A ! [a,...]`

`A(L) ! [a,...]`

For elements $a, \ldots$ in the base ring of the ambient space (or any other scheme) A the expression `A ! [a,...]` creates the set-theoretic point, eg, (a, b) in the affine plane case, $(a : b : c)$ in the projective plane case, or $(a : b : c : d)$ in the product or ruled surface case. If L is an extension ring of the base ring of A then the expression `A(L) ! [a,...]` creates the point with coordinates $(a, \ldots)$ where these coordinates are elements of L (or the base ring of A).

`Origin(A)`

The point $(0, 0, \ldots, 0)$ of the affine space A.

`Coordinates(p)`

`p[i]`

The complete sequence of base ring elements corresponding to the coordinates of the point p or the ith coordinate or p alone.

Example H114E1

In this example we make some points of an affine plane A. The first exhibits the constructor which creates points in the point set of the base ring of A. The second creates a point in some extension of the base ring of A.

```
> k := FiniteField(2);
> A := AffineSpace(k,3);
> p := A ! [1,2,3];
> p;
(1, 0, 1)
> L<w> := ext< k | 2 >;
```

```
> q := A(L) ! [1,2,w];
> q[3];
w
```

114.3 Algebraic Curves

A general curve C (type `Crv`) is defined by the vanishing of a finite number of polynomials $f_1, \ldots, f_n$ or a polynomial ideal I in a general ambient space. As a scheme, this must have dimension 1.

A plane curve C (type `CrvPln`) is defined by the vanishing of a single polynomial f in one of the available ambient planes:

$$C : (f_1 = f_2 = \ldots = f_n = 0) \subset A.$$

The polynomials or ideal must lie in the coordinate ring of A. The notation C for a curve and f or $f_1, \ldots, f_n$ for its defining equation(s) will be maintained. The coefficient ring of the parent of polynomials will be denoted k. Irrespective of type, the ambient space will be denoted A.

114.3.1 Creation

In this section the most basic methods of creating a curve are presented. For specialised types — conics, elliptic curves, hyperelliptic curves — there are additional functions documented in the corresponding chapters. Curves may also be created implicitly, such as when they arise as the images of maps.

`Curve(A,f)`

`Nonsingular`	BoolElt	*Default :* `false`
`Reduced`	BoolElt	*Default :* `false`
`Irreducible`	BoolElt	*Default :* `false`
`GeometricallyIrreducible`	BoolElt	*Default :* `false`
`Saturated`	BoolElt	*Default :* `false`

Create the plane curve $f = 0$ in the ambient plane A where f is a polynomial in the coordinate ring of A.

`Curve(A,I)`

`Nonsingular`	BoolElt	*Default :* `false`
`Reduced`	BoolElt	*Default :* `false`
`Irreducible`	BoolElt	*Default :* `false`
`GeometricallyIrreducible`	BoolElt	*Default :* `false`
`Saturated`	BoolElt	*Default :* `false`

Create the curve in the ambient space A determined by the ideal I of the coordinate ring of A. An error results if the result

`Curve(X,S)`

`Nonsingular`	BoolElt	*Default :* `false`
`Reduced`	BoolElt	*Default :* `false`
`Irreducible`	BoolElt	*Default :* `false`
`GeometricallyIrreducible`	BoolElt	*Default :* `false`
`Saturated`	BoolElt	*Default :* `false`

Create the curve defined by the sequence S in the ambient space of X, where S is a sequence of polynomials in the coordinate ring. Here X can be any scheme, not necessarily an ambient space itelf. An error results if the result is not actually a 1-dimensional scheme.

Note: An important special case is when A is an affine or a projective space of dimension 1 and S is empty. This gives the affine or projective line as a curve - as a scheme it is just the ambient space A. Alternatively, the constructor `Curve(A)` described below may be used. Note that the initial construction of A never returns it as a `Crv` type, even though it is 1-dimensional. This is for internal technical reasons – a `Crv` cannot be considered as an ambient space by Magma. is not a 1-dimensional scheme.

`IsCurve(X)`

Returns `true` if and only if X is a one-dimensional scheme.

`Curve(X)`

The smallest scheme in the inclusion chain above the scheme X which is a curve. If X is a curve (ie 1-dimensional) then X will be returned as a `Crv` type. If X has been created as a subscheme of a curve then this curve will be returned.

`Line(C,p,q)`

`Line(P,S)`

The line through the distinct points p, q on the curve C, or the points of S as a subscheme of the projective space P if they are collinear. If the points are points of a curve rather than the ambient space, the line will be interpreted as the tangent line in the case that the points are equal.

`Conic(P,S)`

Given a projective plane P and a set S of points in P, this function returns the conic P through the points of the set S if such a conic exists and is unique. The traditional setup corresponds to the case where S is a set of 5 points in general position, that is, no three of them are collinear. If the resulting conic curve is nonsingular, then it will be returned as a special type. See Chapter 119 for details of the special functions that apply in that case.

`Union(C,D)`

Create the union of the curves C and D. The result will usually be non-irreducible, so although it will be interpreted as a curve, most of the advanced functions below will not apply to it.

114.3.2 Base Change

Let A be some ambient space in MAGMA. For example, think of A as being the affine plane. Let k be its base ring and R_A its coordinate ring. If $m : k \to L$ is a map of rings (a coercion map, for instance) then there is a new ambient space denoted A_L and called the *base change of A to L* which has coordinate ring R_A but with coefficients L instead of k. (Mathematically, one simply tensors R_A with L over k. In MAGMA the equivalent function at the level of polynomial rings is `ChangeRing`.) There is a base change function described below which takes A and L (or the map $k \to L$) as arguments and creates this new space A_L. Note that there is a map from the coordinate ring of A to that of A_L determined by the map m.

This operation is called *base extension* since one often thinks of the map m as being an extension of fields. Of course, the map m could be many other things. One key example where the name *extension* is a little unusual would be when m is the map from the integers to some finite field.

Now let X be a scheme in MAGMA. Thus X is defined by some polynomials $f_1, \ldots, f_r$ on some ambient space A. Given a ring map $k \to L$ there is a base change operation for X which returns the *base change of X to L*, denoted X_L. This is done by first making the base change of A to L and then using the map from the coordinate ring of A to that of A_L to translate the polynomials f_i into polynomials defined on A_L. These polynomials can then be used to define a scheme in A_L. It is this resulting scheme which is the base change of X to L.

If one has a number of curves in the same ambient space and wants to base change them all at the same time, a little care is required. The function which takes a curve and a

map of rings as argument will create a new ambient space each time so should be avoided. A better approach is to apply base change to the ambient space and then invoke the base change function which takes the curve and the desired new ambient space as argument. (This latter base change function appears to be different to the other. In fact it is not. We described base change above as a function of maps of rings. Of course, there is a natural extension to maps of schemes. With that extension, this final base change intrinsic really is base change with respect to map of ambient spaces.)

`BaseChange(C, K)`

The base change of the curve C to the new base ring K. This is only possible if elements of the current base ring of C can be coerced automatically into K. The resulting curve will lie in a newly created plane (see the example below).

`BaseChange(C, m)`

The base change of the curve C by the map of base rings m. The resulting curve will lie in a newly created plane.

`BaseChange(C, A)`

`BaseChange(C, A, m)`

The base change of the curve C to a curve in the new ambient space A. The space A must be of the same type as the ambient of C and its base ring must either admit coercion from the base ring of C or have the map m between the two explicitly given.

`BaseChange(C, n)`

The base change of C, where the base ring of C is a finite field to the finite field which is a degree n extension of the base field of C.

Example H114E2

We give an example of a singular curve, over the rationals, whose singular points are only defined over the field extension given by adjoining a square root of -1.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,y^2 - (x^2+1)^3);
> SingularPoints(C);
{}
> HasSingularPointsOverExtension(C);
true
```

Here we assume that the user knows which extension to move to. The first method of finding the points is to search in the particular point set as follows.

```
> Qi<i> := QuadraticField(-1);
> SingularPoints(C,Qi);
```

```
{ (i, 0), (-i, 0) }
```

The second method is to create a new curve by base change and to search the base ring point set for that curve. For a single calculation this method is rather clumsy, but if further computation were to take place at these points it might be preferable.

```
> B<u,v> := BaseChange(A,Qi);
> Ci := BaseChange(C,B);
> SingularPoints(Ci);
{ (i, 0), (-i, 0) }
```

114.3.3 Basic Attributes

The first few functions below recover data from the ambient space of the curve (and could equally well be applied to the ambient space). Any curves lying in the same ambient space will return identical results when evaluated in these functions. The remaining functions recover data about the equation defining the curve.

The coordinate ring of a curve is described here, but its function field is discussed much later in Section 114.8.

`AmbientSpace(C)`

The ambient space containing the curve C.

`BaseRing(C)`

`CoefficientRing(C)`

`BaseField(C)`

The base ring of the curve C. This is recovered as the base ring of the ambient plane. The third function will report an error if the base ring is not a field.

`DefiningPolynomial(C)`

The defining polynomial of the plane curve C.

`DefiningIdeal(C)`

The defining ideal of the curve C, as an ideal in the coordinate ring of its ambient space.

`CoordinateRing(C)`

The coordinate ring of the curve C. Even creating this requires the use of Gröbner basis techniques.

`Degree(C)`

The degree of the of the curve C which must be defined in an ordinary projective ambient space.

`JacobianIdeal(C)`

The ideal of partial derivatives of the defining polynomials of the curve C.

`JacobianMatrix(C)`

The matrix of partial derivatives of the defining polynomials of the curve C.

`HessianMatrix(C)`

The symmetric matrix of second partial derivatives of the defining polynomial of the plane curve C.

Example H114E3

In this example we start by creating a plane curve C and check that its ideal really is principal. We have chosen an example which is in Weierstrass form.

```
> A<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(A,z*y^2 - x^3 - x*z^2 - z^3);
> IsNonsingular(C);
true
> DefiningIdeal(C);
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Basis:
[
    -x^3 - x*z^2 + y^2*z - z^3
]
> IsPrincipal($1);
true x^3 + x*z^2 - y^2*z + z^3
```

Next we compute the determinant of the Hessian matrix of C. That is a polynomial which we use to create another curve D. The intersection of C and D are the points of inflection, or *flexes*, of C. Over an algebraic closure there will be nine of these, but we only see one — the family "flex at infinity" — over the rationals.

```
> M := HessianMatrix(C);
> Determinant(M);
24*x^2*z + 24*x*y^2 + 72*x*z^2 - 8*z^3
> D := Curve(A,Determinant(M));
> IntersectionPoints(C,D);
{ (0 : 1 : 0) }
```

114.3.4 Basic Invariants

`IsReduced(C)`

Returns `true` if and only if the ideal defining the curve C is reduced.

`IsIrreducible(C)`

Returns `true` if and only if the curve C is irreducible (as a scheme).

`IsSingular(C)`

Returns `true` if and only if the curve C contains at least one singularity over an algebraic closure of its base field.

`IsNonsingular(C)`

Returns `true` if and only if the curve C has no singularities over an algebraic closure of its base field.

114.3.5 Random Curves

This section described several functions for the generation of random curves of given degree and/or genus over finite fields and the rationals. The implementations follow [ST02].

`RandomNodalCurve(d, g, P)`

`RandomBound` RngIntElt *Default :* 9

Generates a random plane curve in the projective plane P of degree d and genus g with only nodes as singularities. The genus g must satisfy $g \leq (d-1)(d-2)/2$ and then the number of nodes will be $(d-1)(d-2)/2 - g$. These nodes are chosen as a random set of points in P. At the same time g must be $\geq 1 + (d(d-6)/3)$ to guarantee a non-empty linear system for a general set of nodes. For $0 \leq g \leq 10$, a good choice to obtain a general curve of genus g is to take $d = g + 2 - [g/3]$ (see [ST02]).

The base field may be a finite field or $\mathbf{Q}$. Over $\mathbf{Q}$, the construction uses polynomials which will have integer coefficients randomly chosen in the range $[-r \ldots r]$, where r is the value of `RandomBound`. Over a finite field, `RandomBound` is ignored.

`IsNodalCurve(C)`

Given a plane curve C, this function returns `true` if either C is non-singular or C only has nodes as singularities.

`RandomOrdinaryPlaneCurve(d, S, P)`

`Adjoint`	BoolElt	*Default :* `true`
`Proof`	BoolElt	*Default :* `true`
`RandomBound`	RngIntElt	*Default :* 9

Generates a random plane curve in the projective plane P of degree d and with ordinary singularities specified by the sequence *sings* as follows: If $S = [s_2, s_3, s_4, \ldots]$ then the curve will have s_2 nodes, s_3 triple points, s_4 ordinary singularities of multiplicity 4, etc. For example $[2, 0, 1]$ specifies 2 nodes and one ordinary quadruple point.

For such a curve to exist we require $\binom{d-1}{2} \geq \sum_i s_i \binom{i+1}{2}$.

If `Proof` is `false` then the full check that the singularities are ordinary is skipped.

If `Adjoint` is `true` then the adjoint ideal, an ideal of the coordinate ring of P, is also computed and returned as a second value. The r-th graded parts of this homogeneous ideal realises the linear system $K + (r - d + 3)H$ on the normalisation of the curve, where K is the canonical divisor and H is the hyperplane section divisor corresponding to the (singular) embedding into P. This can be used to compute various adjoint maps (for example, $r = d - 3$ gives the canonical map) and its computation by this function is more efficient than using the general method of blowing-up in `Adjoints` (this function now also tests for ordinariness and uses the adjoint ideal by default).

The base field can be finite or $\mathbf{Q}$ and `RandomBound` is as before.

`RandomCurveByGenus(g, K)`

`RandomBound`	RngIntElt	*Default :* 9

Given a positive integer g and a field K this function generates a random projective curve over K of genus g, for $0 \leq g \leq 13$. When $g \leq 10$, a plane nodal curve is returned as given by the function `RandomNodalCurve` with degree $g + 2 - [g/3]$. For $11 \leq g \leq 13$, a curve in $\mathbf{P}^3$ is returned, computed by syzygy computations as described in [ST02].

The field K must be a finite field or $\mathbf{Q}$. The parameter `RandomBound` applies when K is chosen to be $\mathbf{Q}$. Note that, although $\mathbf{Q}$ is allowed in all cases, for the higher values of g, particularly for $g \geq 11$, the heights of the coefficients of the defining polynomials for the curve produced tend to be very large, even for small values of `RandomBound`.

Example H114E4

```
> SetSeed(1);
> C := RandomCurveByGenus(4, Rationals());
> C;
Curve over Rational Field defined by
x^5 + 34965/512*x*y*z^3 - 59355/512*x*z^4 + y^5 - 16705/48*y^3*z^2 -
    1831885/1536*y^2*z^3 - 1553135/2304*y*z^4 + 655145/4608*z^5
```

```
> Genus(C);
4
> C := RandomCurveByGenus(8, GF(23));
> C;
Curve over GF(23) defined by
17*x^8 + 5*x^7*y + 5*x^7*z + 13*x^6*y^2 + 11*x^6*y*z + 8*x^6*z^2 + 16*x^5*y^3 +
    17*x^5*y^2*z + 22*x^5*y*z^2 + 6*x^5*z^3 + 3*x^4*y^4 + 2*x^4*y^3*z +
    18*x^4*y^2*z^2 + 3*x^4*y*z^3 + 14*x^4*z^4 + 19*x^3*y^5 + 19*x^3*y^4*z +
    19*x^3*y^3*z^2 + 21*x^3*y^2*z^3 + 21*x^3*y*z^4 + 10*x^3*z^5 + 18*x^2*y^6 +
    4*x^2*y^5*z + 9*x^2*y^4*z^2 + 2*x^2*y^3*z^3 + 21*x^2*y^2*z^4 + 22*x^2*y*z^5
    + 6*x^2*z^6 + 14*x*y^7 + 4*x*y^6*z + 17*x*y^5*z^2 + 20*x*y^4*z^3 +
    14*x*y^2*z^5 + 15*x*y*z^6 + 3*x*z^7 + 18*y^8 + 2*y^7*z + 11*y^6*z^2 +
    18*y^4*z^4 + 18*y^3*z^5 + 5*y^2*z^6 + 22*y*z^7 + 2*z^8
> Genus(C);
8
> C := RandomCurveByGenus(12, GF(23));
> Ambient(C);
Projective Space of dimension 3
Variables : x, y, z, t
> Degree(C); Genus(C);
12
12
```

114.3.6 Ordinary Plane Curves

The term ordinary plane curve refers to a curve in the projective plane all of whose singularities are ordinary. This means that a singularity of multiplicity $m \geq 2$ has m "distinct tangent directions" – the equation of the curve expanded in local coordinates at the singularity begins with a binary form of degree m which splits into m distinct linear factors over the algebraic closure of the ground field.

A significant property of such curves is that all of their singularities are resolved by a single blow-up. Their adjoint linear systems/adjoint ideal can be computed in a more direct fashion than for more general plane curves. These linear systems give important projective maps such as the canonical map for curves of genus at least 2 and embeddings as rational normal curves for curves of genus 0.

This section contains some functions relating to ordinary curves and, in particular, to nodal curves, all of whose singularities are ordinary of multiplicity 2 (nodes). The list of functions is likely to be extended in future versions of Magma.

`HasOnlyOrdinarySingularities(C)`

`Adjoint` BoolElt *Default :* `true`

Given a plane curve C this function returns `true` in C has only ordinary singularities. If that is the case it also returns the maximum of the multiplicities of the singularities of C (1 means that C is non-singular). Further, if C is ordinary and if `Adjoint` is `true`, then the (saturated) adjoint ideal is also computed and returned as the third value.

`HasOnlyOrdinarySingularitiesMonteCarlo(C)`

Given a plane curve C defined over $\mathbf{Q}$ perform a Monte Carlo test for ordinariness. This will generally be faster than the intrinsic `OnlyOrdinarySingularities`. The function does not compute the adjoint ideal. Five primes are chosen for which the mod p reduction of C is still a curve and which has Jacobian ideal of the same degree as that of C. The five reductions are tested for ordinary singularities. If all pass, then `true` is returned. Otherwise `false` is returned. If `false` is returned, then C is definitely not ordinary. If it succeeds, then C is very likely to be ordinary but this is not 100% guaranteed.

`AdjointIdeal(C)`

Returns the (saturated) adjoint ideal of an ordinary plane curve C. If C is not ordinary then an error results.

`AdjointIdealForNodalCurve(C)`

`AdjointLinearSystemForNodalCurve(C, d)`

Given a plane curve C that is *assumed* to be nodal, these are slightly faster intrinsics for computing the adjoint ideal and adjoint linear system, respectively. The first function returns the adjoint ideal I and the second returns the degree d adjoint linear system, which is the linear subsystem of the complete plane linear system of degree d given by the degree d graded part of I.

`AdjointLinearSystemFromIdeal(I, d)`

Given an ideal I and a positive integer d, this function returns the degree d adjoint linear system for a plane curve whose (saturated) adjoint ideal is I.

`CanonicalLinearSystemFromIdeal(I, d)`

Given an ideal I and a positive integer d, this function returns the canonical linear system for a plane curve of degree d whose (saturated) adjoint ideal is I. This is the same as intrinsic `AdjointLinearSystemFromIdeal` with I and $d-3$ as arguments. It will be empty if the curve has genus 0.

`CanonicalLinearSystem(C)`

`AdjointLinearSystem(C)`

`Adjoints(C,d)`

Given a plane curve C the first two functions return the canonical linear system for any plane curve C and the third gives the general degree d adjoint linear system. If C is ordinary, then the functions compute the adjoint ideal and takes its graded piece as above. If not, they have to work out in detail the graph of the full resolution of singularities of C, which can take some time.

Example H114E5

In this example, we generate a random ordinary plane curve of degree 7 with 3 nodes and one ordinary singularity of order 4. We use its adjoint ideal to get the canonical map and compute its canonical image in $\mathbf{P}^5$. The computation of the canonical map this way is generally faster and gives a much simpler map description than the computation for general curves using the function field machinery.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C, I := RandomPlaneCurve(7,[3,0,1],P : RandomBound := 2);
> C;
Curve over Rational Field defined by
x^7 + x^5*y*z + x^4*y^3 + x^3*y^2*z^2 - 708*x^3*z^4 - 3401/18*x^2*y^2*z^3 +
    3274/9*x^2*y*z^4 + 29054/9*x^2*z^5 - 5861/15552*x*y^6 - 5279/432*x*y^5*z -
    71851/1296*x*y^4*z^2 + 173479/486*x*y^3*z^3 + 87337/108*x*y^2*z^4 -
    294685/81*x*y*z^5 - 662891/243*x*z^6 + 8117/15552*y^7 + 5543/972*y^6*z +
    41419/1296*y^5*z^2 - 17087/243*y^4*z^3 - 646577/972*y^3*z^4 +
    77026/81*y^2*z^5 + 423635/243*y*z^6 + 156920/243*z^7
> Genus(C);
6
> //check with OnlyOrdinarySingularities function
> boo,d,I1 := HasOnlyOrdinarySingularities(C);
> boo; d;
true
4
> I eq I1;
true
> // polynomials for canonical map come from the degree d-3=4
> // graded piece of the adjoint ideal.
> can_pols := AdjointLinearSystemFromIdeal(I, 4);
> Sections(can_pols);
[
    x^4 - 24*x^2*z^2 - 83/36*x*y^2*z + 83/9*x*y*z^2 + 493/9*x*z^3 - 167/432*y^4
        + 145/108*y^3*z + 35/6*y^2*z^2 - 731/27*y*z^3 - 587/27*z^4,
    x^3*y - 12*x^2*z^2 - 73/18*x*y^2*z + 38/9*x*y*z^2 + 286/9*x*z^3 - 55/216*y^4
        + 65/54*y^3*z + 7*y^2*z^2 - 494/27*y*z^3 - 350/27*z^4,
    x^3*z - 6*x^2*z^2 - 5/18*x*y^2*z + 10/9*x*y*z^2 + 98/9*x*z^3 - 7/108*y^4 +
        5/27*y^3*z + y^2*z^2 - 112/27*y*z^3 - 112/27*z^4,
```

```
    x^2*y^2 - 4*x^2*z^2 - 11/3*x*y^2*z - 4/3*x*y*z^2 + 52/3*x*z^3 - 5/36*y^4 -
        5/9*y^3*z + 10*y^2*z^2 - 116/9*y*z^3 - 68/9*z^4,
    x^2*y*z - 2*x^2*z^2 - 2/3*x*y^2*z - 4/3*x*y*z^2 + 16/3*x*z^3 - 1/18*y^4 +
        5/18*y^3*z + y^2*z^2 - 14/9*y*z^3 - 20/9*z^4,
    x*y^3 - 12*x*y*z^2 + 16*x*z^3 - 1/2*y^4 - 2*y^3*z + 12*y^2*z^2 - 8*y*z^3 -
        8*z^4
]
> X := CanonicalImage(C, Sections(can_pols));
> X;
Curve over Rational Field defined by
x[1]*x[4] - x[2]^2 + 5/36*x[2]*x[6] + 4*x[3]^2 + 11/3*x[3]*x[4] - 68/3*x[3]*x[5]
    - 68/9*x[3]*x[6] - 95/216*x[4]*x[6] + 140/3*x[5]^2 + 2*x[5]*x[6] +
    131/324*x[6]^2,
x[1]*x[5] - x[2]*x[3] + 1/18*x[2]*x[6] + 2*x[3]^2 + 2/3*x[3]*x[4] -
    14/3*x[3]*x[5] - 5/9*x[3]*x[6] - 1/27*x[4]*x[6] + 7/6*x[5]^2 +
    1/24*x[5]*x[6] + 2/81*x[6]^2,
-x[1]*x[6] + x[2]*x[4] - 1/2*x[2]*x[6] - 8*x[3]*x[5] + 5/3*x[3]*x[6] -
    1/9*x[4]*x[6] + 14*x[5]^2 + 4*x[5]*x[6] + 43/216*x[6]^2,
x[2]*x[5] - x[3]*x[4] + 2*x[3]*x[5] + 2/3*x[3]*x[6] + 1/18*x[4]*x[6] - 7*x[5]^2
    - 1/4*x[5]*x[6] - 1/27*x[6]^2,
-x[2]*x[6] + x[4]^2 - 1/2*x[4]*x[6] - 4*x[5]^2 + 16/3*x[5]*x[6] + 1/36*x[6]^2,
-x[3]*x[6] + x[4]*x[5] + 2*x[5]^2 + 1/6*x[5]*x[6] + 1/18*x[6]^2,
x[1]^3 - 10*x[1]^2*x[3] + x[1]*x[2]*x[3] + 52*x[1]*x[3]^2 + 95/216*x[2]^3 +
    15/2*x[2]^2*x[3] + 337/432*x[2]^2*x[4] - 491/18*x[2]*x[3]^2 -
    425/36*x[2]*x[3]*x[4] - 71/1296*x[2]*x[4]^2 + 676/27*x[3]^3 +
    3907/108*x[3]^2*x[4] - 506/27*x[3]^2*x[5] + 341/36*x[3]*x[4]^2 -
    529/36*x[3]*x[4]*x[5] + 2525/162*x[3]*x[5]^2 - 103/15552*x[4]^3 -
    18385/1944*x[4]^2*x[5] + 4751/11664*x[4]^2*x[6] - 44317/1296*x[4]*x[5]^2 +
    317441/46656*x[4]*x[5]*x[6] - 3677/139968*x[4]*x[6]^2 - 11905/162*x[5]^3 -
    52453/11664*x[5]^2*x[6] - 1113761/279936*x[5]*x[6]^2 +
    813889/1679616*x[6]^3,
x[1]^2*x[2] - 2*x[1]^2*x[3] - 8*x[1]*x[2]*x[3] + 16*x[1]*x[3]^2 - 31/36*x[2]^3 +
    29/3*x[2]^2*x[3] + 113/72*x[2]^2*x[4] + 29/3*x[2]*x[3]^2 -
    5/9*x[2]*x[3]*x[4] + 5/54*x[2]*x[4]^2 - 460/9*x[3]^3 - 577/18*x[3]^2*x[4] +
    1688/9*x[3]^2*x[5] - 1391/108*x[3]*x[4]^2 + 2257/18*x[3]*x[4]*x[5] -
    1717/9*x[3]*x[5]^2 - 437/648*x[4]^3 + 15593/648*x[4]^2*x[5] +
    13321/15552*x[4]^2*x[6] - 10073/108*x[4]*x[5]^2 - 93221/7776*x[4]*x[5]*x[6]
    + 93403/93312*x[4]*x[6]^2 + 1582/9*x[5]^3 - 11059/1296*x[5]^2*x[6] -
    193157/46656*x[5]*x[6]^2 + 363041/559872*x[6]^3,
x[1]*x[2]^2 - 4*x[1]*x[2]*x[3] + 4*x[1]*x[3]^2 - 7/6*x[2]^3 + x[2]^2*x[3] +
    1/2*x[2]^2*x[4] + 10*x[2]*x[3]^2 + 26/3*x[2]*x[3]*x[4] + 10/9*x[2]*x[4]^2 -
    44/3*x[3]^3 - 36*x[3]^2*x[4] + 200/3*x[3]^2*x[5] - 301/18*x[3]*x[4]^2 +
    331/3*x[3]*x[4]*x[5] - 538/3*x[3]*x[5]^2 - 83/54*x[4]^3 +
    1109/54*x[4]^2*x[5] + 13/8*x[4]^2*x[6] - 1117/12*x[4]*x[5]^2 -
    4051/648*x[4]*x[5]*x[6] - 25/1296*x[4]*x[6]^2 + 2236/9*x[5]^3 +
    14917/648*x[5]^2*x[6] - 1351/162*x[5]*x[6]^2 + 45109/46656*x[6]^3
```

114.4 Local Geometry

Here we discuss some basic functions providing analysis of a point p lying on a curve C. Firstly we describe how to create points on curves and their basic access functions. One should also refer to the comments in Section 112.7 of the general schemes chapter about the point sets of point arguments of these functions where there is a fuller discussion of point sets.

Most functions usually have two arguments, a curve and a point on that curve. In fact, the point need not actually be in a point set of the curve since coercion will be attempted if it is not. Moreover, the curve argument is not strictly necessary either, since if the point does lie in a point set of the curve, it can be recovered automatically. So these functions also work with the curve argument omitted. However, omitting the curve argument should be thought of merely as a convenient shorthand and should be used with care — it is very easy to use a point from some other space for which the function still makes sense but returns a misleading answer.

114.4.1 Creation of Points on Curves

Points of a curve C, and indeed points of any scheme in MAGMA, lie in point sets associated to C rather than C itself. Each point set is the parent of points whose coordinates lie in a particular extension ring of the base ring of the curve. Thus, if k is the base ring of the curve C, points whose coordinates lie in k are elements of the "base ring point set" denoted $C(k)$. If L is an extension ring of k (in the sense of admitting coercion from k or being the codomain of a ring homomorphism from k) then points with coordinates in L lie in the point set $C(L)$.

Here we give the basic point creation methods and access functions. For more information, consult the discussion of points and point sets in Section 112.7 of Chapter 112 on schemes.

`C ! [a,...]`

For a sequence of elements $a, \ldots$ of the base ring of C, this creates the point of C with coordinates $(a, \ldots)$. The parent of the resulting point is the base point set of the curve C rather than C itself.

`C(L) ! [a,...]`

For a sequence of elements $a, \ldots$ of the extension ring L of the base ring of C, this function creates the point of C with coordinates $(a, \ldots)$. The parent of the resulting point is the point set $C(L)$ of the curve C rather than C itself. The phrase 'extension ring' here means that either L admits automatic coercion from the base ring of C, or that L is the codomain of a ring homomorphism from that base ring.

`Curve(p)`

The smallest scheme in the inclusion chain above the scheme on which the point p lies which is a curve. If p lies on a curve then the curve will be returned.

`Curve(P)`

The smallest scheme in the inclusion chain above the scheme P is a point set of which is a curve. If P is a point set of a curve then this curve will be returned.

`Coordinates(p)`

The sequence of ring elements corresponding to the coordinates of the point p.

`p[i]`

`Coordinate(p,i)`

The ith coordinate of the point p.

`p eq q`

Returns `true` if and only if the two points p and q lie in schemes contained in a common ambient space, have coordinates that can be compared (either by lying in the same ring, or by an automatic coercion) and these coordinates are equal.

`FormalPoint(P)`

Given a non–singular point P in $C(K)$, where C is a curve and K is some extension of the field of definition of C, returns a point in `C(LaurentSeriesRing(K))`, such that specializing the variable to 0 yields P.

114.4.2 Operations at a Point

Most of the functions in this section report an error if p does not lie on C. Functions having arguments C, p allow the omission of C as long as the parent of p is a point set of C. A few of the functions apply only to plane curves.

`p in C`

`S in C`

Returns `true` if and only if the point p or the sequence of coordinates S lies on the curve C. That is, return `true` if and only if the coordinates of p satisfy the equation of C.

`IsNonsingular(C,p)`

Returns `true` if and only if p is a nonsingular point of the curve C.

`IsSingular(C,p)`

Returns `true` if and only if the point p is a singular point on the curve C.

`IsInflectionPoint(C,p)`

`IsFlex(C,p)`

Returns `true` if and only if the point p is a flex of the plane curve C. An error is reported if p is a singular point of C. The second return value is the order of the flex, that is, the local intersection number at p of C with its tangent line at p.

`TangentLine(p)`

`TangentLine(C,p)`

The tangent line to the curve C at the point p embedded as a curve in the same space; an error if p is a singular point of C.

`TangentCone(C,p)`

The tangent cone to the curve C at the point p embedded in the same ambient space.

`IsTangent(C,D,p)`

Returns `true` if and only if the plane curves C and D are nonsingular and tangent at the point p.

114.4.3 Singularity Analysis

These functions report an error if p is not a singular point of C. Again, the arguments can be abbreviated to just the point if care is taken about its parent.

`Multiplicity(C,p)`

The multiplicity of the curve C at the point p.

`IsDoublePoint(C,p)`

Returns `true` if and only if the point p is a double point of the curve C.

`IsOrdinarySingularity(C,p)`

Returns `true` if and only if the point p is a singular point of the curve C with reduced tangent cone.

`IsNode(C,p)`

Returns `true` if and only if the point p is an ordinary double point of the curve C.

`IsCusp(C,p)`

Returns `true` if and only if the point p is a nonordinary double point of the curve C.

`IsAnalyticallyIrreducible(C,p)`

Returns `true` if and only if the plane curve C has exactly one place at the point p, or equivalently if the resolution of singularities is injective above p.

Example H114E6

Each of the two curves in this example has a double point at the origin. One of these is a node and one is a cusp.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,x^2-y^3);
> p := Origin(A);
> IsCusp(C,p);
true
> IsDoublePoint(C,p);
true
> IsReduced(TangentCone(C,p));
false
> D := Curve(A,x^2 - y^3 - y^2);
> IsAnalyticallyIrreducible(D,p);
false
> IsNode(D,p);
true
```

114.4.4 Resolution of Singularities

Again, all functions in this section only apply to plane curves.

`Blowup(C)`

Given the affine plane curve C, return the two affine plane curves lying on the standard patches of the blowup of the affine plane at the origin. Note that the two curves returned are the *birational* transforms of C on the blowup patches. The patches are contained in the same affine space as the curve itself. If C does not contain the origin this returns an error message.

`Blowup(C,M)`

This returns the weighted blowup of the plane curve C at the origin defined by the 2×2 matrix of integers M. Again, the birational transform of C is returned inside the ambient plane of C. An error is reported if M does not have determinant ± 1.

Example H114E7

It often happens that one can replace a string of ordinary blowups used to resolve a curve singularity by a single weighted blowup.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,y^2 - x^7);
> f := map< A -> A | [x^2*y,x^7*y^3] >;
> C @@ f;
Curve over Rational Field defined by
x^14*y^7 - x^14*y^6
```

```
> M := Matrix(2,[2,1,7,3]);
> Blowup(C,M);
Curve over Rational Field defined by
-y + 1
14 6
```

The blowup function takes the total pullback as the underlying map and then removes all copies of the x and y axes. The pair of numbers displayed in the final line is the multiplicity of these factors in the total pullback. The curve returned is the birational pullback of C on some patch of a rational surface arising by a number of blowups above the origin of A. It is clearly nonsingular — it's linear! — so this map resolves the singularity at the origin of C.
In fact, MAGMA has machinery for interpreting strings of blowups in terms of a graph, the *resolution graph*.

```
> ResolutionGraph(C);
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -2, 7, 4, 0 ])     2 ;
2 ([ -1, 14, 8, 1 ])    3 ;
3 ([ -3, 6, 3, 0 ])     4 ;
4 ([ -2, 4, 2, 0 ])     5 ;
5 ([ -2, 2, 1, 0 ])     ;
```

Consult Chapter 115 for the full interpretation of this graph. Briefly, one should see this as representing a chain of five blowups which resolve the curve. Each vertex of the graph corresponds to one of the exceptional curves coming from these blowups. The curve extracted by the weighted blowup we saw above corresponds to vertex number 2. Indeed, we can see the multiplicity 14 in the total pullback as the second entry of the labelling sequence. (The multiplicity 6 which we saw above is the corresponding entry in exceptional curve 3.) The fourth entry of that sequence, 1, reports that the birational transform of C to the blownup surface intersects the exceptional curve with multiplicity 1. This is the only nonzero fourth entry of any vertex label, so we conclude that there is exactly one place above the singularity at the origin. This can be confirmed (more quickly!) by the divisor machinery which will be discussed in Section 114.9.

```
> Places(C ! Origin(A));
[
    Place at (0 : 0 : 1)
]
> Degree($1[1]);
1
```

114.4.5 Log Canonical Thresholds

For background on log canonical singularities, see for example [Kollár 1997, Singularities of pairs] or [Kollár 1998, Birational geometry of algebraic varieties].

Let V be a variety with at worst log canonical singularities, P a point on V and D an effective $\mathbf{Q}$-Cartier divisor on V. Then the *log canonical threshold* (lct) of the log pair (V, D) at P is the number

$$\mathrm{lct}_P(V, D) = \sup\{\lambda \in \mathbf{Q} \mid (V, \lambda D) \text{ is log canonical at } P\} \in \mathbf{Q} \cup \{+\infty\}.$$

We can also consider the lct of D along the whole of V:

$$\begin{aligned}\mathrm{lct}(V, D) &= \inf\{\mathrm{lct}_P(V, D) \mid P \in V\} \\ &= \sup\{\lambda \in \mathbf{Q} \mid (V, \lambda D) \text{ is log canonical}\}.\end{aligned}$$

`LogCanonicalThreshold(C)`

The log canonical threshold of the curve C computed at its singular k-points, where k is the base field of C.

`LogCanonicalThresholdAtOrigin(C)`

The local log canonical threshold of the affine curve C computed at the origin.

`LogCanonicalThreshold(C, P)`

The local log canonical threshold of the curve C computed at the point P.

`LogCanonicalThresholdOverExtension(C)`

The log canonical threshold of the curve C computed at all singular points including those defined over some base field extension.

Example H114E8

Consider a cubic curve C on the projective plane, then the singularities of C resemble one of the following examples: a smooth curve, e.g.,

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> A := Curve(P2,x^3-y^2*z-3*x*z^2);
> IsNonsingular(A);
true
```

a curve with ordinary double points (i.e. nodes), e.g.,

```
> B := Curve(P2,x^3-y^2*z-3*x*z^2+2*z^3);
> IsNodalCurve(B);
true
```

a curve with one cuspidal point, e.g.,

```
> C := Curve(P2,x^3-y^2*z);
> #SingularPoints(C) eq 1;
```

```
true
> IsCusp(C,SingularPoints(C)[1]);
true
```

a conic and a line that are tangent, e.g.,

```
> D := Curve(P2,(x^2+(y-z)^2-z^2)*y);
> #PrimeComponents(D) eq 2;
true
> TangentCone(PrimeComponents(D)[1],P2![0,0,1]) eq PrimeComponents(D)[2];
true
```

three lines intersecting at one (Eckardt) point, e.g.

```
> E := Curve(P2,x*y*(x-y));
> IsOrdinarySingularity(E,P2![0,0,1]);
true
> Multiplicity(E,P2![0,0,1]);
3
```

a curve whose support consists of two lines, e.g.,

```
> F := Curve(P2,x^2*y);
> IsReduced(F);
false
> #SingularPoints(ReducedSubscheme(F)) eq 1;
true
> IsNodalCurve(Curve(ReducedSubscheme(F)));
true
```

or a curve whose support consists of three lines, e.g.,

```
> G := Curve(P2,x^3);
> IsReduced(G);
false
> IsNonsingular(ReducedSubscheme(G));
true
```

It is known that a curve is log canonical whenever its singularities are at worst nodal, thus $\mathrm{lct}(P^2, A) = \mathrm{lct}(P^2, B) = 1$. For the remaining reduced curves we can resolve their singularities and calculate their discrepancies to find their log canonical thresholds.

```
> curves := [* A,B,C,D,E,F,G *];
> [LogCanonicalThreshold(curve) : curve in curves];
```

It follows that $\mathrm{lct}(P^2, -K_{P^2}) \leq \frac{1}{3}$. In fact, it is not hard to see that equality holds.

Example H114E9

Here we exhibit a curve C over the rationals, $\mathbf{Q}$, that has singularities defined over a splitting field, k, where $\mathrm{lct}(C)$ (over k) $< \mathrm{lct}(C)$ (over $\mathbf{Q}$). We take a curve C in the projective plane $P2$ with one ordinary double point and two triple point singularities. Such a curve can be obtained by calling:

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := RandomPlaneCurve(6,[1,2],P2);
```

For this example we use the fixed curve C defined below.

```
> f := x*y^5 + y^6 + x^5*z + x^2*y^3*z + 2095/3402*y^5*z + x^4*z^2 -
>      6244382419/8614788*x^3*y*z^2 -
>      28401292681/8614788*x^2*y^2*z^2 -
>      89017753225/25844364*x*y^3*z^2 -
>      243243649115/232599276*y^4*z^2 -
>      2798099890675/70354102*x^3*z^3 -
>      22754590185549/281416408*x^2*y*z^3 -
>      7190675316787/140708204*x*y^2*z^3 -
>      75304687887883/7598243016*y^3*z^3 +
>      17778098933653/140708204*x^2*z^4 +
>      6098447759659/35177051*x*y*z^4 +
>      24308031251845/422124612*y^2*z^4 -
>      4694415764252/35177051*x*z^5 -
>      77497995284599/844249224*y*z^5 +
>      6592790982389/140708204*z^6;
> C := Curve(P2,f);
> IsSingular(C);
true
> LogCanonicalThreshold(C);
1
> IsNodalCurve(C);
false
```

Thus C must have singularities defined over some field extension.

```
> HasSingularPointsOverExtension(C);
true
> LogCanonicalThresholdOverExtension(C);
2/3
```

114.4.6 Local Intersection Theory

The main function here for a single point uses a standard Euclidean algorithm to calculate local intersection numbers at points where two plane curves meet. It was taken from unpublished lecture notes of Franz Winkler, "Introduction to Commutative Algebra and Algebraic Geometry"; the same algorithm is in Fulton's book [Ful69]. These numbers are also called *intersection multiplicities* in the literature. In the following sections there are functions for finding the intersection points of two curves.

There is now a variant that uses an algorithm of Jan Hilmar and Chris Smyth described in [HS10]. This computes all intersection points of the two curves as a set of Galois conjugacy classes and their intersection numbers in a single computation. The implementation, adapted into MAGMA from code contributed by Chris Smyth, returns the sequence of points along with the corresponding local intersection multiplicities.

`IsIntersection(C,D,p)`

Returns `true` if and only if the point p lies on both curves C and D.

`IsTransverse(C,D,p)`

Returns `true` if and only if the point p is a nonsingular point of both plane curves C and D and the curves have distinct tangents there.

`IntersectionNumber(C,D,p)`

The local intersection number $I_p(C, D)$ of the plane curves C and D at the point p. This reports an error if C or D have a common component at p.

`IntersectionNumbers(C,D)`

`IntersectionNumbers(F,G)`

`Global` BOOLELT *Default :* `false`

These intrinsics use the algorithm of Hilmar and Smyth to compute in one go a list of all intersection places along with the corresponding local intersection multiplicities of two projective plane curves C and D defined over k: a finite field, an algebraic field or the rationals $\mathbf{Q}$. Here, intersection place means a point in $\mathbf{P}^2(K)$, where $\mathbf{P}^2$ is the ambient of C and D and K is a finite extension of k, which represents a Galois conjugacy class of ($[K : k]$) points in the intersection of C and D.

The first intrinsic takes C and D as arguments (which must have no common irreducible component) and returns the result as a list of pairs $\langle p, m\rangle$, where p is an element of a pointset $\mathbf{P}^2(K)$ giving a place in the intersection and m is the corresponding local intersection multiplicity.

The second intrinsic avoids the use of pointsets. It takes two homogeneous polynomials F and G which are relatively prime and lie in the same multivariate polynomial ring $P = k[x, y, z]$. They represent two plane curves in $Proj(P)$ and the result is a list of intersection places of these curves with intersection multiplicity. The elements of the list are again pairs $\langle p, m\rangle$, but here p is represented as in Hilmar and Smyth's paper [HS10], p is a list of three elements of one of three types:

i) $[*1, 0, 0*]$. Represents the plane projective point with these homogeneous coordinates.

ii) $[*f(y), 1, 0*]$ with $f(y)$ an irreducible polynomial in $k[y]$. Represents the conjugate points with homogeneous coordinates $[\alpha, 1, 0]$ where α ranges over the roots of f.

iii) $[*h(x, y), g(y), 1*]$ with $g(y)$ an irreducible polynomial in $k[y]$ and $h(x, y)$ a polynomial in $k[x, y]$ whose image in $(k[y]/(g(y)))[x]$ is irreducible. Represents the conjugate points with homogeneous coordinates $[\gamma, \beta, 1]$ where β ranges over the roots of g and, for each β, γ ranges over the roots of $h(x, \beta)$.

The parameter `Global` applies to the polynomial version. If it takes its default value `false`, both the two-variable and one-variable polynomial rings used in the type ii) and iii) representations will be non-global versions with the y and x, y labelling of variables as shown. This may have the disadvantage that elements returned by *different* calls to the intrinsic cannot be directly compared because they lie in different rings. If `Global` is `true`, the global one- and two-variable polynomial rings are used but in this case the variables are not labelled by the intrinsic.

Example H114E10

The local intersection of two curves at a point where they share a common tangent is calculated. If the curves did not share a tangent, the intersection would be the product of multiplicities which it is not in this case.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A, y^2 - x^5);
> D := Curve(A, y - x^2);
> p := Origin(A);
> IntersectionNumber(C,D,p);
4
> Multiplicity(C,p) * Multiplicity(D,p);
2
```

These intersection numbers are often defined to be the length of a particular affine algebra. (See [Har77] Chapter I, Exercise 5.4.) Below it is checked that this definition produces the same result in this case. Note that the algebra is not localised at p so the length calculated is the sum of intersection numbers at all intersection points. At the end one sees that the discrepancy of 1 is accounted for by a single transverse intersection away from the origin.

```
> RA := CoordinateRing(A);
> I := ideal< RA | DefiningPolynomial(C), DefiningPolynomial(D) >;
> Dimension(RA/I);
5
> IP := IntersectionPoints(C,D);
> IP;
{@ (0, 0), (1, 1) @}
> IsTransverse(C,D,IP[1]);
false
> IsTransverse(C,D,IP[2]);
```

```
true
```

Example H114E11

This example is taken from the paper of Hilmar and Smyth. We use the polynomial version for which the output is more explicit.

```
> P<x,y,z> := PolynomialRing(Rationals(),3,"grevlex");
> A := (y-z)*x^5+(y^2-y*z)*x^4+(y^3-y^2*z)*x^3+(-y^2*z^2+y*z^3)*x^2+
>   (-y^3*z^2+y^2*z^3)*x-y^4*z^2+y^3*z^3;
> B := (y^2-2*z^2)*x^2+(y^3-2*y*z^2)*x+y^4-y^2*z^2-2*z^4;
> c := IntersectionNumbers(A,B);
> c;
[* <[* x + y, y^2 + 1, 1 *], 1>, <[* x - y^3, y^4 + 1, 1 *], 1>,
  <[* y^2 + y + 1, 1, 0 *], 2>, <[* x^2 + x + 2, y - 1, 1 *], 1>,
  <[* 1, 0, 0 *], 2>, <[* x^2 + x*y + 2, y^2 - 2, 1 *], 1>,
  <[* x^3 - y, y^2 - 2, 1 *], 1>  *]
```

114.5 Global Geometry

In this section functions which determine global properties of curves such as their genus and whether their equation has a particular form are presented.

114.5.1 Genus and Singularities

`Genus(C)`

`GeometricGenus(C)`

The topological genus of the curve C. More precisely, this is the arithmetic genus of the projective normalisation $\tilde{C}$, which is unique up to k-isomorphism, where k is the basefield of C. C must be an integral curve (reduced and irreducible as a scheme).

Note that, if k is not a perfect field, $\tilde{C}$ may have singularities over an inseparable extension field of k (in technical terms, $\tilde{C}$ is a non-singular scheme, but it may not be k-smooth), in which case the genus of C may drop after some (inseparable) basefield extensions.

`ArithmeticGenus(C)`

The arithmetic genus of the curve C or its projective closure if C is affine. In the case of a plane projective curve of degree d, this number is just $(d-1)(d-2)/2$.

This is really the arithmetic genus of (projective) scheme C and not of its normalisation.

`NumberOfPunctures(C)`

The number of punctures of the affine plane curve C over an algebraic closure of its ground field, that is, the number of points supporting its reduced scheme at infinity. This is just the reduced degree of the polynomial of C at infinity.

`SingularPoints(C)`

The singular points of the curve C which are defined over the base field of C.

`HasSingularPointsOverExtension(C)`

Returns `false` if and only if the scheme of singularities of the curve C has support defined over the base field of C. This function requires that C be reduced.

`Flexes(C)`

`InflectionPoints(C)`

For a plane curve C, this returns the subscheme of C defined by the vanishing of the determinant of the Hessian matrix. This contains the "flex points" of C, which by definition are the nonsingular points at which the tangent line intersects C with multiplicity at least 3.

`C eq D`

Returns `true` if and only if the curves C and D are defined by identical ideals in the same ambient space. (For plane curves, this simply compares defining polynomials of the two curves up to a factor so Gröbner basis calculations are avoided.)

`IsSubscheme(C,D)`

Returns `true` if and only if the curve C is contained (scheme-theoretically) in the curve D.

Example H114E12

We take a plane affine cubic C with a single cusp and non-singular at infinity. Here the projective normalisation of C is isomorphic to the projective line with genus 0, although the arithmetic genus of C is 1.

```
> A<x,y> := AffineSpace(GF(3),2);
> C := Curve(A,y^2 - x^3 - 1);
> Genus(C);
0
> ArithmeticGenus(C);
1
```

Now we consider a similar cubic over the non-perfect field $K = k(t)$ with a single cuspidal singularity defined over an inseparable cubic extension. Now C is a normal and non-singular scheme (but non-smooth), which only loses its normality after an inseparable basefield extension. Here both the genus and arithmetic genus are 1.

```
> K<t> := RationalFunctionField(GF(3));
```

```
> A<x,y> := AffineSpace(K,2);
> C := Curve(A,y^2 - x^3 - t);
> Genus(C);
1
```

114.5.2 Projective Closure and Affine Patches

In MAGMA, any affine space has a unique projective closure. This may be assigned different variable names just like any projective space. The projective closure functions applied to affine curves will return projective curves in the projective closure of the affine ambient. Conversely, a projective space has standard affine patches. These will also appear as the ambient spaces of the standard affine patches of a projective curve.

`ProjectiveClosure(A)`

The projective space that is the projective closure of the ambient A. Unless A is already expressed as a particular patch on some projective space, this is the standard closure defined by the homogenisation of the coordinate ring of A with a new coordinate and unit weights.

`ProjectiveClosure(C)`

The closure of the curve C in the projective closure of its ambient affine space.

Example H114E13

Since the closure of the ambient space is unique, the ambient space of the closure of curves lying in a common affine space is independent of how it is constructed. Here is an odd example but one that occurs in practice when curves and spaces are passed between functions: the functions `ProjectiveClosure()` and `AmbientSpace()` commute!

```
> A<a,b> := AffineSpace(GF(5),2);
> C := Curve(A,a^3 - b^4);
> AmbientSpace(ProjectiveClosure(C)) eq ProjectiveClosure(AmbientSpace(C));
true
```

`LineAtInfinity(A)`

The line which is the complement of the affine plane A embedded in the projective closure of A.

`PointsAtInfinity(C)`

The set of points at infinity defined over the base field of the curve C. The number of these points can also be recovered by the `NumberOfPunctures()` function in the plane case.

`AffinePatch(C,i)`

The i-th affine patch of the projective curve C. For ordinary projective space, the first patch is the one centred on the point $(0 : 0 : \ldots : 0 : 1)$, the second at the point $(0 : 0 : \ldots : 1 : 0)$ and so on.

Example H114E14

Usually one looks at the first affine patch of a curve. If the curve is described, as below, using homogeneous coordinates x,y,z then this is often realised by "setting $z = 1$". Note that we have to assign names to the coordinates explicitly on the affine patches if we want them.

```
> P<x,y,z> := ProjectiveSpace(GF(11),2);
> C := Curve(P,x^3*z^2 - y^5);
> AffinePatch(C,1);
Curve over GF(11) defined by 10*$.1^3 + $.2^5
> C1<u,v> := AffinePatch(C,1);
> C1;
Curve over GF(11) defined by 10*u^3 + v^5
> SingularPoints(C);
{ (1 : 0 : 0), (0 : 0 : 1) }
```

One can also look at other patches. Indeed, sometimes it is necessary. In this example, the curve C has an interesting singularity "at infinity", the point $(1 : 0 : 0)$. If we want to view it on an affine curve then we must take one of the other patches.

```
> C3<Y,Z> := AffinePatch(C,3);
> C3;
Curve over GF(11) defined by Y^5 + 10*Z^2
> IsSingular(C3 ! [0,0]);
true
```

Both affine curves $C1$ and $C3$ have the projective curve C as their projective closure.

```
> ProjectiveClosure(C1) eq ProjectiveClosure(C3);
true
```

114.5.3 Special Forms of Curves

The functions in this section check whether a curve is written in a particular normal form, and also whether it belongs to one of the more specialised families of curve.

`IsEllipticWeierstrass(C)`

Returns `true` if the curve C is nonsingular plane curve of genus 1 in Weierstrass form. This tests the coefficients of the polynomial of C. The conditions guarantee a flex at the point $(0 : 1 : 0)$ either on C or on its projective closure. These are precisely the conditions required by the linear equivalence algorithms for divisors in a later section.

`IsHyperellipticWeierstrass(C)`

Returns `true` if the curve C is a hyperelliptic curve in plane Weierstrass form. The conditions chosen are that the (a) first affine patch be nonsingular, (b) the point $(0:1:0)$ is the only point at infinity and has tangent cone supported at the line at infinity and (c), the projection of C away from that point has degree 2.

`EllipticCurve(C)`

`EllipticCurve(C,p)`

`EllipticCurve(C,p)`

See the description of `EllipticCurve` in Chapter 120.

`IsHyperelliptic(C)`

`Eqn`	BoolElt	*Default :* `true`

`IsGeometricallyHyperelliptic(C)`

`Map`	BoolElt	*Default :* `true`
Verbose	`IsHyp`	*Maximum :* 2

The second function determines whether the curve C is a hyperelliptic curve over the algebraic closure of its base field. If so and if `Map` is `true`, a plane conic or the projective line and a degree 2 map from C to it (all defined over the base field) are returned. The map is to the line if the genus of C is even and to a conic if the genus is odd.

The first function determines whether the curve C is hyperelliptic over its base field K (ie has a degree 2 map to the projective line defined over K). If so, and if the `Eqn` parameter is `true`, it also returns a hyperelliptic Weierstrass model H over K and an isomorphic scheme map from C to H.

Here, hyperelliptic entails genus ≥ 2.

The basic method in both cases is to find the image of C under the canonical map (using functions to be described later) and to check if this is of arithmetic genus zero. If so, this image curve (which is rational normal) is mapped down to a plane conic or the line by repeated adjunction maps. For the second function, the final equation is determined by differential computations in the function field of C once the explicit map to the projective line, which gives the base x function, has been determined.

Example H114E15

```
> P<a,b,c,d,e,f> := ProjectiveSpace(Rationals(),5);
> C := Curve(P,[
> a^2 + a*c - c*e + 3*d*e + 2*d*f - 2*e^2 - 2*e*f - f^2,
> a*c - b^2,
> a*d - b*c,
> a*e - c^2,
```

```
> a*e - b*d,
> b*e - c*d,
> c*e - d^2
> ] );
> boo,hy,mp := IsHyperelliptic(C);
> boo;
true
> hy;
Hyperelliptic Curve defined by y^2 = x^8 + x^6 + x - 1 over Rational Field
> mp;
Mapping from: Sch: C to CrvHyp: hy
with equations :
c*e - d^2 + d*f
-d*f + e*f + f^2
e*f
```

114.6 Maps and Curves

114.6.1 Elementary Maps

The first group of functions create selfmaps of the affine plane. Such a map f can be used to move a curve around the plane simply by applying it to the curve. See Chapter 112 on schemes for more details about maps.

`IdentityAutomorphism(A)`

`Translation(A,p)`

`FlipCoordinates(A)`

`Automorphism(A,q)`

These are the basic automorphisms of the affine plane A taking (x, y) to (x, y), $(x - a, y - b)$, (y, x) and $(x + q(y), y)$ respectively, where p is the point (a, b) and q is a polynomial on A involving y only.

`TranslationToInfinity(C,p)`

The image of C under the change of coordinates which translates p to the point $(0 : 1 : 0)$ in the projective plane and makes the tangent line there equal to the line at infinity. An error is reported if p is a singular point of C. The change of coordinates map is given so that other curves can be mapped by the same change of coordinates.

Example H114E16

In this example we show how one could begin to work out a Weierstrass equation for a Fermat cubic. First we define that cubic curve C in the projective plane and choose a point p on C.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^3 + y^3 + z^3);
> p := C ! [1,-1,0];
> IsFlex(C,p);
true 3
```

The point we have chosen is a flex — the second return value of 3 is the local intersection number of the curve C with its tangent line at p. We use the intrinsic `TranslationToInfinity` to make an automorphism of P which takes the point p to the point $(0 : 1 : 0)$ and takes the curve C to a curve which has tangent line $z = 0$ at the image of p.

```
> C1,phi := TranslationToInfinity(C,p);
> phi(p);
(0 : 1 : 0)
> C1;
Curve over Rational Field defined by
x^3 + 3*y^2*z - 3*y*z^2 + z^3
```

This is almost in Weierstrass form already. It is a pleasant exercise to make coordinate changes which "absorb" some of the coefficients. Alternatively, one can use the intrinsic `EllipticCurve` to perform the entire transformation in one step.

`EvaluateByPowerSeries(m, P)`

Given a map $m : C \to D$, and a nonsingular point P on C, where C is a curve, return $m(P)$, evaluating $m(P)$ using a power series expansion if necessary. This allows a rational map on C to be evaluated at nonsingular base points.

Example H114E17

The following example shows a map evaluated at a point using power series methods.

```
> P2<X,Y,Z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P2,X^3+Y^3-2*Z^3);
> D:=Curve(P2,Y^2*Z-X^3+27*Z^3);
> phi:=map<C->D|[-6*X^2-6*X*Z+6*Y^2+6*Y*Z,
>                9*X^2+18*X*Y+18*X*Z+9*Y^2+18*Y*Z+36*Z^2,
>                X^2-2*X*Z-Y^2+2*Y*Z
>               ]>;
> P:=C![-1,1,0];
> P in BaseScheme(phi);
true (-1 : 1 : 0)
> Q:=EvaluateByPowerSeries(phi,P);
> Q;
(3 : 0 : 1)
```

```
>  phi(P);
>> phi(P);
      ^
Runtime error in map application: Image of map does not lie in the codomain
> pullbackQ:=Q@@phi;
> pullbackQ;
Scheme over Rational Field defined by
-9*X^2 + 9*Y^2,
9*X^2 + 18*X*Y + 18*X*Z + 9*Y^2 + 18*Y*Z + 36*Z^2,
X^3 + Y^3 - 2*Z^3
> IsSubscheme(BaseScheme(phi), pullbackQ);
true
> P in pullbackQ;
true (-1 : 1 : 0)
> Degree(BaseScheme(phi))+1 eq Degree(pullbackQ);
true
```

114.6.2 Maps Induced by Morphisms

Given a non-constant map $\phi : D \to C$ between curves, there are several induced maps between the function fields of C and D and the divisor groups $\mathrm{Div}(C)$ and $\mathrm{Div}(D)$. We refer to the contravariant maps ϕ^* as *Pullback*s and to the covariant maps ϕ_*, corresponding to the Norm between the function fields, as *Pushforward*s. Divisor groups and other function field related items are discussed in Section 114.8.

`Degree(m)`

Returns the degree of a non-constant dominant map m between curves.

`RamificationDivisor(m)`

Returns the ramification divisor of a non-constant dominant map m between irreducible curves.

`Pullback(phi, X)`

Given a map $\phi : D \to C$ between curves and a function, differential, place or divisor X on C, this function returns the pullback of X along ϕ.

`Pushforward(phi, X)`

Given a map $\phi : D \to C$ between curves and a function, place or divisor X on C, this function returns the pushforward of X along ϕ. In older versions, the function applied to a place used to only work with the image of the point (or cluster) below the place for speed and would give an error when ϕ was undefined there. Now, if this is true, the function reverts to working entirely with places and should never fail.

Example H114E18

As an illustration of these routines, consider the following example

```
> Puvw<u,v,w>:=ProjectiveSpace(Rationals(),2);
> Pxyz<x,y,z>:=ProjectiveSpace(Rationals(),2);
> D:=Curve(Puvw,u^4+v^4-w^4);
> C:=Curve(Pxyz,x^4-y^4+y^2*z^2);
> phiAmb:=map<Puvw->Pxyz|[y*z,z^2,x^2]>;
> phi:=Restriction(phiAmb,D,C);
> KC:=FunctionField(C);
> KD:=FunctionField(D);
> Omega:=BasisOfHolomorphicDifferentials(C)[1];
```

Here we see a holomorphic differential pulls back to holomorphic.

```
> IsEffective(Divisor(Pullback(phi,Omega)));
true
```

Ramification divisors are actually quite easy to compute.

```
> RamificationDivisor(phi) eq
>     Divisor(Pullback(phi,Omega))-Pullback(phi,Divisor(Omega));
true
```

Verifying Riemann-Hurwitz:

```
> 2*Genus(D)-2 eq Degree(phi)*(2*Genus(C)-2)+Degree(RamificationDivisor(phi));
true
```

Pulling back and pushing forward is taking powers on the function field.

```
> f:=KC.1;
> Pushforward(phi,Pullback(phi,f)) eq f^Degree(phi);
true
```

`Divisor` and `Pushforward` commute.

```
> g:=KD.1;
> Divisor(Pushforward(phi,g)) eq Pushforward(phi,Divisor(g));
true
```

114.7 Automorphism Groups of Curves

MAGMA now (V2.13) contains functionality for computing the automorphism group of a (reduced and irreducible) curve over a variety of base fields. New structures have been added to aid the user in working with these groups. An automorphism group is of type `GrpAutCrv` and there may be several of these associated to a given curve: the full automorphism group or a subgroup generated by a given set of automorphisms. The automorphism group structure acts as a container for a permutation group representation of the group of curve automorphisms along with the map between the representing group and a list of the actual automorphisms stored in a compressed format.

Elements of the group are of type `GrpAutCrvElt`, a subtype of `MapAutSch`. For these, all of the usual scheme-map operations are available. However, a number of these are specially implemented for the new type and there are also several new operations. In particular, composition and powering of elements makes use of the internal group representation for speed and there are functions for the actions on points, places, divisors etc. which are more efficient than those for a general (finite) map between curves. There is also a function to compute the matrix representation of an automorphism group on the space of holomorphic differentials of the curve.

The full automorphism group is computed at function field level. The algebraic function field versions of some of the functions here are described in Section 42.7.2.

It should be stressed that these groups are groups of *birational* automorphisms of a curve C, in the usual way. This means that they are correspond to actual automorphisms (ie, everywhere defined with an everywhere defined inverse) of the unique normalisation of C and the full automorphism group is the full automorphism group of this normalisation. However, if C is singular, then some of these automorphisms may not be defined at particular singular points.

There are also functions for computing isomorphisms between distinct curves.

All functions require C to have a function field, whether the full group of automorphisms is computed or not.

All of the functions computing the full set of automorphisms require the base field to be perfect (characteristic zero or finite).

114.7.1 Group Creation Functions

`AutomorphismGroup(C)`

Given a reduced, irreducible projective curve C, this function returns the full automorphism group of C over its base field. If the genus of C is less than 2, an error results unless the base field is finite. Note that the full automorphism group is cached so is only ever computed once.

`AutomorphismGroup(C,auts)`

With C as above, this function returns the automorphism group of C generated by the sequence of automorphisms *auts*. The sequence *auts* can consist of either scheme automorphisms of C of type `MapAutSch` or `GrpAutCrvElt` elements lying in

a previously constructed automorphism group of C. The same restrictions on the genus apply.

`Automorphisms(C)`

`Bound` RngIntElt *Default :* ∞

For C as above, this function computes at most `Bound` automorphisms of C over its base field and returns them as a sequence of scheme maps (type `MapSch`). If `Bound` is `Infinity` or at least the order of the full automorphism group, all automorphisms will be returned.

`IsIsomorphic(C, D)`

Given irreducible curves C and D this function returns `true` is C and D are isomorphic over their common base field. If so, it also returns a scheme map giving an isomorphism between them. The curves C and D must be reduced. Currently the function requires that the curves are not both genus 0 nor both genus 1 unless the base field is finite.

`Isomorphisms(C, D)`

`Bound` RngIntElt *Default :* ∞

Given reduced, irreducible curves C and D this function returns at most `Bound` isomorphisms from C to D over their common base field. These are returned as a sequence of scheme maps. The genus restrictions are as for `IsIsomorphic`.

114.7.2 Automorphisms

`A . i`

Let A be a group of automorphisms of curve C and let i be an integer such that $-n \leq i \leq n$, where n is the number of generators of A. This operator returns the i-th generator for A. A negative subscript indicates that the inverse of the generator is to be created. Finally, $A.0$ denotes the identity of A.

Note that if A is an automorphism group that was generated from a list of specified automorphisms, *auts*, then the generators of A will not necessarily be these "user" generators, but will be those coming from the permutation representation of A.

`Identity(A)`

`Id(A)`

`A ! 1`

The identity element of the automorphism group A.

`A ! f`

Let A be an automorphism group of a curve C. Given a scheme map f from C to C or an element of some (other) automorphism group of C, this function returns the `GrpAutCrvElt` element of A corresponding to f. An error will result if f is not in A.

`Order(f)`

The order of the curve automorphism f.

`Inverse(f)`

The inverse of the curve automorphism f.

`f * g`

The product of the curve automorphisms f and g in automorphism group A. If f and g are regarded as maps, this function returns their composite: first apply f, then apply g. Note that this composition uses the permutation representation of A and is generally performed faster than the usual scheme map composition.

`f ^ n`

The nth power of the curve automorphism f. The integer n may be positive or negative. Again, this uses the permutation representation of A, the parent of f.

`g eq h`

Given curve automorphisms g and h belonging to automorphism groups of the same curve C, return `true` if g and h represent the same automorphism, `false` otherwise. Note that g and h do not have to belong to the same automorphism group.

`g ne h`

The logical negation of the preceding function.

`SchemeMap(f)`

It is sometimes useful for the user to convert a curve automorphism back to an object of plain scheme isomorphism type. This is a convenience function that simply returns the curve automorphism f as a `MapAutSch`.

114.7.3 Automorphism Group Operations

`Curve(A)`

The curve of which A is a group of automorphisms.

`Order(A)`

The order of A.

`FactoredOrder(A)`

The factored order of A.

`NumberOfGenerators(A)`

`Ngens(A)`

The number of generators of A. If A was defined as the group generated by a given sequence of automorphisms, this sequence will not necessarily coincide with the set of generators, which is determined by the internal permutation representation.

`Generators(A)`

The generators of A - a small set of `GrpAutCrvElt` elements of A, which generate it as a group - returned in a sequence.

`PermutationGroup(A)`

The permutation group of A which gives the abstract group representation of the automorphism group.

`PermutationRepresentation(A)`

The permutation group representation of A consisting of the actual permutation group G and an invertible map which takes elements of G to the corresponding curve automorphism (as a `GrpAutCrvElt` element).

`MatrixRepresentation(A)`

For an automorphism group A on a curve C of genus at least 2, computes the matrix representation of A acting on the space of holomorphic differentials by pullback. The differentials are represented by row vectors with respect to a basis B and the matrix associated to $g \in A$ gives the pullback action of g by right multiplication of the row vectors. The (finite) matrix group image G is returned along with a map from A to G giving the representation and an enumerated sequence containing the basis of differentials B used.

`a in A`

Returns whether a curve automorphism, given as a `GrpAutCrvElt` or `MapSch`, is equal to an automorphism lying in A.

`A subset B`

Returns whether A is actually a subgroup of B as automorphism groups of the same curve.

114.7.4 Pullbacks and Pushforwards

For `GrpAutCrvElt` elements, pullbacks and pushforwards (images) of functions, places, divisors and differentials of the curve C are handled more directly than for general curve maps. The functions to perform these operations use the more direct `f(x)` and `x @@ f` syntax and these should be used in preference to the `Pushforward` and `Pullback` functions in Section 114.6.2.

In addition, the computation of images of points on C by a curve automorphism is handled slightly differently. If the point doesn't lie in the original domain of definition of the scheme map giving the automorphism, the image is still computed without extending the map (although for some singular points of the curve model, the image may still not exist for the "birational" automorphism).

`f(X)`
`X @ f`

Given a curve automorphism f of curve C and a point, function, differential, place or divisor X on C, this function returns the image (or pushforward) of X under f.

`X @@ f`

Given a curve automorphism f of curve C and a function, differential, place or divisor X on C, this function returns the inverse image (or pullback) of X under f.

Example H114E19

Here are some examples of the above functions applied to the genus 3 Fermat curve in characteristic 0.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^4+y^4+z^4);
> L := Automorphisms(C);
> #L;
24
> // to get all automorphisms, we base change to Q(zeta_8)
> K := CyclotomicField(8);
> C1 := BaseChange(C,K);
> L1 := Automorphisms(C1);
> #L1;
96
> // next, we get the automorphism as a group
> G := AutomorphismGroup(C1);
> g := G!iso<C1 -> C1 | [y,z,x],[z,x,y]>;
> Gp,rep := PermutationRepresentation(G);
> Gp;
Permutation group Gp acting on a set of cardinality 12
Order = 96 = 2^5 * 3
    (2, 4)(3, 5)(6, 8)(7, 10)
    (2, 5, 10, 8, 4, 3, 7, 6)(11, 12)
    (1, 2, 3)(4, 5, 9)(6, 11, 7)(8, 12, 10)
```

```
> rep(g);
(1, 8, 4)(2, 9, 6)(3, 10, 11)(5, 7, 12)
> Inverse(rep)(Gp.2);
Mapping from: CrvPln: C1 to CrvPln: C1
with equations :
zeta_8^2*$.1^3*$.3
$.1^3*$.2
$.2^4 + $.3^4
and inverse
zeta_8^2*$.1^3*$.3
-zeta_8^2*$.1^3*$.2
$.2^4 + $.3^4
> $1 eq G.2;
true
```

Example H114E20

In the next example we look at a superelliptic curve over $GF(7^2)$ with isomorphism group a central extension of $PGL_2(\mathbf{F}_7)$ by a cyclic group of order 4.

```
> SetSeed(1);
> k := GF(7^2);
> A<x,y> := AffineSpace(k,2);
> C := ProjectiveClosure(Curve(A,y^4-x^7+x));
> G := AutomorphismGroup(C);
> Gp,rep := PermutationRepresentation(G);
> Gp;
Permutation group G acting on a set of cardinality 192
Order = 1344 = 2^6 * 3 * 7
> [Order(G.i) : i in [1..Ngens(G)]];
[ 7, 24, 2 ]
> Z := Centre(Gp); Z;
Permutation group Z acting on a set of cardinality 192
Order = 4 = 2^2
> (Z.1)@@rep;
Mapping from: CrvPln: C to CrvPln: C
with equations :
$.1
k.1^12*$.2
$.3
and inverse
$.1
k.1^36*$.2
$.3
> Gp1 := quo<Gp|Z>;
> boo := IsIsomorphic(Gp1,PGL(2,GF(7)));
> boo;
true
```

```
> // Find the representation on the 176 Weierstrass places.
> // Only need the action of the generators of Gp (or G)
> wpls := WeierstrassPlaces(C);
> wpls_perms := [[Index(wpls,g(w)) : w in wpls]: g in Generators(G)];
> G_wpls := SymmetricGroup(#wpls);
> weier_rep := hom<Gp->G_wpls|[G_wpls!p : p in wpls_perms]>;
> //Check that its faithful
> K := Kernel(weier_rep);
> #K;
1
```

Example H114E21

The next example illustrates the use of `IsIsomorphic` in finding an isomorphism between the well-known plane model of genus 3 curve $X(7)$ and a degree 6 model of it in $\mathbf{P}^3$. The isomorphism between C and D is only computed in one direction, but we can then use `IsInvertible`.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,x^3*y+y^3*z+z^3*x);
> P3<a,b,c,d> := ProjectiveSpace(Rationals(),3);
> D := Curve(P3,[b^2-a*d,a*b*c+b*d^2+c^3]);
> boo,im := IsIsomorphic(C,D);
> boo;
true
> im;
Mapping from: CrvPln: C to Crv: D
with equations :
x^2
x*z
y*z
z^2
> _,imi := IsInvertible(im);
> Inverse(imi);
Mapping from: CrvPln: C to Crv: D
with equations :
x^2
x*z
y*z
z^2
and inverse
b
c
d
```

We now compute the automorphism group of C over $GF(11^3)$ where the full set of 168 automorphisms exists (the group is isomorphic to $PSL_2(\mathbf{F}_7)$) and generate some subgroups.

```
> P2<x,y,z> := ProjectiveSpace(GF(11^3),2);
> C := Curve(P2,x^3*y+y^3*z+z^3*x);
```

```
> G := AutomorphismGroup(C);
> Order(G);
168
> [Order(g) : g in Generators(G)];
[ 7, 3, 7 ]
> G1 := AutomorphismGroup(C,[G.1,G.2]);
> Order(G1);
21
> PermutationGroup(G1);
Permutation group acting on a set of cardinality 8
Order = 21 = 3 * 7
    (2, 4, 3, 7, 5, 8, 6)
    (3, 5, 4)(6, 8, 7)
> // can also find the normaliser of <G.3> via the
> // permutation rep
> Gp,rep := PermutationRepresentation(G);
> H := Normaliser(Gp,sub<Gp|Gp.3>);
> #H;
21
> Hgens := [g@@rep : g in Generators(H)];
> [Order(g) : g in Hgens];
[ 3, 7 ]
> h := Hgens[1];
> //check directly that h normalises <g> in G
> g := G.3;
> Index([g^i : i in [1..7]],h*g*(h^-1));
4
```

114.7.5 Quotients of Curves

For G an arbitrary group of automorphisms of a curve C of genus $>= 2$, the main intrinsic in this section computes a model of the quotient curve C/G along with the explicit projection map $C \to C/G$.

`CurveQuotient(G)`

G is a group of automorphisms of a curve C/k, which must be of genus $g >= 2$. The function computes a projective, non-singular model of C/G, the scheme theoretic quotient of C by G. This is returned with the G-invariant projection map from C down to it.

If $k(C)$ is the function field of C then C/G can be thought of as the curve with function field $k(C)^{G^*}$ where G^* is the group of field automorphisms of $k(C)/k$ induced by G under function pull-back.

The implementation utilises MAGMA's Function Field and Invariant Theory functionality and uses a variety of methods. There is no restriction on the characteristic p

and the function works in positive characteristic as long as the following assumption is true:

$$C \to C/G \text{ is tamely ramified when genus}(C/G) > 1$$

This is equivalent to saying that, for all points P on (the non-singular, projective model of) C, the subgroup G_P of G fixing P has order prime to p. In practise, this will only be a problem for very small $p > 0$ when $p|\#G$.

The algorithm used is slightly different depending on g_G, the genus of the quotient curve.

When $g_G >= 2$ and the quotient is non-hyperelliptic, the canonical image of C/G is computed and this is the model returned. This uses function field functionality only.

The case $g_G >= 2$ and the quotient is hyperelliptic is similar, only extra work is needed to find the "y-coordinate". Again everything is done purely with function fields. The model returned is a `CrvHyp` Weierstrass model if the quotient is hyperelliptic over k, or a bi-quadratic model in P^3 if it is only geometrically hyperelliptic.

When g_G is 0 or 1, the canonical map methodology used in the above cases fails and we use a combination of Invariant theory and function field methods instead.

For $g_G = 0$, the model returned for C/G is either the projective line P^1 or a conic in P^2. In this case, the function does not automatically search for k-rational points so, if a conic is returned, the quotient may still be isomorphic to P^1 over k.

For $g_G = 1$, the model returned is a projectively normal embedding by quadrics in P^n, $n >= 3$, or a cubic in P^2. Again, the function does not search for k-rational points in order to try to convert the quotient into elliptic curve form.

Example H114E22

We start with our old friend the Klein quartic again - this time over Q. The quotient by an automorphism of order 3 is a genus 1 curve. We find this quotient and produce an isomorphism from this to an elliptic curve using the rational point that is the projection of $(0 : 0 : 1)$.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,x^3*y+y^3*z+z^3*x);
> phi := iso<C->C|[y,z,x],[z,x,y]>;
> // we will take the quotient by phi
> G := AutomorphismGroup(C,[phi]);
> CG,prj := CurveQuotient(G);
> CG;
Curve over Rational Field defined by
x[1]^2 - 13*x[1]*x[2] + 8*x[1]*x[3] + 10*x[2]*x[3] - 6*x[3]^2 + 15*x[1]*x[4] +
    3*x[2]*x[4] - 6*x[3]*x[4] - 6*x[4]^2,
x[1]^2 - 12*x[1]*x[2] + 3*x[2]^2 + 8*x[1]*x[3] + 12*x[1]*x[4]
> Genus(CG);
1
> // find an minimal elliptic Weierstrass model
> ptCG := prj(C![0,0,1]);
> E1, psi1 := EllipticCurve(CG,ptCG);
```

```
> E, psi := MinimalModel(E1);
> prj := Expand(prj * psi1 * psi); // get the composite map C -> E
> E;
Elliptic Curve defined by y^2 + x*y = x^3 - x^2 - 2*x - 1 over Rational Field
> prj;
Mapping from: CrvPln: C to CrvEll: E
with equations :
-2*x^2*y^8 + 2*x*y^8*z - 2*y^9*z - 2*x*y^7*z^2 - 2*x^2*y^5*z^3 + 2*x*y^6*z^3 -
    2*y^7*z^3 + 2*x^2*y^4*z^4 - 2*x^2*y^3*z^5 - 2*y^4*z^6 + 2*y^3*z^7 -
    2*y^2*z^8
4*x^2*y^8 - 2*x*y^9 - 2*x^2*y^7*z - 2*x*y^8*z + 4*y^9*z + 2*x^2*y^6*z^2 +
    4*x*y^7*z^2 - 2*y^8*z^2 + 2*x^2*y^5*z^3 - 4*x*y^6*z^3 + 4*y^7*z^3 -
    4*x^2*y^4*z^4 + 2*x*y^5*z^4 - 2*y^6*z^4 + 4*x^2*y^3*z^5 + 2*y^5*z^5 -
    2*x^2*y^2*z^6 + 2*y^4*z^6 - 4*y^3*z^7 + 4*y^2*z^8 - 2*y*z^9
2*x*y^8*z - 2*x^2*y^5*z^3 - 2*y^4*z^6
> Conductor(E);
49
```

In fact, C is the modular curve $X(7)$ and E is $X_0(49)$. That C/G and E are isogenous elliptic curves also follows from modular form theory.

Example H114E23

As a second example we consider the genus 4 modular curve $X_0(54)$ with the two commuting Atkin-Lehner involutions W_2, W_{27}. We compute the quotients by $< W_2 >$, $< W_{27} >$ and $< W_2, W_{27} >$ which respectively have genera 2,1,0.

```
> P<[x]> := ProjectiveSpace(Rationals(),3);
> X054 := Curve(P,[
>   x[2]*x[3] - x[1]*x[4], 4*x[1]^2*x[2] + 2*x[1]*x[2]^2 +
>       x[2]^3 - x[3]^3 + x[3]^2*x[4] - x[3]*x[4]^2]);
> W2 := iso<X054->X054|[1/2*x[4],-x[3],-x[2],2*x[1]],
>                 [1/2*x[4],-x[3],-x[2],2*x[1]]>;
> W27 := iso<X054->X054|
> [
>     -1/3*x[1] - 1/3*x[2] - 1/3*x[3] - 1/3*x[4],
>     -2/3*x[1] - 2/3*x[2] + 1/3*x[3] + 1/3*x[4],
>     -2/3*x[1] + 1/3*x[2] - 2/3*x[3] + 1/3*x[4],
>     -4/3*x[1] + 2/3*x[2] + 2/3*x[3] - 1/3*x[4]
> ],
> [
>     -1/3*x[1] - 1/3*x[2] - 1/3*x[3] - 1/3*x[4],
>     -2/3*x[1] - 2/3*x[2] + 1/3*x[3] + 1/3*x[4],
>     -2/3*x[1] + 1/3*x[2] - 2/3*x[3] + 1/3*x[4],
>     -4/3*x[1] + 2/3*x[2] + 2/3*x[3] - 1/3*x[4]
> ]>;
> // 1. Quotient by <W2>
> G := AutomorphismGroup(X054,[W2]);
> CG,prj := CurveQuotient(G);
```

```
> CG;
Hyperelliptic Curve defined by y^2 + (-x^2 - 1)*y = 387*x^6 + 999*x^5 + 785*x^4
    + 294*x^3 + 58*x^2 + 6*x over Rational Field
> Genus(CG);
2
> // 2. Quotient by <W27>
> G := AutomorphismGroup(X054,[W27]);
> CG,prj := CurveQuotient(G);
> CG;
Curve over Rational Field defined by
1878243840*x[1]*x[2] - 68400*x[2]^2 - 680252400*x[1]*x[3] - 774110424*x[2]*x[3]
    + 246079248*x[3]^2 + 1252162560*x[1]*x[4] - 298086240*x[2]*x[4] -
    834823168*x[3]*x[4] - 2254334400*x[1]*x[5] - 447968988*x[2]*x[5] +
    18*x[3]*x[5] + 936*x[4]*x[5] + 346615560*x[1]*x[6] + 63486504*x[2]*x[6] -
    1620*x[5]*x[6] + 14787*x[6]^2,
-4225630080*x[1]*x[2] + 152304*x[2]^2 + 1530412160*x[1]*x[3] +
    1741572584*x[2]*x[3] - 553624072*x[3]^2 - 2817086720*x[1]*x[4] +
    670624096*x[2]*x[4] + 1878164832*x[3]*x[4] + 5071740600*x[1]*x[5] +
    1007828352*x[2]*x[5] - 1824*x[4]*x[5] - 779805580*x[1]*x[6] -
    142830348*x[2]*x[6] + 4*x[3]*x[6] + 3870*x[5]*x[6] - 33021*x[6]^2,
-14377204800*x[1]*x[2] + 512928*x[2]^2 + 5207031920*x[1]*x[3] +
    5925482424*x[2]*x[3] - 1883642044*x[3]^2 - 9584803200*x[1]*x[4] +
    2281707168*x[2]*x[4] + 6390228448*x[3]*x[4] + 64*x[4]^2 +
    17255963160*x[1]*x[5] + 3429009288*x[2]*x[5] - 5472*x[4]*x[5] -
    2653188900*x[1]*x[6] - 485962956*x[2]*x[6] + 14040*x[5]*x[6] -
    111537*x[6]^2,
762945840*x[1]*x[2] - 27648*x[2]^2 - 276319320*x[1]*x[3] - 314444676*x[2]*x[3] +
    99957860*x[3]^2 + 508630560*x[1]*x[4] - 121082832*x[2]*x[4] -
    339106480*x[3]*x[4] - 915713640*x[1]*x[5] - 181965582*x[2]*x[5] +
    360*x[4]*x[5] + 140795640*x[1]*x[6] + 25788312*x[2]*x[6] - 675*x[5]*x[6] +
    5985*x[6]^2,
559393920*x[1]*x[2] - 18864*x[2]^2 - 202594400*x[1]*x[3] - 230548712*x[2]*x[3] +
    73289896*x[3]^2 + 372929280*x[1]*x[4] - 88775136*x[2]*x[4] -
    248632672*x[3]*x[4] - 671395320*x[1]*x[5] - 133415856*x[2]*x[5] +
    103229820*x[1]*x[6] + 18907956*x[2]*x[6] + 16*x[4]*x[6] - 702*x[5]*x[6] +
    4167*x[6]^2,
488083680*x[1]*x[2] - 16272*x[2]^2 - 176767680*x[1]*x[3] - 201158520*x[2]*x[3] +
    63947136*x[3]^2 + 325389120*x[1]*x[4] - 77457888*x[2]*x[4] -
    216937408*x[3]*x[4] - 585806400*x[1]*x[5] - 116408124*x[2]*x[5] + 81*x[5]^2
    + 90070080*x[1]*x[6] + 16497576*x[2]*x[6] - 648*x[5]*x[6] + 3609*x[6]^2,
-103229280*x[1]*x[2] + 3456*x[2]^2 + 37386240*x[1]*x[3] + 42544872*x[2]*x[3] -
    13524760*x[3]^2 - 68819520*x[1]*x[4] + 16382304*x[2]*x[4] +
    45882080*x[3]*x[4] + 123897600*x[1]*x[5] + 24620220*x[2]*x[5] -
    19049760*x[1]*x[6] - 3489228*x[2]*x[6] + 135*x[5]*x[6] - 765*x[6]^2,
5219280*x[1]*x[2] - 216*x[2]^2 - 1890360*x[1]*x[3] - 2151168*x[2]*x[3] +
    683800*x[3]^2 + 3479520*x[1]*x[4] - 828384*x[2]*x[4] - 2319840*x[3]*x[4] -
    6264540*x[1]*x[5] - 1244862*x[2]*x[5] + 963210*x[1]*x[6] + 176418*x[2]*x[6]
    + 45*x[6]^2,
```

```
52200*x[1]*x[2] - 18900*x[1]*x[3] - 21510*x[2]*x[3] + 6840*x[3]^2 +
    34800*x[1]*x[4] - 8280*x[2]*x[4] - 23200*x[3]*x[4] - 62640*x[1]*x[5] -
    12447*x[2]*x[5] + 9630*x[1]*x[6] + 1764*x[2]*x[6]
> Genus(CG);
1
> // 3. Quotient by <W2,W27>
> G := AutomorphismGroup(X054,[W2,W27]);
> CG,prj := CurveQuotient(G);
> CG;
Curve over Rational Field defined by
0
> Ambient(CG);
Projective Space of dimension 1
Variables : $.1, $.2
```

114.8 Function Fields

An integral curve C has a coordinate ring that is an integral domain. The function field of the curve is the corresponding field of fractions in the affine case and the homogeneous degree 0 part of this in projective cases. The function field of an affine curve is isomorphic to that of its projective closure. As with schemes generally, a function field is attached to projective curves and the same object represents the function field of all of its affine patchs.

Furthermore, in the curve case, there is a unique (up to abstract scheme isomorphism) (ordinary) projective non-singular curve $\tilde{C}$ which is birationally equivalent to C (ie there are maps from C to $\tilde{C}$ which are defined at all but finitely many points and whose composite is the identity where defined) $\Leftrightarrow \tilde{C}$ has the same function field (up to isomorphism) as C.

When C is projective, $\tilde{C}$ is just the *normalisation* of C. The normalisation $\tilde{C}$ differs from C only at singular points of the latter and C can be thought of as a singular model of $\tilde{C}$ and, as is usual with curves, most of the functionality provided by the function field and the objects attached - places, divisors etc. - can be more properly thought of as relating to $\tilde{C}$. The first section below treats function fields and some basic functions related to them and their elements.

From now on we assume that the reader is familiar with the notion of divisor, linear equivalence and their relationship with function fields. If not, there are very brief discussions of them at the beginning of each section and also in the introduction to this chapter, but you may also consult standard texts such as [Har77] Chapter II, 6 (especially from page 136) and Chapter IV for a more serious treatment.

For functions working with elements of a function field of a scheme and the scheme itself see Section 112.6.

114.8.1 Function Fields

For the purposes of this section, function fields are fields of rational functions on a curve C. Let f be an element and p a nonsingular rational point of C. Then one can evaluate f at p and compute the order of a zero or a pole of f at p, an integer which is positive for zeros and negative for poles. This allows points of the curve to be considered as valuations of the function field.

In fact, the proper language for discussing valuations and function fields is that of *places* and *divisors*, which really correspond to points of the projective normalisation $\tilde{C}$ of C and formal sums of these, and they are discussed in later sections. Functions in this section which take a point of a curve as an argument are convenient shorthands for functions taking a place and are only allowed when there is no ambiguity about which place is intended, which is why p is required to be non-singular. Functions which compute the zeros and poles of rational functions properly return divisors, so will be discussed later.

Finally, function fields and divisor groups are cached so that recomputation is avoided. Although it is transparent in practice, it is worth remembering that the function fields and divisor groups are always attached to the projective model of a curve, rather than to any of the affine models. Since these are all tightly related, it doesn't make any difference. The support points of divisors will be returned as points on the projective model since they can quite easily lie at infinity on any particular affine model. For a completely clean treatment, it is possible to work exclusively with the projective model, although it is certainly not necessary. Indeed, for some time the elliptic curve machinery in MAGMA has happily presented affine models of curves together with projective points.

```
FunctionField(C)
```

The function field of the curve C, a field isomorphic to the field of fractions of the coordinate ring of C. It can be assigned generator names using the diamond bracket notation, as in the example below. The function field will only exist when C is integral (reduced and irreducible) and this can be checked directly or by calling the function below if it is in doubt.

```
HasFunctionField(C)
```

```
Curve(F)
```

The curve used to create the function field F, or the projective closure of that curve (if it was affine). The function field is stored on projective curves so that the same field is returned whenever it is called for from any patch of the projective curve.

```
F ! r
```

Coerce the ring element r into the function field F of a scheme. For the coercion to be successful r must be in a ring related to the scheme of F, e.g. the base ring or coordinate ring of (or a field of fractions of) the scheme or one of its affine patches or a subscheme or super scheme.

`ProjectiveFunction(f)`

Return the function f in the function field of a scheme as a function in projective coordinates (as an element in the field of fractions of the coordinate ring of the projective scheme having function field the parent of f).

Example H114E24

After creating a curve in the usual way we make its function field F.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^4 + 2*x*y^2*z + 5*y*z^3);
> F<a,b> := FunctionField(C);
> F;
Function Field of Curve over Rational Field defined by
x^4 + 2*x*y^2*z + 5*y*z^3
> Curve(F);
Curve over Rational Field defined by
x^4 + 2*x*y^2*z + 5*y*z^3
> b^2;
b^2;
```

Once constructed, the function field will be stored with the curve (or its projective closure). Thus the same field will be returned from multiple function calls.

```
> FunctionField(C) eq FunctionField(AffinePatch(C,3));
true
```

`p @ f`

`f(p)`

`Evaluate(f, p)`

The ring element $f(p)$ where f is an element of the function field of the curve on which p is a point. If f has a pole at p the value infinity is returned.

`Expand(f, p)`

Given an element f on a curve C and a place p of C return a series which is the expansion of f at p and the uniformizing element of p.

`Completion(F, p)`

`Precision` RngIntElt *Default :* 20

The completion of the function field F of the curve C at the place p of C and a map from F into its completion.

`Degree(f)`

Given an element f of the function field of a curve, return the degree of f. This is the degree of the map given by f to $\mathbf{P}^1$ or the degree of the numerator and denominator of the principal divisor of f. If f is constant, then 0 is returned.

`Valuation(f, p)`

The degree of the zero of the function f at the point p where f is a function on the curve on which p is a point. A negative value indicates there is a pole of f at p.

`Valuation(p)`

The valuation of the function field of the curve on which p lies centred at the point p. This is a map from the function field to the integers.

`UniformizingParameter(p)`

A rational function on the curve of the nonsingular point p which having valuation 1 at p.

`Module(S)`

`Preimages`	BoolElt	*Default :* `false`
`IsBasis`	BoolElt	*Default :* `false`

Given a sequence S of elements of a function field of a curve C return the module over the base ring of C generated by the elements of S. Also return the map from the module to the function field and a sequence of preimages of the elements of S if `Preimages` is `true`.

If `IsBasis` is `true` then the elements of S will be assumed to be a basis of the module.

`Relations(S)`

`Relations(S, m)`

Given a sequence S of elements of a function field of a curve C return the module over the base ring R of C of R-linear relations between the elements of S.

`Genus(C)`

The genus of the curve C.

`FieldOfGeometricIrreducibility(C)`

Return the algebraic closure of the base ring of C in the function field of C along with the map including the closure in the function field.

`IsAbsolutelyIrreducible(C)`

Returns `true` if the field of geometric irreducibility of the curve C is the base ring of C.

`DimensionOfFieldOfGeometricIrreducibility(C)`

The dimension of the field of geometric irreducibility of the curve C over the base ring of C.

Example H114E25

Having made a curve and its function field we make an element of the function field using its named generators a, b. The function is put into a convenient form which, for this F, ensures that the denominator is a polynomial in a alone.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^4 + 2*x*y^2*z + 5*y*z^3);
> F<a,b> := FunctionField(C);
> f := a/b;
> f;
(-2*a*b - 5)/a^3
```

Now we choose a point of the curve and find that f has a pole there of order 3.

```
> p := C ! [0,0,1];
> Evaluate(f,p);
Infinity
> Valuation(f,p);
-3
```

Computing the valuations of the generators we notice that a is a uniformising parameter at p — indeed, it is the parameter automatically returned. Clearly the valuation of $f = a/b$ at p should be $1 - 4 = -3$ as computed in the previous line.

```
> vp := Valuation(p);
> vp(a), vp(b);
1 4
> UniformizingParameter(p);
a
```

`GapNumbers(C)`

The gap numbers of the curve C.

`WronskianOrders(C)`

The Wronskian orders of the curve C.

`NumberOfPlacesOfDegreeOverExactConstantField(C, m)`

`NumberOfPlacesDegECF(C, m)`

The number of places of degree m of the curve C defined over a finite field. Contrary to the `Degree` function the degree is here taken over the field of geometric irreducibility.

`NumberOfPlacesOfDegreeOneOverExactConstantField(C)`

`NumberOfPlacesOfDegreeOneECF(C)`

The number of places of degree one of the curve C defined over a finite field. Contrary to the `Degree()` function the degree is here taken over the field of geometric irreducibility.

`NumberOfPlacesOfDegreeOneOverExactConstantField(C, m)`

`NumberOfPlacesOfDegreeOneECF(C, m)`

The number of places of degree one in the constant field extension of degree m of the curve C. Contrary to the `Degree()` function the degree is here taken over the field of geometric irreducibility.

`NumberOfPlacesOfDegreeOneECFBound(C)`

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(C)`

`NumberOfPlacesOfDegreeOneECFBound(C, m)`

`NumberOfPlacesOfDegreeOneOverExactConstantFieldBound(C, m)`

An upper bound on the number of places of degree one in the constant field extension of degree m (if given) of the curve C. Contrary to the `Degree` function the degree is here taken over the respective exact constant fields.

`DivisorOfDegreeOne(C)`

Return a divisor of the curve C of degree 1 over its field of geometric irreducibility.

`SerreBound(C)`

`SerreBound(C, m)`

`IharaBound(C)`

`IharaBound(C, m)`

The Serre and Ihara bounds of the number of places of degree 1 over the field of geometric irreducibility of the curve C over the extension of degree m of the base ring of C, which must be a finite field.

`LPolynomial(C)`

`LPolynomial(C, m)`

`ZetaFunction(C)`

`ZetaFunction(C, m)`

The L-polynomial and the ζ function of the curve C over the extension of degree m of the base ring of C, which must be a finite field.

114.8.2 Representations of the Function Field

The function field of a scheme has very little direct functionality. But when the scheme is a curve the function field is isomorphic to an algebraic function field as described in Chapter 42. This isomorphism is used internally in many computations for curves. Although the isomorphic algebraic function field can be retrieved from the function field of the scheme, this should not be necessary during ordinary usage of the curves.

`AlgorithmicFunctionField(F)`

Given the function field F of a curve C, this function returns the background algebraic function field, AF. As this is the object where such calculations as those involving places, divisors and differentials are performed, we refer to it as the algorithmic or arithmetic function field. Since there are curve functions which provide an interface to most of the functionality of this field via C, F and its elements, the user can usually avoid accessing AF directly. However, when it is required, it is also usually necessary to translate between elements of F and AF. A map from F to AF is thus also returned. This map is invertible and its inverse is used to map elements the other way.

`FunctionFieldPlace(p)`
`CurvePlace(C, p)`
`FunctionFieldDivisor(d)`
`CurveDivisor(C, d)`
`FunctionFieldDifferential(d)`
`CurveDifferential(C, d)`

Return the place, divisor or differential of the algebraic function field corresponding to the place, divisor or differential of a curve or convert the place p, divisor or differential d of an algebraic function field into a place, divisor or differential of the curve C.

114.8.3 Differentials

The space of differentials in MAGMA is the vector space of elements df over the function field of a curve, where f is any element of the function field and where the operator d satisfies the usual derivation conditions. This vector space is called the *differential space* and corresponds to the Kähler differentials of [Har77], II.8. Note that the differential space is not explicitly a vector space in MAGMA. Rather, as so often when there are many different structures to be considered, a vector space together with a map to the space of differentials is given. (Of course, basic vector space arithmetic works on the space of differentials.) In fact, this is appropriate: after all, Kähler differentials are merely a model of an object which one might prefer to define by its universal properties.

114.8.3.1 Creation of Differentials

`DifferentialSpace(C)`

The space of differentials of the curve C.

`SpaceOfDifferentialsFirstKind(C)`

`SpaceOfHolomorphicDifferentials(C)`

Given a curve C, this function returns a vector space V and a map from V to the space of differentials of C with image the holomorphic differentials on C.

`BasisOfDifferentialsFirstKind(C)`

`BasisOfHolomorphicDifferentials(C)`

Given a curve C, this function returns a basis for the space of holomorphic differentials of C.

`DifferentialSpace(D)`

Given a divisor D associated with curve C, this function returns a vector space V and a map from V to the space of differentials of the curve C containing the divisor D with image the differentials of $\omega_C(D)$. Colloquially, this refers to the differentials whose zeros are at least the positive (or effective) part of D and whose poles are no worse than the negative part of D.

`DifferentialBasis(D)`

Given a divisor D on a curve, this function returns the basis of the differential space of D.

`Differential(a)`

The exact differential $d(a)$ of the function field element a.

114.8.3.2 Operations on Differentials

The space of differentials admits vector space operations over the function field of the curve. As such, it is one-dimensional so one can even divide two non-zero differentials to recover an element of the function field.

`Identity(S)`

The identity differential of the differential space S of a curve.

`Curve(S)`

The curve for which S is the space of differentials.

`Curve(a)`

The curve to which the differential a belongs.

`f * x` `x * f` `x + y` `- x` `x - y` `x / r` `x / y`

The basic arithmetic in the space of differentials. Thought of as a vector space over the function field, this space is one-dimensional. The final operation uses this fact to return a function field element as the quotient of two differentials.

`S eq T`

Returns `true` if and only if the two spaces of differentials S and T are the same.

`a eq b`

Returns `true` if and only if the differentials a and b are equal.

`a in S`

Returns `true` if and only if a is an element of the differential space S of a curve.

`IsExact(a)`

Returns `true` if and only if a is known to be an exact differential, that is, if it is known to be of the form df. If this is not already known, no further attempt is made to determine that.

`IsZero(a)`

Returns `true` if and only if the differential a is the zero differential.

`Valuation(d, P)`

The valuation of the differential d of a curve at the place P of the same curve.

`Residue(d, P)`

The residue of the differential d of a curve at the place P of the same curve.

`Divisor(d)`

The divisor $(f) + (dx)$ of the differential $d = f dx$ of a curve.

`Module(L)`

`IsBasis`	BoolElt	*Default :* `false`
`PreImages`	BoolElt	*Default :* `false`

Given a sequence L of differentials of a curve C, return the module over the base ring of C generated by the differentials in L as an abstract module and a map from the module into the space of differentials of C.

If the parameter `IsBasis` is set to `true` then the elements in L are assumed to be a basis for the module returned. If `PreImages` is set to `true` then a sequence of the preimages of the basis elements is also returned.

`Relations(L)`

`Relations(L, m)`

Given a sequence L of differentials of a curve C, return the module over the base ring R of C of R-linear relations between the elements of L. If given, the argument m is used to compute a generating system for the relation module such that the corresponding generating system of $\{ \sum_{i=1}^{m} v_i a_i \mid v = (v_i)_i \in V \}$ consists of "small" elements where a_i are the elements of L.

`Cartier(a)`

`Cartier(a, r)`

Given a differential a belonging to a curve C and a positive integer r, this function returns the result of applying the Cartier divisor to a r times (or just once if the argument r is omitted). More precisely, let C be a curve over the perfect field k with function field F, $x \in F$ be a separating variable and $a = g\,dx \in \Omega(C)$ with $g \in F$ be a differential. The Cartier operator is defined by

$$CA(a) = \left(-d^{p-1}g/dx^{p-1}\right)^{1/p} dx.$$

This function computes the r-th iterated application of CA to a.

`CartierRepresentation(C)`

`CartierRepresentation(C, r)`

Given a curve C and a positive integer r, this function determines a row representation matrix for action of the Cartier operator on a basis of the space of holomorphic differentials of C, (applied r times). More precisely, let C be a curve over the perfect field k, $\omega_1, \ldots, \omega_g \in \Omega(C)$ be a basis for the holomorphic differentials and $r \in \mathbf{Z}^{\geq 1}$. Let $M = (\lambda_{i,j})_{i,j} \in k^{g \times g}$ be the matrix such that

$$CA^r(\omega_i) = \sum_{m=1}^{g} \lambda_{i,m}\omega_m$$

for all $1 \leq i \leq g$. This function returns M and $(\omega_1, \ldots, \omega_g)$.

Example H114E26

In this example we create a curve known to have genus 3 (a nonsingular plane quartic). So it should have a three-dimensional space of holomorphic differentials.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^4+y^4+z^4);
> Omega_C,phi := SpaceOfHolomorphicDifferentials(C);
> Omega_C;
KModule of dimension 3 over Rational Field
> F<a,b> := FunctionField(C);
```

```
> phi;
Mapping from: ModFld: Omega_C to Space of differentials of F
```

That is good. Now we make a differential and check whether it is exact (which it obviously is since that's how we made it).

```
> f := a/b;
> df := Differential(f);
> df;
(-a/(b^6 + b^2)) d(b)
> Curve(df) eq C;
true
> IsExact(df);
true
```

114.9 Divisors

We work implicitly on the resolution of a particular, usually singular, model of a curve C that has been referred to as $\tilde{C}$. To handle prime divisors at the singularities properly we use the notion of *places* of the curve and devote the first section below to their construction. Places include the case of prime divisors at points of degree greater than 1, that is, points whose coordinates do not lie in the base ring. Following this are sections on constructions of divisors and their basic arithmetic. It may seem a little strange at first to distinguish places from other divisors, but in practice when doing arithmetic the difference is not noticeable.

The most important invariant associated with a divisor D is its *Riemann–Roch* space, often denoted by $L(D)$ or $H^0(D)$. This is a vector subspace of the function field of a curve. Its computation has an enormous number of applications. One we give as an illustration is the computation of the canonical embedding of C (in the case that C is non-hyperelliptic).

This section, together with Section 114.10, is devoted to MAGMA's facility to work with divisors on curves. A *divisor* on a nonsingular curve C is a formal sum of points $\Sigma n_i p_i$, where each $p_i \in C$ and each n_i is an integer. It is clear that divisors form a group under componentwise addition. This group, and various variants of it, is an important invariant of the curve C. For a singular (but still irreducible) curve, one can make a similar definition in which *points* are replaced by *places*, a notion that makes precise the vague idea that singularities arise in different ways, both by "gluing" nonsingular points together and by "pinching" tangent vectors at a particular point. The sections of this chapter contain very brief discussions which will help to orient the user who already knows something about divisors. For a more complete account one should consult an advanced textbook such as [Har77] Chapter II 6, or [Ful69] Chapter 8.

We start with a description of *prime divisors* or *places*. Then we show how to create divisor groups and divisors in them and go on to explain the basic arithmetic of divisors. All of the functions here are based on equivalent functions which apply to algebraic function fields. All calculations are done in that context using the functionality of Chapter 42.

A translation is made in the background for all functions described here. For information about function fields, their representations and translations see Section 114.8. The translations should not be explicitly needed in the ordinary course of working with curves and their divisors.

The main concern when working with divisors is with questions of linear equivalence. In this respect, one should at least have in mind that the most substantial calculations really make sense on projective curves. However, the functions below allow constructions using an affine curve and its points which are reinterpreted in terms of the projective closure.

Section 114.10 describes those functions which are related to linear equivalence of divisors. A divisor is called *principal* if it is linearly equivalent to the zero divisor. In the case of a curve defined over a finite field, we can compute the class group, that is, the group of divisors modulo principal divisors. In fact, we compute a finitely presented abelian group isomorphic to the class group and a map which transforms elements between the divisor group and the class group.

For any divisor on any curve, there are functions to compute Riemann-Roch spaces and a host of related entities.

114.9.1 Places

A *place* is a point of the resolution of a curve or, equivalently, a valuation of the function field of a curve. It is characterised by a point on the curve and, if that point is singular, some data that distinguishes a single resolved point above it.

114.9.1.1 Sets of Places

`Places(C)`

The set of places of the curve C.

`Curve(P)`

The curve on which the places in the set of places P lie. This will be a projective curve.

`P eq Q`

`P ne Q`

Returns `true` if and only if the sets of places P and Q are (not) equal.

114.9.1.2 Places

There are some explicit methods for constructing places. As we show later, places arise implicitly as components of divisors and this is a common way of getting hold of them.

`Places(C, m)`

A sequence of all places of degree m on C, a curve defined over a finite field, where m is a positive integer.

`HasPlace(C, m)`

`RandomPlace(C, m)`

If a place of degree m exists on the curve C over a finite field, then this returns `true` and a single such place, where m is a positive integer. Otherwise it returns `false`.

`Place(p)`

The place corresponding to the nonsingular point p of some curve.

`Places(p)`

A sequence containing the places corresponding to the point p of some curve.

`Place(C, I)`

Return the place of the curve C defined by the ideal I of the coordinate ring of C.

`WeierstrassPlaces(C)`

The Weierstrass places of the curve C, which are the Weierstrass places of the zero divisor of C.

`Place(Q)`

The place of a curve C determined by the sequence of elements of the sequence Q, which should all be contained in the function field of C.

`Ideal(P)`

Given a place P of a curve C return the prime ideal of the coordinate ring of the ambient of C of coordinate functions which vanish at the place P.

`TwoGenerators(P)`

Return two elements of the function field of P which determine the place P. The sequence containing these two elements can be used as input to `Place` to create a place equal to P.

Example H114E27

In this example we show how to use rational functions to create a place on a curve. This is not directly a very geometric operation. However, it is very useful since a pair of rational functions which determine a place form a very concise description of that place. Thus one often uses this method to recreate a given place on a curve in one step. We illustrate that by first finding a place and later recreating it from rational functions.

```
> P2<x,y,z> := ProjectivePlane(FiniteField(17));
> C := Curve(P2,x^5 + x^2*y^3 - z^5);
> p := C ! [1,0,1];
> Places(p);
[
    Place at (1 : 0 : 1)
]
```

```
> P := $1[1];
> P:Minimal;
Place at (1 : 0 : 1)
> TwoGenerators(P);
$.1 + 16
$.1^2*$.2
```

So now we have a place and some rational functions. As usual, these functions are elements of the function field of the curve C, so to be able to read them conveniently we assign names to that function field.

```
> FC<a,b> := FunctionField(C);
> TwoGenerators(P);
a + 16
a^2*b
```

We can use this sequence to recreate the place P. The real convenience of the first line of code below is that it could be in a different MAGMA session in which only the curve C has been defined together with the names a, b of its function field. (You can confirm this by running the four relevant lines in a separate MAGMA session.) The final line is simply to confirm that we really have created the same place P as we started with.

```
> Place([a+16,a^2*b]);
Place at (1 : 0 : 1)
> Place([a+16,a^2*b]) eq P;
true
> Place([a+16,a*b,a^2*b^2]) eq P;
true
```

Notice that in the final line we create exactly the same place using more than the two elements that `TwoGenerators` returned.

`Zeros(f)`

`Poles(f)`

A sequence of places of the curve C containing the zeros or poles of f where f is an element of the function field of C.

`Zeros(C, f)`

`Poles(C, f)`

A sequence of places of the curve C containing the zeros or poles of f where f is some function on C, i.e. f is coercible into the function field of C.

`CommonZeros(L)`

`CommonZeros(C, L)`

Given a sequence L of elements of the function field of some curve C or a curve C and a sequence L of functions on C, return the zeros which are common to all elements of L as places of C.

Example H114E28

The second argument to the intrinsic `Poles(C,f)` can be either an element of the function field of the curve C or an element of the function field of its ambient space.

```
> A<x,y> := AffineSpace(GF(2),2);
> C := Curve(A,x^8*y^3 + x^3*y^2 + y + 1);
> FA<X,Y> := FunctionField(A);
> FC<a,b> := FunctionField(C);
> Poles(C,X/Y);
[
    Place at (1 : 0 : 0)
]
> Poles(C,a/b);
[
    Place at (1 : 0 : 0)
]
> $1 eq $2;
true
```

In particular, we did not use the ambient coordinates x, y in the arguments.

`p1 + p2` `- p1` `p1 - p2` `k * p`
`p div k` `p mod k` `Quotrem(p1, k)`

`Curve(P)`

The projective curve on which the place P lies.

`RepresentativePoint(P)`

A representative point of the projective model of the curve underlying the place.

`P eq Q`
`P ne Q`

Returns `true` if and only if the two places are (not) the same.

`P in S` `P notin S`

`Valuation(f, P)`

The order of vanishing of the function f on the curve C at the place P of C. A negative value indicates a pole at P.

`Valuation(P)`

The valuation of the function field centred at the place P. This is a map from the function field to the integers.

`Valuation(a, P)`

The valuation of the differential a at the place P.

`Residue(a, P)`

The residue of the differential a at the degree 1 place P.

`UniformizingParameter(P)`

A function on the curve of the place P having valuation 1 at P.

`IsWeierstrassPlace(P)`
`IsWeierstrassPlace(D, P)`

Returns `true` if and only if the place P (which must have degree 1) is a Weierstrass place (of divisor D if given).

`ResidueClassField(P)`

The residue class field of the place P.

`Evaluate(a, P)`

Evaluate the element a in the function field of the curve of the place P, returning an element of the residue class field of P.

`Lift(a, P)`

Lift the element a of the residue class field of the place P (including infinity) to a function on the curve of P.

`Degree(P)`

The degree of the place P of a curve over the base ring of the curve of P.

`GapNumbers(C, P)`
`GapNumbers(P)`

The gap numbers of the curve C at the degree 1 place P.

`Parametrization(C, p)`
`Parametrization(C, p, P)`

Returns a map parametrizing the rational curve C at the rational point p or place p of degree 1. If p is a *singular* point on C, then it must have a unique place above it of degree 1. If P is also given it must be the projective line of dimension 1 as a curve (ie of type `Crv`) and then the domain of the map will be P.

114.9.2 Divisor Group

A curve has an associated group of divisors which is simply the formal abelian group generated by the places of the curve C. Divisors are elements of this group. In other words, divisors are expressions of the form $\Sigma n_i p_i$ where n_i are integers, p_i are places of the curve which one usually assumes to be distinct. Each term np is called a *summand* of d or the *component of d corresponding to p*. The integer n will be called the *coefficient* or *multiplicity* of the summand.

Divisors are created by specifying the curve for which they will a divisor (if that is not clear) and then giving sufficient data to identify precisely the divisor in question. This data could be a list of points or places together with integers, but there are many other creation methods. Divisors are printed as a linear combination of the places which support them, if such a combination is known. However, giving this information can be extremely expensive so often printing simply refers to the curve.

`DivisorGroup(C)`

The group of divisors of the curve C. This curve may be either an affine or projective curve.

`Curve(Div)`

The curve that was used to create the divisor group Div, or its projective model.

`Div1 eq Div2`

`Div1 ne Div2`

Returns `true` if and only the divisor groups $Div1$ and $Div2$ are (not) equal.

114.9.3 Creation of Divisors

`DivisorGroup(D)`

The divisor group in which the divisor D lies.

`Curve(D)`

The (projective) curve on which the divisor D lies.

`Identity(D)`

`Id(D)`

`D ! 0`

The zero divisor of the divisor group D of a curve.

`Div ! p`

`Divisor(p)`

The prime divisor in the divisor group Div of the curve C corresponding to the place or nonsingular point p of some curve C.

`Divisor(D, S)`

`Divisor(C, S)`

`Divisor(S)`

The divisor of the curve C or the curve of the divisor group D described by the factorization sequence S. The sequence should contain tuples of the form `<place,integer>`.

Example H114E29

One can use the intrinsic `Divisor(S)` to reconstruct a divisor from concise data related to it. The intrinsics `Support` and `CanonicalDivisor` are defined below.

```
> P<x,y,z> := ProjectivePlane(FiniteField(17));
> C := Curve(P,x^5 + x^2*y^3 - z^5);
> F<a,b> := FunctionField(C);
> K := CanonicalDivisor(C);
> supp, exps := Support(K);
> Q := [ < RationalFunctions(supp[i]),exps[i] > : i in [1..#supp] ];
> Q;
[
    <[ a, 2*a^2*b^2 + 4*a*b ], 2>,
    <[ a + 16, a^2*b ], 2>,
    <[ a^4 + a^3 + a^2 + a + 1, a^2*b ], 2>,
    <[ 1/a, (a + b)/a ], -2>,
    <[ 1/a, (a^2 + 16*a*b + b^2)/a^2 ], -2>
]
> K;
Divisor 2*Place at (0 : 1 : 0) + 2*Place at (1 : 0 : 1) + 2*Place at
($.1 : 0 : 1) - 2*Place at (16 : 1 : 0) - 2*Place at (16*$.1 + 1 : 1 : 0)
```

Now we can reconstruct the divisor K using this sequence.

```
> Divisor([<Place(f[1]), f[2]> : f in Q]);
Divisor on Curve over GF(17) defined by
x^5 + x^2*y^3 + 16*z^5
> K eq $1;
true
```

`PrincipalDivisor(C, f)`
`PrincipalDivisor(D, f)`
`PrincipalDivisor(f)`
`Divisor(C, f)`
`Divisor(D, f)`
`Divisor(f)`

The principal divisor corresponding to f, that is, the divisor of the curve C of zeros and poles of the function field element f, where C is the curve of the divisor group D if given.

`Divisor(a)`

The divisor of the curve C corresponding to the differential a of C.

`Divisor(C, X)`
`Divisor(D, X)`

The divisor described by intersection of the curve C with the scheme X, (where C is the curve of the divisor group D if the group is given instead of the curve).

`Divisor(C, p, q)`
`Divisor(D, p, q)`

The principal divisor corresponding to the line through points p and q (the tangent line to the curve C there if they coincide) where C is the curve of the divisor group D if given.

`Divisor(C, I)`
`Divisor(D, I)`

The divisor of the curve C defined by the ideal I of the ambient coordinate ring where C is the curve of the divisor group D if given.

`Decomposition(D)`

The decomposition sequence of D as a sequence of tuples of the form `<place, multiplicity>` characterizing the divisor D.

`Support(D)`

The sequence of places in the support of D, followed by their sequence of multiplicities in D.

Example H114E30

A curve, its divisor group and some divisors are created.

```
> P<x,y,z> := ProjectiveSpace(GF(7), 2);
> C := Curve(P,y^2*z - x^3 - x*z^2 - z^3);
> Div := DivisorGroup(C);
> Div;
Group of divisors of Curve over GF(7) defined by
6*x^3 + 6*x*z^2 + y^2*z + 6*z^3
> FP<a,b> := FunctionField(P);
> D := Divisor(C,a);
> D;
Divisor of Curve over GF(7) defined by
6*x^3 + 6*x*z^2 + y^2*z + 6*z^3
> Decomposition(D);
[
    <Place at (0 : 1 : 0), -2>,
    <Place at (0 : 6 : 1), 1>,
    <Place at (0 : 1 : 1), 1>
]
> D;
Divisor -2*Place at (0 : 1 : 0) + 1*Place at (0 : 6 : 1) + 1*Place at (0 : 1 : 1)
```

The support of a divisor is written in the style of the factorization of other objects in MAGMA; compare with the factorization of the integer 84 below. This expression is called the *factorization* of a divisor and provides a method of accessing the individual components.

```
> Factorization(84);
[ <2, 2>, <3, 1>, <7, 1> ]
> Support(D)[2];
Place at (0 : 6 : 1)
```

One can access the point underlying a given place.

```
> p := Support(D)[1];
> p;
Place at (0 : 1 : 0)
> RepresentativePoint(p);
(0 : 1 : 0)
```

`CanonicalDivisor(C)`

A divisor in the canonical divisor class of the curve C.

`RamificationDivisor(C)`

The ramification divisor of the curve C.

114.9.4 Arithmetic of Divisors

`D + E` `- D` `D - E` `n * D` `D div n` `D mod n`

Basic formal arithmetic of divisors; D and E are divisors (or places) and n is an integer.

`Quotrem(D, n)`

The quotient and remainder on dividing the divisor D by the integer n.

`Degree(D)`

The sum of coefficients of the divisor D multiplied by the degrees of the places of the corresponding components.

`IsEffective(D)`
`IsPositive(D)`

Returns `true` if and only if all coefficients of the divisor D are nonnegative.

`Numerator(D)`
`Denominator(D)`

The numerator, respectively denominator, of the divisor D of a curve. The numerator and denominator are both positive divisors such that D is the difference between numerator and denominator.

`SignDecomposition(D)`

The minimal effective divisors A and B such that the equality of divisors $D = A - B$ holds.

Example H114E31

The sign decomposition of the previous example is calculated.

```
> P<x,y,z> := ProjectiveSpace(GF(7),2);
> C := Curve(P,y^2*z - x^3 - x*z^2 - z^3);
> Div := DivisorGroup(C);
> phi := hom< Parent(x/z) -> FP | [FP.1,FP.2,1] >
>             where FP is FunctionField(P);
> D := Divisor(Div,phi(x/z));
> Decomposition(D);
[
    <Place at (0 : 1 : 0), -2>,
    <Place at (0 : 6 : 1), 1>,
    <Place at (0 : 1 : 1), 1>
]
> Decomposition(D div 2);
[
    <Place at (0 : 1 : 0), -1>
```

```
]
> A, B := SignDecomposition(D);
> IsEffective(A);
true
> IsEffective(B);
true
> A - B eq D;
true
```

`d in D` `d notin D`

`D eq E`
`D ne E`

Returns `true` if and only the divisors D and E are (not) equal. Note that this means equality in the group of divisors and is not the same as being linearly equivalent.

`D lt E` `D le E` `D gt E` `D ge E`

`IsZero(D)`

Returns `true` if and only if all coefficients of the divisor D are zero.

`IsCanonical(D)`

Returns `true` if and only if the divisor D is the divisor of a differential, in which case also return a differential realising this.

`GCD(D1, D2)`
`Gcd(D1, D2)`
`GreatestCommonDivisor(D1, D2)`

The greatest common divisor of the divisors $D1$ and $D2$. This is the divisor supported on the places common to the support of both divisors with coefficients the minimum of those occurring in $D1$ and $D2$.

`LCM(D1, D2)`
`Lcm(D1, D2)`
`LeastCommonMultiple(D1, D2)`

The least common multiple of the divisors $D1$ and $D2$. This is the divisor supported on all the places in the supports of $D1$ and $D2$ with coefficients the maximum of those occurring in the input divisors.

Example H114E32

We find that a given divisor is actually a canonical divisor.

```
> P<x,y,z> := ProjectiveSpace(GF(7),2);
> C := Curve(P,y^2*z - x^3 - x*z^2 - z^3);
> Div := DivisorGroup(C);
> phi := hom< Parent(x/z) -> FP | [FP.1,FP.2,1] >
>               where FP is FunctionField(P);
> D := Divisor(Div,phi(x/z));
> IsCanonical(D);
true (($.1) ^ 1 * ($.1^3 + $.1 + 1) ^ -1 * ($.1) ^ 1) d($.1)
```

The printing of the differential in the last line above is not very clear since names have not been assigned, but nonetheless, it can be used as an argument to intrinsics.

```
> _, dd := IsCanonical(D);
> Valuation(dd,Support(D)[1]);
-2
```

114.9.5 Other Operations on Divisors

`Ideal(D)`

Given a divisor D of a curve C, return the ideal of the coordinate ring of the ambient of C of coordinate functions which cuts out D.

`Valuation(D,p)`

`Valuation(D,P)`

The coefficient of the divisor summand of the divisor D corresponding to the point p or place P.

`ComplementaryDivisor(D,p)`

`ComplementaryDivisor(D,P)`

The divisor after removing from the divisor D the component corresponding to the point p or place P.

114.10 Linear Equivalence of Divisors

114.10.1 Linear Equivalence and Class Group

`IsPrincipal(D)`

Returns `true` if and only if the divisor D is the divisor of zeros and poles of some rational function. A rational function which performs that role will also be returned. Recall that any two such functions differ only by a scalar factor.

`IsLinearlyEquivalent(D1,D2)`

Returns `true` if and only if the difference $D1 - D2$ of the two divisor arguments is a principal divisor. In that case return also the rational function giving this equivalence.

`IsHypersurfaceDivisor(D)`

For an effective hypersurface divisor D on an ordinary projective curve C, returns `true` if and only if D is the scheme theoretic intersection of C with a hypersurface H of the ambient projective space. If so, also returns an equation for H and the degree of H.

Example H114E33

We check that an effective canonical divisor on a non-singular degree 4 plane curve is indeed a divisor coming from the intersection of the curve with a hyperplane.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^3*y+y^3*z+z^3*x);
> D0 := CanonicalDivisor(C);
> f := Basis(D0)[1];
> D := D0+PrincipalDivisor(f);
> IsEffective(D);
true
> IsHypersurfaceDivisor(D);
true -2/9*x
1
```

`ClassGroup(C)`

The *divisor class group*, or simply *class group* of a curve is the group of divisors modulo principal divisors. If C is a curve defined over a finite field, then this function returns an abelian group isomorphic to its divisor class group, a map of representatives from the class group to the divisor group and the homomorphism from the divisor group onto the class group. The second map has an inverse, so translation between the group and the divisors can be achieved using only this map.

`ClassNumber(C)`

The order of the class group of the curve C.

`GlobalUnitGroup(C)`

The group of global units of the function field of the curve C, that is the multiplicative group of the field of geometric irreducibility of C as an abelian group together with the map into the function field of C.

Example H114E34

We compute the class group of a curve defined over a finite field. The calculation takes a few seconds.

```
> A<x,y> := AffineSpace(GF(2,5),2);
> C := Curve(A,x^7 + x^4*y^3 + x*y^2 + y);
> Genus(C);
3
> Cl, _, phi := ClassGroup(C);
> Cl;
Abelian Group isomorphic to Z/26425 + Z
Defined on 2 generators
Relations:
    26425*Cl.1 = 0
```

We can use the map ϕ to pull elements of the abelian group back to the divisor group. For curves over finite fields, this is one way of constructing interesting divisors.

```
> Div := DivisorGroup(C);
> Div eq Domain(phi);
true
> D := Cl.1 @@ phi;
> D;
Divisor of Curve over GF(2^5) defined by
x^7 + x^4*y^3 + x*y^2 + y
```

The unpleasant printing with the dollar signs is happening because no names have yet been assigned to the projective closure of C. Notice that the printing is rather uninformative. This is because a factorization of d is not yet known and could be an extremely expensive computation.

```
> Decomposition(D);
[
    <Place at (0 : 0 : 1), -3>,
    <Place at ($.1 : $.1^10*$.1^2 + $.1^26*$.1 + $.1^22 : 1), 1>
]
> Degree(D);
0
```

The degree function simply returns the sum of the integer valuations in the factorization. Those valuations can be seen as the second entry of each tuple in the support.

`ClassGroupAbelianInvariants(C)`

The abelian invariants of the curve C.

`ClassGroupPRank(C)`

The p-rank of the class group of C.

`HasseWittInvariant(C)`

The Hasse–Witt invariant of the curve C.

114.10.2 Riemann–Roch Spaces

If D is a divisor on C then colloquially speaking the Riemann–Roch space $L(D)$ is the finite dimensional vector subspace of the function field of C consisting of functions with poles no worse that D (and at least as many zeros as the negative part of D if D is not effective). To be precise, we say that

$$L(D) = \{f \in k(C) \mid \mathrm{div}(f) + D \geq 0\}$$

where $\mathrm{div}(f)$ is the principal divisor of zeros and poles of f and the condition ≥ 0 is simply shorthand to mean that the left-hand side is an effective divisor.

For any divisor on a projective curve defined over a field this vector space is finite dimensional. Its dimension $\ell(D)$ appears in the Riemann–Roch formula

$$\ell(D) - \ell(K_C - D) = \deg(D) + 1 - g$$

where K_C is the canonical divisor and g is the genus of C.

The space of *effective* divisors linearly equivalent to D, the complete linear system of D, is the projectivisation of the Riemann–Roch space $L(D)$. This can be used to create a map from the curve C to a projective space of dimension $\ell(D) - 1$.

`Reduction(D)`

`Reduction(D, A)`

Let D be a divisor. Denote the result of both functions by $\tilde{D}$, r, A and a (for the second function the input A always equals the output A). A has (must have) positive degree and the following holds:

(i) $D = \tilde{D} + rA - (a)$,

(ii) $\tilde{D} \geq 0$ and $\deg(\tilde{D}) < g + \deg(A)$ (over the field of geometric irreducibility),

(iii) $\tilde{D}$ has minimal degree among all such divisors satisfying (i), (ii).

`RiemannRochSpace(D)`

A vector space V and an isomorphism from V to the Riemann–Roch space of D in the function field of the curve on which the divisor D lies.

`Basis(D)`

A sequence containing a basis of the Riemann-Roch space $L(D)$ of the divisor D.

`ShortBasis(D)`

A sequence containing a basis of the Riemann-Roch space $L(D)$ of the divisor D in short form.

`Dimension(D)`

The dimension $\ell(d)$ of the Riemann-Roch space of the divisor D.

`DifferentialSpace(D)`

Returns a vector space V and a map from V to the space of differentials of the curve C containing the divisor D with image the differentials of $\omega_C(D)$. Colloquially, this means the differentials whose zeros are at least the positive (or effective) part of D and whose poles are no worse than the negative part of D.

`DifferentialBasis(D)`

A basis of the space of differentials of the divisor D.

`IndexOfSpeciality(D)`

The index of speciality of the divisor D, that is the dimension $\ell(K_C - D)$ appearing in the Riemann–Roch formula.

`IsSpecial(D)`

Returns `true` if and only if the divisor D is special.

`GapNumbers(D)`
`GapNumbers(D,p)`

The gap numbers of the divisor D, at the place p if included.

`GapNumbers(p)`

The gap numbers of the nonsingular point p.

`WeierstrassPlaces(D)`
`WeierstrassPoints(D)`

A sequence containing the Weierstrass places (or underlying points) of the divisor D.

`WronskianOrders(D)`

The Wronskian orders of the divisor D.

`RamificationDivisor(D)`

The ramification divisor of the divisor D.

`DivisorMap(D)`

`DivisorMap(D,P)`

A map from the curve of the divisor D to the projective space P (which will be created in the background if not given as an argument). The dimension of P must be $\ell(D) - 1$.

`CanonicalMap(C)`

`CanonicalMap(C,P)`

The canonical map from the curve C to the projective space P (which will be created in the background if not given as an argument). This is the map determined by (a basis of the) Riemann–Roch space of the canonical divisor K_C. In particular, the dimension of P must be $\ell(K_C) - 1 = g - 1$.

`CanonicalImage(C, phi)`

`CanonicalImage(C, eqns)`

These functions compute the canonical image of a projective curve C of genus at least 2 much more efficiently than the generic image machinery applied to the canonical map. In the first function, the second argument should be the canonical map and in the alternative it should be a sequence of polynomials defining the canonical map.

A second return value is a boolean which is true if and only if the image is a rational curve or equivalently if and only if C is a hyperelliptic curve. Note that these functions may also be used for computing the rational normal curve image of a genus 0 curve C under the map given by any (non-trivial) complete linear system.

Example H114E35

We first make a curve and compute its genus.

```
> P2<X,Y,Z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P2,X^7 + X^3*Y^2*Z^2 + Z^7);
> Genus(C);
3
```

If C is not hyperelliptic then its canonical map will embed it as a nonsingular quartic curve in the projective plane. We make the canonical map. We even include the plane $P2$ that we have already created as an argument so that it will be set as the codomain of the map.

```
> phi := CanonicalMap(C,P2);
> phi;
Mapping from: Prj: P2 to Prj: P2
with equations :
X^2 X*Z Z^2
> phi(C);
```

```
Curve over Rational Field defined by -X*Z + Y^2
```

That curve is certainly not a plane quartic. Indeed, it is evidently a rational curve, so C must have been hyperelliptic.

```
> D := phi(C);
> Genus(D);
0
```

The bicanonical map will embed C since its genus is strictly bigger than 2.

```
> D := 2 * CanonicalDivisor(C);
> phi2 := DivisorMap(D);
> Dimension(Codomain(phi2));
5
> P5<a,b,c,d,e,f> := Codomain(phi2);
> phi2(C);
Scheme over Rational Field defined by
a^2 + b^2 + e*f
-b*d + c^2
-b*e + c*d
-b*f + d^2
-b*f + c*e
-c*f + d*e
-d*f + e^2
> Dimension(phi2(C));
1
> IsNonsingular(phi2(C));
true
```

If you are familiar with the equations of rational normal curves, you will recognise this as a quadric section of a standard scroll—the cone on the rational normal curve of degree 4—which misses the vertex of the cone. This is exactly what gives the curve its hyperelliptic structure. One could go on to make a ruled surface with a map to $\mathbf{P}^5$ with this scroll as its image and pull back the curve to the ruled surface on which the ruling cuts out the degree 2 linear system of the hyperelliptic curve. This is carried out for a particular trigonal curve later in Section 114.11.1.

114.10.3 Index Calculus

MAGMA contains functionality to compute discrete logarithms in the degree 0 divisor class group of plane curves over finite fields. The algorithm used for this is Diem's version of index calculus [Die06], which looks for relations by considering lines through the points of the factor base. In order to use this algorithm, the group order must be know and given as input.

As is always the case with index calculus, the algorithm consists of two main stages: the sieving stage, during which relations are obtained and stored in a matrix, and the linear algebra stage, during which a non-trivial element of the kernel of this matrix is computed. It is possible to only do the sieving and compute the matrix. For this it is not necessary

to know the group order. This can be used to obtain information on how fast sieving can be done. Generally, the linear algebra stage will be the bottle neck for larger examples.

The running time of the algorithm mainly depends on the degree of the curve. In [Die06] it is explained how one can obtain a model of a given curve with relatively low degree. This can greatly speed up the running time of the algorithm.

The algorithm will not work if the field is too small, as there will not be enough lines to produce relations in that case. One requires that at least $q \geq d!$ for a degree d curve over $GF(q)$.

Elements in the degree 0 class group can be represented by divisors of degree 0. But we also allow divisors of non-zero degree D as input, together with a fixed divisor $D0$ of degree 1. This will be interpreted as representing the divisor class $D - \deg(D)D0$. Note that once $D0$ is fixed, and D is effective of minimal degree, then D is uniquely determined by the divisor class. Starting out with an arbitrary divisor E, this unique D can be computed with the `Reduction(E,D0)` command.

`IndexCalculus(D1, D2, D0, np)`

`IndexCalculus(D1, D2, D0, np, n, rr)`

Compute the discrete logarithm of $D2 - \deg(D2)D0$ with base $D1 - \deg(D1)D0$. The group order np must be given. Optionally, one can also give n as approximate size of the factor base to be used, and rr as the number of required relations.

`IndexCalculusMatrix(D1, D2, D0, n, rr)`

Find the sparse matrix with relations found in the sieving stage. Input is the same as for `IndexCalculus`, except that the group order is not given. Outputs M, pos, fb, Da, Db, a, b, where M is the sparse relation matrix. Da and Db are divisors that split over the factor base, and $Da - \deg(Da)D0$ and $Db - \deg(Db)D0$ are linearly equivalent to $a(D1 - \deg(D1)D0)$ and $b(D2 - \deg(D2)D0)$, respectively. fb is a sequence of points that contains the support of Da, Db and $D0$. pos is a sequence of integers that indicates the position of the points of fb in the factor base. The last two rows of M correspond to the divisors $Da - \deg(Da)D0$ and $Db - \deg(Db)D0$.

`MultiplyDivisor(n, D , D0)`

Effective divisor E of minimal degree such that $E - \deg(E)D0$ is equivalent to $n(D - \deg(\mathrm{D})D0)$.

Example H114E36

In this example we compute a discrete logarithm in the class group of a curve of degree 6. The curve is over $GF(2^{13})$, but it is a base change of a curve over $GF(2)$

```
> A2<x,y>:=AffinePlane(GF(2));
> C1:=ProjectiveClosure(Curve(A2,x^5*y + x*y^2 + y^6 + y + 1));
> L:=LPolynomial(C1);
> Evaluate(L,1);
```

```
752
> K<z>:=CyclotomicField(13);
> np:=Numerator(Norm(Evaluate(L,z)));
> Factorisation(np);
[ <175310448404461045718069548360655837, 1> ]
```

So the class group of $C1$ has order 752 over $GF(2)$, and is has order $752np$ over $GF(2^{13})$, where np is a large prime. Now we create some divisors, and make sure that $D2 - \deg(D2)D0$ is in the subgroup generated by $D1 - \deg(D1)D0$.

```
> F13<u>:=GF(2^13);
> C13:=ChangeRing(C1,F13);
> D1:=Divisor(C13![u^4758,u^3]);
> D2:=752*Divisor(C13![u^1325,u^6]);
> D0:=Divisor(C13![u^2456,u^11]);
```

In order to get an idea what is going on, we set the verbose flag, and we perform the discrete logarithm computation. In the end, we verify the result.

```
> SetVerbose("CurveIndexcal",1);
> time dl:=IndexCalculus(D1,D2,D0,752*np);
Try to find factor base of size 5087
Try to find 5138 relations.
First input divisor already splits.
Second input divisor already splits.
First index calculus input: Divisor 1*Place at (u^4758 : u^3 : 1)
Multiple of original input: 1
Second index calculus input: Divisor 752*Place at (u^1325 : u^6 : 1)
Multiple of original input: 1
Sieving
Number of relations found: 5138
Number of elements in factor base: 5089
Find element of kernel of matrix
Found kernel element
DLP mod 175310448404461045718069548360655837:
125220860410617991206452745026979 42
Time: 164.900
> MultiplyDivisor(dl,D1,D0) eq Reduction(D2,D0);
true
```

114.11 Advanced Examples

These examples are intended to demonstrate basic programming in MAGMA using the functions of this chapter together with a few from Chapter 112. There is little or no explanation of the geometry behind the examples. We assume here that you are familiar with that and are really interested in the problem of realising it in MAGMA.

114.11.1 Trigonal Curves

We discuss a particular example covered by Petri's theorem. In fact, we write down a curve C of genus 8 which is trigonal. We discover this easily since its canonical embedding is not cut out by quadrics. It would be nice to have automatic functions to recognise the equations of the surface scroll cut out by the quadrics, but at the moment they don't exist so we have to make that calculation by hand.

Example H114E37

First we make the curve and compute its canonical model.

```
> k := Rationals();
> P<X,Y,Z> := ProjectiveSpace(k,2);
> C := Curve(P,X^8 + X^4*Y^3*Z + Z^8);
> Genus(C);
8
> phi := CanonicalMap(C);
> P7<a,b,c,d,e,f,g,h> := Codomain(phi);
> CC := phi(C);
> CC;
a^3*e + d^4 + d^2*h^2
a^3*f + d^3*e + d*e*h^2
a^3*g + d^3*f + d*f*h^2
a^3*h + d^3*g + d*g*h^2
a^2*b + d^3 + d*h^2
a^2*c + d^2*e + e*h^2
a*b*c + d^2*f + f*h^2
a*c^2 + d^2*g + g*h^2
b*c^2 + d^2*h + h^3
-a*c + b^2  -a*e + b*d  -a*f + c*d  -a*f + b*e
-a*g + c*e  -d*f + e^2  -a*g + b*f  -a*h + c*f
-d*g + e*f  -d*h + f^2  -a*h + b*g  -b*h + c*g
-d*h + e*g  -e*h + f*g  -f*h + g^2
```

In this example, we can see all the quadrics which cut out the canonical model CC. But even if we could not, or if computing the full canonical ideal was too difficult, we can compute the conics in the canonical ideal separately using only linear algebra.

```
> SC := Image(phi,C,2);
> SC;
a*c - b^2  a*e - b*d  a*f - c*d  a*g - c*e
```

```
a*h - c*f  b*e - c*d  b*f - c*e  b*g - c*f
b*h - c*g  d*f - e^2  d*g - e*f  d*h - f^2
e*g - f^2  e*h - f*g  f*h - g^2
> Dimension(SC);
2
```

We would like to identify the scroll SC. Even better, we would like to find a map from a ruled surface to this scroll and pull the image curve CC back to this ruled surface. Then the fibres of the ruling will cut out the g^1_3 on C giving its trigonal structure. We will also see the Maroni invariant of C directly. In this case one immediately recognises the equations of the scroll so can write down the ruled surface and choose the bidegree of linear system which gives the map to the scroll.

```
> F<r,s,u,v> := RuledSurface(k,2,4);
> psi := map< F -> P7 | [ m : m in MonomialsOfWeightedDegree(F,[0,1]) ] >;
> SF := psi(F);
> DefiningIdeal(SF) eq DefiningIdeal(SC);
true
```

To realise the curve's trigonal structure, we need to create a divisor by intersecting it with a fibre of the ruling. The natural way would be to pull the curve back to F via psi and work there. However, MAGMA currently cannot create divisors that lie on curves on scrolls.
Luckily, we *can* work with the image `CC` in `P7` and obtain the divisor `D` of the image of a fibre under `psi` intersected with `CC` for our g^1_3. This is then used to define a $3-1$ map to the projective line.

```
> fib := psi(Scheme(F,r));
> Dimension(fib);
1
> D := Divisor(CC,fib);
> Degree(D);
3
> #Basis(D);
2
```

So `D` really does give us a g^1_3. To get the map to P^1, we can pull it back to `C`, but it is faster to compose the divisor map on `CC` with `phi`. Proceeding this way, it is then a good idea to use the function field of `C` to simplify the map description.

```
> phiD := DivisorMap(D);
> mpD := Expand(Restriction(phi,C,CC)*phiD);
> FC := FunctionField(C);
> rat := FC!(p1/p2) where p1,p2 := Explode(DefiningPolynomials(mpD));
> mpD := map<C->Codomain(mpD)|[rat,1]>; mpD;
Mapping from: CrvPln: C to Projective Space of dimension 1
Variables : $.1, $.2
with equations :
X^7 + X^3*Y^3*Z
Z^7
```

114.11.2 Algebraic Geometric Codes

MAGMA includes functions for working with codes which arise from algebraic geometry. Discussion of these functions is left to the chapter on error-correcting linear codes, Chapter 152. As is well known, these codes are often created using Riemann–Roch spaces of divisors on curves. Here we demonstrate the creation of such a code taken from the book of van Lint and van der Geer [vLvdG88]. This is a famous example which arises many times in that book, as Example II (3.12) for instance.

Example H114E38

This code is based on the Klein quartic over the finite field $F8$ of 8 elements, a curve that we define immediately. Notice that we start by defining a curve C over the field of 2 elements. This is so that we can investigate C over small fields while still being able to work over $F8$ later.

```
> F2 := FiniteField(2);
> F4<t4> := FiniteField(4);
> F8<t8> := FiniteField(8);
> P<x,y,z> := ProjectiveSpace(F2,2);
> C := Curve(P,x^3*y + y^3*z + z^3*x);
> C8<a,b,c> := BaseChange(C,F8);
> C8;
Curve over GF(2^3) defined by
a^3*b + a*c^3 + b^3*c
```

The code will have length 24, corresponding to the 24 rational points of C.

```
> #RationalPoints(C8);
24
```

In constructing such codes, one must have a collection of points, in this case the 24 rational points we have just found, and a divisor whose support is disjoint from these points. As the divisor, we take some multiple of a conjugate pair of points defined over the finite field of 4 elements. In MAGMA, it is convenient to use the function field machinery to describe this as a place of degree 2. It is constructed as the intersection of C with one of its bitangents.

```
> L := Curve(P,x+y+z);
> IntersectionPoints(C,L);
{@ @}
> C4 := BaseChange(C,FiniteField(4));
> P4 := Ambient(C4);
> L4 := BaseChange(L,P4);
> IntersectionPoints(C4,L4);
{@ (t4 : t4^2 : 1), (t4^2 : t4 : 1) @}
> [ IntersectionNumber(C4,L4,p) : p in $1 ];
[ 2, 2 ]
```

So we see that L is indeed a bitangent of C and that the two points of intersection are defined properly over the finite field of 4 elements. In order to be able to compute the Riemann-Roch

space of a divisor, we compute this intersection as a place on $C8$. Notice how the support of the divisor D is not used in its display until it has been explicitly computed.

```
> FP8<X,Y> := FunctionField(Ambient(C8));
> Div := DivisorGroup(C8);
> D := Divisor(Div,X + Y + 1);
> D;
Divisor on Curve over GF(2^3) defined by
a^3*b + a*c^3 + b^3*c
> Decomposition(D);
[
    <Place at (0 : 1 : 0), -3>,
    <Place at (1 : 0 : 0), -1>,
    <Place at ($.1 + 1 : $.1 : 1), 2>
]
> D;
Divisor -3*Place at (0 : 1 : 0) + 2*Place at ($.1 + 1 : $.1 : 1) -
1*Place at (1 : 0 : 0)
```

This divisor contains the place we require with multiplicity 2 and some negative contributions at infinity. We extract the degree 2 place we need.

```
> p := Support(D)[2];
> Degree(p);
2
```

Now we can make the code. Rather than using the rational points of $C8$, the code creation intrinsic takes a sequence of degree 1 places. But they are also easily retrieved. The divisor we use will be $3p$.

```
> G := 3*p;
> S := Places(C8,1);
> #S;
24
> AGC := AlgebraicGeometricCode(S,G);
> Dimension(AGC);
4
```

In fact, it is the dual code that is really wanted, but this is easy.

```
> Dual(AGC);
[24, 20] Linear Code over GF(2^3)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^5 t8 t8^5 t8^3]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^5 t8^2 t8^6 t8^5]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^3 t8^6 t8^5]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 t8^6 t8^3 t8^4]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^2 t8^3 t8 t8^2]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^6 t8^6 t8^2 t8^6]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 t8^2 t8^4 t8^4 t8^6]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 t8^6 t8^3 t8^3 t8^2]
```

```
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 t8^4 0 t8^4]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 t8^4 t8^3 t8^6 1]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 t8 t8^2 t8^2 t8^3]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 t8^6 t8^6 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 t8^2 t8^4 t8^3 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 t8^6 t8 t8^2 t8]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 t8^5 t8 t8^6]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 t8 1 1 t8^3]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 t8^3 1 t8^5 t8^2]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 t8^2 t8^2 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 t8^3 t8 t8^5 t8^5]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 t8^4 t8^3 t8 t8^4]
```

114.12 Curves over Global Fields

114.12.1 Finding Rational Points

There are also various point search routines for more general schemes.

`PointsCubicModel(C, B :` *parameters*`)`

`OnlyOne`	BoolElt	*Default :* `false`
`ExactBound`	BoolElt	*Default :* `false`
Verbose	`CubicModelSearch`	*Maximum :* 1

Given a plane cubic over the rationals, this function searches, by a reasonably efficient method due to Elkies [Elk00], for a point on C of naïve height up to B — the asymptotic running time is $O(B)$.

If `OnlyOne` is set to `true`, the search stops as soon as it finds one point; however, the algorithm is p-adic and there is no guarantee that points with small co-ordinates in $\mathbf{Z}$ will be found first. If `ExactBound` is set to `true`, then points that are found with height larger than B will be ignored.

Example H114E39

```
> P<x,y,z>:=ProjectiveSpace(Rationals(),2);
> C:=Curve(P,x^3+9*y^3+73*z^3);
> time PointsCubicModel(C,10^4);
[ (31/2 : -15/2 : 1), (-353/824 : -1655/824 : 1), (-463/106 : 111/106
: 1), (-1 : -2 : 1), (-2347/526 : 635/526 : 1), (-206/5 : 99/5 : 1),
(22/5 : -13/5 : 1), (-43/16 : -29/16 : 1), (-25 : 12 : 1), (-4493/1076
: -299/1076 : 1), (-215/47 : 64/47 : 1), (-631/151 : -22/151 : 1), (-4
: -1 : 1), (278/193 : -393/193 : 1), (-328/55 : 137/55 : 1), (311/88 :
-207/88 : 1), (145/71 : -148/71 : 1) ]
Time: 2.790
```

114.12.2 Regular Models of Arithmetic Surfaces

Let F be a global field (the rationals, a number field or a function field) with ring of integers O_F. Given a curve C over F with integral defining equations, the associated *arithmetic surface* is the scheme of relative dimension 1 over SpecO_F defined by the same equations. In MAGMA, one can compute a regular model of the arithmetic surface locally at a given prime of O_F, and extract information from it. The 'model' is a data structure which contains several affine patches with maps between them, as well as the components of the special fibre, and other data. This raw data is quite bulky; the interesting information is accessed via the functions described in this section. This functionality may be expanded (on request) in later versions.

The MAGMA category for these data structures is `CrvRegModel`.

Caveat: in the initial implementation there are some restrictions on which curves, and which fields, can be handled. These restrictions are not documented here, and may be lifted in the near future.

114.12.2.1 Creation of Regular Models

`RegularModel(C, P)`

Verbose `RegularModel` *Maximum :* 2

This computes a regular model of the curve C at the prime P. Here C is a curve over a field F (the rationals, a number field or a univariate rational function field), and P is a prime of the maximal order O_F of F (given as an element or as an ideal). The defining equations of C must have integral coefficients, and the reduction of C modulo P must have dimension 1.

The function returns a model of C: this consists of several affine patches, given by integral equations, which together describe a scheme over O_F whose generic fibre is isomorphic to C. (The gluing maps, and the isomorphism to C, are part of the stored data). The model is regular on the special fibre above P. However it is not necessarily a minimal model.

In some cases, the function may replace F by a finite extension L/F in which P is unramified, and return a regular model for the base change C_L. (This occurs when there is a non-regular point or component in the special fibre that is not geometrically irreducible over the residue field.) When this occurs, a warning message is printed.

114.12.2.2 Using Regular Models

`IntersectionMatrix(M)`

Given a regular model, this returns a matrix whose entries are the intersection multiplicities of the (reduced, irreducible) components of the special fibre. Secondly, it returns a sequence giving the multiplicities of the components. (The components of the model always come in the same order.)

`ComponentGroup(M)`

Given a regular model of a curve C at a prime P, this returns (as an abstract abelian group) the group of components of the Neron model of the Jacobian of C over the completion at P. (This is computed from the `IntersectionMatrix` of the model.)

`PointOnRegularModel(M, x)`

Given a regular model of a curve C over a global field F, and a point $x \in C(F)$, this lifts x to a point on the regular model. More precisely, it finds a patch of the model, and a point on the generic fibre of that patch which maps to x (under the isomorphism between the generic fibres of M and C).

Three objects are returned: a sequence giving coordinates of the point, a sequence containing the equations of the relevant patch, and (for internal use) the 'index' of the patch.

114.12.3 Minimization and Reduction

Minimization and reduction is a search for a linear transformation, that leads to nice equations. The general strategy is described in Section 116.4.4.

Here, we describe the routines for the minimization of plane quartic curves and the reduction for plane curves of degree at least 3. The reduction is done by constructing a cluster of special points on the curve. Thus, we start with this.

`ReduceCluster(X)`

`eps`	FldReElt	*Default :* 1e-6
`c`	FldReElt	*Default :* 1
Verbose	`ClusterReduction`	*Maximum :* 3

Here X is a sequence of n-dimensional vectors of complex numbers. The routine returns the cluster in a better embedded form, the transformation matrix applied and its inverse.

Optional arguments: `eps` is the bound used to decide whether a floating point number is zero, and `c` is the initial value of the acceleration factor.

The algorithm is due to Stoll (see [Sto11]).

This routine can be used for reduction of any variety that has a finite and stable set of special points, by using the transformation that reduces the point set.

`ReducePlaneCurve(f)`

Here f is a homogeneous polynomial in 3 variables of degree > 2 with integral or rational coefficients. A new polynomial and the transformation matrix applied are retured.

The routine computes the intersection points of f with its hessian. Then the cluster reduction is applied to this point set.

114.12.3.1 Minimization and Reduction for Plane Quartics

`MinimizePlaneQuartic(f,p)`

Verbose `PlaneQuartic` *Maximum :* 1

Given a plane quartic defined by the polynomial f with integer coefficients this routine computes an at p minimized model of the quartic. The new quartic and the transformation used are returned.

`MinimizeReducePlaneQuartic(f)`

Verbose `PlaneQuartic` *Maximum :* 1

Given a smooth plane quartic curve defined by the polynomial f with integer coefficients this routine computes a minimized and reduced model of the curve. The transformation matrix returned applied to f will evaluate to a scalar multiple of the returned polynomial.

For the reduction step, `ReducePlaneCurve` is used.

Example H114E40

Here we test the code by taking a bad embedding and a of curve.

```
> _<x,y,z> := PolynomialRing(Integers(), 3);
> C := -3*x^4 + 7*x^3*y - 2*x^3*z + 6*x^2*y^2 + 9*x^2*y*z - 9*x^2*z^2
>       + 10*x*y^3 - 7*x*y^2*z + 5*x*y*z^2 - 6*x*z^3 - 3*y^4 + 5*y^3*z
>       - 3*y^2*z^2 + 4*y*z^3 + 6*z^4;
> C2 := Evaluate(C,[45*x+346*y,74*y+43*z,62324*z+3462*x]);
> C2;
850482855369981*x^4 - 77028319604430*x^3*y + 61459466820119559*x^3*z -
11625449190228*x^2*y^2 - 4102113209795298*x^2*y*z + 16654003843623327 72*x^2*z^2
- 62468022936*x*y^3 - 417499281622764*x*y^2*z - 72808467360772908*x*y*z^2 +
2005588071197635933 2*x*z^3 - 16293798512*y^4 - 875035770696*y^3*z -
3749491014537304*y^2*z^2 - 430694749052979580*y*z^3 + 905674491172905110 49*z^4
> MinimizeReducePlaneQuartic(C2);
6*x^4 - 6*x^3*y + 4*x^3*z - 9*x^2*y^2 + 5*x^2*y*z - 3*x^2*z^2 - 2*x*y^3 +
9*x*y^2*z - 7*x*y*z^2 + 5*x*z^3 - 3*y^4 + 7*y^3*z + 6*y^2*z^2 + 10*y*z^3 - 3*z^4
[    14878   4611976 -21564104]
[    -1935    148866   2804580]
[     3330   -256188   1197852]
```

We do not get the initial curve, but we get a curve with coefficients of the same size.

114.13 Minimal Degree Functions and Plane Models

This section contains functionality to compute smallest degree covering maps from a curve to the projective line $\mathbf{P}^1$, which are equivalent to smallest degree functions on the curve. We refer to such maps as *gonal* maps (the degree of such a map is usually referred to as the *gonality* of the curve). There are intrinsics for all general type (genus ≥ 2) curves of genus less than seven and for all trigonal (gonality 3) curves. The trigonal cases use the Lie algebra method to construct degree 3 maps following the algorithm in [SS]. The 4-gonal cases for genus 5 and 6 curves use the algorithms in [Hara]. Hyperelliptic cases (gonality 2) are dealt with directly using the canonical map and parametrisation of rational curves.

Gonal maps may not exist over the base field and the intrinsics here will return a gonal map over a finite extension in such cases. In some cases, the degree of a minimal extension over which a gonal map may be constructed is determined exactly. In others, this may involve difficult arithmetic problems (e.g., finding a point of minimal degree on a plane sextic over $\mathbf{Q}$) and the extension used may not be of minimal degree. This is discussed further in the descriptions of the intrinsics. The intrinsics are designed for input curves defined over number fields or finite fields.

There are also some intrinsics to compute minimal degree plane models of curves of genus 5 and 6. In the genus 6 case, these birational models may be defined over a small extension field of the base field (an exact minimal degree extension is found here). In the genus 5 case, the intrinsic computes models over the base field given a rational point or divisor of degree 2 in the generic case that may be input by the user. The functionality here is incomplete and doesn't currently cover every type of genus 5 or 6 curve or lower genus curves. We plan to extend this in later releases.

114.13.1 General Functions and Clifford Index One

The algorithm of Schicho and Sevilla used for trigonal curves actually covers the slightly more general case of Clifford index one. An algebraic curve has Clifford index one iff it is trigonal or it is of genus 6 and isomorphic to a non-singular plane quintic. In the latter case, the curve has gonality 4. Classical theory (e.g. Petri's theorem) tells us that a curve of genus greater than one has Clifford index 1 precisely when its canonical image is not defined by quadrics alone. For genus > 3, a minimal basis for the ideal defining the canonical image will then consist of quadrics and cubics. A non-hyperelliptic genus 3 curve is always trigonal and it's canonical image is defined by a smooth quartic in the projective plane.

This gives a simple computational test for Clifford index 1. There is an intrinsic described below that may be called directly by the user for Clifford index 1 canonical curves that returns a gonal map to $\mathbf{P}^1$ in the trigonal case or gives a birational map to a smooth plane quintic.

`GenusAndCanonicalMap(C)`

Convenience function for the user. Returns the genus g of the curve C, a Boolean value which is `true` iff $g \leq 1$ or C is hyperelliptic, and the canonical map from C to its canonical image if $g > 1$.

`CliffordIndexOne(C)`

`CliffordIndexOne(C,X)`

The curve C should be a (non-singular) canonical model of a curve of Clifford index 1 (this condition may be tested as described in the introduction). Computes and returns a degree 3 map to the projective line $\mathbf{P}^1$ or a birational map onto a smooth plane quintic, depending on whether C is trigonal or not. In the trigonal case, the map may be defined over a quadratic extension of the base field for curves of even genus. This will occur only if no such map exists over the base field. In the plane quintic case, the map is always defined over the base field. The trigonal case requires that the characteristic of the base field is not 2.

The algorithm used is that of Schicho and Sevilla that applies the Lie algebra method to explicitly compute the fibration map to $\mathbf{P}^1$ of the rational scroll surface X that is defined by the quadrics in the defining ideal of the canonical curve C. The second version of the intrinsic also takes X as an argument in case the user has already computed it.

Example H114E41

```
> P5<x,y,z,s,t,u> := ProjectiveSpace(Rationals(),5);
> C := Curve(P5,[
> -y*z+x*s-y*s+z*t-4*s*t+t^2+2*s*u-3*t*u+2*u^2,
> -z^2+s^2+x*t+2*z*t-2*t^2-4*s*u-2*t*u+4*u^2,
> -z*s-s^2+z*t+t^2+x*u-2*s*u-7*t*u+6*u^2,
> -z*s+s^2+y*t+2*s*t-t^2-2*s*u+t*u,
> -s^2+s*t+y*u-s*u-t*u+u^2,
> -s*t+t^2+z*u-s*u-3*t*u+2*u^2,
> x^3-3*x^2*y+7*x*y^2+3*y^3+x*y*z-y*z^2+9*z^3+3*y^2*s-3*y*z*s-7*z^2*s-7*y*s^2+
> 17*z*s^2-86*s^3-4*z*s*t-16*s^2*t-185*z*t^2+807*s*t^2-406*t^3+17*s^2*u-
> 666*s*t*u+1391*t^2*u+228*s*u^2-1378*t*u^2+393*u^3,
> x^2*z-2*x*y*z+5*y^2*z+y*z^2+3*z^3+8*y^2*s+y*z*s-7*z^2*s+3*y*s^2+17*z*s^2-
> 47*s^3+6*z^2*t+9*z*s*t+s^2*t+32*z*t^2+11*s*t^2-103*t^3-51*s^2*u-171*s*t*u+
> 433*t^2*u+89*s*u^2-577*t*u^2+246*u^3,
> x*z^2-y*z^2-z^3+4*y*z*s+z^2*s+8*y*s^2-3*z*s^2-5*s^3+7*z^2*t-3*z*s*t+17*s^2*t+
> 8*z*t^2-2*s*t^2+62*t^3-41*s^2*u-34*s*t*u-28*t^2*u+14*s*u^2-127*t*u^2+92*u^3
> ]);
> // C a genus 6 canonical curve of gonality 3
> mp3 := CliffordIndexOne(C);
> mp3;
Mapping from: Crv: C to Curve over Rational Field defined by
0
with equations :
x - 2*z + 10*s - 7*t + 10*u
-2/5*x - 24/5*s + 26/5*t - 44/5*u
```

We can check that `mp3` gives a degree 3 map by seeing that the degree 10 linear pencil of divisors generated by D_1 and D_2, the hyperplane sections of C given by the two defining equations of the

map, has a common degree 7 factor.

```
> defs := DefiningPolynomials(mp3);
> D1 := Scheme(C,defs[1]);
> D2 := Scheme(C,defs[2]);
> D12 := Scheme(C,defs);
> Degree(D1); Degree(D2);
10
10
> Dimension(D12); Degree(D12);
0
7
```

114.13.2 Small Genus Functions

This section contains intrinsics to compute gonal maps for curves with genus greater than one and less than seven.

`Genus2GonalMap(C)`

Returns a degree 2 map from genus 2 curve C to the projective line. The map is defined over a quadratic extension of the base field k iff no such map exists over k. The map is just the canonical map followed by an inverse parametrisation of its image.

`Genus3GonalMap(C)`

`IsCanonical` BoolElt *Default :* `false`

For a genus 3 curve C, returns the gonality (2 or 3) and a gonal map to $\mathbf{P}^1$. C is trigonal precisely when it is non-hyperelliptic, when its canonical image is a plane quartic Q. Gonal maps are given by the canonical map onto Q followed by projection from a point on Q. Constructing one requires finding a point on Q. If Q is defined over the rationals, a point search up to a small height is applied, and if Q is defined over a finite field a point enumeration is applied to try to find a point over the base field. If this fails to locate a point or if the base field is a number field, a point of Q over an extension of degree ≤ 4 of the base field is used. Thus the map returned may be over an extension of the base field and this extension may not be of minimal degree.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

`Genus4GonalMap(C)`

`IsCanonical`	BoolElt	*Default :* `false`

For a genus 4 curve C, returns the gonality (2 or 3) and a gonal map to $\mathbf{P}^1$. C is trigonal precisely when it is non-hyperelliptic. In the trigonal case, a gonal map is computed using the `CliffordIndexOne` intrinsic. This involves computing a fibration map for a quadric surface in $\mathbf{P}^3$ containing the canonical image of C, which may be defined over a biquadratic extension of the base field in bad cases. So the map returned may be defined over an extension of degree 4, whereas the map on C should always be defined over an extension of degree at most 2. We will try to fix this in later releases.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

`Genus5GonalMap(C)`

`DataOnly`	BoolElt	*Default :* `false`
`IsCanonical`	BoolElt	*Default :* `false`

For a genus 5 curve C, returns the gonality (2, 3 or 4) and a gonal map to $\mathbf{P}^1$. In the gonality 4 case, it also returns some extra data that gives the parametrisation of the set of degree 4 linear pencils (g^1_4s) which give the gonal maps.

The hyperelliptic case (gonality 2) is handled as usual by parametrising the canonical image which gives a gonal map that may be defined over a quadratic extension of the base field.

The trigonal case is easy to deal with here since the rational scroll X that contains the canonical image of C is of codimension 2 in its ambient. We directly compute the fibration map of X from the minimal free polynomial resolution of the canonical coordinate ring of C. This gives a gonal map on C that is always defined over the base field.

In the general (4-gonal case), there are infinitely many equivalence classes of gonal maps which are parametrised by a plane quintic curve F. This along with a function f which takes a point in $F(K)$ as an argument are returned as third and fourth return values. f evaluated at $p \in F(K)$ will return the corresponding gonal map on C which is defined over K or a quadratic extension of it. The actual gonal map returned as the second return value is given by searching for a point on F over a small extesnion of the base field. We use a point search as for `Genus3GonalMap` if the base field is the rationals or a finite field. Otherwise or if this fails, we find a point over an extension field by decomposing a random hyperplane section of F. We try to use singular points on F, if they exist, which correspond to g^1_4s whose associated rational scroll is defined by a rank 3 rather than a rank 4 quadric.

If the parameter `DataOnly` is set to `true` (the default is `false`), then only the gonality is returned when it is less than 4, and only the gonality, F and f are computed and returned in the 4-gonal case.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

`Genus6GonalMap(C)`

`DataOnly`	BoolElt	*Default :* `false`
`IsCanonical`	BoolElt	*Default :* `false`

For a genus 6 curve C, returns the gonality (2, 3 or 4), a second type identifer and a gonal map to $\mathbf{P}^1$. The second type number is irrelevant in the gonality 2 and 3 cases, when it is always 1. It is 1,2 or 3 in the 4-gonal case.

For 4-gonal curves of secondary type 2, C is a double cover of a genus 1 curve E through which all gonal maps factor. In this case, the map giving this double covering is returned as a fourth return value. Secondary type 3 curves are isomorphic to (smooth) plane quintics and an isomorphism to such a quintic is returned as the fourth return value. These maps are defined over the base field.

The hyperelliptic case (gonality 2) is handled as for `Genus5GonalMap`.

The trigonal case requires the characteristic of the base field to not be 2. It uses the `CliffordIndexOne` intrinsic and constructs a gonal map that is defined over a quadratic extension of the base field k if no such map exists over k.

There are 3 distinct subcases of the 4-gonal case, which are distinguished by the second return number.

Type 2 curves are double covers of a genus 1 curve E and there are infinitely many 4-gonal maps which are given by composing this double cover with a degree 2 map of E to $\mathbf{P}^1$. E is constructed as a projective normal curve of degree 5 in $\mathbf{P}^4$. The gonal map returned is found by looking for k-rational points on E by general point search methods and, if this fails, taking a point on E defined over a finite extension of k lying in a random hyperplane section of E.

Type 3 curves are (birationally) isomorphic to a plane quintic C_5. The infinitely many 4-gonal maps are given by projection from a point on C_5. The gonal map returned is constructed by finding a k-rational point on C_5 or one over a finite extension as for type 2.

Type 1 (general type) curves have only finitely many gonal maps up to equivalence (5 at most, in fact). The algorithm explicitly finds algebraic data defining these map classes. It chooses one defined over k if possible. Otherwise, we take one over a minimal degree extension of k. The gonal map returned is thus defined over a minimal degree extension of k for such maps.

If the parameter `DataOnly` is set to `true` (the default is `false`), then only the gonality (and secondary type 1) is returned when it is less than 4, and only the gonality, secondary type t and fourth return value when $t = 2$ or 3 are computed and returned in the 4-gonal case.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

Example H114E42

We give some examples with 4-gonal curves of genus 5 and 6. Firstly, we take a random canonical curve of genus 5 given by the intersection of 3 quadrics in $\mathbf{P}^4$.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> C := Curve(P4,[
> -x^2-x*y-y*z+z^2-x*t-y*t-z*t+t^2-x*u+y*u-t*u,
> -x*y+y^2+y*z+x*t+y*t-z*t+z*u+t*u+u^2,
> -x^2-x*y-y*z+z^2+x*t-y*t+t^2-x*u+y*u+z*u+t*u+u^2]);
> g,mp4,F,f := Genus5GonalMap(C);
> g;
4
> mp4;
Mapping from: Crv: C to Curve over Rational Field defined by
0
with equations :
z + u
t
> F;
Curve over Rational Field defined by
7*u^5+6*u^4*v-50*u^3*v^2+40*u^2*v^3+3*u*v^4+2*v^5+48*u^4*w-22*u^3*v*w-
 50*u^2*v^2*w+30*u*v^3*w+10*v^4*w+134*u^3*w^2-24*u^2*v*w^2-36*u*v^2*w^2+
 8*v^3*w^2+187*u^2*w^3+6*u*v*w^3-19*v^2*w^3+129*u*w^4+6*v*w^4+35*w^5
```

Next, we look at a genus 6 curve that is a degree 2 cover of a genus 1 curve.

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> C := Curve(P3,[ x^2*y^2-x^2*t^2-z^2*t^2+2*t^4,
>    y^4-x*z^2*t-2*y^2*t^2+2*x*t^3+t^4,
>    x^3-y^2*t+t^3 ]);
> Genus(C);
6
> g,t,mp4,mpE := Genus6GonalMap(C);
> g; t;
4
2
> mp4;
Mapping from: Crv: C to Curve over Rational Field defined by
0
with equations :
x
t
> mpE;
Mapping from: Crv: C to Curve over Rational Field defined by
$.1^2 + $.2*$.3 - $.4*$.5,
$.1*$.2 + $.3^2 - $.5^2,
$.2^2 - $.1*$.3,
$.2*$.4 - $.1*$.5,
$.3*$.4 - $.2*$.5
```

```
with equations :
x^2
x*t
t^2
x*y
y*t
```

114.13.3 Small Genus Plane Models

This section contains intrinsics to compute minimal degree birational plane models for curves of genus 5 or 6.

`Genus6PlaneCurveModel(C)`

`IsCanonical` BoolElt *Default :* `false`

For a genus 6 curve C of gonality 4 which isn't the double cover of a genus 1 curve (i.e. subtype 2 for `Genus6GonalMap`), returns a birational map to a plane curve of minimal degree (5 or 6). This map may be defined over a finite extension of the base field k, but it will always be a minimal degree extension for the existence of minimal degree plane models. The first return value is a boolean, which is `true` only if C is of gonality 4 and not of subtype 2. If so, the second return value is the map to the curve.

When C is birationally isomorphic to a plane quintic C_5, (subtype 3 for `Genus6GonalMap`), C_5 is a minimal degree plane model and it and the birational map to it are defined over k. This is a Clifford index 1 case, and the computation is performed via the `CliffordIndexOne` intrinsic.

In the general case, (subtype 1 for `Genus6GonalMap`), the smallest degree plane models are of degree 6, there are only finitely many birational maps from C to degree 6 plane curves up to linear equivalence and these are in 1-1 correspondence with the finitely many equivalence classes of gonal maps. The birational maps to plane models are computed by a slight variant of the same algorithm used to compute the gonal maps (see [Hara]). The return value is over the base field k, if possible, and otherwise over a smallest degree (at most 5) possible field extension.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

`Genus5PlaneCurveModel(C)`

`Genus5PlaneCurveModel(C,P)`

`Genus5PlaneCurveModel(C,Z)`

`IsCanonical` BoolElt *Default :* `false`

For a genus 5 curve C, tries to compute a birational map over the base field k to a plane curve of minimal degree (5 or 6) for plane curve models over $\bar{k}$. The first

return value is a boolean which is `true` if the construction succeeds and, if so, the second return value is the birational map to the plane curve.

If C is hyperelliptic, the return value is always `false`. Gonality 3 curves are always birational over k to a plane curve of degree 5 with a single singular point and the computation always succeeds. The map to $\mathbf{P}^2$ on a canonical model C_c of C lying in $\mathbf{P}^4$ is given by projection from a line lying in the rational scroll surface X that contains C_c.

If C is 4-gonal and isn't a double cover of a genus 1 curve, the minimal degree plane model is of degree 6 and there are infinitely many of these (up to linear equivalence) that are given by projection from a secant line or tangent line of the canonical model C_c.

To find a birational map to such a model over k, we need a secant or tangent line defined over k, which will correspond to a k-rational point on C_c or a k-rational reduced divisor of degree 2 (i.e. 2 distinct points that are k-rational or conjugate over a quadratic extension of k).

The second and third versions of the function have a second argument that should be a k-rational *non-singular* point on C or a reduced subscheme of C of dimension 0 and degree 2, whose points (over $\bar{k}$) lie in the non-singular locus of C. These are used to define the k-rational tangent line or secant line on the canonical model and the intrinsic will always succeed and return the map to a degree 6 model if they are provided by the user.

In the version with only the curve C as input, the intrinsic attempts to find a (non-singular) k-rational point on C using `PointSearch` if the base field is the rationals or using point enumeration if the base field is a finite field. If this search fails, or k is a field of a different type, the intrinsic will fail and return `false`.

If the input C is already a canonical model, parameter `IsCanonical` may be set to `true`. This will simplify the internal processing. **NB** The defining equations of C must be a minimal basis for its defining ideal.

Example H114E43

We use the intrinsic for genus 6 curves to get a degree 6 plane model for the modular curve $X_0(58)$, starting from its canonical model.

```
> X := X0NQuotient(58,[]);
> X;
Curve over Rational Field defined by
x[1]^2-x[1]*x[3]+x[2]*x[4]+x[2]*x[5]-x[1]*x[6]+x[4]*x[6]+x[5]*x[6],
x[1]^2-x[1]*x[2]-x[2]^2-x[1]*x[3]-x[1]*x[4]+x[3]*x[4]+x[2]*x[5]+x[3]*x[5],
-x[1]^2+x[1]*x[2]+x[2]^2+x[4]^2+x[4]*x[5]-x[2]*x[6],
-x[1]^2-x[1]*x[2]+x[2]^2+x[2]*x[3]-x[1]*x[4]+x[2]*x[4]+x[3]*x[4]+x[4]^2-x[5]*x[6],
x[2]^2-x[1]*x[3]+x[2]*x[3]+x[3]^2-x[1]*x[4]+x[3]*x[4]+x[2]*x[6]+x[3]*x[6],
x[1]*x[2]-x[2]*x[3]-x[1]*x[4]+x[3]*x[4]+x[2]*x[5]+x[3]*x[5]+x[4]*x[5]-x[1]*x[6]+
x[2]*x[6]+x[3]*x[6]+x[4]*x[6]
> boo,mp := Genus6PlaneCurveModel(X : IsCanonical := true);
> boo;
```

```
true
> C<x,y,z> := Codomain(mp); //the plane model
> C;
Curve over Rational Field defined by
x^6-3*x^5*y+x^4*y^2+3*x^3*y^3-7/4*x^2*y^4-1/4*x*y^5-1/4*y^6+3/2*x^5*z-x^4*y*z-
9/2*x^3*y^2*z+7/2*x^2*y^3*z+5/8*x*y^4*z+3/4*y^5*z+5/4*x^4*z^2+1/4*x^3*y*z^2-
3*x^2*y^2*z^2+3/4*x*y^3*z^2-3/4*y^4*z^2+5/8*x^3*z^3+5/4*x^2*y*z^3-7/4*x*y^2*z^3+
1/4*y^3*z^3+1/8*x^2*z^4+5/8*x*y*z^4-1/8*y^2*z^4+1/8*y*z^5
> mp;
Mapping from: Crv: X to Crv: C
with equations :
x[1] - x[3]
x[2] + x[6]
x[5] + x[6]
```

Do the same for genus 5 4-gonal modular curve $X_0(42)$.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> X42 := Curve(P4,[x*z+z^2+x*t-x*u,
>   y^2-2*y*z+z^2-x*t-2*y*t+z*t-2*y*u+z*u+t*u,
>   x^2+x*t+y*t+y*u ]);
> // use pointsearch to pick rational point on X42
> boo,mp := Genus5PlaneCurveModel(X42 : IsCanonical := true);
> boo;
true
> C<x,y,z> := Codomain(mp);
> C;
x^6+3*x^5*y+9/2*x^4*y^2+2*x^3*y^3+2*x^2*y^4+11/8*x^5*z+31/8*x^4*y*z+23/4*x^3*y^2*z
+1/2*x^2*y^3*z+3*x*y^4*z+85/128*x^4*z^2+29/16*x^3*y*z^2+41/16*x^2*y^2*z^2-
9/4*x*y^3*z^2+9/8*y^4*z^2+9/64*x^3*z^3+17/32*x^2*y*z^3+15/16*x*y^2*z^3-
9/8*y^3*z^3+1/128*x^2*z^4+3/32*x*y*z^4+9/32*y^2*z^4
> mp;
Mapping from: Crv: X42 to Crv: C
with equations :
x + 1/2*t - 1/2*u
y - 1/4*t + 1/4*u
z - t + u
> // use nicer chosen point for nicer map
> boo,mp := Genus5PlaneCurveModel(X42,X42![0,0,0,0,1] : IsCanonical := true);
> C<x,y,z> := Codomain(mp);
> C;
x^5*y-2*x^4*y^2+x^3*y^3-x^2*y^4+2*x*y^5-y^6-x^5*z+3*x^4*y*z-6*x^3*y^2*z-
11*x^2*y^3*z-12*x*y^4*z-2*x^4*z^2+9*x^3*y*z^2+20*x^2*y^2*z^2+16*x*y^3*z^2+
2*y^4*z^2-4*x^3*z^3-7*x^2*y*z^3-4*x*y^2*z^3-x^2*z^4-2*x*y*z^4-y^2*z^4
> mp;
Mapping from: Crv: X42 to Crv: C
with equations :
x
y
```

```
z + t
```

114.14 Bibliography

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**Die06**] Claus Diem. An Index Calculus Algorithm for Plane Curves of S mall Degree. In Hess et al. [HPP06], pages 543–557.

[**Elk00**] N. Elkies. Rational Points Near Curves and Small Nonzero $|x^3 - y^2|$ via Lattice Reduction. In Bosma [Bos00], pages 33–63.

[**Ful69**] William Fulton. *Algebraic Curves*. Mathematics Lecture Note Series. W. A. Benjamin, New York-Amsterdam, 1969.

[**Har**] M. C. Harrison. Explicit solution by radicals, gonal maps and plane models of algebraic curves of genus 5 or 6. Preprint: online at arXiv as arXiv:1103.4946v3[math. AG].

[**Har77**] Robin Hartshorne. *Algebraic Geometry, GTM 52*. Springer, ASpringer, 1977.

[**HPP06**] F. Hess, S. Pauli, and M. Pohst, editors. *ANTS VII*, volume 4076 of *LNCS*. Springer-Verlag, 2006.

[**HS10**] J. Hilmar and C. Smyth. Euclid Meets Bézout: Intersecting Algebraic Plane Curves with the Euclidean Algorithm. *The American Mathematical Monthly*, 117:250–260, (March) 2010.

[**SS**] J. Schicho and D. Sevilla. Effective radical parametrization of trigonal curves. Preprint: online at arXiv as arXiv:1104.2470v1[math.AG].

[**ST02**] Frank-Olaf Schreyer and Fabio Tonoli. Needles in a Haystack: Special Varieties via Small Fields. In Eisenbud et al., editors, *Computations in Algebraic Geometry with Macaulay2*, volume 8 of *Springer Algorithms and Computation in Mathematics Series*, pages 251–277. Springer-Verlag, 2002.

[**Sto11**] Michael Stoll. Reduction theory of point clusters in projective space. 2011.

[**vLvdG88**] J. H. van Lint and G. van der Geer. *Introduction to Coding Theory and Algebraic Geometry*, volume 12 of *DMV Seminar*. Birkhaeuser, Basel, 1988.

115 RESOLUTION GRAPHS AND SPLICE DIAGRAMS

Chapter 115
RESOLUTION GRAPHS AND SPLICE DIAGRAMS

115.1 Introduction

Both resolution graphs and splice diagrams are labelled graph-like diagrams used to encode geometric data closely related to some resolution of singularities procedure in algebraic geometry. They are commonly used to visualise this data. Of course, there are other tools in MAGMA, Puiseux expansions for instance, which can be used if preferred. A typical example is when a configuration of curves on a surface is the given data. In this case, *dual graph* of the configuration has vertices corresponding to the individual curves and edges corresponding to their intersections. The vertices may be labelled with the selfintersections of the corresponding curves and possibly also with the multiplicities with which the curves appear in the configuration.

This chapter discusses two different enhanced graph types: `GrphRes` for resolution graphs and `GrphSpl` for splice diagrams. Neither of them is literally a graph in MAGMA; in particular, functions taking graphs as argument cannot be applied directly to objects defined here. Instead they work by having a directed graph, referred to as the *underlying graph*, as a primary attribute and by caching other data (which is typically associated to particular vertices and edges of the graph) in sequences as secondary attributes. There are also vertex types which allow the convenient idiom of MAGMA's graph package to be used. Note, however, that unlike other graph types, these do not have edge types.

Graph surgery routines cannot be used directly since they must manage both the underlying graph and the associated data. A collection of appropriate surgery functions, those used in resolution routines, have been provided in this context; they are usually brief, simply concatenating attribute data whenever it is present on both sides.

Functions to recover data related to the graph, whether globally or at a particular vertex or edge, are also provided. Thus the user does not access labels of the graph but rather uses the intrinsics listed later in this chapter. Of course, usually these do no more than unload an attribute.

115.2 Resolution Graphs

A resolution graph g is a graph with data associated to its vertices. The underlying graph is the dual graph of a blowup process. The vertices correspond to rational curves E, "exceptional curves", on some surface S which are contracted by some map $f : S \to P^2$, where P^2 is the projective plane. Such maps arise during the resolution process by blowups of a curve singularity, say $p \in C$ where C is a curve in P^2. In that context, C can be

pulled back to S using f and the pullback can be decomposed (as an effective divisor on S) as

$$f^*C = \widetilde{C} + E_C$$

where $\widetilde{C}$ is the *birational transform* of C and E_C is an effective divisor supported on the union of the exceptional curves. The surface S is never realised as a geometric object.

For a vertex v of g, the corresponding rational curve is often denoted E_v. The edges of g correspond to the intersections between different E: vertices v and w are joined by edges corresponding in some way to the points of intersection of the curves E_v and E_w. In the contexts used below, any two curves will meet at most in one point and that will be a transverse intersection. So the intersection number E_vE_w on S will be the number of edges between v and w, either zero or one, if they are distinct vertices and a selfintersection number associated to v otherwise. Thus the graph enables basic intersection calculations to be carried out implicitly on S.

The data associated to each vertex v is the following where $E = E_v$: the selfintersection E^2; the coefficient of E in some pullback divisor, often E_C but this depends on the context; the coefficient of E in a representative of the canonical class of S locally supported on the exceptional curves; the number of transverse intersections of the birational transform $\widetilde{C}$ of C with E. At each vertex v, this data is often denoted s_v, m_v, k_v, t_v respectively. This is the coarsest kind data one could want. It enables basic intersection theory calculations (including calculating the contribution of a singularity p of a curve C to the genus of C). For more detailed calculations, the map f and the equation of the birational transform of C are held on patches along each E. In each case, the patch is an affine xy-plane with E as the line $y = 0$ in it (although it is actually the closure of this patch and map that is recorded).

Resolution graphs are usually created implicitly by some geometrical algorithm like the resolution of a plane curve singularity. The first creation methods below are of this nature. But graphs can also be created explicitly by providing the required data. How much data is required varies according to the purpose of the graph; some common alternatives are included here.

115.2.1 Graphs, Vertices and Printing

Resolution graphs are less complicated as a type than the other graphs in MAGMA. They do not have associated vertex and edge sets. However, there is a resolution graph vertex type so that vertices can be passed between intrinsics.

Graph printing is similar to that of the underlying directed graph. An example is

```
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -2, 9, 1, 0 ])     2 ;
2 ([ -4, 18, 2, 0 ])    3 ;
3 ([ -2, 63, 9, 0 ])    4 6 ;
4 ([ -2, 42, 6, 0 ])    5 ;
5 ([ -2, 21, 3, 0 ])    ;
```

```
6 ([ -1, 66, 10, 3 ])   ;
```

This is a graph on 6 vertices — they are listed as one of the integers $1, \ldots, 6$ down the left. The integer corresponding to a vertex is called the *index* of the vertex. In brackets by each vertex v is a label of the form $[s, m, k, t]$ where s is the selfintersection, m is the multiplicity (the interpretation of which is dependent on the context), k is the canonical multiplicity and t is the number of transverse intersections at v as described in the introduction. Next comes a space-separated list of the vertices at the far end of edges directed away from v. Until one is used to reading graphs in this way, and also even then, drawing the graph by hand is recommended.

The vertex labels can be shorter if some data is missing. The alternatives are $[s]$, $[s, t]$ and $[s, m, t]$ in the previous notation. The most data consistent with what has been calculated for the graph will be displayed.

Notice that, although these really are printed as graph labels, the data in them should not be accessed as such. These labels are often unassigned, or assigned in an unexpected way. They are only intended for printing. There are dedicated functions below for retrieving data associated to a graph.

`g eq h`

Returns `true` if and only if the resolution graphs g and h are the same object in MAGMA.

`ResolutionGraphVertex(g,i)`

`g ! i`

The vertex of the resolution graph g with index i.

`Vertex(v)`

The underlying directed graph vertex of the resolution graph vertex v.

`ResolutionGraph(v)`

The resolution graph of which v is a vertex.

`IsVertex(g,v)`

Returns `true` if and only if v is a vertex of the resolution graph g.

`Index(v)`

The index of the underlying graph vertex of the resolution graph vertex v; this is the integer appearing as the vertex identifier when the graph is printed.

`v eq w`

Returns `true` if and only if the resolution graph vertices v and w are the same object in MAGMA.

115.2.2 Creation from Curve Singularities

Let C be a reduced plane curve, either affine or projective. See Chapter 114 for details of how to create such a curve. Let p be a singular point of C. The intrinsic below calculates the dual graph of the resolution of p on C together with some auxiliary data. As in the introduction, the sequence of blowups required in the resolution is thought of as a morphism $f : S \to P^2$ of projective surfaces; P^2 is the projective plane (which is the closure of the ambient plane of C if it is affine) while S is a surface not realised explicitly in MAGMA.

The target resolution is the minimum transverse resolution, sometimes called the log resolution, a resolution in which the birational transform $\widetilde{C}$ of C on S is nonsingular and transverse to the exceptional locus. However, there are circumstances under which a larger resolution will be calculated. This makes no difference to the geometric data arising from the resolution.

```
ResolutionGraph(C, p)
```

M	RNGINTELT	*Default :* 1
K	RNGINTELT	*Default :* 1

```
ResolutionGraph(C, p)
```

Calculate a transverse resolution graph of the plane curve singularity of C at the point p. If the point argument is missing and C is affine, the resolution is calculated at the origin, but in that case parameters cannot be assigned and take the default values.

The numerical parameters determine whether additional data is calculated: value 1, the default, enables the calculation while value 0 omits it.

The parameter `M` refers to the pullback multiplicities of C. It returns a sequence $[m_v]$ of rational numbers (although always integral in the current algorithms) corresponding to the vertices of the graph. These numbers have the following meaning: on the blownup surface S, as divisors,

$$f^*C = \widetilde{C} + \sum m_v E_v$$

where the sum is taken over the vertices v of the graph and E_v is the corresponding exceptional curve.

The parameter `K` refers to the canonical multiplicities along g. It returns a sequence $[k_v]$ of rational numbers (although again always integral) corresponding to the vertices of the graph. On the blownup surface S in a neighbourhood of the union of exceptional curves E the canonical class K_S has a representative

$$K_S = \sum k_v E_v$$

where the sum is taken over the vertices of the graph.

The surface S is covered by affine plane patches. The map f to the projective plane can be calculated when restricted to these patches. In fact, the closure of

these maps is calculated as a birational automorphism of the projective plane. All blowup algorithms arrange that the exceptional curve E_v corresponding to a vertex v is (a patch on) L_y, the second coordinate line "$y = 0$", in the projective plane. So pulling C back to the plane using the patch map at v will produce a curve having $m_v L_y$ as a component.

The calculation of maps and pullbacks is expensive so some precautions are taken. First, the maps are only calculated at significant vertices of the graph: significant here means that the blowup procedure branches at that vertex, or that $\widetilde{C}$ meets the exceptional locus there. Second, the maps are calculated as a sequence of maps: the map required is the composition of these. The composition will be carried out automatically if needed. (In the current code, this can only be carried out if no field extensions have been made. This will be updated in due course.)

The calculation is carried out by tying together strings of blowups (done recursively in one go using a standard Newton polygon argument). The Newton polygon argument used automatically makes a curve transverse to all axes, not just to exceptional curves. These extra blowups do not invalidate numerical calculations made with the resolution graph (since minimality is never a condition) and they are essential when resolving irregular fibres of pencils.

Example H115E1

First make a curve with a singularity. The following curve demonstrates the typical way singularities with more than one Puiseux pair — those which need the Newton algorithm to be repeated when calculating a resolution or local parametrisation — arise.

```
> A<x,y> := AffineSpace(Rationals(),2);
> C := Curve(A,(x^2 - y^3)^2 + x*y^6);
```

The interesting singularity is at the origin. The next two lines calculate the resolution graph of this singularity and display it.

```
> g := ResolutionGraph(C,Origin(A));
> g;
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -3, 4, 1, 0 ])     2 ;
2 ([ -2, 12, 4, 0 ])    3 4 ;
3 ([ -2, 6, 2, 0 ])     ;
4 ([ -3, 14, 5, 0 ])    5 ;
5 ([ -1, 30, 12, 1 ])   6 ;
6 ([ -2, 15, 6, 0 ])    ;
```

The resulting graph has 6 vertices. So the transverse resolution of this singularity is achieved by 6 blowups. The order of blowup can be determined by the third column, the canonical class: this number strictly increases throughout the process, so curve E_1 is the first exceptional curve, E_3 is the second and so on until finally E_5 is extracted. This could also be deduced from the sequence of selfintersections, the first column. See how the birational image of C after blowing up only intersects E_5, and then only in a single transverse point. In other words, C has a single place at the origin.

115.2.3 Creation from Pencils

Let P be a jacobian pencil in the affine plane A^2, that is, a pencil of the form $f(x, y) = c$ as c varies. See Chapter 112 for details of how to create jacobian pencils. The resolution graph which can be created automatically is the regular resolution graph: that is, the minimal sequence of transverse blowups which resolve the rational map determined by P from the projective plane (the closure of A^2) to the projective line. However, other resolution graphs can be constructed using more explicit creation functions.

`ResolutionGraph(P)`

The resolution at infinity of the jacobian pencil $P : f = c$ of some polynomial f as the value c varies defined on some affine space A. This resolution graph is thought of as being rooted with root vertex corresponding to the line at infinity of A. The multiplicities are those of the fibre of f at infinity.

`ResolutionGraph(P,a,b)`

The resolution graph at infinity of the union of the two fibres of P above a and b. The multiplicities and canonical class are not calculated automatically for this graph.

Example H115E2

The following is a simple example of a pencil exhibiting interesting behaviour at infinity. It is taken from [Neu99].
First a pencil in some affine plane is made.

```
> A<x,y> := AffineSpace(Rationals(),2);
> f := x^2*y - x;
> P := Pencil(A,f);
> P;
The pencil defined by x^2*y - x
```

Then the regular resolution graph at infinity is calculated.

```
> ResolutionGraph(P);
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -1, 3, -3, 0 ])    2 5 ;
2 ([ -3, 1, -2, 0 ])    3 ;
3 ([ -1, 0, -2, 1 ])    4 ;
4 ([ -2, 0, -1, 0 ])    ;
5 ([ -2, 2, -2, 0 ])    6 ;
6 ([ -2, 1, -1, 0 ])    7 ;
7 ([ -1, 0, 0, 1 ])     ;
```

The resulting graph has 7 vertices. Vertex 1 corresponds to the line at infinity of the plane, or more properly to its birational image after blowing up. The pencil meets this line at two points,

each of which have undergone three blowups in the resolution process. The vertex labels carry auxiliary data. The first column lists the selfintersections of the curves E_v corresponding to the vertices v. The general fibre of the pencil meets a single exceptional curve above each of the points as shown by the fourth column. The fibre at infinity is $3E_1 + E_2 + 2E_5 + E_6$ as shown in the second column. The third column gives the multiplicities of a canonical divisor on the blownup surface: K_S can be represented by the divisor $-3E_1 - 2E_2 - 2E_3 - E_4 - 2E_5 - E_6$.
It is known that the pencil P is irregular at infinity above 0. This can be seen by calculating the explicit resolution of the union of two fibres, $f = 0$ and a general one, say $f = 1$.

```
> ResolutionGraph(P,0,1);
The resolution graph on the Digraph
Vertex  Neighbours
1 ([ -1, 0 ])   2 5 ;
2 ([ -3, 0 ])   3 ;
3 ([ -1, 1 ])   4 ;
4 ([ -2, 2 ])   ;
5 ([ -2, 0 ])   6 ;
6 ([ -2, 0 ])   7 ;
7 ([ -1, 2 ])   ;
```

The multiplicities and canonical class have not been calculated by this function. Later in this chapter there are functions which will do this, although the multiplicities require some interpretation. The selfintersections have been calculated as before (and notice that the blowups at least appear to be exactly those of the previous resolution procedure; in fact they are indeed the same). The number of transverse intersections are now those of the union of the two fibres. Since the fibre above 1 is general it contributes one intersection at vertex 3 and one at vertex 7. So the intersections of the irregular fibre can be deduced, namely two at vertex 4 and one at vertex 7.

115.2.4 Creation by Hand

A resolution graph can be created by hand. This can be fiddly if the underlying graph is complicated. See Chapter 149 for details on how to create a directed graph.

`MakeResolutionGraph(g,s,t)`

`MakeResolutionGraph(g,s)`

The resolution graph on underlying directed graph g. Although it is not checked, the graph g should usually be a directed tree otherwise some reduction algorithms which might be invoked later might not work. In that case, moreover, its root must be the vertex of index 1.

The selfintersections of vertices correspond to the integer entries of the sequence s. If used, the number of transverse intersections of the putative resolved curve with each vertex correspond to the integer entries of the sequence t.

`MakeResolutionGraph(N)`

The resolution graph corresponding to the Newton polygon N.

`UnderlyingGraph(g)`

The underlying directed graph of the resolution graph g.

115.2.5 Modifying Resolution Graphs

Any resolution graph, whether created by hand or not, can have its numerical data calculated or modified. There are also some functions for performing surgery on the underlying graph.

There are also functions which do the linear algebra calculations typical of the numerical calculations associated with resolution graphs. But beware: they each base their calculation on some part of the data of the graph but make no check that all numerical data is consistent at the end of calculation.

`Connect(v,w)`

If v and w are vertices of distinct resolution graphs, return the graph comprising the union of these graphs joined by an edge from v to w. Selfintersections are inherited by the graph from its two components. Multiplicities, canonical class and transverse intersections will be inherited if calculated on both components.

`CalculateCanonicalClass(~g)`

Calculate the canonical class supported on the resolution graph g using the selfintersections of the E_v and the assumption that the E_v are nonsingular rational curves meeting transversely. Note that this calculation uses only the selfintersections already associated to g.

`Disconnect(v,w)`

If v and w are vertices of a resolution graph g, return the resolution graph with any edge joining them removed. The resulting graph may well be disconnected. The only data preserved is the selfintersections and transverse intersections.

`Component(v)`

The connected component of the resolution graph containing vertex v.

`CalculateMultiplicities(~g)`

Calculate the pullback multiplicities of the resolution graph g using the selfintersections of the E_v, the assumption that the E_v are nonsingular rational curves meeting transversely, and the number of transverse intersections of $\widetilde{C}$ with the E_v. It is assumed that g is the resolution graph of a curve singularity during this calculation, although that need not be the case. In general there will be some choice of multiplicities. If g is the resolution of a curve singularity, and if the multiplicity of that singularity is cached, then the correct multiplicities can be identified: the multiplicity along the first blownup curve is the same as that of the singularity; the curve which was blown up first can be identified since it alone has canonical multiplicity 1.

However, if g arose as the resolution at infinity of two fibres of a pencil, for instance, the multiplicities calculated here would depend on which two fibres were used. The result would be the unique divisor supported on g which when added to the birational transform of the two affine fibre patches was linearly equivalent to the zero divisor. In particular, if the two fibres were general, the calculation here would return -2 times the fibre at infinity. Otherwise the calculation will return a combination of that and the exceptional components of irregular finite fibres. The intrinsic which calculates the regular resolution graph of a pencil already takes this into account.

`CalculateTransverseIntersections(~g)`

Calculate the number of transverse intersections of $\widetilde{C}$ with each E_v on the basis of their selfintersection numbers and multiplicities in the resolution graph g.

`ModifySelfintersection(~v,n)`

Change the selfintersection at vertex v of a resolution graph to n.

`ModifyTransverseIntersection(~v,n)`

Change the number of transverse intersections at vertex v of a resolution graph to n.

115.2.6 Numerical Data Associated to a Graph

The meaning of the data given here depends on the context in which the graph was created. The case already discussed of a configuration of rational curves arising from the resolution of a curve singularity is the prototype.

Many of these functions can also be applied to a single vertex of a graph: `Selfintersection`, `CanonicalMultiplicity` and so on.

`Size(g)`

The number of vertices of the underlying graph of the resolution graph g. Typically, this is the number of exceptional curves in the resolution.

`SelfIntersections(g)`

The selfintersections of the vertices of the resolution graph g.

`Multiplicities(g)`

The multiplicities of the vertices of the resolution graph g in some divisor.

`CanonicalClass(g)`

The multiplicities of the vertices of the resolution graph g in a local representative of the canonical class.

`TransverseIntersections(g)`

The number of transverse intersections of some curve (usually used to create g) with the vertices of the resolution graph g.

`GenusContribution(g)`

The contribution to the genus of a plane curve of a singularity having g as its resolution graph.

`CartanMatrix(g)`

The incidence matrix of the (undirected) graph underlying the resolution graph g with selfintersections on the diagonal.

`Determinant(g)`

The determinant of the Cartan matrix of the resolution graph g.

115.3 Splice Diagrams

Splice diagrams are graphs decorated with integer labels at each end of each edge and a number of arrows, often none at all, attached to each vertex. They are fully described in [EN85]. No description of the meaning of splice diagrams will be given here, only the functions that MAGMA has for manipulating them. (There are two common features of splice diagrams that cannot yet be realised. First, omitting edge labels which are 1 is common but is not allowed. Second, and less trivially, arrow weights are not allowed.

Other data that is also stored with the diagram are vertex multiplicities, canonical multiplicities, and total linking numbers of vertices. The distinction between a splice diagram and its underlying directed graph, as well as that between a splice diagram vertex and its underlying vertex, is often left implicit.

115.3.1 Creation of Splice Diagrams

Splice diagrams are usually created from geometric objects like plane curve singularities or jacobian pencils. They can also be built explicitly by hand. The section on translations between graphs describes functions which create splice diagrams associated to resolution graphs.

`SpliceDiagram(C,p)`

The splice diagram of the plane curve singularity of C at the point p.

`RegularSpliceDiagram(P)`

The regular splice diagram at infinity of the jacobian pencil P. This diagram is considered to be rooted at vertex 1 corresponding to the line at infinity of the ambient plane of P. Its underlying graph is directed away from this root.

`MakeSpliceDiagram(g,e,a)`

A splice diagram on directed graph g. The edge labels are taken from the sequence e. The elements of this sequence are sequences of pairs of integers $[a, b]$. The i-th element of e will be assigned to the i-th edge of g (in the order returned by `Edges(g)`) with a as the near label and b as the far label (with respect to the directions of the edges of g). The number of arrows at each vertex are taken from the sequence of integers a.

`MakeSpliceDiagram(e,l,a)`

The splice diagram described by the data in the sequences e, l, a. The first two contain sequences of two integers: e is interpreted as a set of directed edges, the integers appearing in it being the vertex indices of the resulting graph. Edge labels are determined by the sequence l as in the previous function. The number of arrows at each vertex are taken from the sequence of integers a.

`SpliceDiagramVertex(s,i)`

The vertex of the splice diagram s having index i.

`SpliceDiagram(v)`

The splice diagram containing the vertex v.

`UnderlyingGraph(s)`

The underlying directed graph of the splice diagram s.

`UnderlyingVertex(v)`

`Vertex(v)`

The vertex of the underlying graph corresponding to the vertex v of a splice diagram.

`Vertices(s)`

The vertices of the splice diagram s.

`RootVertex(s)`

The root vertex of the splice diagram s as a rooted tree directed away from its root by the directions on the edges.

`Index(v)`

The index of the vertex v of a splice diagram.

`s eq t`

`v eq w`

Returns `true` if and only if the splice diagrams s and t are the same object in MAGMA. This can also be applied to a pair of vertices of a splice diagram.

115.3.2 Numerical Functions of Splice Diagrams

The raw numerical data of a splice diagram is the collection of labels on its edges and vertices together with data recovered from the underlying graph. First we list functions to get hold of this data, and then some more substantial functions which derive many of the natural conclusions of this data, linking number, euler characteristic and so on.

`EdgeLabels(s)`

The integer labels on the edges of the splice diagram s.

`VertexLabels(s)`

The integer labels on the vertices of the splice diagram s.

`TotalLinking(v)`

The total linking number of the vertex v of a splice diagram.

`LinkingNumbers(s)`

The total linking numbers of the vertices of the splice diagram s.

`Linking(u,v)`

The linking number of vertices u and v of a splice diagram.

`EdgeDeterminant(u,v)`

The edge determinant of the edge joining the vertex u to the vertex v of a splice diagram.

`Valency(v)`

The splice valency of the vertex v of a splice diagram. This is the valency of v in the underlying graph plus the number of arrows at v.

`IsRegular(s)`

Returns `true` if and only if the splice diagram s is regular.

`IsReduced(s)`

Returns `true` if and only if the splice diagram s is reduced, that is, it has no valency 2 nodes and no weight 1 leaves.

`HasIrregularFibres(s)`

Returns `true` if and only if the splice diagram s has a vertex with zero linking number.

`Degree(s)`

The linking number of the first vertex of the splice diagram s.

`EulerCharacteristic(s)`

The Euler characteristic of the splice diagram s.

`Size(s)`

The number of vertices of the splice diagram s.

`Arrows(s)`

A sequence of integers containing the number of arrows of the splice diagram s: the i-th sequence entry is the number of arrows at the vertex of index i. This function can also be applied to a single vertex returning a single integer.

`VertexPath(u,v)`

A sequence of vertices on the path from the vertex u to the vertex v of a splice diagram. The second return value is the sequence of products of off-path weights at each vertex.

115.4 Translation Between Graphs

Splice diagrams arise from resolution graphs by a reduction procedure and conversely resolution graphs arise from splice diagrams by a continued fraction calculation. At present, MAGMA only incorporates the former calculation. However, when a splice diagram s has been constructed using a curve singularity, the corresponding resolution graph g is calculated and can be recovered using the function `CorrespondingResolutionGraph`. The vertices of s correspond to a subset of those of g. The correspondence can be recovered with the function `CorrespondingVertices`.

115.4.1 Splice Diagrams from Resolution Graphs

By default MAGMA always makes the reduced splice diagram since otherwise many determinants would be calculated unnecessarily.

The translation of resolution graph g to splice diagram is done in two steps. First the underlying graph of g is reduced by the removal of all vertices of valency 2 (including arrows in the valency calculation). Then the edge labels are calculated using determinants of subgraphs.

`SpliceDiagram(g)`

`L`	RNGINTELT	*Default :* 0
`K`	RNGINTELT	*Default :* 0
`Reduced`	RNGINTELT	*Default :* 1

A splice diagram of the resolution graph g. All parameters can take the value 0 or 1. If `Reduced` is 1 then the splice diagram will be reduced, otherwise it will be the splice diagram on the underlying graph of g.

The parameter `L` refers to the total linking numbers of the vertices of the resulting splice diagram. The parameter `K` refers to the canonical class of the vertices of the resulting splice diagram. Each quantity will be calculated when the corresponding parameter is 1.

`SpliceDiagram(g,v)`

The splice diagram of the resolution graph g with the condition that the vertex v will not be removed by a reduction.

115.5 Bibliography

[**EN85**] D. Eisenbud and W.D. Neumann. *Three-dimensional link theory and invariants of plane curve singularities*, volume 110 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, NJ, 1985.

[**Neu99**] W.D. Neumann. Irregular links at infinity of complex affine plane curves. *Quart. J. Math. Ox. Ser. (2)*, 50:301–320, 1999.

116 ALGEBRAIC SURFACES

Chapter 116
ALGEBRAIC SURFACES

116.1 Introduction

This chapter contains MAGMA geometric functionality for working specifically with algebraic surfaces and specialised subtypes. We hope to greatly expand this area in upcoming years. The general surface type `Srfc` is for 2-dimensional algebraic varieties over a field (i.e. schemes that are geometrically reduced and irreducible).

The current functionality is largely split between that for (singular) hypersurfaces in ordinary projective 3-space, which is older and relies heavily on the formal desingularisation package of Tobias Beck, and that for ordinary projective surfaces in arbitrary dimensional ambients that are "almost non-singular". The latter relies more on MAGMA's coherent sheaf package. Here, almost non-singular means that only simple (A-D-E) singularities are allowed. These are terminal singularities which don't affect computations involving the pluri-canonical sheaves. It is useful to allow simple singularities as they naturally occur in a number of models (anticanonically-embedded degenerate Del Pezzo surfaces; minimal models for surfaces of general type).

The surface type `Srfc` is a subtype of the scheme type `Sch` so the general functionality for schemes (see Chapter 112) and coherent sheaves (see Chapter 113) is of course also available.

The first section of the chapter deals with functions for surfaces in general ambients although, as noted above, there are singularity assumptions and the restriction to ordinary projective surfaces for many of the intrinsics.

The next section deals with surfaces in $\mathbf{P}^3$ with no singularity assumptions. this contains functions to compute formal desingularizations, general adjoint linear systems, classification and reduction of rational surfaces to special type and a general parametrization routine.

The final section deals with specific code for Del Pezzo surfaces, the specialised subtype for which we currently have the most functionality. There are parametrization routines over the rationals (or number fields in some cases) for Del Pezzos by degree and constructions for degree 6 Del Pezzos associated to different twisted torus type. There are also minimisation and reduction routines for degree 3 and 4 Del Pezzos and construction, point-counting and computation of invariants for degree 3.

116.2 General Surfaces

116.2.1 Introduction

We describe here the newer functionality for surfaces (two-dimensional, geometrically integral schemes over a field) in arbitrary dimensional ambients. However, the reader should be aware that there are major restrictions for many of the intrinsics.

The biggest problems are the singularity assumptions (either non-singular or with at worst simple singularities) and restrictions to ordinary projective space which means that large dimensional ambients are unavoidable at times. These are the general issues most in need of address in future development. Since the time (and memory) for singularity checks can often vastly outweigh the time for the main processing, singularity checks are usually *only* performed when the user explicitly asks for them by setting a parameter value to `true`.

The main functionality is for the computation of fundamental invariants (irregularity, geometric genus etc.), checks for different type of 'non-singularity' (e.g. Gorenstein, only simple singularities), Kodaira-Enriques classification, computation of minimal models (including the full canonical model for a surface of general type) and construction of random surfaces from certain families in $\mathbf{P}^4$.

116.2.2 Creation Functions

As for general schemes and curves, surfaces may be created in any of MAGMA's ambient spaces. However, nearly all of the current specialised surface functionality only applies to surfaces in ordinary projective space.

The requirement for a scheme to be a surface is that it is defined over a field, is of dimension 2 and is geometrically integral (reduced and irreducible when base extended to the algebraic closure of the ground field). Due to the difficulty in checking for *geometric* integrality, at present we only test for integrality (reduced and irreducible over the base field).

See Section 116.4.2 for some specific creation intrinsics for Del Pezzo surfaces and Section 116.4.3 for some additional degree 6 Del Pezzo constructors. Additionally, see Section 116.2.7 for intrinsics to create a range of surfaces in $\mathbf{P}^4$ belonging to special families.

`Surface(A,I)`

`Surface(A,f)`

`Surface(A,S)`

`Nonsingular`	BOOLELT	*Default :* `false`
`Check`	BOOLELT	*Default :* `true`
`Saturated`	BOOLELT	*Default :* `false`

Let A be an ambient or a scheme which already has some defining equations. The function returns the surface defined by the ideal I, the single polynomial f or the sequence of polynomials S within the scheme A.

If I or the set of new defining equations added to those of A generate an ideal that is known to be saturated (*c.f.* Section 112.3), `Saturated` can be set to `true`. If the surface is known to be non-singular or singular, much subsequent calculation

can be avoided by setting `Nonsingular` to `true` or `false`. The parameter `Check` is `true` by default and forces the function to check that the surface is integral (reduced and irreducible) by testing primality of the (saturated) defining ideal. This can be an expensive computation in high-dimensional ambients, so it is best to set `Check` to `false` if it is known in advance that the surface is integral. As stated above, our actual requirement is that surfaces are geometrically integral (equivalently, the surface is integral and the base field is integrally closed in the coordinate ring) because many of the surface intrinsics only really make sense for such varieties. However, this is a more difficult property to test. In practice, integrality should usually imply geometric integrality.

`RationalRuledSurface(P,n)`

Returns a rational, ruled surface X in the ordinary, projective ambient $P = \mathbf{P}^m$ with parameters $n, m-1-n$ where n is the second argument. Such a surface is a rational scroll that can be defined in a number of equivalent ways (see Appendix A2H, [Eis05]).

Let $\mathbf{P}^n$ and $\mathbf{P}^{m-1-n}$ be the linear subspaces of P corresponding to the first $n+1$ coordinates and the last $m-n$ coordinates respectively. Then X is given geometrically as the union of the lines L_{QR} joining a point Q on a rational normal curve in $\mathbf{P}^n$ to a point R on a rational normal curve in $\mathbf{P}^{m-1-n}$, where Q and R correspond under a fixed isomorphism of the first rational normal curve to the second. In the cases $n = 0$ and $n = m-1$, the first or second rational normal curve degenerates to a single point and X is the cone of all lines from a rational normal curve in a hyperplane of P to a point (the apex) outside of the hyperplane. The apex is the only singular point of the surface X (and is not a simple singularity in general). In the non-degenerate cases, X is non-singular. n must always be between 0 and $m-1$ (inclusive).

Following the notation of Section 2, Chapter 5 of [Har77], the rational ruled surface with parameters r, s can also be defined as follows. If e is $\mathrm{Max}(r,s)$-$\mathrm{Min}(r,s)$ and v is $\mathrm{Max}(r,s)$, then X is the Hirzebruch surface X_e (Thm. 2.17, *ibid*) mapped into P via the linear system $|C_0 + v * f|$, which gives an embedding precisely in the non-degenerate cases.

The second return value is a scheme map f from X to $\mathbf{P}^1$ which defines the ruling on X: the fibres of f are all lines in P.

`RandomCompleteIntersection(P,ds)`

`Nonsingular`	BoolElt	*Default :* `true`
`RndP`	RngIntElt	*Default :* 1

This is the same as the general scheme intrinsic to generate a random complete intersection scheme in ordinary projective space $P = \mathbf{P}^m$ over a finite field or the rationals. *ds* should be a sequence of positive integers of length $m-2$. The intrinsic will generate random homogeneous polynomials $F_1, \ldots, F_{m-2}$ of degrees $ds[1], \ldots, ds[m-2]$ in the coordinate ring of P and return the subscheme X of P

with the F_i as defining equations. It is checked that X has dimension 2 (in which case all irreducible components have dimension 2). If parameter `Nonsingular` is set to the default value of `true`, the non-singularity (actually smoothness) is also checked. This guarantees that X is geometrically integral and the result is returned as a surface type `Srfc`. If the check is not performed, X is constructed as a plain `Sch`. If X fails the dimension or non-singularity check, a new set of random polynomials is generated.

If the rationals are the base field, the parameter `RndP` is a positive integer used as an upper absolute bound for random coefficients of polynomials. That is, the algorithm uses random integers between $-RndP$ and $+RndP$ inclusive. The default value here is 1.

`KummerSurfaceScheme(C)`

Returns the Kummer surface of the Jacobian J of the genus 2 hyperelliptic curve C. This is a singular model of the surface: a quartic hypersurface in $\mathbf{P}^3$ with 16 simple "A1" singularities corresponding to the 16 points of order 1 or 2 on J. Its desingularisation is a K3 surface.

Example H116E1

We illustrate the basic creation functions with some simple examples. Firstly, we create a degree 3 (Del Pezzo) and a degree 4 (K3) surface in $\mathbf{P}^3$ directly by giving a defining equation.

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Surface(P3,x^3+y^3+z^3+t^3);
> X;
Surface over Rational Field defined by
x^3 + y^3 + z^3 + t^3
> X := Surface(P3,x^4+y^4+z^4+t^4);
> X;
Surface over Rational Field defined by
x^4 + y^4 + z^4 + t^4
```

We can create a degree 5 Del Pezzo surface in $\mathbf{P}^5$ by specifying it as the projective plane blown up in 4 points (and anti-canonically embedded).

```
> P2<x,y,z> := ProjectiveSpace(Rationals(),2);
> pts := [* P2![1,0,0],P2![0,1,0],P2![0,0,1],P2![1,1,1] *];
> X := DelPezzoSurface(P2,pts);
> P5<x,y,z,s,t,u> := Ambient(X);
> X;
Del Pezzo Surface of degree 5 over Rational Field defined by
-y*z + x*s + s^2 - s*t - s*u + t*u,
-y*s + s^2 + x*t - s*t,
-z*s + s^2 + x*u - s*u,
-s^2 + s*t + y*u - t*u,
```

```
-s^2 + z*t + s*u - t*u
```

The next example is a random surface in $\mathbf{P}^5$ which is the complete intersection of hypersurfaces of degrees 2, 2 and 3

```
> P5<x,y,z,s,t,u> := ProjectiveSpace(Rationals(),5);
> X := RandomCompleteIntersection(P5,[2,2,3]);
> X;
Surface over Rational Field defined by
-y*z-z^2-x*s+y*s+z*s-s^2-x*t-y*t-z*t+s*t+t^2-x*u-y*u-z*u-t*u,
-x^2+z^2-y*s+s^2+x*t+y*t-z*t+s*t+z*u-s*u+t*u-u^2,
-x^3-x^2*y+y^2*z-x*z^2+y*z^2-x^2*s+x*z*s+y*z*s+z^2*s+x*s^2-y*s^2+z*s^2+s^3-x^2*t+
x*z*t-z^2*t-x*s*t-x*t^2+y*t^2+z*t^2+s*t^2-t^3+x*y*u-y^2*u+x*z*u+y*z*u+y*s*u+z*s*u
-s^2*u-x*t*u+y*t*u-z*t*u+s*t*u+t^2*u+x*u^2-y*u^2+t*u^2-u^3
```

The next example is a rational ruled surface in $\mathbf{P}^4$ with parameters $2, 1$. This is a nonsingular surface scroll that is abstractly isomorphic to the Hirzebruch surface X_1 (the plane blown up at one point).

```
> P4<x,y,z,s,t> := ProjectiveSpace(Rationals(),4);
> X := RationalRuledSurface(P4,2);
> X;
Surface over Rational Field defined by
-z*s + y*t,
-y*s + x*t,
-y^2 + x*z
```

Finally, we call one of the intrinsics from the section on surfaces in $\mathbf{P}^4$ to get an Abelian surface (2-dimensional Abelian variety) which is the zero locus of a random global section of the famous Horrocks-Mumford vector bundle. There are 18 defining equations of degrees 5 and 6 that we do not list.

```
> P4<x,y,z,s,t> := ProjectiveSpace(Rationals(),4);
> X := RandomAbelianSurface_d10g6(P4);
> #DefiningPolynomials(X);
18
> [TotalDegree(f) : f in DefiningPolynomials(X)];
[ 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6 ]
```

116.2.3 Invariants

The following functions give standard invariants for projective surfaces with only A-D-E singularities (or slightly weaker assumptions). Due to the current limitations of the cohomology and sheaf machinery, most are only available for ordinary projective surfaces. For corresponding functions that give invariants of the desingularization of hypersurfaces with more general singularities, see Section 116.3.4. Key invariants are stored when computed.

`GeometricGenus(S)`

`CheckGor`	BoolElt	*Default :* `false`
`UseCohom`	BoolElt	*Default :* `false`

The surface S should be an ordinary projective surface which is Gorenstein (this guarantees that a canonical sheaf K exists as a dualising sheaf and is invertible). Returns the geometric genus of S, defined as the dimension of the space of global sections of K, $h^0(K)$.

The boolean parameter `CheckGor` (default `false`) can be set to true to force a check that S is Gorenstein if this isn't already known and stored. By default, the computation computes (and stores) K and then does a direct computation of its global sections. The alternative method is to compute the genus via cohomology since the dimension of H^2 of the structure sheaf is equal to the genus. Set `UseCohom` to `true` to apply the second method. The advantage of the first method is that it is currently faster (in general) and also that K is used in many other intrinsics. Note that unless S is non-singular or has only A-D-E singularities, the genus computed here will generally be larger than the geometric genus of a desingularization of S.

`Plurigenus(S,n)`

`CheckGor`	BoolElt	*Default :* `false`

The surface S and parameter `CheckGor` are as in `GeometricGenus` above. The integer n should be non-negative. Returns the dimension of the space of global sections of the nth tensor power of the canonical sheaf K of S. Again this will generally be larger than the nth plurigenus of a desingularization of S unless S has at worst simple singularities.

`ArithmeticGenus(S)`

Given a scheme S, this function returns the arithmetic genus. It is, in fact, the general scheme intrinsic.

`Irregularity(S)`

`CheckGor`	BoolElt	*Default :* `false`
`UseCohom`	BoolElt	*Default :* `false`

The irregularity q of S, an ordinary projective surface, defined as the dimension of the cohomology group $H^1(S, O_S)$, where O_S is the structure sheaf of S.

If S is known to be Gorenstein or the geometric genus has already been computed and stored, this is computed from the geometric genus p_g and arithmetic genus p_a using the formula $q = p_g - p_a$. Note that S will be known Gorenstein if it is known to be non-singular or to only have simple singularities (All of these properties will have been stored if already tested for. See next section.).

If `CheckGor` is set to `true` (the default is `false`), and the above conditions arent satisfied, Gorensteinness will be checked and, if S is Gorenstein, the above procedure will be followed. Otherwise, the cohomology machinery is used directly.

Setting `UseCohom` to `true` (the default is again `false`) will force the cohomology machinery to be used, unless the value of q has already been computed and stored.

`ChernNumber(S,n)`

`CheckADE` BoolElt *Default :* `false`

The surface S should be ordinary projective with at most simple (A-D-E) singularities. The integer n should be 1 or 2. The singularity condition, if not already known, will only be tested for if `CheckADE` is set to `true` (default is `false`). The function returns the nth Chern number of S_1, the *minimal* desingularization of S. For $n = 1$, this is just the intersection product $K.K$, where K is the canonical sheaf of S_1. Thanks to the singularity condition, this can just be computed on S. For $n = 2$, the Chern number $c_2(S)$ is computed from the relation $c_2(S) + K.K = 12 * (1 + p_a)$, where p_a is the arithmetic genus of S.

`MinimalChernNumber(S,n)`

`CheckADE` BoolElt *Default :* `false`

The surface S should be ordinary projective with at most simple (A-D-E) singularities. The integer n should be 1 or 2. The singularity condition, if not already known, will only be tested for if `CheckADE` is set to `true` (default is `false`). The function computes and returns the relevant Chern number for a minimal model S_2 of a desingularisation S_1 of S. As above, these numbers follow from knowing $K_m.K_m$ where K_m is the canonical sheaf of S_2. If k is the base-field and S is not rational or birationally ruled (i.e. of Kodaira dimension -1), then S_2 is defined over k and is unique up to k-isomorphism. In these cases, $K_m.K_m$ is known from the Kodaira dimension and the second plurigenus in the Kodaira dimension 2 (general type) case. For rational and ruled surfaces, the minimal model is not unique up to isomorphism and a geometrically minimal model may not be defined over k. In these cases, we conventionally take for the invariants a minimal model over the algebraic closure of k with maximal $K_m.K_m$, which is therefore 9 for rational S and 8 for non-rational, ruled S.

`HodgeNumber(S,i,j)`

`CheckADE` BoolElt *Default :* `false`

The surface S should be an ordinary projective with at most simple (A-D-E) singularities. The singularity condition, if not already known, will only be tested for if `CheckADE` is set to `true` (default is `false`).

The integers i, j should be such that $0 \leq i, j \leq 2$. The function returns the Hodge number $h^{i,j}$ of the minimal desingularization S_1 of S which is the dimension of the cohomology group $H^j(S_1, D^i)$ where D^i is the ith alternating power of the sheaf of differentials of S_1. These are computed by formula from the fundamental invariants which are the geometric genus, the irregularity and the first Chern number of S (or S_1).

Example H116E2

We take an easy example: the Kummer surface of the Jacobian of a genus 2 hyperelliptic curve, embedded in $\mathbf{P}^3$ as a degree 4 surface with 16 A_1 singularities lying beneath the 16 points of order 2 on the Jacobian. A nonsingular quartic in $\mathbf{P}^3$ is a K3 surface and simple singularities don't affect the quartic being K3. We verify this here for the Kummer surface, finding that the invariants are the standard invariants for a K3 surface.

```
> f := PolynomialRing(Rationals())![-1,0,0,0,0,0,1]; //t^6-1
> X := KummerSurfaceScheme(HyperellipticCurve(f));
> IsSingular(X);
true
> HasOnlySimpleSingularities(X);
true
> GeometricGenus(X);
1
> ArithmeticGenus(X);
1
> Irregularity(X);
0
> [ChernNumber(X,i) : i in [1,2]];
[ 0, 24 ]
> for i in [0..2], j in [0..2] do
>  printf "%o,%o : %o\n",i,j,HodgeNumber(X,i,j);
> end for;
0,0 : 1
0,1 : 0
0,2 : 1
1,0 : 0
1,1 : 20
1,2 : 0
2,0 : 1
2,1 : 0
2,2 : 1
```

116.2.4 Singularity Properties

This section contains intrinsics for testing for various levels of 'singularity' of a surface. There are further intrinsics applying to more general schemes in Chapter 112 for basic singularity/non-singularity as well as tests for whether a scheme is locally/arithmetically Cohen-Macaulay or locally/arithmetically Gorenstein. The tests here that are currently specific to surfaces are for normality and for having only simple (A-D-E) singularities. All of these properties are stored when computed for a surface/scheme and the various implications between them are used to shortcut tests. The intrinsics below rely on being able to compute the singular subscheme of the surface and having each singular point lying in a constructible affine patch, so they apply to surfaces lying in a wide range of ambients.

`IsNormal(S)`

Returns whether the surface S is a normal variety. The normality test used here consists of checking that the singular subscheme of S is empty or has dimension zero and applying a local normality test at each singular point p (over a splitting field). The local test used is simply whether the depth of the local ring is 2. Taking an affine patch at p and translating to the origin, we simply consider the quotient of the coordinate ring by a non-vanishing coordinate variable and check that the maximal homogeneous ideal is not an associated prime by a straightforward saturation computation. We could have also chosen to use our test for being Cohen-Macaulay once it is known that the singular subscheme of S is zero dimensional.

`IsSimpleSurfaceSingularity(p)`

The point p should be a point in the pointset of a surface S. It is referred to as a simple or A-D-E singularity if it is an isolated singularity on S which is analytically of the type A_n, $n \geq 1$, D_n, $n \geq 4$, E_6, E_7 or E_8 as described in Chapter III, Section 7 of [BHPdV04]. For convenience, if p is non-singular on S, we class it as a simple singularity of type A_0. These are all Gorenstein (even l.c.i) singularities. Their significance is that they are the surface singularities that impose no 'adjunction' condition on the canonical sheaf with respect to computing the canonical sheaf of the minimal desingularization S_1 of S. They all resolve to a collection of (-2)-curves on S_1 whose intersection pairing matrix is the negative of that of the root system with which they share a label.

This intrinsic tests whether p is a simple singularity and, if so, returns the type as a string ("A", "D" or "E") along with the index n (e.g. 6, 7 or 8 for type "E"). It requires that the characteristic of the base field of S is not 2. Also, the E_n types can be a little awkward to analyse in characteristic 3. Therefore in char. 3, the intrinsic always returns `false` if p is a possible E type singularity.

The intrinsic first uses `IsHypersurfaceSingularity` to determine whether p is analytically isomorphic to a hypersurface singularity (which is the case for all simple singularities) and then tests for A-D-E type by examining the expansion of the equation that defines the analytically equivalent singularity.

NB: The intrinsic doesn't fully check that p is an isolated singularity (i.e., that it doesn't lie on a curve in the singular locus of S). It may crash or hang in some cases where p is not isolated.

`HasOnlySimpleSingularities(S)`

`ReturnList` BoolElt *Default :* `false`

This intrinsic determines whether the surface S has no singularities worse than isolated simple singularities as described in the previous intrinsic. Again, the characteristic of the base field of S should not be 2. If S has only simple singularities and `ReturnList` is `true` (the default is `false`), a list is also returned containing triples that consist of each singular point of S (in a pointset over an extension of the base field) along with its type, given as a string and index number as described

previously. In some cases, it may be already known (and recorded internally) that there are only simple singularities without their precise type having been computed. For example, if S is a minimal or weighted canonical model of a surface of general type.

Example H116E3

Anticanonically-embedded *degenerate* Del Pezzo surfaces of degree ≥ 3 are singular but have only simple singularities. We verify this for a degree 4 Del Pezzo which has 2 conjugate (over a quadratic extension) A_1 singularities.

```
> P<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> X := Surface(P,[x*z-y^2, t^2-2*u^2+x^2-2*z^2]);
> HasOnlySimpleSingularities(X : ReturnList := true);
true [* <(0 : 0 : 0 : r1 : 1), "A", 1>, <(0 : 0 : 0 : r2 : 1), "A", 1> *]
> _,lst := $1;
> Ring(Parent(lst[1][1]));
Algebraically closed field with 2 variables over Rational Field
Defining relations:
[
    r2^2 - 2,
    r1^2 - 2
]
```

As a second example, we consider a singular rational ruled surface (scroll) that is the cone over a rational normal curve. The intrinsics tell us that the surface is normal and Cohen-Macaulay (i.e., the local ring at the singular point at the apex of the cone satisfies these properties) but that it satisfies none of the stronger "non-singularity" properties.

```
> P4<x,y,z,t,u> := ProjectiveSpace(Rationals(),4);
> X := RationalRuledSurface(P4,0);
> // one singular point
> Degree(ReducedSubscheme(SingularSubscheme(X)));
1
> Support(SingularSubscheme(X));
{ (1 : 0 : 0 : 0 : 0) }
> HasOnlySimpleSingularities(X);
false
> IsArithmeticallyGorenstein(X);
false
> IsGorenstein(X);
false
> IsCohenMacaulay(X);
true
> IsNormal(X);
true
```

116.2.5 Kodaira-Enriques Classification

`KodairaEnriquesType(S)`

`CheckADE` BoolElt *Default :* `false`

The argument S is a surface in ordinary projective space having at most simple (A-D-E) singularities. As it may be a very heavy computation, the latter is only checked if the user sets the `CheckADE` parameter is set to `true` (the default is `false`).

The function computes the type of S (or rather, of the non-singular projective surfaces in its birational equivalence class) according to the classification of Kodaira and Enriques.

The first number returned is the *Kodaira dimension* of S, which is -1, 0, 1, or 2. We use -1 here rather than $-\infty$. A second return value further specifies the type within the Kodaira dimension -1 or 0 cases (and is irrelevant in the other two cases).

Kodaira dimension -1 corresponds to birationally ruled surfaces. The second number returned in this case is the irregularity $q \geq 0$ of S. So S is birationally equivalent to a ruled surface over a smooth curve of genus q and is a *rational* surface if and only if q is zero.

Kodaira dimension 0 corresponds to surfaces which are birationally equivalent to a K3 surface, an Enriques surface, a torus or a bi-elliptic surface. In the final case, there is a partial subclassification in that the canonical sheaf of the minimal model is a torsion sheaf of order r, where r is 2, 3, 4, or 6. The second integer return value in the Kodaira dimension zero case codes the subtypes as follows:-

-3	Enriques surface
-2	K3 surface
-1	Torus
r	$r = 2, 3, 4$ or 6. Bielliptic surface of subtype r

A third return value is a string that gives a verbal description of the surface type (e.g. "Rational" or "Bi-elliptic (type 3)"). Kodaira dimension 1 surfaces are labelled as "Elliptic fibration" (which they all are – though there are also surfaces of Kodaira dimension less than 1 which have elliptic fibrations) and Kodaira dimension 2 surfaces are labelled as "General type", as is traditional.

There are no built-in restrictions on the characteristic of the base field, but there are some special cases for surfaces of Kodaira dimension 0 in characteristics 2 and 3 that may not be dealt with properly.

The function works by computing a number of the invariants of S and sometimes also considering the dimension of the image of appropriate pluri-canonical maps. We try to compute the least number of invariants to fully determine the type. A useful by-product is that, after calling this function, a number of the surface invariants (always including the geometric genus and irregularity) will have been computed and stored for later use. The type information is also stored.

`KodairaEnriquesDimension(S)`

`CheckADE` BoolElt *Default :* `false`

The argument S is a surface in ordinary projective space having at most simple (A-D-E) singularities. The later condition will only be checked if the parameter `CheckADE` is set to `true`. The function simply returns the Kodaira dimension (-1,0,1,2) without further type information. In most cases it does the same amount of work as to compute the full Kodaira-Enriques type, unless the result has already been determined and stored.

Example H116E4

We will present further cases of Kodaira-Enriques typing in our minimal model examples. For now, we just give two simple examples: a Veronese surface in $\mathbf{P}^5$ and the Kummer surface in $\mathbf{P}^3$ with simple singularities from our earlier example.

The scheme X is a Veronese surface, isomorphic to P^2:

```
> P<a,b,c,d,e,f> := ProjectiveSpace(Rationals(),5);
> X := Surface(P,[b^2-a*c, a*d-b*f, b*d-c*f, d^2-c*e, a*e-f^2, b*e-d*f]);
> // X is a Veronese surface, isomorphic to P^2
> KodairaEnriquesType(X);
-1 0 Rational
```

The scheme X is a singular K3 surface:

```
> P<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> X := Surface(P,x^3*t+x^2*z^2-8*x*y^2*z-x*z*t^2+16*y^4+y^2*t^2-z^3*t);
> KodairaEnriquesType(X);
0 -2 K3
```

116.2.6 Minimal Models

In contrast to the curve case, a birational equivalence class of surfaces contains an infinite number of non-isomorphic projective, non-singular surfaces. Any two such surfaces are linked by a birational map that consists of a sequence of blowing up points and blowing down exceptional curves (rational (-1)-curves). For any non-singular, projective surface, a sequence of blow downs of exceptional curves will result in a surface with no more exceptional curves after a finite number of steps. Such a surface is referred to as a *minimal model*. It is also possible to further contract connected cycles of rational (-2)-curves to simple singularities. Sometimes minimal model also refers to a surface on which these contractions have been performed. This is particularly true for surfaces of general type where the pluri-canonical models are minimal in this second sense.

For surfaces of Kodaira dimension greater than or equal to zero, there is a unique minimal model (up to isomorphism) within the birational equivalence class. That is, the minimisation procedure of blowing down exceptional curves will always lead to the same thing starting with any non-singular projective surface within that class. This can be

carried out over an arbitrary base field and the minimal model is a unique representative of the class, which partly explains its importance.

For rational or ruled surfaces, there is not a unique minimal model. Over an algebraically closed field, the minimal models in these cases are the projective plane and the *geometrically ruled surfaces* which are fibrations over a non-singular, projective base curve C, all of whose fibres are irreducible curves isomorphic to the projective line. The rational surfaces are those with C rational (in this case, one of the ruled surfaces is not minimal but is the plane blown up in one point). Over a non-algebraically closed base field k, it may not be possible to blow down all exceptional curves working over k and so there are k-minimal surfaces (certain Del Pezzo surfaces, for example) that are not minimal over the algebraic closure. Models like Del Pezzo surfaces that are close to minimal but may not strictly even be minimal over k are still very important for rational and ruled surfaces because they allow the reduction to a small class of standard isomorphism types. We can think of these as quasi-minimal.

This section describes functions which are designed to construct minimal models or quasi-minimal models of ordinary projective surfaces. The precise meaning of this varies a little depending on the Kodaira dimension of S, so there are distinct functions for the different dimensions. The Kodaira dimension, if unknown, can be determined by use of the intrinsics in the previous section.

More of these intrinsics really should work for surfaces with simple singularities - at least, if the user is happy with a result that also has simple singularities. For the moment, except for surfaces of general type (Kodaira dimension 2), we require S to be non-singular.

The output minimal models are all non-singular except for surfaces of general type where all (-2)-curves are contracted to simple singularities, as is traditional. There is also an intrinsic to compute the full canonical model (which lies in weighted projective space in general) and the canonical coordinate ring of a surface of general type. With the Kodaira dimension 1 minimal models, the user can optionally ask for a map to a smooth projective curve C that presents the minimal model M as an elliptic fibration over C. In this case, M is the global arithmetic minimal model of its generic fibre.

`MinimalModelRationalSurface(S)`

`CheckSing` BoolElt *Default :* `false`

Let S should be a non-singular ordinary projective rational (Kodaira dimension -1 and irregularity 0) surface. Non-singularity is not checked by default so to force a check, set `CheckSing` to `true`. It is also left to the user to check that S is rational (using Section KodairaEnriquesType for example).

The intrinsic does not strictly compute a minimal model M (there may be (-1)-curves that can still be blown down over the base field), but instead it produces a standard model that is terminal for the adjunction process.

This means that M will be a member of a small family of special rational surfaces: the projective plane or its Veronese embedding in $\mathbf{P}^5$, an anticanonically embedded Del Pezzo surface, a rational scroll or a conic bundle. Other functions for special surfaces may then be applied to M – to parametrize it, for example.

The return value is a rational map f from S onto M (M may be recovered as the codomain of f) which will be of type `MapSchGrph` or `MapSch`. The implementation proceeds by simply iterating the adjunction map until simple termination criteria are recognised.

Example H116E5

We take one of the family of non-singular rational surfaces in $\mathbf{P}^4$ that can be generated by the `RandomRationalSurface_d10g9` intrinsic to be described later. These surfaces are very far from minimal. They have exceptional curves of degrees 3, 2 and 1 in the $\mathbf{P}^4$ embedding and the theory tells us that we need 2 adjunction maps to reduce the surface to a degree 5 Del Pezzo in $\mathbf{P}^5$, which is terminal. The degree of non-minimality is measured by the first Chern number $K.K$, with $9 - K.K$ or $8 - K.K$ telling us how many point blowups it takes to get from a geometric minimal model (over $\bar{k}$) to the surface, depending on whether the geometric minimal model is $\mathbf{P}^2$ or a rational ruled surface. Here, we start at -9 and get to 5 for the Del Pezzo, which is $\mathbf{P}^2$ blown up at 4 points over the algebraic closure of the base field. We choose to work over a finite field so that the example completes quickly.

```
> k := GF(37);
> P := ProjectiveSpace(k,4);
> X := RandomRationalSurface_d10g9(P);
> #DefiningPolynomials(X);
11
> [TotalDegree(f): f in DefiningPolynomials(X)];
[ 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
> // X is defined by a quartic and 10 quintics
> ChernNumber(X,1);
-9
> mp := MinimalModelRationalSurface(X);
> Y := Codomain(mp);
> Y;
Surface over GF(37) defined by
y[2]*y[3]+23*y[3]^2+4*y[1]*y[4]+29*y[2]*y[4]+34*y[3]*y[4]+36*y[4]^2+
 18*y[1]*y[5]+22*y[2]*y[5]+y[3]*y[5]+31*y[4]*y[5]+3*y[5]^2+33*y[1]*y[6]+
 6*y[2]*y[6]+31*y[3]*y[6]+20*y[4]*y[6]+28*y[5]*y[6]+33*y[6]^2,
y[1]*y[3]+27*y[3]^2+25*y[1]*y[4]+y[2]*y[4]+19*y[3]*y[4]+34*y[4]^2+13*y[1]*y[5]+
 15*y[2]*y[5]+14*y[3]*y[5]+15*y[4]*y[5]+17*y[5]^2+21*y[1]*y[6]+18*y[2]*y[6]+
 33*y[3]*y[6]+9*y[4]*y[6]+27*y[5]*y[6]+12*y[6]^2,
y[2]^2+13*y[1]*y[4]+10*y[2]*y[4]+9*y[3]*y[4]+26*y[4]^2+32*y[1]*y[5]+27*y[2]*y[5]+
 18*y[4]*y[5]+33*y[5]^2+27*y[1]*y[6]+16*y[2]*y[6]+31*y[3]*y[6]+35*y[4]*y[6]+
 24*y[5]*y[6]+21*y[6]^2,
y[1]*y[2]+26*y[3]^2+16*y[1]*y[4]+23*y[2]*y[4]+24*y[3]*y[4]+32*y[4]^2+27*y[1]*y[5]+
 14*y[2]*y[5]+10*y[3]*y[5]+12*y[4]*y[5]+33*y[5]^2+16*y[1]*y[6]+26*y[2]*y[6]+
 7*y[3]*y[6]+13*y[4]*y[6]+11*y[5]*y[6]+y[6]^2,
y[1]^2+5*y[3]^2+19*y[1]*y[4]+35*y[2]*y[4]+2*y[3]*y[4]+20*y[4]^2+29*y[1]*y[5]+
 27*y[2]*y[5]+22*y[3]*y[5]+14*y[4]*y[5]+6*y[5]^2+32*y[1]*y[6]+31*y[2]*y[6]+
 2*y[3]*y[6]+36*y[5]*y[6]+33*y[6]^2
> Ambient(Y); Degree(Y);
```

```
Projective Space of dimension 5 over Finite field of size 37
Variables: y[1], y[2], y[3], y[4], y[5], y[6]
5
> ChernNumber(Y,1);
5
```

`MinimalModelRuledSurface(S)`

`CheckSing` BoolElt *Default :* `false`

The surface S should be non-singular ordinary projective of Kodaira dimension -1. Non-singularity is not checked by default so to force a check, set `CheckSing` to `true`. It is also left to the user to check that S is of the correct Kodaira dimension (using `KodairaEnriquesType` for example).

If S is rational, the intrinsic `MinimalModelRationalSurface` is applied. For non-rational ruled surfaces, the same adjunction procedure is applied to lead to a terminal model M, which is either a non-rational scroll and genuinely minimal, a conic bundle over a non-rational curve that may not be strictly minimal (it may have degenerate fibres that are the intersecting unions of two (-1)-curves) or a non-split minimal ruled surface over a genus one curve embedded as a degree 9 surface in $\mathbf{P}^6$ in such a way that the fibres of the ruling have degree 3.

The return value is a rational map f from S onto M (M may be recovered as the codomain of f) and will be of type `MapSchGrph` or `MapSch`.

`MinimalModelKodairaDimensionZero(S)`

`CheckSing` BoolElt *Default :* `false`

The surface S should be non-singular ordinary projective of Kodaira dimension 0. Non-singularity is not checked by default so to force a check, set `CheckSing` to `true`. It is also left to the user to check that S is of the correct Kodaira dimension (using `KodairaEnriquesType` for example).

The function computes a minimal model M of S, again by repeatedly applying the adjunction map until minimality occurs (the first Chern number is zero).

The return value is a rational map f from S onto M (M may be recovered as the codomain of f) and will be of type `MapSchGrph` or `MapSch`.

Example H116E6

We start with an example of a non-minimal surface X that is a torus T blown up in one point. Such examples naturally occur when T is the Jacobian J of a genus 2 curve C. The product of the C with itself quotiented by an appropriate involution is such an X. It has a natural embedding in $\mathbf{P}^7$. This turns out to be the projection from the zero point of J embedded in $\mathbf{P}^8$ (by three times the theta divisor).
We work over a finite field for speed (although the example doesn't take too long over the rationals). Our example corresponds to the curve C with Weierstrass equation $y^2 = x^5 - 1$. The

minimal model routine has the effect of *unprojecting* here: it recovers the torus J in its $\mathbf{P}^8$ embedding.

```
> P7<z1,z2,z3,z4,z5,z6,z7,z8> := ProjectiveSpace(GF(37),7);
> X := Surface(P7,
> [
> z3*z4-z2*z5+z1*z6,
> 1/2*z3^2-1/2*z2*z7+z1*z8,
> z1^2+z5^2-z4*z6-2*z3*z7+z2*z8,
> -z3*z5*z6+z2*z6^2+z5^2*z7-z4*z6*z7-z3*z7^2-1/2*z3^2*z8+1/2*z2*z7*z8,
> z3*z4*z6+1/2*z3^2*z7-z4*z5*z7+1/2*z2*z7^2+z2*z3*z8,
> -z4^2*z5+z2^2*z6+z6^3+z2*z4*z7+z6*z7*z8-z5*z8^2,
> -1/2*z3^2*z5-z4*z5^2+z2*z3*z6+z4^2*z6+z3*z4*z7+1/2*z2*z5*z7-z2*z4*z8,
> z1*z2*z5+z5^2*z6-z4*z6^2-z3*z6*z7-z5*z7^2+z3*z5*z8-z2*z6*z8+z4*z7*z8,
> -z4^3+z2^2*z5+z5*z6^2-z6*z7^2+2*z5*z7*z8-z4*z8^2,
> -z3*z4^2+z2^2*z7+z6^2*z7-z3*z8^2,
> -z2*z4^2+z1*z2*z7+z5*z6*z7+z3*z7*z8-z2*z8^2,
> -z2*z4^2+z1*z4*z5+z3*z6^2+z5^2*z8-z4*z6*z8,
> z1*z4^2-z3*z7^2-1/2*z3^2*z8+1/2*z2*z7*z8,
> 1/2*z3^2*z4-z2*z3*z5+z4^2*z5-z6^3-3/2*z2*z4*z7+z1*z5*z7-z6*z7*z8+z5*z8^2,
> -z2*z3*z4+z1*z4*z7+z6*z7^2+z3*z6*z8-z5*z7*z8,
> z2^2*z4-z5*z7^2-z2*z6*z8+z4*z7*z8,
> z1*z2*z4+1/2*z3^2*z6-z3*z5*z7+1/2*z2*z6*z7,
> -1/2*z3^3-z3*z4*z5+2*z2*z4*z6+3/2*z2*z3*z7-z4^2*z7+z1*z7^2,
> 1/2*z3^3+z3*z4*z5-z2*z4*z6-3/2*z2*z3*z7+z2^2*z8,
> -1/2*z2*z3^2-3/2*z3*z4^2+z2*z4*z5+z1*z3*z7+1/2*z6^2*z7-1/2*z3*z8^2,
> z2^2*z3-z2*z4^2+z3*z6^2+2*z3*z7*z8-z2*z8^2,
> z1*z2*z3+z3*z5*z6-z4*z6*z7-z3*z7^2+z3^2*z8+z2*z7*z8,
> z2^3+z3*z5*z6-z5^2*z7+1/2*z3^2*z8+1/2*z2*z7*z8,
> z1*z2^2-z3*z4*z6+z2*z5*z6-z3^2*z7
> ] : Check:= false);
> KodairaEnriquesType(X);
0 -1 Torus
> mp := MinimalModelKodairaDimensionZero(X);
> Y := Codomain(mp);
> Ambient(Y);
Projective Space of dimension 8 over Rational Field
Variables: y[1], y[2], y[3], y[4], y[5], y[6], y[7], y[8], y[9]
> Y;
Surface over Rational Field defined by
y[4]*y[5] - y[3]*y[6] + y[2]*y[7],
y[4]^2 - y[3]*y[8] + 2*y[2]*y[9],
y[3]*y[4] + y[5]^2 + y[1]*y[7] - 2*y[2]*y[8] + y[9]^2,
y[1]*y[4] + 2*y[2]*y[5] + y[7]*y[8],
y[3]^2 - 1/3*y[2]*y[4] + 1/3*y[1]*y[6] + 1/3*y[7]^2 + 1/3*y[8]*y[9],
y[2]*y[3] + y[1]*y[5] + y[6]*y[7] + y[8]^2 + 2*y[4]*y[9],
y[1]*y[3] - y[4]*y[7] + 2*y[6]*y[8],
y[2]^2 + y[6]^2 - y[5]*y[7] - 2*y[4]*y[8] + y[3]*y[9],
```

```
y[1]*y[2] + y[4]*y[6] - 2*y[3]*y[7] + y[5]*y[8],
y[3]*y[5]*y[7] - y[5]^2*y[8] + y[2]*y[8]^2 + 1/3*y[2]*y[4]*y[9] -
  1/3*y[1]*y[6]*y[9] - 1/3*y[7]^2*y[9] - 1/3*y[8]*y[9]^2,
y[2]*y[5]*y[7] + 2/3*y[2]*y[4]*y[8] + 1/3*y[1]*y[6]*y[8] +
  1/3*y[7]^2*y[8] - y[1]*y[5]*y[9] - y[6]*y[7]*y[9] - 2/3*y[8]^2*y[9] -
  2*y[4]*y[9]^2,
y[2]*y[5]*y[6] + y[4]*y[7]^2 + y[1]*y[5]*y[8] + y[8]^3 + y[6]^2*y[9] -
  y[5]*y[7]*y[9] + y[4]*y[8]*y[9] + y[3]*y[9]^2
> MinimalChernNumber(X,1)-ChernNumber(X,1);
1
> ChernNumber(Y,1);
0
```

So we have one exceptional line blown down and Y is now minimal.

Example H116E7

For our second example, we take a surface of Enriques type in $\mathbf{P}^4$ which comes from one of the special families of $\mathbf{P}^4$ surfaces to be described later. Such surfaces are non-minimal, but isomorphic to an Enriques surface blown up at one point. As we are choosing a random surface from a family which will be defined by many non-sparse equations, we work over a finite field for speed.

```
> k := GF(37);
> P := ProjectiveSpace(k,4);
> X := RandomEnriquesSurface_d9g6(P);
> #DefiningPolynomials(X);
15
> [TotalDegree(f): f in DefiningPolynomials(X)];
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
> //surface defined by 15 quintics
> ChernNumber(X,1);
-1
> mp := MinimalModelKodairaDimensionZero(X);
> Y := Codomain(mp);
> #DefiningPolynomials(Y);
10
> [TotalDegree(f): f in DefiningPolynomials(Y)];
[ 3, 3, 3, 3, 3, 3, 3, 3, 3, 3 ]
> // minimal model Y defined by 10 cubics
> Ambient(Y); Degree(Y);
Projective Space of dimension 5 over Finite field of size 37
Variables: y[1], y[2], y[3], y[4], y[5], y[6]
10
> ChernNumber(Y,1);
0
```

`MinimalModelKodairaDimensionOne(S)`

`CheckSing`	BoolElt	*Default :* `false`
`Fibration`	BoolElt	*Default :* `false`

Let S be a non-singular ordinary projective surface of Kodaira dimension 1. Non-singularity is not checked by default and to force a check, set `CheckSing` to `true`. It is also left to the user to check that S is of the correct Kodaira dimension (using `KodairaEnriquesType` for example).

The function computes a minimal model M of S. Such a model always has a connected elliptic fibration to a smooth projective curve C (i.e. there is a morphism g from M onto C such that the general fibre of g is a smooth curve of genus 1). If `Fibration` is set to `true` (the default is `false`), the function also computes g and returns it as a second return value.

The first return value is a rational map f from S onto M (M may be recovered as the codomain of f) and will be of type `MapSchGrph` or `MapSch`.

The computation of f again proceeds by repeating adjunction until the first Chern number is zero. There is a slight speed-up here, though, that we apply when S has positive geometric genus. We use the appropriate modification of the image of an effective canonical divisor to compute a new canonical divisor at each stage of adjunction. Although this doesn't usually speed up the computation of the next adjunction map, it can greatly increase the speed of computation of the new first Chern number, which speeds up testing of the termination criterion.

If required, g may be computed by using an appropriate small multiple of the pluricanonical map on M.

`MinimalModelGeneralType(S)`

`CanonicalWeightedModel(S)`

`CheckADE`	BoolElt	*Default :* `false`

The surface S should be ordinary projective of general type (Kodaira dimension 2) with at worst simple (A-D-E) singularities. The singularity condition is not checked by default and to force a check, set `CheckADE` to `true`. It is also left to the user to check that S is of general type (using `KodairaEnriquesType` for example).

The intrinsics construct a minimal model M for S which is of the canonical type: it has any (-2)-curves contracted to simple singularities.

The first intrinsic produces M as an ordinary projective surface as before, which is an m-canonical embedding with m equal to 3 generally but 4 or 5 when S has certain small invariants. This is implemented simply by computing the m-canonical map on S (this map automatically factors through a non-singular minimal model). The drawback of this is that M is usually defined in a high-dimensional projective space. There is a very simple check to see whether the canonical sheaf or the bi-canonical sheaf is very ample, in which case M is just taken as S.

The second intrinsic actually computes the *full* canonical model of S. This is a (generally weighted) projective model of M equal to *Proj* of the canonical coordinate

ring $\sum_{n=0}^{\infty} H^0(S, K_S^{\otimes n})$, where K_S is the canonical sheaf of K. This is more intrinsic and it generally gives a model in a much lower-dimensional space with the drawback that this space is now weighted. The cases where the canonical or bicanonical image is actually isomorphic to M will show up in this version whereas it will produce an actual ordinary projective model in the first case. Currently, this intrinsic has the small restriction that S cannot have geometric genus (and irregularity) zero. The implementation involves calculating with Riemann-Roch spaces for small multiples of an effective canonical divisor.

For either intrinsic, the first return value is a rational map f from S onto M (M may be recovered as the codomain of f) which will be of type `MapSchGrph` or `MapSch` for the first intrinsic and of type `MapSch` only for the second. A second return value is a boolean with value `true` if and only if the original model X is minimal in the sense of containing no (-1)-curves.

`CanonicalCoordinateIdeal(S)`

`CheckADE` BoolElt *Default :* `false`

The surface S should be ordinary projective of general type (Kodaira dimension 2) with at worst simple (A-D-E) singularities. The singularity condition is not checked by default and to force a check, set `CheckADE` to `true`. It is also left to the user to check that S is of general type (using `KodairaEnriquesType` for example).

This intrinsic constructs the full canonical coordinate ring of S as described for `CanonicalWeightedModel`. What is returned is a homogeneous ideal I in a polynomial ring R with variable weightings over the base field of S such that the canonical coordinate ring is isomorphic to the quotient R/I as a graded ring.

Example H116E8

We give two examples of computing minimal models for some surfaces of general type. We begin with a non-singular type I Horikawa surface with first Chern number K^2 equal to 3. It is already minimal and the bi-canonical map gives an embedding into $\mathbf{P}^6$. We start with a bicanonical model. The first intrinsic luckily recognises that the surface is bi-canonical and just returns it as a minimal model. The second intrinsic finds the well-known weighted-projective embedding for such surfaces as a sextic hypersurface in a $\mathbf{P}(1,1,1,2)$ weighted projective space. (see Section 9, Chapter VII, [BHPdV04] or [Hor76]).

```
> P<x1,x2,x3,x4,x5,x6,x7> := ProjectiveSpace(Rationals(),6);
> X := Surface(P,[
>  -x4*x5 + x2*x6, x1*x5 - x4*x6,
>   x3*x4 - x5*x6, x2*x3 - x5^2,
>   x1*x3 - x6^2, x1*x2 - x4^2,
>   x1^3+x1*x2^2+x2^2*x3-x2*x3^2+x3^3+x2^2*x4+x1*x3*x4-x2*x3*x4+x1*x4^2-
>    x2*x4^2-x3*x4^2-x1*x2*x5-x1*x3*x5+x2*x3*x5+x1*x4*x5-x2*x4*x5+x4^2*x5-
>    x1*x5^2+x2*x5^2-x3*x5^2-x5^3-x1^2*x6+x1*x2*x6-x2*x3*x6-x2*x4*x6+
>    x1*x5*x6-x3*x5*x6-x4*x5*x6+x5^2*x6+x1*x6^2+x2*x6^2-x3*x6^2+x5*x6^2-
>    x1*x2*x7+x1*x3*x7+x2*x3*x7-x3^2*x7+x1*x4*x7-x2*x4*x7-x4^2*x7+x5^2*x7+
>    x2*x6*x7+x3*x6*x7-x4*x6*x7+x5*x6*x7-x6^2*x7+x3*x7^2+x4*x7^2+x5*x7^2-
```

```
>    x6*x7^2-x7^3 ]);
> IsSingular(X);
false
> KodairaEnriquesType(X);
2 0 General type
> mp := MinimalModelGeneralType(X);
> X1 := Codomain(mp); //the minimal model
> X1 eq X;
true
> mp1 := CanonicalWeightedModel(X);
> Y := Codomain(mp1);
> PW<a,b,c,d> := Ambient(Y);
> PW;
Projective Space of dimension 3 over Rational Field
Variables: a, b, c, d
The grading is:
    1, 1, 1, 2
```

The fact that the full weighted canonical model has only weights 1 and 2 also shows that the bi-canonical map is an embedding.

```
> Y;
Surface over Rational Field defined by
a^6 + a^4*b^2 + a*b^5 - a^5*c + 2*a^3*b^2*c - a^2*b^3*c - a*b^4*c + a^4*c^2 +
 2*a^3*b*c^2 - 2*a^2*b^2*c^2 - a*b^3*c^2 + 2*b^4*c^2 - a^2*c^4 - a*b*c^4 -
 2*b^2*c^4 + c^6 + a^3*b*d - 2*a^2*b^2*d - a*b^3*d - a^2*b*c*d + a*b^2*c*d +
 a*b*c^2*d + 2*b^2*c^2*d + a*c^3*d - c^4*d + a*b*d^2 - a*c*d^2 + b*c*d^2 +
 c^2*d^2 - d^3
```

We can also ask for the canonical coordinate ideal that defines the canonical coordinate ring which is the coordinate ring of Y. The call takes no time as the canonical weighted model Y and map `mp1` from X to Y have been stored internally.

```
> time CanonicalCoordinateIdeal(X);
Ideal of Graded Polynomial ring of rank 4 over Rational Field
Order: Grevlex with weights [1, 1, 1, 2]
Variables: a, b, c, d
Variable weights: [1, 1, 1, 2]
Homogeneous, Dimension >0
Groebner basis:
[
a^6 + a^4*b^2 + a*b^5 - a^5*c + 2*a^3*b^2*c - a^2*b^3*c - a*b^4*c + a^4*c^2 +
 2*a^3*b*c^2 - 2*a^2*b^2*c^2 - a*b^3*c^2 + 2*b^4*c^2 - a^2*c^4 - a*b*c^4 -
 2*b^2*c^4 + c^6 + a^3*b*d - 2*a^2*b^2*d - a*b^3*d - a^2*b*c*d + a*b^2*c*d +
 a*b*c^2*d + 2*b^2*c^2*d + a*c^3*d - c^4*d + a*b*d^2 - a*c*d^2 + b*c*d^2 +
 c^2*d^2 - d^3
]
Time: 0.000
```

Example H116E9

This example is a non-singular but non-minimal surface. We start with a degree 5 hypersurface in $\mathbf{P}^3$ and then blow up a point using the standard intrinsic to give a model X of the blow-up as a surface in $\mathbf{P}^8$. The original hypersurface is a (simply-weighted) canonical model of X and `WeightedCanonicalModel` does just return a slight linear transformation of it. The computation takes a little time.

```
> P3<x,y,z,t> := ProjectiveSpace(Rationals(),3);
> Y := Surface(P3,x^5+y^5+z^5+t^5 : Nonsingular := true); //the hypersurface
> X := BlowUp(Y,Y![0,0,-1,1]);
> P<x1,x2,x3,x4,x5,x6,x7,x8,x9> := Ambient(X);
> X;
Surface over Rational Field defined by
-x6*x8+x5*x9, -x3*x8+x2*x9, -x1*x8+x3*x9, -x6*x7+x4*x9, -x5*x7+x4*x8,
-x3*x7+x5*x9+x8*x9, -x2*x7+x5*x8+x8^2, -x1*x7+x6*x9+x9^2,
-x3*x5+x2*x6, -x1*x5+x3*x6, -x3*x4+x5*x6+x6*x8, -x2*x4+x5^2+x5*x8,
-x1*x4+x6^2+x6*x9, -x1*x2+x3^2,
x1^4+x2*x3^3+x4*x6^3-x4*x6^2*x9+x4*x6*x9^2-x4*x9^3+x7*x9^3,
x1^3*x3+x2^2*x3^2+x4*x5*x6^2-x4*x5*x6*x9+x4*x5*x9^2-x4*x8*x9^2+x7*x8*x9^2,
x2*x3^2*x5+x1^3*x6+x4^2*x6^2+x2*x3^2*x8+x1^3*x9-x4^2*x6*x9+x4^2*x9^2-
x4*x7*x9^2+x7^2*x9^2,
x2^3*x3+x1^2*x3^2+x4*x5^2*x6-x4*x5^2*x9+x4*x5*x8*x9-x4*x8^2*x9+x7*x8^2*x9,
x2^2*x3*x5+x1^2*x3*x6+x4^2*x5*x6+x2^2*x3*x8+x1^2*x3*x9-x4^2*x5*x9+x4^2*x8*x9-
x4*x7*x8*x9+x7^2*x8*x9,
x2*x3*x5^2+x4^3*x6+x1^2*x6^2+2*x2*x3*x5*x8+x2*x3*x8^2-x4^3*x9+2*x1^2*x6*x9+
x4^2*x7*x9-x4*x7^2*x9+x7^3*x9+x1^2*x9^2,
x2^4+x1*x3^3+x4*x5^3-x4*x5^2*x8+x4*x5*x8^2-x4*x8^3+x7*x8^3,
x2^3*x5+x4^2*x5^2+x1*x3^2*x6+x2^3*x8-x4^2*x5*x8+x4^2*x8^2-x4*x7*x8^2+
x7^2*x8^2+x1*x3^2*x9,
x4^3*x5+x2^2*x5^2+x1*x3*x6^2-x4^3*x8+2*x2^2*x5*x8+x4^2*x7*x8-x4*x7^2*x8+
x7^3*x8+x2^2*x8^2+2*x1*x3*x6*x9+x1*x3*x9^2,
x4^4+x2*x5^3+x1*x6^3-x4^3*x7+x4^2*x7^2-x4*x7^3+x7^4+3*x2*x5^2*x8+
3*x2*x5*x8^2+x2*x8^3+3*x1*x6^2*x9+3*x1*x6*x9^2+x1*x9^3
> KodairaEnriquesType(X);
2 0 General type
> mp, is_min := CanonicalWeightedModel(X);
> is_min;
false
> X1 := Codomain(mp); //the canonical model
> P<a,b,c,d> := Ambient(X1);
> X1;
Surface over Rational Field defined by
a^5 + b^5 + c^5 + 5*c^4*d + 10*c^3*d^2 + 10*c^2*d^3 + 5*c*d^4 + 2*d^5
> MinimalChernNumber(X,1) - ChernNumber(X,1);
1
```

The last line confirms that there was one exceptional divisor blown down.

116.2.7 Special Surfaces in Projective 4-space

There has been much study of the families of nonsingular surfaces of non-general type that lie in $\mathbf{P}^4$ (a complete intersection of two nonsingular hypersurfaces of degrees d and e – the obvious construction – leads to general-type surfaces unless $d + e \leq 5$). These families are a very useful tool for generating random surfaces of a particular Kodaira-Enriques type with a non-singular ordinary projective model lying in a small-dimensional ambient.

Decker, Ein and Schreyer have given many such families in their paper [DES93]. Their constructions make use of the theory of codimension two schemes and the Beilinson spectral sequence for coherent sheaves. The result is that the defining ideal of a surface in the family is isomorphic as a graded module to the cokernel of a random map between two modules over $k[x_0, \ldots, x_4]$ that are direct sums of twists of the modules representing alternating powers of the sheaf of differentials on $\mathbf{P}^4$ (or something slightly more complex derived from these modules). The ideals constructed give Cohen-Macaulay surfaces and the surface is actually non-singular for a general map. The families are generally described by the degree, sectional genus (arithmetic genus of a hyperplane section) and Kodaira-Enriques type of the surfaces they contain. Given the type, knowing the degree and sectional genus is equivalent to knowing the Hilbert polynomial.

Intrinsics are provided to generate a random surface from a selection of the families described in the paper above, following the implementations in the Macaulay 2 computer algebra package.

`RandomRationalSurface_d10g9(P)`

`RndP`	RngIntElt	*Default :* 2
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. The intrinsic generates a random (non-minimal) degree 10 rational surface with sectional genus 9 in P from the family described by Ranestad. If the rationals are the base field, the parameter `RndP` is a positive integer used as an upper absolute bound for random coefficients of polynomials. That is, the algorithm uses random integers between $-RndP$ and $+RndP$m, inclusive. The default value here is 2. Parameter `Check` being `true` (the default) means that the random prospective surfaces generated are tested for irreducibility and non-singularity and rejected if they fail (the process tries 10 surfaces before giving up). The user can set this to `false` for a slight speed-up: it is rare that a singular surface occurs.

`RandomEnriquesSurface_d9g6(P)`

`RndP`	RngIntElt	*Default :* 2
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (non-minimal) degree 9 Enriques surface with sectional genus 6 in P. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

`RandomAbelianSurface_d10g6(P)`

`RndP`	RngIntElt	*Default :* 5
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random degree 10 torus with sectional genus 6. This is just the zero section of an element of the Horrocks-Mumford bundle. The parameters have the same meaning as for `RandomRationalSurface_d10g9` although the default for `RndP` is now 5.

`RandomEllipticFibration_d7g6(P)`

`RndP`	RngIntElt	*Default :* 2
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (minimal) degree 7 elliptic surface with sectional genus 6 in P. These have Kodaira dimension 1, geometric genus 2 and irregularity 0. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

`RandomEllipticFibration_d8g7(P)`

`RndP`	RngIntElt	*Default :* 2
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (minimal) degree 8 elliptic surface with sectional genus 7 in P. These have Kodaira dimension 1, geometric genus 2 and irregularity 0. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

`RandomEllipticFibration_d9g7(P)`

`RndP`	RngIntElt	*Default :* 2
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (minimal) degree 9 elliptic surface with sectional genus 7 in P. These have Kodaira dimension 1, geometric genus 1 and irregularity 0. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

`RandomEllipticFibration_d10g10(P)`

`RndP`	RngIntElt	*Default :* 2
`Check`	BoolElt	*Default :* `true`

The argument P should be a 4-dimensional ordinary projective space defined over the rational numbers or a finite field. Generates a random (non-minimal) degree 10 elliptic surface with sectional genus 10 in P. These have Kodaira dimension 1, geometric genus 2 and irregularity 0. The parameters have the same meaning as for `RandomRationalSurface_d10g9` and the same defaults.

116.3 Surfaces in $\mathbf{P}^3$

116.3.1 Introduction

This section describes several packages of functionality developed for working with (hyper)surfaces in three-dimensional projective space $\mathbf{P}^3$.

At the core is a package to compute a **formal desingularization** of such a hypersurface X, expressed via a collection of algebraic power series giving the formal completion of the components of *some* desingularization lying over the components of the singular subscheme of the hypersurface. This allows the computation of important birational invariants of *any* desingularization of X like the arithmetic and geometric genera and higher geometric plurigenera. The algorithm is based on the method of Jung and was designed and implemented by Tobias Beck. It is fully described in [Bec07].

An important application of the desingularization data is the computation of *m-adjoint* maps as rational maps on X. A function is provided for this. Theoretical and algorithmic details may be found in [BS08].

There are functions to determine whether X is of Kodaira dimension $-\infty$, *i.e.,* birationally ruled. For the important special case of *rational* surfaces, there is a suite of functions to determine whether a parameterization exists over the base field and to explicitly construct one in the affirmative case. This is based on the work of Josef Schicho described in [Sch98] and [Sch00].

A surface X is mapped to a standard model by applying an appropriate m-adjoint map. These are then parameterized by special case code. The main special cases are Del Pezzo surfaces (including some singular cases) and line and conic bundles. The functions can be called directly by the user. Apart from the previously existing Del Pezzo code (for degrees 6, 8 and 9) and the special singular code for degrees 3 and 4, these functions were implemented by Tobias Beck and Josef Schicho.

116.3.2 Embedded Formal Desingularization of Curves

A formal embedded desingularization of plane curves, as described below, is used in the Jung surface resolution process. The main function is available to the user and provides another alternative to the existing function field and resolution graph curve functionality.

Before describing the function, we introduce some terminology. Let $C \subset P$ be a plane algebraic curve (where $P = \mathbf{A}^2_{\mathbf{E}}$ or $P = \mathbf{P}^2_{\mathbf{E}}$ for some field $\mathbf{E}$ of characteristic zero) and

$\pi : Q \to P$ an *embedded desingularization*, *i.e.*, π is proper birational, Q is regular and $D := \pi^{-1}(C) \subset Q$ is a normal crossing divisor. Further let $\{p_1, \ldots, p_r\} \in Q$ be the generic points of the decomposition of D into irreducible components and $\{q_1, \ldots, q_s\} \in Q$ the closed points of the normal crossings of D.

From π we can construct morphisms Spec $\widehat{\mathcal{O}_{Q,p_i}} \to P$ and Spec $\widehat{\mathcal{O}_{Q,q_i}} \to P$. The set of all these morphisms (up to isomorphism of the domain) is called a *formal embedded desingularization* of $C \subset P$. Each of these morphisms has a centre on P which is defined to be the image of the closed point.

The two classes of morphisms are represented, respectively, by homomorphisms $A \to \widehat{\mathcal{O}_{Q,p_i}}$ and $A \to \widehat{\mathcal{O}_{Q,q_i}}$ (where A is either the normal polynomial ring $\mathbf{E}[x, y]$ or the graded polynomial ring $\mathbf{E}[x, y, w]$ and the inverse image of the maximal ideal of the completion ring is the prime ideal defining the centre), and we are free to choose an isomorphic representation of the codomain. We refer to the homomorphisms as μ_i and ν_i respectively.

`ResolveAffineCurve(p)`

`Factors`	SeqEnum	*Default :* []
`Ps`	RngMPolElt	*Default :* 0
`Focus`	RngMPolElt	*Default :* 0
`ExtName`	MonStgElt	*Default :* "*alpha*"
`ExtCount`	RngIntElt	*Default :* 0
Verbose	`Resolve`	*Maximum :* 1

Given the curve defined by $p \in \mathbf{E}[x, y]$ (a non-zero bivariate polynomial over a number field), this intrinsic essentially computes a formal embedded resolution of the curve using a succession of point blow ups. Only morphisms whose centres vanish on the ideal generated by `Focus` are considered. Note that `Focus` may be a single polynomial or a sequence of polynomials.

The three returned lists contain elements of the form $(b_1, (y, m_{11}), (p_1, m_{12}))$, $(b_2, (y, m_2))$ and $(b_2, (p_3, m_3))$ respectively. Here b_1, b_2 and b_3 are homomorphisms $\mathbf{E}[x, y] \to \mathbf{E}'[x, y]$ to some bivariate polynomial ring over an algebraic field extension $\mathbf{E}'$ over $\mathbf{E}$.

The first list gathers normal crossings. Precisely, the extended homomorphism $b_1 : \mathbf{E}[x, y] \to \mathbf{E}'[[x, y]]$ corresponds to a ν_i from above. Moreover we have $\langle b_1(p) \rangle = \langle y^{m_{11}} p_1^{m_{12}} \rangle$ where $y = 0$ and $p_1 = 0$ have a normal crossing.

The second list gathers exceptional divisors. The extended homomorphism $b_2 : \mathbf{E}[x, y] \to \mathbf{E}'(x)[[y]]$ corresponds to a μ_i from above. Moreover we have $\langle b_2(p) \rangle = \langle y^{m_2} \rangle$ and $y = 0$ corresponds to an exceptional divisor.

Finally, the last list corresponds to the components of the original curve. The extended homomorphism $b_3 : \mathbf{E}[x, y] \to \mathbf{E}'\widehat{[x, y]_{\langle p_3 \rangle}}$ (where $\mathbf{E}' = \mathbf{E}$ in this case) corresponds to another μ_i from above. Moreover we have that $b_3(p)$ has multiplicity m_3 in $\mathbf{E}'\widehat{[x, y]_{\langle p_3 \rangle}}$ and corresponds to an original curve component.

If known, a factorization of p (as returned by the `Factorization` command) can be passed using the parameter `Factors` and the squarefree part of p (as returned by the `SquarefreePart` command) using `Ps`.

If the ground field has to be extended, the algebraic elements will be displayed as `ExtName_i` where `i` starts from `ExtCount`. The last return value is the value of `ExtCount` plus the number of field extensions that have been introduced, which can be useful for consecutive naming when making a series of resolution calls.

Example H116E10

We compute an embedded resolution of an affine plane curve.

```
> Q := Rationals();
> Qxy<x,y> := PolynomialRing(Q, 2, "glex");
> f := (y^2-x^3)*(x^2-y^2-y^3);
> NCs, EXs, DCs := ResolveAffineCurve(f : Factors := Factorization(f));
> #NCs, #EXs, #DCs;
7 4 2
> NCs[2]; EXs[3]; DCs[1];
[*
    Mapping from: RngMPol: Qxy to RngMPol: Qxy,
    <y, 4>,
    <-x^2 + 2*x + y, 1>
*]
[*
    Mapping from: RngMPol: Qxy to RngMPol: Qxy,
    <y, 10>
*]
[*
    Mapping from: RngMPol: Qxy to RngMPol: Qxy,
    <y^3 - x^2 + y^2, 1>
*]
> NCs[2][1](x), NCs[2][1](y);
x*y - y
y
```

Here we have passed the factorization of f only for illustrative purposes. The curve is the union of a cusp and a node at the origin. It has two singular points over $\mathbf{Q}$, the origin and another intersection point of the two curves which has a residue field of degree 5 over $\mathbf{Q}$.
We have computed the local information of a (not necessarily minimal) embedded resolution and find that it contains the 2 components of the strict transform, further 4 exceptional divisors and 7 normal crossings. For example, the pushforward of f under the chart map $x \mapsto xy - y, y \mapsto y$ is equal to $y^4(-x^2 + 2x + y)$ up to a local unit. The corresponding germ is isomorphic to a normal crossing in the embedded desingularization. We also see that one of the exceptional divisors has multiplicity 10.
If we were only interested in a local resolution, we would do the following:

```
> NCs, EXs, DCs := ResolveAffineCurve(f : Focus := [x,y]);
> #NCs, #EXs, #DCs;
```

```
5 3 0
```

We focus on the origin, hence, any curve components are not considered . We have 1 less exceptional divisor and 2 less normal crossings. This is because the second intersection point of the above two curve components was already a normal crossing, but our algorithm has nevertheless blown it up in the previous example.

`ResolveProjectiveCurve(p)`

`Focus`	RngMPolElt	*Default :* 0
`ExtName`	MonStgElt	*Default :* "*alpha*"
`ExtCount`	RngIntElt	*Default :* 0
Verbose	`Resolve`	*Maximum :* 1

Given the curve defined by $p \in \mathbf{E}[x, y, z]$ (a non-zero trivariate homogeneous polynomial over a number field), this intrinsic essentially computes a formal embedded resolution of the curve using a succession of point blow ups. This is the same as `ResolveAffineCurve` above, but now p is a homogeneous polynomial in three variables that defines a projective curve. Accordingly, the b_j map from the respective homogeneous coordinate ring to some $\mathbf{E}'[x, y]$.

Example H116E11

We can also desingularize the projectivisation of the above curve.

```
> Q := RationalField();
> QXYZ<X,Y,Z> := PolynomialRing(Q, 3);
> F := (Y^2*Z-X^3)*(X^2*Z-Y^2*Z-Y^3);
> NCs, EXs, DCs := ResolveProjectiveCurve(F); #NCs, #EXs, #DCs;
7 4 2
> NCs[3];
[*
    Mapping from: RngMPol: QXYZ to Polynomial ring of rank 2 over
    Rational Field ...,
    <y, 4>,
    <x^2 + 2*x - y, 1>
*]
> NCs[3][1](X);
x*y + y
> NCs[3][1](Y);
y
> NCs[3][1](Z);
1
```

The homomorphisms take a slightly different shape (because they have now $\mathbf{Q}[X, Y, Z]$ as domain), but otherwise they are the same. This is because the curve has no singularities at infinity.

116.3.3 Formal Desingularization of Surfaces

For curves we have described embedded formal desingularization. For surfaces instead we produce only *formal desingularizations*. Let $S \subset P$ be a hypersurface (where $P = \mathbf{A}^3_{\mathbf{E}}$ or $P = \mathbf{P}^3_{\mathbf{E}}$) and $C \subset S$ a closed subset (which typically contains the singular locus). Further let $\pi : T \to S$ be a *desingularization*, *i.e.*, π is proper birational and T is regular. By $\{p_1, \ldots, p_r\} \in T$ we denote the generic points of the curve components of the decomposition of $D := \pi^{-1}(C)$ into irreducibles.

From π we can construct morphisms Spec $\widehat{\mathcal{O}_{T,p_i}} \to S$. The set of all these morphisms (up to isomorphism of the domain) is called a *formal desingularization* of S over $C \subset S$. Such a morphism has a centre on S which is defined as the image of the closed point (and actually is contained in C).

The morphisms are represented by homomorphisms $A \to \widehat{\mathcal{O}_{T,p_i}}$ (where A is either the algebra $\mathbf{E}[x,y,z]/\langle p\rangle$ or the graded algebra $\mathbf{E}[x,y,z,w]/\langle p\rangle$ with p a defining polynomial), and we are free to choose an isomorphic representation of the codomain. We refer to such a homomorphisms as μ_i.

In the actual algorithm, C is the ramification locus of a finite projection, pr, to an affine or projective plane (C contains the singular subscheme of S). The underlying desingularization T (which is not computed explicitly) is a Jung resolution which is constructed in two stages. Firstly, an embedded resolution of the image of C in the plane is performed by blow-ups and T_1 is taken as the normalization of the pullback of this by pr. So T_1 then has only point singularities of a simple type (*toric singularities*), lying over the (normal-crossing) intersections of components of the embedded resolution. These are resolved by a finite succession of blow-ups on T_1 to give T.

The algorithm computes the formal desingularization, as described above, corresponding to T, using the embedded formal desingularization for curves followed by algebraic power series operations for the normalization and final resolution of the toric singularities. This is described fully in [Bec07]. The μ_i homomorphisms are defined by algebraic power series images of the variables of P.

It is important to note that the Jung desingularization T is not a minimal desingularization and, in any case, the set of morphisms returned for the formal desingularization generally contain some elements whose centre on S is already non-singular (because, for example, components of C are often generically non-singular). However, there is an parameter option with the main function `ResolveProjectiveSurface`, which removes "non-singular" morphisms and possibly others that have no effect on the computation of birational invariants and m-adjoint maps.

`ResolveAffineMonicSurface(s)`

`Focus`	RngMPolElt	*Default :* 0
`ExtName`	MonStgElt	*Default :* "*alpha*"
`ExtCount`	RngIntElt	*Default :* 0
Verbose	`Resolve`	*Maximum :* 1

The main user resolution function `ResolveProjectiveSurface` is for projective hypersurfaces. This affine version, however, may be useful in some circumstances.

The input is a monic, squarefree polynomial $s \in \mathbf{E}[x,y][z]$ where $\mathbf{E}$ is a number field (*i.e.*, s is univariate over a bivariate polynomial ring). Let $S \subset \mathbf{A}^3_{\mathbf{E}}$ denote the surface defined by it and $C \subset S$ the closed subset defined by $\mathrm{disc}_z(s)$ (*i.e.*, the intersection of S with the cylinder over the discriminant curve when considering the projection $S \to \mathbf{A}^2_{\mathbf{E}}$ in z-direction). The function computes a formal desingularization of S over C (see above).

The first return value is a list of elements of the form $((X,Y,Z),o)$ where $X,Y,Z \in \mathbf{F}[[t]]$ are univariate power series (over some field extension $\mathbf{F}$ of transcendence degree 1 over $\mathbf{E}$) s.t. $s(X,Y,Z) = 0$ and o is an integer. The induced homomorphism $\mathbf{E}[x,y][z]/(s) \to \mathbf{F}[[t]]$ corresponds to a μ_i from above and o is its *adjoint order*, *i.e.*, the negation of the order of a special differential form (see Section 116.3.4).

One can specify a focus ideal $F \subset \mathbf{E}[x,y]$ by passing a single generator or sequence of generators in `Focus` (as for `ResolveAffineCurve`). In this case C is taken to be the intersection of S and the cylinder over the zero set of $F + \langle \mathrm{disc}_z(s) \rangle$.

If the ground field has to be extended, the algebraic elements will be displayed as `ExtName_i` where `i` starts from `ExtCount`. The last return value is the value of `ExtCount` plus the number of field extensions that have been introduced, which can be useful for consecutive naming when making a series of resolution calls. A transcendental element will always be displayed as `s`.

Example H116E12

We compute a formal desingularization for the affine surface $z^2 - xy = 0$.

```
> Q := Rationals();
> Qxy<x,y> := PolynomialRing(Q, 2, "glex");
> Qxyz<z> := PolynomialRing(Qxy);
> f := z^2 - x*y;
> desing := ResolveAffineMonicSurface(f); #desing;
3
```

We have computed 3 morphisms. Two of them are centred over the coordinate axes $x = 0$ and $y = 0$. But they might not be of interest, because the surface is normal and has an isolated singularity over the origin.

```
> #ResolveAffineMonicSurface(f : Focus := [x,y]);
1
```

The only remaining morphism corresponds to the exceptional divisor obtained by blowing up the singularity.
Elements in the returned list which define the morphisms of the formal desingularization are examined more closely in the projective surface example below.

`ResolveProjectiveSurface(S)`

`AdjComp`	BoolElt	*Default :* `false`
`ExtName`	MonStgElt	*Default :* "*gamma*"
`ExtCount`	RngIntElt	*Default :* 0
Verbose	`Resolve`	*Maximum :* 1

The principal function for hypersurface desingularization, similar in description to `ResolveAffineMonicSurface` above. The argument is either a projective surface $S \subset \mathbf{P}^3_{\mathbf{E}}$ or an irreducible, homogeneous polynomial $s \in \mathbf{E}[x, y, z, w]$ which defines such a surface S. Computes a formal desingularization (see above) of S. It will be a formal desingularization over an automatically chosen subset $C \subset S$ (using again the cylinder over the discriminant curve w.r.t. a nice projection onto some $\mathbf{P}^2_{\mathbf{E}}$). Accordingly the elements of the return list of formal desingularization data are now of the form $((X, Y, Z, W), o)$.

If `AdjComp` is `true`, then only a sublist is returned that is still sufficient for the computation of birational invariants and adjoint spaces (see Section 116.3.4). The parameters `ExtName` and `ExtCount` and the second return value have the same meaning as in the affine case.

As stated above, the algorithm is based on formally computing a Jung resolution and is described in [Bec07].

Example H116E13

Computing a formal desingularization is easy.

```
> P<x,y,z,w> := PolynomialRing(Rationals(), 4);
> F := w^3*y^2*z+(x*z+w^2)^3;
> desing := ResolveProjectiveSurface(F); #desing;
26
```

Hence, the formal desingularization of the projective surface defined by `F` contains 26 morphisms. They are represented by tuples of power series that vanish on `F`. We have a closer look at the first morphism.

```
> prm, ord := Explode(desing[1]);
> IsZero(AlgComb(F, prm)); ord;
true
4
> X, Y, Z, W := Explode(prm);
> Expand(X, 6); Expand(Y, 6); Expand(Z, 6); Expand(W, 6);
true 1
true -s*t^2
true -t^2
true -1/64*s^2*gamma_0*t^5 + 1/2*gamma_0^2*t^3 + gamma_0*t^2 + t
> Domain(W);
Polynomial ring of rank 1 over Algebraic function field defined
over Univariate rational function field over Rational Field
```

```
by $.1^3 - 1/8*s^2
Graded Lexicographical Order
Variables: t
```

One of the morphisms is of type Spec$\mathbf{Q}(s)[\texttt{gamma_0}][[t]] \to \text{Proj}\mathbf{Q}[x,y,z,w]/\langle F\rangle$ where $\texttt{gamma\_0}^3 - 1/8s^2 = 0$. In particular, $\mathbf{Q}(s)[\texttt{gamma\_0}]$ is isomorphic to the residue field of the corresponding prime divisor on the desingularization. From this one can for example deduce that it is a rational curve. The morphism is given by the ring homomorphism $x \mapsto 1$, $y \mapsto -st^2$, $z \mapsto -t^2$ and $w \mapsto t + \texttt{gamma\_0}t^2 + 1/2\texttt{gamma\_0}^2t^3 - 1/64s^2\texttt{gamma\_0}t^5 + \ldots$.
The adjoint order for this morphism is 4. Consider the chart $x \neq 0$. The special differential form (see Section 116.3.4) in this chart obtained by dehomogenizing is

$$\frac{x^5}{(\partial F/\partial w)(x,y,z,w)}\mathrm{d}y/x \wedge \mathrm{d}z/x.$$

Substituting the values `X`, `Y`, `Z` and `W` we see that it is mapped to

$$\frac{\texttt{X}^5}{(\partial F/\partial w)(\texttt{X},\texttt{Y},\texttt{Z},\texttt{W})}\mathrm{d}\texttt{Y}/\texttt{X} \wedge \mathrm{d}\texttt{Z}/\texttt{X}$$

$$= \frac{1}{(\partial F/\partial w)(\texttt{X},\texttt{Y},\texttt{Z},\texttt{W})}\mathrm{d}(-st^2) \wedge \mathrm{d}(-t^2)$$

$$= \frac{1}{(\partial F/\partial w)(\texttt{X},\texttt{Y},\texttt{Z},\texttt{W})}(2st\mathrm{d}t + t^2\mathrm{d}s) \wedge 2t\mathrm{d}t$$

$$= \frac{1}{(\partial F/\partial w)(\texttt{X},\texttt{Y},\texttt{Z},\texttt{W})}2t^3\mathrm{d}s \wedge \mathrm{d}t$$

The adjoint order is minus the overall order of this expression, hence, -3 plus the order of $(\partial F/\partial w)(\texttt{X},\texttt{Y},\texttt{Z},\texttt{W})$. We check the computation.

```
> Order(AlgComb(Derivative(F,w), prm));
7
```

If we needed the formal desingularization only in order to compute birational invariants or adjoint spaces we could set the parameter `AdjComp` and forget about some morphisms.

```
> #ResolveProjectiveSurface(F : AdjComp := true);
18
```

116.3.4 Adjoint Systems and Birational Invariants

In this section we describe computation of adjoint spaces. Let $S \subset \mathbf{P}^3_{\mathbf{E}}$ be a surface defined by a homogeneous irreducible polynomial $F \in \mathbf{E}[x_0, x_1, x_2, x_3]$ of degree d and $\Omega_{\mathbf{E}(S)|\mathbf{E}}$ the vector space of rational differential forms of the function field (over the ground field $\mathbf{E}$ of characteristic zero). We can consider $\Omega_{\mathbf{E}(S)|\mathbf{E}}$ a constant sheaf of $\mathcal{O}_S$-modules. Let $U_i \subset S$ be the affine open subsets of the standard covering w.r.t. this choice of variables.

Let $\omega^0_S \subset \Omega^{\wedge 2}_{\mathbf{E}(S)|\mathbf{E}}$ be the subsheaf which is locally generated on U_i by

$$\left(\frac{\partial F/\partial x_j}{x_i^{d-1}}\right)^{-1} \bigwedge_{k\in\{0,\ldots,3\}\setminus\{i,j\}} \mathrm{d}\frac{x_k}{x_i}$$

(for an arbitrary choice of $j \neq i$). By sending this generator to x_i^{d-4} one finds that $\omega^0_S \cong \mathcal{O}_S(d-4)$. Further let $F_{S,m} \subset (\Omega^{\wedge 2}_{\mathbf{E}(S)|\mathbf{E}})^{\otimes m}$ be the subsheaf of those forms whose pullbacks are regular on some desingularization of S. It is called the *sheaf of m-adjoints*. It is in fact well-defined, *i.e.*, doesn't depend on any specific desingularization, and one can show $F_{S,m} \subseteq (\omega^0_S)^{\otimes m}$. For more details we refer to [BS08].

Now since $F_{S,m}$ is a coherent sheaf on the projective scheme $S \subset \mathbf{P}^3_{\mathbf{E}}$ it can be defined by its associated graded module $M_{S,m}$ and by the above discussion $F_{S,m}$ is isomorphic to a subsheaf of $\mathcal{O}_S(m(d-4))$. The module $M_{S,m}$ is thus naturally a graded submodule of $(\mathbf{E}[x_0, x_1, x_2, x_3]/\langle F\rangle)(m(d-4))$. The n-th graded piece of $M_{S,m}$, a linear subsytem of the standard linear system of degree $n+m(d-4)$ homogeneous polynomials on S, corresponds to global sections of the Serre twist $F_{S,m}(n)$. This, under pullback, corresponds to the space of global sections of the twisted m-adjoint sheaf $(\omega_X)^{\otimes m}(n)$ for any desingularization X of S, where (n) now signifies twisting by the n-th tensor power of $\mathcal{L}$, the invertible sheaf on X which gives the map into projective space projecting X down onto S.

These adjoint linear systems immediately give the plurigenera of any desingularization X as well as an explicit representation of the important twisted m-adjoint maps into projective space as rational maps from S (defined by the sequence of homogeneous polynomials forming a basis of the adjoint system). These maps are used to take any rational hypersurface to a standard model, as described in the next section.

All functions in this section (and several in the following sections) allow the user to enter precomputed formal desingularization data. It is a good idea to do this if performing several operations on the same hypersurface to avoid repeated computation of this desingularization.

`HomAdjoints(m,n,S)`

`FormalDesing`	SeqEnum	*Default :* 0
Verbose	`Classify`	*Maximum :* 1

Given a surface S of degree d in $\mathbf{P}^3$ defined over a number field $\mathbf{E}$ together with integers m and n, the intrinsic returns a basis for the vector space of the degree-n graded summand of the graded ring associated to $F_{S,m}$ (*i.e.*, $\Gamma(S, \mathcal{O}_S(n) \otimes F_{S,m})$) as a subspace of the homogeneous forms in $\mathbf{E}[x_0, x_1, x_2, x_3]$ (the coordinate ring of the $\mathbf{P}^3$ ambient) of degree $n + m(d-4)$ (see above).

The parameter FormalDesing may be set to a precomputed formal desingularization (as returned by ResolveProjectiveSurface). The desingularization passed in can be computed with the AdjComp parameter set to true. The default value for FormalDesing is the integer 0, in which case a formal desingularization needs to be computed during function execution.

The function computes the adjoint space as a linear subspace of homogeneous polynomials of the appropriate degree by using the formal divisor morphisms of the formal desingularization to give additional linear conditions at the singular places of S. This is explained fully in [BS08].

GeometricGenusOfDesingularization(S)

FormalDesing SeqEnum *Default :* 0

Given a hypersurface S in $\mathbf{P}^3$, the intrinsic returns the geometric genus of (any) desingularization of S. The function just computes the dimension of the $(1,0)$ adjoint space.

As in the case of HomAdjoints, a precomputed desingularization (of S or its defining polynomial) can be passed in via the FormalDesing parameter.

PlurigenusOfDesingularization(S,m)

FormalDesing SeqEnum *Default :* 0

Given a hypersurface S in $\mathbf{P}^3$, the intrinsic returns the m-th plurigenus of (any) desingularization, X, of S. This is the dimension of the global sections of the sheaf $(\omega_X)^{\otimes m}$ and is just computed as the dimension of the $(m,0)$ adjoint space.

As for HomAdjoints, a precomputed desingularization (of S or its defining polynomial) can be passed in via the FormalDesing parameter.

ArithmeticGenusOfDesingularization(S)

FormalDesing SeqEnum *Default :* 0

Given a hypersurface S in $\mathbf{P}^3$, the intrinsic returns the arithmetic genus of (any) desingularization of S. This is computed from a simple formula involving the dimensions of the (1,1)- and (1,2)-adjoints coming from the Riemann-Roch theorem.

As for HomAdjoints, a precomputed desingularization (of S or its defining polynomial) can be passed in via the FormalDesing parameter.

Example H116E14

We compute several adjoint spaces of a surface. We precompute a formal desingularization and pass it to the calls to HomAdjoints.

```
> P<x,y,z,w> := ProjectiveSpace(Rationals(), 3);
> F := w^3*y^2*z+(x*z+w^2)^3;
> S := Surface(P,F);
> desing := ResolveProjectiveSurface(S : AdjComp := true);
> HomAdjoints(1, 0, S : FormalDesing := desing);
[]
```

```
> HomAdjoints(1, 1, S : FormalDesing := desing);
[
    x*z*w + w^3
]
> HomAdjoints(1, 2, S : FormalDesing := desing);
[
    x^2*z^2 - w^4, x^2*z*w + x*w^3, x*y*z*w, x*z^2*w + z*w^3,
    x*z*w^2 + w^4, y*z*w^2, y*w^3
]
> HomAdjoints(1, 3, S : FormalDesing := desing);
[
    x^3*z^2 - x*w^4, x^2*y*z^2, x^2*z^3 - z*w^4,
    x^3*z*w + x^2*w^3, x^2*y*z*w, x*y^2*z*w, x^2*z^2*w - w^5,
    x*y*z^2*w, x*z^3*w + z^2*w^3, x^2*z*w^2 + x*w^4, x*y*z*w^2,
    y^2*z*w^2, x*z^2*w^2 + z*w^4, y*z^2*w^2, x*y*w^3, y^2*w^3,
    x*z*w^3 + w^5, y*z*w^3, y*w^4
]
>
> HomAdjoints(2, 0, S : FormalDesing := desing);
[]
> HomAdjoints(2, 1, S : FormalDesing := desing);
[]
> HomAdjoints(2, 2, S : FormalDesing := desing);
[
    x^2*z^2*w^2 + 2*x*z*w^4 + w^6
]
> HomAdjoints(2, 3, S : FormalDesing := desing);
[
    x^3*z^2*w^2 + 2*x^2*z*w^4 + x*w^6, x^2*y*z^2*w^2 - y*w^6,
    x^2*z^3*w^2 + 2*x*z^2*w^4 + z*w^6,
    x^2*z^2*w^3 + 2*x*z*w^5 + w^7, x*y*z^2*w^3 + y*z*w^5,
    x*y*z*w^4 + y*w^6, y^2*z*w^4
]
```

116.3.5 Classification and Parameterization of Rational Surfaces

This section contains functions for the recognition of rational surfaces in $\mathbf{P}^3$, the classification and transformation to standard models using appropriate m-adjunction maps and, finally, special case code for these standard models, to determine a parametrization of the original hypersurface.

A *non-singular* surface in $\mathbf{P}^r$ with $r \geq 4$ may be transformed to a standard model using the intrinsic `MinimalModelRationalSurface` (see Section 116.2.6).

`IsRational(X)`

`FormalDesing`	SeqEnum	*Default :* 0
`CheckADE`	BoolElt	*Default :* `false`

Returns `true` if the ordinary projective surface X is (geometrically) rational, i.e. birationally isomorphic to the projective plane over the algebraic closure of its base field. This simply uses the Castelnuevo criterion that X is rational if and only if both the arithmetic genus and second plurigenus of any desingularization are zero.

If the ambient of X is $\mathbf{P}^3$ over a number field, there is no assumption about the singularity of X and a formal desingularization will be used to compute plurigenera. Otherwise, X should have at worst simple (A-D-E) singularities and the algorithms of early sections are used for the computations. In this latter case, the singularity status is assumed by default. To force a check for only A-D-E singularities, the user should set the parameter `CheckADE` to `true`. This can be a very heavy verification in higher dimensional ambients.

The computation in the former case uses a formal desingularization of X. To avoid recalculation, a precomputed formal desingularization can be supplied using `FormalDesing` parameter, as with the `HomAdjoints` intrinsic.

116.3.6 Reduction to Special Models

In this section, we describe the function for the birational transformation over the base field of a rational hypersurface in $\mathbf{P}^3$ into a special model of one of the types in the standard classification as listed by Josef Schicho in [Sch98]. He enumerates 5 basic cases [Sch98, p. 17 and Lem. 5.2-5.7] and splits the last case in two, choosing labels "1", "2", "3", "4", "5A" and "5B". The lemmas describing cases 3 and 5A involve a further case distinction. Also, a label "0" is useful for the non-rational case. From this, we get the label set

$$\mathcal{L} := \{\text{"0"}, \text{"1"}, \text{"2"}, \text{"3a"}, \text{"3b"}, \text{"4"}, \text{"5Aa"}, \text{"5Ab"}, \text{"5Ac"}, \text{"5B"}\}$$

Let S be a surface in $\mathbf{P}^3$. In each of the above cases the author specifies a set of adjoint spaces defining interesting maps, either birationally to a special surface or to a rational normal curve (giving a pencil of rational curves on the surface). More precisely, the maps can be computed using $V_{n,m} :=$ `HomAdjoints` (m, n, S) for certain choices of m and n. Let μ to be the smallest integer s.t. $V_{1,\mu+1} \neq [\ ]$. Then the important $V_{n,m}$ for the different cases are as follows:

0	1	2	3a	3b	4
$[\ ]$	$[V_{1,\mu}]$	$[V_{1,\mu}]$	$[V_{2,2\mu+1}, V_{1,\mu}]$	$[V_{2,2\mu+1}]$	$[V_{1,\mu}, V_{2,2\mu-1}]$

5Aa	5Ab	5Ac	5B
$[V_{1,\mu-1}]$	$[V_{1,\mu-1}, V_{2,2\mu-2}]$	$[V_{1,\mu-1}, V_{2,2\mu-2}, V_{3,3\mu-3}]$	$[V_{2,1}]$

The function to find a parameterization of a rational hypersurface, which uses the reduction function below as a first stage, is described in the next section.

`ClassifyRationalSurface(S)`

`FormalDesing`	SeqEnum	*Default :* 0
Verbose	`Classify`	*Maximum :* 1

Given an ordinary projective surface S in $\mathbf{P}^3$ over a number field, the intrinsic returns the special rational surface type to which S is birationally equivalent and associated data. If S is not (geometrically) rational, the return values are S itself, a list containing only the identity map on S and the string "Not rational".

If S is rational and Schicho's algorithm reduces it to a standard surface Y over the base field k, Y is the first return value. The second return value is a list of one or two scheme maps. The first is always a birational map from S to Y. There is a second map if and only if Y is a rational scroll or conic bundle. Then S (and Y) have fibration maps to a rational normal curve such that the general fibre is a rational curve (and a line or conic for Y). The fibration map on S is the second return value. Note that, if the base field is $\mathbf{Q}$, in these cases, S can be parameterized by calling `ParametrizePencil` with the fibration map as argument.

The third return value is a string describing the type of Y. It is "P2" (for the projective plane!), "Quadric surface" (for a degree 2 surface in $\mathbf{P}^3$), "Rational scroll", "Conic bundle" or "Del Pezzo of degree d" where $1 \leq d \leq 9$. The Del Pezzos might be degenerate (with simple singularities) and are anticanonically embedded in $\mathbf{P}^d$ except for degrees 1 and 2 when they have their standard weighted-projective embeddings.

As usual, to avoid recalculation, a precomputed formal desingularization can be given using the `FormalDesing` parameter, as in the case of the `HomAdjoints` intrinsic.

Example H116E15

Here are a few examples.

```
> P<x,y,z,w> := ProjectiveSpace(Rationals(),3);
```

The first surface:

```
> p1 := x^4 + y^4 - z^2*w^2;
> _,_,typ := ClassifyRationalSurface(Surface(P,p1));
> typ;
Not rational
```

The second surface:

```
> p2 := 2*x + y + 8*z + 5*w;
> Y,_,typ := ClassifyRationalSurface(Surface(P,p2));
> typ; Y;
P^2
Surface over Rational Field defined by
0
```

The third surface:

```
> p3 := x^2 - 4*x*z + 3*x*w + y*z - y*w + 2*z^2 - 3*z*w + w^2;
```

```
> _,_,typ := ClassifyRationalSurface(Surface(P,p3));
> typ;
Quadric surface
```

The fourth surface:

```
> p4 := (y^2 - w*z)*(w^2 - y*x) + (x*z - y*w)^2;
> S := Surface(P,p4);
> Y,mps,typ := ClassifyRationalSurface(S);
> typ;
Rational scroll
> mps[2]; // the fibration map
Mapping from: Srfc: S to Scheme over Rational Field defined by
$.1*$.2 - $.3^2
with equations :
x*y - w^2
y^2 - z*w
x*z - y*w
```

The fifth surface:

```
> p5 := x^3*y - 4*x^3*z - 6*x^3*w - 3*x^2*y^2 - 2*x^2*y*z
>     - 3*x^2*y*w + 50*x^2*z^2 + 146*x^2*z*w + 108*x^2*w^2
>     - 11*x*y^2*z + 2*x*y^2*w + 61*x*y*z^2 + 149*x*y*z*w
>     + 65*x*y*w^2 + 68*x*z^3 + 228*x*z^2*w + 260*x*z*w^2
>     + 112*x*w^3 + 4*y^4 - 13*y^3*z - 19*y^3*w + 20*y^2*z^2
>     + 77*y^2*z*w + 55*y^2*w^2 + 40*y*z^3 + 106*y*z^2*w
>     + 58*y*z*w^2 - 2*y*w^3 + 22*z^4 + 84*z^3*w + 130*z^2*w^2
>     + 108*z*w^3 + 38*w^4;
> S := Surface(P,p5);
> _,mps,typ := ClassifyRationalSurface(S);
> typ;
P2
> mps[1]; //birational map from Y to P2
Mapping from: Srfc: S to Surface over Rational Field defined by
0
with equations :
x^2-1114/45*x*z-232/15*y*z-241/15*z^2-1543/45*x*w-319/15*y*w-1327/45*z*w-
    457/45*w^2
x*y-182/45*x*z-11/15*y*z-38/15*z^2-284/45*x*w-17/15*y*w-266/45*z*w-146/45*w^2
y^2-16/45*x*z-28/15*y*z-4/15*z^2-22/45*x*w-61/15*y*w+32/45*z*w+92/45*w^2
```

The sixth surface:

```
> p6 := x^2*y^2 + 8*x^3*y + 4*x^4 + x*y*z^2 - x^2*z^2 - y^2*w^2
>     - 7*x*y*w^2 + 8*x^2*w^2;
> S := Surface(P,p6);
> Y,mps,typ := ClassifyRationalSurface(S);
> typ; Y;
Conic bundle
Surface over Rational Field defined by
```

```
$.1^2 + 2*$.1*$.2 + 1/4*$.1*$.3 - 1/4*$.4^2 + 1/4*$.4*$.5 +
    2*$.6^2 - 7/4*$.6*$.7 - 1/4*$.7^2,
$.2^2 - $.1*$.3,
$.2*$.4 - $.1*$.5,
$.3*$.4 - $.2*$.5,
$.2*$.6 - $.1*$.7,
$.3*$.6 - $.2*$.7,
$.5*$.6 - $.4*$.7
> mps[2]; //the fibration map
Mapping from: Srfc: S to Projective Space of dimension 1 over Rational Field
Variables: $.1, $.2
with equations :
x
y
```

The seventh surface:

```
> p7 := x^2*w^3 + y^3*w^2 + z^5;
> Y,_,typ := ClassifyRationalSurface(Surface(P,p7));
> typ; Y; Ambient(Y);
Del Pezzo degree 1
Surface over Rational Field defined by
$.1^5*$.2 + $.3^3 + $.4^2
Projective Space of dimension 3 over Rational Field
Variables: $.1, $.2, $.3, $.4
The grading is:
    1, 1, 2, 3
```

The seventh surface:

```
> p8 := w^3*y^2*z + (x*z + w^2)^3;
> Y,_,typ := ClassifyRationalSurface(Surface(P,p8));
> typ; Y;
Del Pezzo degree 6
Surface over Rational Field defined by
$.1^2 - 4*$.5^2 + $.3*$.6 - 3*$.6*$.7,
$.1*$.2 + 2*$.2*$.5 + $.3*$.7,
$.1*$.4 + 2*$.4*$.5 + $.6^2,
$.2*$.4 + $.5^2 + $.6*$.7,
$.3*$.4 - $.1*$.6 - $.4*$.7,
$.1*$.5 + 2*$.5^2 + $.6*$.7,
$.3*$.5 - $.1*$.7 - $.5*$.7,
$.2*$.6 - $.1*$.7 - $.5*$.7,
$.5*$.6 - $.4*$.7
```

116.3.7 Parametrization of Rational Surfaces

The package also includes functions to directly parametrize rational surfaces (over $\mathbf{Q}$). These use the reduction to special type, described in the last section, followed by specialised algorithms for the special cases, which are described in the following sections and the section on Del Pezzo surfaces. There is a version for hypersurfaces in $\mathbf{P}^3$ and one for more general rational surfaces which are first birationally projected to a hypersurface. Note, however, that the projection method can be very inefficient because it often introduces nasty singularities that cause the resolution process to hang. If it is known that the surface S is non-singular, it is usually much better to use MinimalModelRationalSurface to get a birational map from S to a standard model Y and call the relevant special case parametrization routine for Y directly.

```
ParametrizeProjectiveHypersurface(X, P2)
```

FormalDesing	SeqEnum	*Default :* 0
Verbose	Classify	*Maximum :* 1

Given a surface X in $\mathbf{P}^3_{\mathbf{Q}}$ and a projective plane $P2$ over $\mathbf{Q}$, the intrinsic returns false if the surface is not rational over $\mathbf{Q}$, otherwise returns true and a birational parameterization $P2 \to X$.

The function begins by mapping X, as in ClassifyRationalSurface, to a surface of special type. As for that function, if a formal desingularization (for the defining polynomial p of X) is already known, it can be passed as parameter FormalDesing.

It is assumed that X is defined over $\mathbf{Q}$ because some of the special type routines assume this, partly for simplicity. This will probably be generalised in future releases.

```
ParametrizeProjectiveSurface(X, P2)
```

Verbose	Classify	*Maximum :* 1

Given an ordinary projective surface X in $\mathbf{P}^n_{\mathbf{Q}}$ for some $n \geq 2$ and a projective plane $P2$ over $\mathbf{Q}$, returns false if the surface is not rational over $\mathbf{Q}$, otherwise return true and a birational parametrization $P2 \to X$.

The function finds a birational projection to a hypersurface in $\mathbf{P}^3$ and then calls ParametrizeProjectiveHypersurface.

It should be noted that the birational projection may produce a very singular hypersurface defined by a polynomial with large coefficients. Then, the desingularisation, classification and special parameterization routines may each be very slow. As noted above, it may be better to try to use MinimalModelRationalSurface followed by one of the specialised parametrization routines for a non-singular rational X for larger n.

Example H116E16

We try to parameterize the hypersurfaces given by polynomials p1 - p8 from the previous example.

The surface defined by $p1$:

```
> P2<X,Y,W> := ProjectiveSpace(Rationals(), 2);
> ParametrizeProjectiveHypersurface(Surface(P, p1), P2);
false
```

The surface defined by $p2$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p2), P2);
true Mapping from: Prj: P2 to Surface over Rational Field
defined by 2*x + y + 8*z + 5*w
with equations :
-1/2*X - 4*Y - 5/2*W
X
Y
W
```

The surface defined by $p3$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p3), P2);
true Mapping from: Prj: P2 to Surface over Rational Field
defined by x^2 - 4*x*z + 3*x*w + y*z - y*w + 2*z^2 - 3*z*w + w^2
with equations :
X^2 - 2*X*W
X^2 + X*Y - 4*X*W - Y*W + 2*W^2
X^2 - 3*X*W + Y*W
X^2 - 4*X*W + Y*W + 2*W^2
```

The surface defined by $p4$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p4), P2);
true Mapping from: Prj: P2 to Surface over Rational Field
defined by x^2*z^2 - x*y^3 - x*y*z*w + 2*y^2*w^2 - z*w^3
with equations :
2*X^2*Y^2*W^2 - Y*W^5
X^4*Y^2 - X^2*Y*W^3
X^3*Y*W^2
X^3*Y^2*W
```

The surface defined by $p5$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p5), P2);
true Mapping from: Prj: P2 to Surface over Rational Field
defined by ...
with equations :
-1/4*X^2 + 9/2*X*Y + 1/2*X*W - 69/4*Y^2 - 87/8*Y*W + 1/2*W^2
1/2*X*Y + 11/4*X*W - 5/8*Y^2 - 131/8*Y*W - 63/8*W^2
-11/8*X*Y + 7/8*X*W + 13/4*Y^2 - 11/2*W^2
X*Y - 1/8*X*W - 23/8*Y^2 + 4*W^2
```

The surface defined by $p6$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p6), P2);
```

```
true Mapping from: Prj: P2 to Surface over Rational Field
defined by 4*x^4 + 8*x^3*y + x^2*y^2 - x^2*z^2 + 8*x^2*w^2
           + x*y*z^2 - 7*x*y*w^2 - y^2*w^2
with equations :
-3/8*X^2*Y^2*W - 2*X^2*Y*W^2 - 8/3*X^2*W^3 - 59/24*X*Y^2*W^2
    - 38/3*X*Y*W^3 - 16*X*W^4 + 17/6*Y^2*W^3 + 44/3*Y*W^4
    + 56/3*W^5
-3/8*X^3*Y^2 - 2*X^3*Y*W - 8/3*X^3*W^2 - 59/24*X^2*Y^2*W
    - 38/3*X^2*Y*W^2 - 16*X^2*W^3 + 17/6*X*Y^2*W^2
    + 44/3*X*Y*W^3 + 56/3*X*W^4
-1/8*X^2*Y^2*W - 4/3*X^2*Y*W^2 - 8/3*X^2*W^3 - 1/12*X*Y^2*W^2
    - 16/3*X*Y*W^3 - 40/3*X*W^4 + 19/3*Y^2*W^3 + 104/3*Y*W^4
    + 48*W^5
-3/8*X^2*Y^2*W - 2*X^2*Y*W^2 - 8/3*X^2*W^3 - 8/3*X*Y^2*W^2
    - 14*X*Y*W^3 - 56/3*X*W^4 + Y^2*W^3 + 20/3*Y*W^4 + 32/3*W^5
```

The surface defined by $p7$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p7), P2);
true Mapping from: Prj: P2 to Surface over Rational Field defined by
z^5 + y^3*w^2 + x^2*w^3
with equations :
-X^362*Y^144*W^79 - 9*X^363*Y^141*W^81 - 36*X^364*Y^138*W^83 -
    84*X^365*Y^135*W^85 - 126*X^366*Y^132*W^87 - 126*X^367*Y^129*W^89 -
    84*X^368*Y^126*W^91 - 36*X^369*Y^123*W^93 - 9*X^370*Y^120*W^95 -
    X^371*Y^117*W^97
X^359*Y^148*W^78 + 10*X^360*Y^145*W^80 + 45*X^361*Y^142*W^82 +
    120*X^362*Y^139*W^84 + 210*X^363*Y^136*W^86 + 252*X^364*Y^133*W^88 +
    210*X^365*Y^130*W^90 + 120*X^366*Y^127*W^92 + 45*X^367*Y^124*W^94 +
    10*X^368*Y^121*W^96 + X^369*Y^118*W^98
-X^357*Y^150*W^78 - 11*X^358*Y^147*W^80 - 55*X^359*Y^144*W^82 -
    165*X^360*Y^141*W^84 - 330*X^361*Y^138*W^86 - 462*X^362*Y^135*W^88 -
    462*X^363*Y^132*W^90 - 330*X^364*Y^129*W^92 - 165*X^365*Y^126*W^94 -
    55*X^366*Y^123*W^96 - 11*X^367*Y^120*W^98 - X^368*Y^117*W^100
X^354*Y^153*W^78 + 12*X^355*Y^150*W^80 + 66*X^356*Y^147*W^82 +
    220*X^357*Y^144*W^84 + 495*X^358*Y^141*W^86 + 792*X^359*Y^138*W^88 +
    924*X^360*Y^135*W^90 + 792*X^361*Y^132*W^92 + 495*X^362*Y^129*W^94 +
    220*X^363*Y^126*W^96 + 66*X^364*Y^123*W^98 + 12*X^365*Y^120*W^100 +
    X^366*Y^117*W^102
and inverse
z^4*w^8
y*z^2*w^9
x*z*w^10
```

The surface defined by $p8$:

```
> ParametrizeProjectiveHypersurface(Surface(P, p8), P2);
true Mapping from: Prj: P2 to Surface over Rational Field defined by
x^3*z^3 + 3*x^2*z^2*w^2 + 3*x*z*w^4 + y^2*z*w^3 + w^6
with equations :
```

```
2*X^20*Y^3*W - 4*X^19*Y^3*W^2 + 4*X^17*Y^3*W^4 - 2*X^16*Y^3*W^5
4*X^19*Y^4*W - 8*X^18*Y^4*W^2 + 4*X^17*Y^4*W^3
1/2*X^16*Y^7*W - X^15*Y^7*W^2 + 1/2*X^14*Y^7*W^3
-X^18*Y^5*W + 3*X^17*Y^5*W^2 - 3*X^16*Y^5*W^3 + X^15*Y^5*W^4
> IsRational(Surface(P, p7));
true
```

Here are two easy examples of parameterizing non-hypersurfaces.

```
> P4<u,v,w,x,y> := ProjectiveSpace(Rationals(),4);
> P2<X,Y,Z> := ProjectiveSpace(Rationals(), 2);
> S := Surface(P4,[u^2 + v^2 + w^2 - x^2, y - x]);
> ParametrizeProjectiveSurface(S, P2);
true Mapping from: Prj: P2 to Srfc: S
with equations :
-2*X*Z + 2*Z^2
-2*Y*Z
X^2 + Y^2 - 2*X*Z
-X^2 - Y^2 + 2*X*Z - 2*Z^2
-X^2 - Y^2 + 2*X*Z - 2*Z^2
and inverse
u + w + y
v
w + x
```

Here we parametrize a particularly easy surface – $\mathbf{P}^2$ itself!

```
> S := Surface(P2,[]);
> ParametrizeProjectiveSurface(S, P2);
true Mapping from: Prj: P2 to Srfc: S
with equations :
X
Y
Z
and inverse
X
Y
Z
```

`Solve(p, F)`

For convenience, this intrinsic provides a purely algebraic version of parameterization of an affine hypersurface.

Given a polynomial $p \in \mathbf{Q}[x, y, z]$, the equation of a (not necessarily irreducible affine hypersurface S) and a two-variable rational function field $F = \mathbf{Q}(u, v)$, the function finds birational parameterizations of the irreducible components of S (that are parametrizable over $\mathbf{Q}$).

A sequence of triples $(X, Y, Z) \in F^3$ is returned such that $p(X, Y, Z) = 0$ and each triple gives an isomorphism of F to the function field of a component of S.

The routine may again result in a runtime error if it involves parameterizations of special surface types that are not yet implemented, as for the preceding functions.

Example H116E17

The following affine hypersurface has three irreducible factors: one not rational and two that are rational and parameterizable over $\mathbf{Q}$.

```
> Q := RationalField();
> P<x,y,z> := PolynomialRing(Q, 3);
> F<s,t> := RationalFunctionField(Q, 2);
> p := (x^4+y^4-z^2)*(2*x + y + 8*z + 5)
>       *(x^2 - 4*x*z + 3*x + y*z - y + 2*z^2 - 3*z + 1);
> Solve(p, F);
[
    [ -1/2*s - 4*t - 5/2, s, t ],
    [
        (s^2 - 2*s)/(s^2 - 4*s + t + 2),
        (s^2 + s*t - 4*s - t + 2)/(s^2 - 4*s + t + 2),
        (s^2 - 3*s + t)/(s^2 - 4*s + t + 2)
    ]
]
```

116.3.8 Parametrization of Special Surfaces

In this section we describe routines for the explicit parameterization of the special classes of rational surfaces that arise from the reduction of the general case via m-adjoint maps. The algorithms are the work of Josef Schicho, in collaboration with others in some cases. The functions are used in the general parameterization routines but can also be called directly by the user.

`ParametrizeQuadric(X,P2)`

Suppose the scheme $X \subset \mathbf{P}^3_{\mathbf{Q}}$ is a geometrically irreducible quadric (degree 2) projective hypersurface and $P2$ is a projective plane. The intrinsic returns `false` if X is not parametrizable over the rationals, otherwise it returns `true` together with a birational parameterization $P2 \to X$.

Given a rational point p on X, the intrinsic is based on a simple, well-known algorithm (see [Sch98, Sec. 3.1]). Finding the point p is equivalent to finding a non-trivial isotropic vector for F, the quadric form in four variables defining X. This is achieved by a reduction to the solution of two quadrics in three variables, which is performed using standard lattice methods. The solubility routines here assume that the quadric is defined over $\mathbf{Q}$.

Example H116E18

We give a few simple examples.

```
> Q := Rationals();
> P2<X,Y,W> := ProjectiveSpace(Q, 2);
> P3<x,y,z,w> := ProjectiveSpace(Q, 3);
> X1 := Scheme(P3, x^2 + y^2 + z^2 + w^2);
> X2 := Scheme(P3, x^2 + y^2 + z^2 - w^2);
> X3 := Scheme(P3, x^2 + y^2 + z^2);
> X4 := Scheme(P3, x^2 + y^2 - z^2);
> X5 := Scheme(P3, x^2 - 4*x*z + 3*x*w + y*z - y*w + 2*z^2
>                    - 3*z*w + w^2);
> ParametrizeQuadric(X1, P2);
false
> ParametrizeQuadric(X2, P2);
true Mapping from: Prj: P2 to Sch: X2
with equations :
2*X*W
2*Y*W
-X^2 - Y^2 + W^2
X^2 + Y^2 + W^2
> ParametrizeQuadric(X3, P2);
false
> ParametrizeQuadric(X4, P2);
true Mapping from: Prj: P2 to Sch: X4
with equations :
X*Y
-1/2*X^2 + 1/2*Y^2
1/2*X^2 + 1/2*Y^2
Y*W
> ParametrizeQuadric(X5, P2);
true Mapping from: Prj: P2 to Sch: X5
with equations :
X^2 - 2*X*W
X^2 + X*Y - 4*X*W - Y*W + 2*W^2
X^2 - 3*X*W + Y*W
X^2 - 4*X*W + Y*W + 2*W^2
```

`ParametrizePencil(phi, P2)`

Let X be an ordinary projective birationally ruled surface, given as the domain of a rational pencil ϕ defined over $\mathbf{Q}$ (*i.e.*, a rational map $X \to \mathbf{P}^n_{\mathbf{Q}}$ for some n with image a rational normal curve) and $P2$ is a projective plane over $\mathbf{Q}$. The intrinsic returns `false` if X is not parameterizable over the rationals. Otherwise, it returns `true` and a birational parameterization $P2 \to X$.

These intrinsics take care of rational scrolls and conic bundles in the classification of special surfaces. The algorithm is described in [Sch00]. The scheme X is not assumed to be non-singular.

Example H116E19

We start with a (singular) degree 4 hypersurface X in P^3 and construct a pencil from P^2 to X.

```
> Q := Rationals();
> P3<x,y,z,w> := ProjectiveSpace(Q, 3);
> P2<X,Y,Z> := ProjectiveSpace(Q, 2);
> X := Scheme(P3, x^2*z^2 - x*y^3 - x*y*z*w + 2*y^2*w^2 - z*w^3);
> pencil := map<X -> P2 | [x*y - w^2, y^2 - z*w, x*z - y*w]>;
> DefiningPolynomial(Image(pencil));
X*Y - Z^2
> ParametrizePencil(pencil, P2);
true Mapping from: Prj: P2 to Sch: X
with equations :
-X^4*Y + 2*X^2*Z^3
-X^2*Y*Z^2 + Z^5
X*Y*Z^3
X*Z^4
```

`ParametrizeDelPezzo(X, P2)`

The argument X should be a (anticanonically-embedded) Del Pezzo surface (of type `Sch` or `Srfc` and $P2$ a projective plane both defined over $\mathbf{Q}$. The function returns `false` if X is not parametrizable over the rationals and returns `true` and a birational parameterization $P2 \to X$ otherwise.

This intrinsic is the main interface to a suite of functions parameterizing Del Pezzo surfaces (over $\mathbf{Q}$). These include the anticanonically Del Pezzos of degrees 1 and 2, which lie in non-trivially weighted projective spaces. A degree d Del Pezzo surface refers to a rational surface that is embedded in projective space by its anticanonical divisor. For degrees 1 and 2 this means an *ample* embedding into weighted projective space. For $3 <= d <= 9$, this is a very-ample embedding giving X as a degree d surface in ordinary projective space of dimension d.

It should be noted that not only do the routines handle the usual non-singular cases, but they also deal with degenerate singular cases (arising in degrees $d >= 3$. In the latter case, rather than blowing up $9 - d$ *distinct* points in the plane, some

of the blown-up points are "infinitely near" points: lying on the exceptional curves corresponding to already blown-up points). This is important as these degenerate cases can arise in the general parameterization of hypersurfaces in $\mathbf{P}^3$.

The package contains routines to find and blow down sets (defined over $\mathbf{Q}$) of exceptional curves, reducing to a DelPezzo of degree d, $5 <= d <= 9$. After this reduction, these cases are handled by the MAGMA routines described in Section 116.4.3 (which now also handle singular Del Pezzos). There is an exception to the above rule. For *singular* degree 3 and degree 4 Del Pezzo surfaces, it is more efficient to apply special case code directly rather than trying to blow down lines to get to higher degree. Starting in V2.17, special functions are provided do this that can be called directly for a singular anti-canonical degree 3 or 4 surface and are described in the next section.

The routines are not yet fully documented. For the location of exceptional curves and the blowing-down, some details may be found in [Sch98, Sec. 3.5] while [Man86] contains the general theory. There are also implementation notes in the appendix of the software documentation report [Bec08] from which this documentation has been adapted. For the nonsingular degrees 6, 8 and 9 cases, which use the Lie algebra method and the degree 5 case, see the references in Section 116.4.3.

116.4 Del Pezzo Surfaces

116.4.1 Introduction

This section contains a collection of geometric and arithmetic routines for Del Pezzo surfaces in their anti-canonical embeddings (the full weighted projective anticanonical embedding for degrees 1 and 2).

There are routines for creation, parametrization, minimisation and reduction, construction, computation of invariants for degree 3, and point-counting for degree 3 surfaces over a finite field.

There is a specialised type for Del Pezzos, `SrfDelPezzo`, which is a subtype of type `Srfc`. Some intrinsics use this type for arguments while some use the more general `Srfc` (or even `Sch`).

116.4.2 Creation of General Del Pezzos

`DelPezzoSurface(P,L)`

`DelPezzoSurface(S)`

`DelPezzoSurface(Z)`

The Del Pezzo surface of degree $9-d$, embedded by its anticanonical system, which arises by blowing up the projective plane P in the d points. The arguments are either plane P and a list of points L, the set of points Q, or the length d zero-dimensional scheme Z defining the points. If the points are not in sufficiently general position (so that the anticanonical image is not smooth) then an error is reported.

`DelPezzoSurface(f)`

The argument f should be a degree three homogeneous polynomial in a 4-variable polynomial ring P (with grevlex ordering). Creates the degree 3 Del Pezzo surface with defining polynomial f inside $\mathbf{P}^3 = Proj(P)$. If this surface is not smooth an error is reported.

`IsDelPezzo(Y)`

Returns `true` if and only if scheme Y in ordinary projective space is an abstract Del Pezzo surface. If so, it also returns the image X of the standard (pluri-)anticanonical embedding of Y, and the map $Y \to X$. Note that this can be a computationally very heavy function if Y is in a reasonably high-dimensional ambient.

116.4.3 Parametrization of Del Pezzo Surfaces

Del Pezzo surfaces are a special type of non-singular projective surface. A good reference for their general properties is [Man86]. A Del Pezzo surface X has a degree $1 \leq d \leq 9$. For $d \geq 3$, the standard representation is as a degree d surface in $\mathbf{P}^d$ for which a hyperplane section is an anti-canonical divisor. When we talk about Del Pezzos in this section, we mean a surface in that anti-canonical form.

Del Pezzo surfaces are birationally equivalent to the projective plane $\mathbf{P}^2$ over an algebraically-closed field. That is, there exists an invertible scheme map from $\mathbf{P}^2$ to X : a *parametrization*. Most of the functions in this section are concerned with the existence of parametrizations of X over a number field.

The significance of Del Pezzo surfaces comes from *adjunction theory* (see [SvdV87]). The adjunction map for surfaces is a general construct that contracts certain lines, known as exceptional lines, to points. Repeated application of the adjunction map to a rational surface results in a reduction to a surface in one of a small number of families, including the Del Pezzos. For a non-singular ordinary projective surface Y, the intrinsic `MinimalModelRationalSurface` is now available to produce the birational map from Y to one of the special, terminal cases that include the Del Pezzos.

Hence the parametrization problem for general rational surfaces reduces via adjunction (a purely algebraic construction) to parametrization of surfaces in the specific families, which is an arithmetic problem (ie dependent on the ground field). This is the surface analog of the simpler situation for curves. Any rational curve can be algebraically reduced to the projective line or a plane conic and the parametrization of plane conics is also an arithmetic problem.

Thus, the parametrization of Del Pezzo surfaces is an important component in that of general rational surfaces. General parametrization code for rational hypersurfaces in $\mathbf{P}^3$ is described in the previous section and it makes use of the routines described here.

If X is parametrizable with $d \geq 3$, then we can blow down (contract) exceptional lines on it to arrive at a surface with $d = 5$, 6, 8 or 9. The functions described here deal with parametrization in those cases using computational methods based around the Lie algebra of the automorphism group of X. For the theory behind these algorithms, see

[dG06], [dGP], [HS06] and [GSHPBS12]. In one of the examples, we parametrize a cubic hypersurface (degree 3 Del Pezzo) by blowing down to a degree 6 surface.

Although reduction to degree 9 is always possible, for $d = 7$ it is more efficient to work directly using the Lie algebra method. There is now also an intrinsic for $d = 7$ provided by Josef Schicho. Schicho has also written code for the $d = 5$ case, minimal or not, that uses the more geometrical method described in the above reference and a corresponding intrinsic is provided.

Furthermore, the degree $5 - 8$ intrinsics now also cover degenerate cases of singular Del Pezzos in their anticanonical projective embeddings. The additional code for the singular cases is also due to Josef Schicho.

A feature (from V2.17) is the provision of special case code for degree 3 and degree 4 singular Del Pezzos. For some of these, it is not possible to blow down any exceptional lines over the base field but the surface is still parametrizable. Additionally, it is usually much more efficient to handle these cases directly without blowing down curves to get to higher degree.

A general intrinsic `ParametrizeDelPezzo` for parametrizing any Del Pezzo surface of degree $d \geq 1$ in its anticanonical weighted embedding (for $d = 1, 2$ the anticanonical divisor is no longer very ample, but gives an ample embedding into weighted projective space) is described in the previous section. This blows down exceptional lines to reach degree $d \geq 5$ and then invokes one of the intrinsics described in this section. It is more efficient to call the appropriate intrinsic directly if the starting point is either the $d \geq 5$ case or the $d = 3,\ 4$ singular cases.

`SetVerbose("ParamDP", v)`

Set the verbose printing level for the Del Pezzo parametrizing functions. Currently the legal values for v are `true`, `false`, 0, 1 and 2 (`false` is the same as 0, and `true` is the same as 1).

`ParametrizeDegree9DelPezzo(X)`

Let X be a degree 9 Del Pezzo surface anticanonically embedded in 9-dimensional projective space. For this function the base field should be $\mathbf{Q}$. The surface X is defined by 27 degree 2 polynomials. The function performs only basic checks that the input X is valid.

If X is parametrizable over $\mathbf{Q}$ then there is a parametrization $\phi : \mathbf{P}^2 \to X$ which is everywhere-defined and given by cubic polynomials in the variables of $\mathbf{P}^2$. The function returns whether such a ϕ exists and, if so, ϕ also.

`ParametrizeDegree8DelPezzo(X)`

Let X be a degree 8 Del Pezzo surface anticanonically embedded in 8-dimensional projective space over a number field. The surface X is defined by 20 degree 2 polynomials. The function performs only basic checks that the input X is valid.

For degree 8, there are two types of non-singular Del Pezzo surface, the second type splitting into subfamilies:

1) X is isomorphic to $\mathbf{P}^2$ with a single rational point blown up.

2i) X is isomorphic to $T_1 \times T_2$ with T_i Galois twists of $\mathbf{P}^1$.

2ii) X is isomorphic to a Galois twist of $\mathbf{P}^1 \times \mathbf{P}^1$ where Galois acts transitively on the two $\mathbf{P}^1$ factors.

In case 1), X is always parametrizable.
In case 2i), X is parametrizable $\Leftrightarrow$ both T_i are trivial twists of $\mathbf{P}^1 \Leftrightarrow X$ is isomorphic to $\mathbf{P}^1 \times \mathbf{P}^1$.
In case 2ii), there is an infinite family of parametrizable X_a classified by $a \in \mathbf{Q}^*/\mathbf{Q}^{*2}$. The scheme X_a is isomorphic (properly, not just birationally) to the surface in $\mathbf{P}^3$ given by the equation $x_0^2 - ax_1^2 = x_2x_3$.

The main function determines whether X is parametrizable over $\mathbf{Q}$. If so, there is a parametrization $\phi : \mathbf{P}^2 \to X$ (given by cubic polynomials in case 1 and by degree 4 polynomials in case 2) and this is also returned.

The intrinsic also handles the degenerate case of a singular degree 8 Del Pezzo surface. This case is recognised directly from the Lie algebra computation which is part of the main routine and an appropriate adaptation of the general method is used.

Example H116E20

In this example, we parametrize an anticanonical sphere X_2, in the above notation. This is obviously an artificial illustration, as we start with X_2 in the form $F = x_0^2 - 2x_1^2 - x_2x_3 = 0$, which is trivial to parametrize directly! The surface in this form is embedded anticanonically in $\mathbf{P}^8$ by any 9-dimensional vector space complement of $< F >$ in the 10-dimensional linear system of all degree 2 polynomials in $x_0 \ldots x_3$. The parametrizing map is undefined at precisely 2 points of the plane. Geometrically, the map consists of a blowup of these 2 points followed by a blowdown of the line joining them.

```
> P3<x0,x1,x2,x3> := ProjectiveSpace(Rationals(),3);
> X2 := Scheme(P3,x0^2-2*x1^2-x2*x3);
> L := LinearSystem(P3,2);
> L := LinearSystemTrace(L,X2);
> P8<x1,x2,x3,x4,x5,x6,x7,x8,x9> := ProjectiveSpace(Rationals(),8);
> X := map<X2->P8|Sections(L)>(X2); X;
Scheme over Rational Field defined by
x1^2 - 2*x4^2 - x4*x8,
x1*x2 - 2*x4*x5 - x5*x8,
x2^2 - 2*x4*x7 - x7*x8,
x1*x3 - 2*x4*x6 - x5*x9,
x2*x3 - 2*x4*x8 - x7*x9,
x3^2 - 2*x4*x9 - x8*x9,
-x1*x5 + x2*x4,
-x1*x6 + x3*x4,
-x1*x7 + x2*x5,
-x1*x8 + x3*x5,
-x4*x7 + x5^2,
-x1*x8 + x2*x6,
-x1*x9 + x3*x6,
```

```
-x4*x8 + x5*x6,
-x4*x9 + x6^2,
-x2*x8 + x3*x7,
-x5*x8 + x6*x7,
-x2*x9 + x3*x8,
-x5*x9 + x6*x8,
-x7*x9 + x8^2
> boo,prm := ParametrizeDegree8DelPezzo(X);
> boo;
true
> prm;
Mapping from: Prj: P2 to Sch: X
with equations :
-1/4*U*V*W^2
-1/16*V*W^3
2*U^2*V*W - 4*V^3*W
-1/8*U^2*W^2
-1/32*U*W^3
U^3*W - 2*U*V^2*W
-1/128*W^4
1/4*U^2*W^2 - 1/2*V^2*W^2
-8*U^4 + 32*U^2*V^2 - 32*V^4
> bs := ReducedSubscheme(BaseScheme(prm)); bs;
Scheme over Rational Field defined by
U^2 - 2*V^2,
W
```

`ParametrizeDegree7DelPezzo(X)`

Let X be a degree 7 Del Pezzo surface anticanonically embedded in 7-dimensional projective space over a number field. We allow that X can be a degenerate (singular) Del Pezzo surface here. The scheme X is always parametrizable over the base field and this intrinsic returns such a parametrisation without reduction to degree 8 or 9 but directly from the Lie Algebra method.

`ParametrizeDegree6DelPezzo(X)`

`ExistenceOnly` BoolElt *Default :* `false`

Let X be a degree 6 Del Pezzo surface anticanonically embedded in 6-dimensional projective space. For this function the base field K may be $\mathbf{Q}$ or a number field. The surface X is defined by nine degree 2 polynomials. The function performs only basic checks that the input X is valid. **NB:** This intrinsic only handles the non-singular case. For a singular (degenerate) degree 6 Del Pezzo, use the intrinsic that follows.

The connected component of the automorphism group of X is a 2-dimensional torus over K. For any of the possible tori, there is a family of degree 6 Del Pezzos which correspond to principal homogeneous spaces of the torus up to isomorphism.

The parametrizability of X is equivalent to X corresponding to the trivial homogeneous space of its torus.

The function determines whether a parametrization $\phi : \mathbf{P}^2 \to X$ exists over K and returns one when this is the case. The degree of the polynomials defining a "minimal" ϕ (one which is undefined at the smallest number of points) is 3,4 or 6 depending on the torus type. The parametrization returned is always of this minimal degree.

The surface X is parametrizable if and only if it contains a point over K. Furthermore, it satisfies the local-global principle: it has a point over $K \Leftrightarrow$ it has a point over each p-adic completion of K. (These statements are also true for degree 8 and 9 Del Pezzos)

The `ExistenceOnly` option allows the function to just perform this local solubility check, deciding upon the existence of a parametrization without explicitly constructing one. Depending on the torus type, simultaneous norm equations over a degree 6 field extension of K or a single norm equation over a degree 3 extension of K may have to be solved to construct a parametrization. This is a hard computation, especially if K is not $\mathbf{Q}$, whereas the pure existence check is quite fast.

`Degree6DelPezzoType2_1(K,pt)`

`Degree6DelPezzoType2_2(K,pt)`

`Degree6DelPezzoType2_3(K,pt)`

`Degree6DelPezzoType3(K,pt)`

`Degree6DelPezzoType4(K,K1,pt)`

`Degree6DelPezzoType6(K,pt)`

These functions generate the parametrizable degree 6 Del Pezzo surface X whose (connected) automorphism group is the torus T, which comes from field data K, and which contains point pt.

The point pt must be in 6-dimensional projective space over the base field k of a number field K. Its first projective coordinate may not be 0 and, depending on the torus type, certain of its other coordinates must also be non-zero.

The torus types and corresponding fields K for the various functions are as follows ($pt = [a_0, \ldots, a_6]$):

`Type2_1`. K/k should be a quadratic extension. $T(k) = K^*$ and T acts on $\mathbf{P}^6$ to give an X with degree 3 minimal parametrization. pt satisfies not($a_1 = a_2 = 0$ or $a_3 = a_4 = 0$ or $a_5 = 0$ or $a_6 = 0$).

`Type2_2`. K/k should be a quadratic extension. $T(k) = K^*$ and T acts on $\mathbf{P}^6$ to give an X with degree 4 minimal parametrization. pt satisfies not($a_1 = a_2 = 0$ or $a_3 = a_4 = 0$ or $a_5 = a_6 = 0$).

`Type2_3`. K/k should be a quadratic extension. $T(k) = K^{*N_{K/k}=1} \times K^{*N_{K/k}=1}$. pt satisfies not($a_1 = a_2 = 0$ or $a_3 = a_4 = 0$ or $a_5 = a_6 = 0$).

`Type3`. K/k should be a cubic extension. $T(k) = K^{*N_{K/k}=1}$. *pt* satisfies not($a_1 = a_2 = a_3 = 0$ or $a_4 = a_5 = a_6 = 0$).

`Type4`. K/k and $K1/k$ should be distinct quadratic extensions. $T(k) = L^{*N_{L/K}=1}$ where L is $K.K1$. *pt* satisfies not($a_1 = a_2 = a_3 = a_4 = 0$ or $a_5 = a_6 = 0$).

`Type6`. K/k should be a degree 6 extension which contains cubic and quadratic subextensions K_3 and K_2. For simplicity, the precise condition is that the generator $y = K.1$ must have minimal polynomial of the form $x^6 + 2ax^4 + a^2x^2 - d$ and then $K_3 = k(y^2)$ and $K_2 = k(y^3 + ay)$. $T(k) = K^{*N_{K/K_3}=N_{K/K_2}=1}$. *pt* satisfies not($a_1 = a_2 = a_3 = a_4 = a_5 = a_6 = 0$).

`ParametrizeDelPezzoDeg6(X)`

This variant for parametrizing a degree 6 Del Pezzo also handles the degenerate (singular) case. Note however, that it doesn't recognise singularity from the Lie algebra computation as occurs for degrees 7 and 8. It tests for singularity at the start using the generic non-singularity computation that can be very slow. Therefore for known non-degenerate Del Pezzos of degree 6, it is always better to use the above `ParametrizeDegree6DelPezzo` directly.

That is also used here, if X turns out to be non-singular. Otherwise, projection from a singular point to $\mathbf{P}^5$ reduces the problem to that of parametrizing a rational scroll.

Example H116E21

In the this example, we start with a degree 3 Del Pezzo surface - a non-singular hypersurface in $\mathbf{P}^3$ - which contains the 3 disjoint lines $x = y = 0$, $z = t = 0$ and $x = z, y = t$. These are blown down to give a degree 6 Del Pezzo surface, the parametrisation of which gives a parametrisation of the original surface. As well as demonstrating the degree 6 code, this is a nice example of blowing down exceptional lines on surfaces, something for which more general code will be added at a future date.

```
> R3<x,y,z,t> := PolynomialRing(Rationals(),4,"grevlex");
> P3 := Proj(R3);
> //equation of the degree 3 surface:
> F := -x^2*z + x*z^2 - y*z^2 + x^2*t - y^2*t - y*z*t + x*t^2 + y*t^2;
> X3 := Scheme(P3,F);
> // get the ideal defing the union of the 3 lines:
> I1 := ideal<R3|[x,y]>;
> I2 := ideal<R3|[z,t]>;
> I3 := ideal<R3|[x-z,y-t]>;
> I := I1*I2*I3;
> I := Saturation(I);
```

General surface theory tells us that if H is the hyperplane divisor on X_3, then the blowing down is given by the projective map associated to the divisor $H + L_1 + L_2 + L_3$, where the L_i are our 3 lines. We need the global sections of the sheaf of this: if $L_1 + L_2 + L_3 \sim 2H - D$ (linear equivalence of divisors) for an effective divisor D, then the space of global sections" is" the degree 3 graded

part of the ideal of D (mod the equation of X_3). The ideal I_D of a suitable D is computed by requiring that $ID \cap I = (F, F_2)$ with F_2 a degree 2 polynomial in I.

```
> F2 := Basis(I)[5]; F2;
y*z - x*t
> ID := ColonIdeal(ideal<R3|[F,F2]>,I);
> ideal<R3|[F,F2]> eq (ID meet I);
true
> // get basis of degree 3 graded part of ID
> ID3 := ID meet ideal<R3|Setseq(MonomialsOfDegree(R3,3))>;
> B3 := MinimalBasis(ID3);
> B3;
[
    y*z*t - x*t^2,
    z^3 - z^2*t + t^3,
    y*z^2 - x*z*t,
    x*z^2 - x*z*t + y*t^2,
    y^2*z - x*y*t,
    x*y*z - x^2*t,
    x^2*z - x^2*t + y^2*t,
    x^3 - x^2*y + y^3
]
> // and a complementary subspace of F
> F in ideal<R3|Remove(B3,7)>;
false
> B3 := Remove(B3,7);
> // now map to the degree 6 Del Pezzo
> P6<a,b,c,d,e,f,g> := ProjectiveSpace(Rationals(),6);
> blow_down := map<X3->P6|B3>;
> X6 := blow_down(X3);
> Dimension(X6); Degree(X6);
2
6
```

We also need the inverse of blow down. The general `IsInvertible` function could be used here but again the general theory tells us that the inverse is given by linear equations and it is faster to find them directly by a Grobner basis plus linear algebra computation. We omit this for brevity and just assume the result.

```
> X3toX6 := iso<X3->X6|B3,[f,e,c,a]>;
> // now parametrise X6
> boo,prm := ParametrizeDegree6DelPezzo(X6);
> boo;
true
> p2toX3 := Expand(prm*Inverse(X3toX6));
> p2toX3;
Mapping from: Projective Space of dimension 2
Variables : $.1, $.2, $.3 to Sch: X3
with equations :
```

```
-77/9*$.1^3 + 59/6*$.1^2*$.2 + 10/3*$.1^2*$.3 + 8/9*$.1*$.2^2 -
    73/18*$.1*$.2*$.3 - 113/18*$.1*$.3^2 - 59/9*$.2^3 + 383/18*$.2^2*$.3 -
    259/9*$.2*$.3^2 + 329/18*$.3^3
253/18*$.1^3 - 193/6*$.1^2*$.2 - 17/3*$.1^2*$.3 + 695/18*$.1*$.2^2 -
    244/9*$.1*$.2*$.3 + 353/18*$.1*$.3^2 - 151/9*$.2^3 + 185/9*$.2^2*$.3 -
    41/9*$.2*$.3^2 - 79/9*$.3^3
-11/6*$.1^3 + 37/6*$.1^2*$.2 + 10/3*$.1^2*$.3 - 28/3*$.1*$.2^2 + 4*$.1*$.2*$.3 -
    7*$.1*$.3^2 + 8/3*$.2^3 + 8/3*$.2^2*$.3 - 11/2*$.2*$.3^2 + 9/2*$.3^3
11/18*$.1^3 + 8/3*$.1^2*$.2 - 1/6*$.1^2*$.3 - 28/9*$.1*$.2^2 - 2/9*$.1*$.2*$.3 -
    59/18*$.1*$.3^2 - 2/9*$.2^3 + 34/9*$.2^2*$.3 - 53/18*$.2*$.3^2 + 53/18*$.3^3
and inverse
-884/23043*x^3 + 884/23043*x^2*y - 884/23043*y^3 - 4436/23043*x*y*z -
    4334/23043*y^2*z + 6902/23043*x*z^2 - 3560/7681*y*z^2 - 4420/23043*z^3 +
    4436/23043*x^2*t + 4334/23043*x*y*t + 3778/23043*x*z*t + 4420/23043*y*z*t +
    4420/23043*z^2*t - 4420/23043*x*t^2 + 6902/23043*y*t^2 - 4420/23043*t^3
-442/23043*x^3 + 442/23043*x^2*y - 442/23043*y^3 - 3544/23043*x*y*z -
    6808/23043*y^2*z + 8800/23043*x*z^2 - 4392/7681*y*z^2 - 6290/23043*z^3 +
    3544/23043*x^2*t + 6808/23043*x*y*t + 4376/23043*x*z*t + 8744/23043*y*z*t +
    6290/23043*z^2*t - 8744/23043*x*t^2 + 8800/23043*y*t^2 - 6290/23043*t^3
-884/23043*x^3 + 884/23043*x^2*y - 884/23043*y^3 - 458/23043*x*y*z -
    5660/23043*y^2*z + 5828/23043*x*z^2 - 2854/7681*y*z^2 - 3910/23043*z^3 +
    458/23043*x^2*t + 5660/23043*x*y*t + 2734/23043*x*z*t + 6208/23043*y*z*t +
    3910/23043*z^2*t - 6208/23043*x*t^2 + 5828/23043*y*t^2 - 3910/23043*t^3
and alternative inverse equations:
...
```

`ParametrizeDegree5DelPezzo(X)`

Let X be a degree 5 Del Pezzo surface anticanonically embedded in 5-dimensional projective space over a number field. We allow that X can be a degenerate (singular) Del Pezzo here. The scheme X is always parametrizable over the base field and this intrinsic returns such a parametrisation without reduction to higher degree.

The scheme X has a finite automorphism group in this case, so the Lie Algebra method cannot be applied. However, there is a more geometric method using projections that works well for degree 5 and that is used here.

`ParametrizeSingularDegree3DelPezzo(X,P2)`

`ParametrizeSingularDegree4DelPezzo(X,P2)`

These two intrinsics compute whether a degree 3 (resp. 4) anticanonically embedded *singular* Del Pezzo X has a parametrization over the base number field k and, if so, return such a parametrization as a scheme map with inverse from $P2$ to X. A projective plane $P2$ over the same base field k is the second argument of the intrinsic and will be used as the domain of the map returned.

The conditions on X mean that is an irreducible degree 3 hypersurface in P^3 in the first case or an irreducible complete intersection of 2 quadrics in P^4 in the second

case, having only a finite number of singularities that are canonical A-D-E type in either case. The condition that there is a finite non-empty set of singularities is checked but whether these singularities are canonical is not checked. In the unlikely event that a degree 3 hypersurface or degree 4 complete intersection has finitely many singularities but one is non-canonical, the functions will fail at some point.

If there is a singular point p defined over the base field, projection from p gives an immediate inverse parametrization of X in the degree 3 case and maps X onto a line or conic bundle in P^3 in the degree 4 case, which is then parameterized by the special routines for those cases. There remain a small number of configurations of conjugate singularities in the contrary case, corresponding to certain special root subsytems of E_6 or D_5. For these, individual methods have been devised and implemented. These include an adaptation of the Lie algebra method for the degree 3 and 4 singular Del Pezzo surfaces that are actually toric.

Example H116E22

The following is an example of a degree 3 hypersurface in P^3 over Q, that is a singular Del Pezzo with 4 conjugate A_1 singularities. It is handled very easily by the special case code.

```
> Q := RationalField();
> P2<a,b,c> := ProjectiveSpace(Q,2);
> P3<x,y,z,t> := ProjectiveSpace(Q,3);
> X := Scheme(P3, -4*x^2*y + 16*x*y^2 - y^3 + 2*x^2*z - 2*x*y*z +
>     7/2*y^2*z - 252*x*z^2 + 16*y*z^2 - 55*z^3 + 10*x^2*t + 14*x*y*t -
>     61/2*y^2*t - 3400*x*z*t + 216*y*z*t - 261*z^2*t - 11468*x*t^2 +
>     728*y*t^2 + 3987*z*t^2 + 21889*t^3);
> ParametrizeSingularDegree3DelPezzo(X,P2);
true Mapping from: Prj: P2 to Sch: X
with equations :
1/4*a^3 + 435/2*a^2*b - 4743/4*a*b^2 + 968*b^3 + 257/16*a^2*c + 183/8*a*b*c -
    3647/16*b^2*c + 8*a*c^2 - 8*b*c^2 + c^3
-257/32*a^3 + 2547/32*a^2*b - 10419/32*a*b^2 + 8129/32*b^3 - 4*a^2*c + 22*a*b*c
    - 66*b^2*c - 1/2*a*c^2 + 1/2*b*c^2
-57/32*a^3 - 173*a^2*b + 26459/32*a*b^2 - 1529/16*b^3 - 29/2*a^2*c + 83/4*a*b*c
    - 25/4*b^2*c - 7/2*a*c^2 + b*c^2
-1/32*a^3 + 457/16*a^2*b - 4081/32*a*b^2 + 33/2*b^3 + 2*a^2*c - 7/4*a*b*c -
    1/4*b^2*c + 1/2*a*c^2
and inverse
x*y*z - 2*y^2*z + 63/2*z^3 + 2*x^2*t - 9*x*y*t + 35/2*y^2*t + 299/2*z^2*t -
    4567/2*z*t^2 - 25075/2*t^3
x^2*z - 1/4*y^2*z + 4*z^3 + 7*x^2*t - 2*x*y*t + 9/4*y^2*t + 19*z^2*t - 290*z*t^2
    - 1593*t^3
x^3 - 63/4*x*y^2 + y^3 + 256*x*z^2 - 65/4*y*z^2 + 3454*x*z*t - 439/2*y*z*t +
    11650*x*t^2 - 2957/4*y*t^2
```

116.4.4 Minimization and Reduction of Surfaces

Given an algebraic variety defined by several polynomials with integer coefficients, *reduction* asks for another embedding of this **Z**-scheme, such that the defining polynomials have smaller coefficients. *Minimization* asks for an isomorphic **Q**-scheme with minimal invariants. Many constructions of algebraic varieties lead to very bad models and thus it becomes necessary to perform minimization and reduction. Otherwise, subsequent calculations become impractical. The result of the minimization process is usually not unique.

Minimization is done locally for each prime of bad reduction. The local minimization routines and the reduction routines are directly accessible. They may be helpful for local computations or if the computation of all bad primes is too slow. Note that these subroutines do not check for semi-stability (in the sense of Mumford's geometric invariant theory). Unstable varieties may lead to infinite loops. As smooth hypersurfaces are known to be stable the initial computation of the bad primes will fail if an unstable variety is given.

In this section, minimization and reduction routines are described for Del Pezzo surfaces of degrees 3 (cubic surfaces) and 4.

Minimization and reduction is also available for various kinds of genus one curves (see Section 124.6) and plane quartics (see Section 114.12.3).

`MinimizeCubicSurface(f, p)`

Verbose `MinRedCubSurf` *Maximum :* 2

Given a cubic surface f as a homogeneous polynomial with integer coefficients, this routine performs a minimization at the place p. The new equation and the transformation matrix are returned. No checks of stability are done so that an unstable surface will lead to an infinite loop.

`ReduceCubicSurface(f)`

Verbose `MinRedCubSurf` *Maximum :* 2

Given a cubic surface f as a homogeneous polynomial with integral coefficients, this function computes a reduction of the surface. The second returned value is the transformation used.

`MinimizeReduceCubicSurface(f)`

Verbose `MinRedCubSurf` *Maximum :* 2

Given a smooth cubic surface f as a homogeneous polynomial with integer coefficients, this function computes a minimized and reduced model of the surface. The second return value is the transformation matrix. The transformation matrix applied to f will evaluate to a scalar multiple of the returned polynomial.

The algorithm is based on [Els].

`MinimizeDeg4delPezzo(f, p)`

Verbose `MinRedDeg4delPezzo` *Maximum :* 1

Given a degree 4 del Pezzo surface f as a sequence of two quadrics with integer coefficients, this function will compute a partially local minimized model for the place p. The second return value is the transformation matrix.

`MinimizeReduceDeg4delPezzo(f)`

Verbose `MinRedDeg4delPezzo` *Maximum :* 1

Given a degree 4 del Pezzo surface as a sequence of two quadrics with integral coefficients, this function computes a practically minimized and reduced model of the surface. The second return value is the transformation matrix.

The transformation matrix applied to the initial polynomials will evaluate to polynomials defining the same **Q** scheme as that defined by the returned quadrics.

For the reduction step, `ReduceQuadrics` is called.

`MinimizeReduce(S)`

Verbose `MinRedCubSurf` *Maximum :* 2

Verbose `MinRedDeg4delPezzo` *Maximum :* 1

Given a del Pezzo surface S of degree 3 or 4 this function will call the minimization and reduction routines described above and converts the output scheme X to a del Pezzo surface. The second returned value is the matrix hat maps S to the result.

Example H116E23

This example demonstrates minimization and reduction on del Pezzo surfaces of degree 3 and 4 obtained by blowing up rational points.

```
> P2 := ProjectiveSpace(RationalField(),2);
> pts := [P2| [-5,-10,-8], [-4,10,-4], [8,-2,-5], [0,-10,0], [1,5,7], [-7,-8,-6]];
> S := DelPezzoSurface(pts);
> _<W, X, Y, Z> := AmbientSpace(S); // give names to the variables
> S;
Del Pezzo Surface of degree 3 over Rational Field defined by
-W*X^2 + 318827/104630*X^3 + 46774615/29003436*W*X*Y -
    2039633371/290034360*X^2*Y - 2588798/7250859*W*Y^2 +
    246700427/58006872*X*Y^2 - 4904503/7250859*Y^3 + W^2*Z -
    318827/104630*W*X*Z + 34829/52315*X^2*Z + 117476057/58006872*W*Y*Z -
    2004449/27622320*X*Y*Z + 4769241/12890416*Y^2*Z - 34829/52315*W*Z^2 -
    44696243/72508590*Y*Z^2
> MinimizeReduce(S);
Del Pezzo Surface of degree 3 over Rational Field defined by
-22*W^2*X + 38*W*X^2 - 4*X^3 - 28*W^2*Y + 103*W*X*Y - 48*X^2*Y + 44*W*Y^2 -
    59*X*Y^2 - 24*Y^3 + 24*W^2*Z + 80*W*X*Z + 44*X^2*Z - 57*W*Y*Z + 73*X*Y*Z
```

```
    - 59*Y^2*Z - 79*W*Z^2 - 21*X*Z^2 - 5*Y*Z^2
```

Now we consider the surface of degree 4 obtained by blowing up only the first five points.

```
> T := DelPezzoSurface(pts[1..5]);
> _<V, W, X, Y, Z> := AmbientSpace(T);
> T;
Del Pezzo Surface of degree 4 over Rational Field defined by
-W^2 + 7031/194432*X^2 + V*Y + 10877/13888*X*Y + 47/280*W*Z + 412801/277760*X*Z
     - 78153/138880*Y*Z - 86217/9721600*Z^2,
-W*X + 1693/6944*X^2 + 23/496*X*Y + V*Z + 26763/9920*X*Z - 4003/4960*Y*Z
     - 233411/347200*Z^2
> MinimizeReduce(T);
Del Pezzo Surface of degree 4 over Rational Field defined by
-5*V^2 + 4*V*W + 8*W^2 + V*X - 8*X^2 - 8*V*Y + 3*W*Y + 3*X*Y + 15*Y^2 -
    2*V*Z - 16*X*Z - 10*Y*Z,
-2*V^2 + V*W + 3*W^2 + 2*V*X + 2*W*X - 6*X^2 + 3*V*Y + 3*W*Y - 17*X*Y -
    5*Y^2 + 5*V*Z - 22*X*Z - Y*Z + 13*Z^2
```

116.4.5 Cubic Surfaces over Finite Fields

In this section all cubic surface are represented by a homogeneous polynomial of degree 3 in a rank 4 polynomial ring. The coefficients are elements of a finite field.

`NumberOfPointsOnCubicSurface(f)`

Given a smooth cubic surface f over a finite field this routine computes the Frobenius action on the lines. The return values are the number of points of the surface and the Swinnerton-Dyer number of conjugacy class of the Weil group $W(E_6)$ that contains the Frobenius.

Example H116E24

```
> p := NextPrime(3^100);
> r<x,y,z,w> := PolynomialRing(GF(p),4);
> S := x^3 + 2* y^3 + 7* z^3 + 11 * w^3 - 5 * (-x-y-z-w)^3;
> NumberOfPointsOnCubicSurface(S);
265613988875874769338781322035779626829233452805946242111250315725884985311926\0
    79714208525578202
13
```

So we get a large number of points and the Frobenius has Swinnerton-Dyer number 13.

`IsIsomorphicCubicSurface(f,g)`

`UseLines` BoolElt *Default :* `false`

Given cubic surfaces f and g defined over finite fields, the intrinsic returns `true` if the surfaces are isomorphic. If f and G are isomorphic, there is a second return value comprising a list of matrices such that g^m will evaluate to a scalar multiple of f for each matrix m in the list. In the case where there are several isomorphisms over the algebraic closure of the basefield, one matrix for each isomorphism is returned.

Note that an isomorphism of smooth cubic surfaces is always given by a linear map.

The computation is based on an analysis of a finite set of points associated to the surface. Here we used the singularities of the hessian. If the hessian degenerates, the 135 intersection points of the lines are used. Setting `UseLines` to `true` indicates that the second algorithm is to be used.

As the algorithm involves huge field extensions it is only practical for surfaces over finite fields.

Example H116E25

```
> _<x,y,z,w> := PolynomialRing(GF(101),4);
> S := x^3 + y^3 + z^3 + w^3 - (x+y+z+w)^3;
> time a,b := IsIsomorphicCubicSurface(S, S);
Time: 0.530
> #b;
120
> S := x^3 + 2*y^3 + 7*z^3 + 5*w^3 - y*z*w + x^2*w + 2*y*z^2;
> time a,b := IsIsomorphicCubicSurface(S, S);
Time: 0.480
> #b;
1
> S := x^3 + y^3 + z^3 + w^3;
> time a,b := IsIsomorphicCubicSurface(S, S);
Time: 22.830
> #b;
648
```

Thus the diagonal cubic surface has 648 automorphisms and the Clebsch cubic surface has 120. Both examples are exceptional, a general cubic surface has a trivial automorphism group. The first example is much slower because the hessian degenerates and the 27 lines are used.

116.4.6 Construction of Cubic Surfaces

`CubicSurfaceByHexahedralCoefficients(pol)`

Given a separable polynomial p of degree 6 this intrinsic constructs a cubic surface having the roots of the p as hexahedral coefficients. These surfaces automatically have a Galois invariant set of 12 lines.

See [EJ10] for details.

`CoblesRadicand(p)`

Given a separable polynomial p of degree 6 this routine evaluates the Cobles quartic at the roots of p. Up to a square factor this value is the discriminant of the cubic surface constructed using hexahedral coefficients.

Example H116E26

```
> q<tt> := PolynomialRing(RationalField());
> p6 := tt^6 + 34*tt^4 + 180*tt^3 + 458*tt^2 + 524*tt + 212;
> CoblesRadicand(p6);
-676
> eqn := CubicSurfaceByHexahedralCoefficients(p6);
> Max([AbsoluteValue(c) : c in Coefficients(eqn)]);
13021618703131414093372560000 20
```

We have to use minimization and reduction to make further computations faster.

```
> S := MinimizeReduce(DelPezzoSurface(eqn));
> Equation(S);
6*y[1]^3 - 14*y[1]^2*y[2] + 6*y[1]*y[2]^2 - 6*y[2]^3 - 14*y[1]^2*y[3] +
    9*y[1]*y[2]*y[3] + 11*y[2]^2*y[3] - 21*y[1]*y[3]^2 + 14*y[2]*y[3]^2 +
    3*y[3]^3 - 3*y[1]*y[2]*y[4] + 10*y[2]^2*y[4] + 8*y[1]*y[3]*y[4] -
    53*y[2]*y[3]*y[4] + 40*y[3]^2*y[4] + 9*y[1]*y[4]^2 + 39*y[2]*y[4]^2 -
    23*y[3]*y[4]^2 - 16*y[4]^3
> M := PicardGaloisModule(S);
> Order(Group(M));
72
> CohomologyGroup( CohomologyModule(Group(M),M), 1);
Full Quotient RSpace of degree 2 over Integer Ring
Column moduli:
[ 2, 2 ]
```

Here the hexahedral approach gives us a cubic surface with nontrivial cohomology.

116.4.7 Invariant Theory of Cubic Surfaces

For background on this classical topic we refer to [Hun96, Appendix B] and [Sal58].

In this section a cubic surface is represented by a homogeneous polynomial of degree 3 in a rank 4 polynomial ring.

116.4.7.1 Invariants

By a theorem of Clebsch the ring of invariants of a cubic surface is generated by 5 invariants having degrees 8, 16, 24, 32 and 40. An explicit system of generators was found by Salmon. By Geometric Invariant Theory, stable cubic surfaces are isomorphic if and only if their invariants determine the same point in the weighted projective space $\mathbf{P}(1, 2, 3, 4, 5)$.

`ClebschSalmonInvariants(f)`

Computes a sequence of the numerical values of Salmon's invariants of the cubic surface given by the polynomial f. The second returned value is the discriminant of the surface.

`SkewInvariant100(f)`

Computes the numerical value a degree 100 skew invariant I_{100} of the cubic surface f. The square of I_{100} is an element of Clebsch invariant ring. It vanishes if and only if the cubic surface has an Eckardt point.

`CubicSurfaceFromClebschSalmon(inv)`

Computes a cubic surface whose invariants are equal to the given sequence. The algorithm requires the last invariant to be non-zero.

Example H116E27

```
> r4<x,y,z,w> := PolynomialRing(Rationals(),4);
> surf := r4!CubicSurfaceFromClebschSalmon([1,2,3,4,5]);
> surf := r4!MinimizeReduceCubicSurface(surf);
> surf;
-79*x^3 - 64*x^2*y + 228*x^2*z - 197*x^2*w + 320*x*y^2 - 470*x*y*z + 492*x*y*w
  - 180*x*z^2 - 94*x*z*w - 242*x*w^2 - 125*y^3 + 100*y^2*z + 94*y^2*w + 530*y*z^2
  - 886*y*z*w + 390*y*w^2 - 235*z^3 + 526*z^2*w - 825*z*w^2 +  279*w^3
> inv := ClebschSalmonInvariants(surf);
> inv;
[ 976235771549603375/3, 1906072563306098780753436239622781250/9,
9303881097343297830094619181364801014510415259433593 75/9,
3633112616588281944217451032208493848831995668840667046827293985351562500/81,
4433468122977074711513128802595347058077853330280167372742436742276136 4246\
3575851440429 6875/243 ]
> [inv[i] / inv[1]^i : i in [1..5]];
[ 1, 2, 3, 4, 5 ]
> SkewInvariant100(surf);
-14747658751687702477522103639772055954980186626723314226839431502066804 81\
8692101231381870506424535465849889285142677691430466259169068950411766 1282\
5810551009923477912337277997297305679991266945261596766795264557003974 9145\
50781250/531441
```

Thus the constructed surface has equivalent invariants (as they are in $\mathbf{P}(1, 2, 3, 4, 5)$) and no Eckardt points.

116.4.7.2 Covariants

A covariant is in the same ambient space as the initial surface. For example the equation itself is a covariant. In the case of a cubic surface this gives a degree 1 order 3 covariant. Products of covariants are again covariants. They form a ring over the ring of invariants.

`LinearCovariants(f)`

The intrinsic constructs a sequence containing Salmon's 4 linear covariants for the cubic surface f.

`ClassicalCovariantsOfCubicSurface(f)`

The intrinsic constructs a sequence containing the 4 classical covariants of the cubic surface f. The first one is the hessian. The next two are classically known as T and Θ. The last one is a degree 9 surface, which intersects f precisely in its 27 lines.

116.4.7.3 Contravariants

A contravariant is in the dual projective space of the initial surface. All contravariants form a ring over the ring of invariants. Contravariants can be constructed from invariants of varieties of the same degree but dimension one less by the Clebsch transfer principle.

`NumericClebschTransfer(f, inv, p)`

Given a form f and a user program `inv` that evaluates an invariant of a form of the same degree but one variable less, this function evaluates the corresponding contravariant of f at the point p. If this is repeated using sufficiently many different knots, it is possible to reconstruct a polynomial representation of the contravariant by interpolation.

`ContravariantsOfCubicSurface(f)`

Computes a sequence of 3 contravariants of the cubic surface f. By Clebsch transfer they correspond to the invariants S, T, and the discriminant of plane cubic curves. Thus the first one describes all hyperplanes such that the intersection with $f = 0$ gives a cubic curve with j-invariant equal to zero. The second gives all hyperplanes intersecting $f = 0$ in a cubic curve with j-invariant 1728 (as long as the intersection is smooth). The last one is $S^2 - 6T$. This is the degree 12 polynomial of the (formal) dual surface. It describes all hyperplanes such that the intersection with $f = 0$ is singular. For smooth surfaces this is equivalent to tangency. If $f = 0$ is singular the result will be reducible or even zero.

Example H116E28

This is the Cayley cubic surface. It has 4 singularities of type A_1 and each results in a linear factor of multiplicity two in the (formal) dual surface.

```
> r4<x,y,z,w> := PolynomialRing(Rationals(),4);
> surf := x*y*z + x*y*w + x*z*w + y*z*w;
> cont := ContravariantsOfCubicSurface(surf);
> Factorization(cont[3]);
```

```
[
    <w, 2>,
    <z, 2>,
    <y, 2>,
    <x, 2>,
    <x^4 - 4*x^3*y - 4*x^3*z - 4*x^3*w + 6*x^2*y^2 + 4*x^2*y*z + 4*x^2*y*w
     + 6*x^2*z^2 + 4*x^2*z*w + 6*x^2*w^2 - 4*x*y^3 + 4*x*y^2*z + 4*x*y^2*w
     + 4*x*y*z^2 - 40*x*y*z*w + 4*x*y*w^2 - 4*x*z^3 + 4*x*z^2*w + 4*x*z*w^2
     - 4*x*w^3 + y^4 - 4*y^3*z - 4*y^3*w + 6*y^2*z^2 + 4*y^2*z*w + 6*y^2*w^2
     - 4*y*z^3 + 4*y*z^2*w + 4*y*z*w^2 - 4*y*w^3 + z^4 - 4*z^3*w + 6*z^2*w^2
     - 4*z*w^3 + w^4, 1>
]
```

116.4.7.4 Interaction of Covariants and Contravariants

One can apply a contravariant to a covariant (or vice versa). The result is a new covariant (resp. contravariant) or an invariant. Its degree is the sum of the degrees of the arguments. The order is the difference of the two orders. If the order of the result is zero, it is an invariant.

One way to define the action is to interpret the contravariant as a differential operator, i.e. x_i^k acts as $\frac{\partial^k}{\partial x_i^k}$. Then one applies this differential operator to the covariant.

`ApplyContravariant(c, d)`

Given a covariant c and a polynomial d, this intrinsic interprets d as a differential operator (i.e., x is replaced by d/dx). It applies this operator to the polynomial c and returns the resulting polynomial.

In invariant theory d is a contravariant and c is a covariant.

Example H116E29

Here we compute Salmon's first invariant by applying a degree 4 order 4 contravariant to the hessian which is a degree 4 order 4 covariant.

```
> r4<x,y,z,w> := PolynomialRing(RationalField(),4);
> surf := x^3 + 2*y^3 + 3*z^3 + 5*w^3 - 2*x*y*(z-w) + (x+y+z+w)^3;
> cont := ContravariantsOfCubicSurface(surf);
> cov := ClassicalCovariantsOfCubicSurface(surf);
> ApplyContravariant(cont[1],cov[1]) / (2^11 * 3^9);
1438753/729
> ClebschSalmonInvariants(surf)[1];
1438753/729
```

116.4.8 The Pentahedron of a Cubic Surface

A general cubic surface can be written as a sum of 5 cubes of linear forms. These are unique up to scaling by third roots of unity and permutation. Thus we can associate 5 points in the dual projective space to a given cubic surface. They are called the faces of its pentahedron. In general the faces of the pentahedron are defined over a larger field.

The algorithm is described in [RS00].

`PentahedronIdeal(f)`

Computes the ideal of the faces of the pentahedron of the cubic surface f.

Example H116E30

The first example is a randomly chosen cubic surface. By construction it has a proper rational pentahedron.

```
> r4<x,y,z,w> := PolynomialRing(Rationals(),4);
> surf := x^3 + (x-y+2*z)^3 + (y-w)^3 + z^3 + (x - 3*y-2*z-7*w)^3;
> p_id := PentahedronIdeal(surf);
> Points(Cluster(ProjectiveSpace(Rationals(),3),Basis(p_id)));
{@ (-1/7 : 3/7 : 2/7 : 1), (0 : -1 : 0 : 1), (0 : 0 : 1 : 0),
   (1/2 : -1/2 : 1 : 0), (1 : 0 : 0 : 0) @}
```

The next example shows that the pentahedron of the diagonal cubic surface degenerates.

```
> diag := x^3 + y^3 + z^3 + w^3;
> p_id2 := PentahedronIdeal(diag);
> Points(Cluster(ProjectiveSpace(Rationals(),3),Basis(p_id2)));
{@ (0 : 0 : 0 : 1), (0 : 0 : 1 : 0), (0 : 1 : 0 : 0), (1 : 0 : 0 : 0) @}
```

The final example is a surface without a pentahedron.

```
> degen := x^3+y^3+z^3 + x*y*z+ w^3;
> p_id3 := PentahedronIdeal(degen);
> Points(Cluster(ProjectiveSpace(Rationals(),3),Basis(p_id3)));
{@ (0 : 0 : 0 : 1) @}
```

116.5 Bibliography

[**Bec07**] Tobias Beck. Formal Desingularization of Surfaces – The Jung Method Revisited –. Technical Report 2007-31, RICAM, December 2007. URL:http://www.ricam.oeaw.ac.at/publications/reports/.

[**Bec08**] Tobias Beck. Software Documentation. Technical Report 2008-8, RICAM, May 2008. URL:http://www.ricam.oeaw.ac.at/publications/reports/.

[**BHPdV04**] Barth, Hulek, Peters, and Van de Ven. *Compact Complex Surfaces*. Ergebnisse der Mathematik und ihrer Grenzgebiete 4. Springer, second edition, 2004.

[**BS08**] Tobias Beck and Josef Schicho. Adjoint Computation for Hypersurfaces Using Formal Desingularizations. Technical Report 2008-2, RICAM, January 2008. URL:http://www.ricam.oeaw.ac.at/publications/reports/.

[**DES93**] Decker, Ein, and Schreyer. Construction of surfaces in P4. *J. Algebraic Geometry*, 2:185–237, 1993.

[**dG06**] W. de Graaf, M. Harrison, J. Pilnikova and J. Schicho. A Lie Algebra Method for Rational Parametrization of Severi-Brauer Surfaces. *J. Alg.*, 303(2):514–529, 2006.

[**dGP**] W.A. de Graaf and J. Pilnikova. Parametrizing Del Pezzo surfaces of degree 8 using Lie algebras. URL:http://arxiv.org/abs/math.NT/0512477.

[**Eis05**] David Eisenbud. *The geometry of syzygies: a second course in commutative algebra and algebraic geometry*, volume 225 of *Graduate Texts in Mathematics*. Springer, New York–Berlin–Heidelberg, 2005.

[**EJ10**] Andreas-Stephan Elsenhans and Jörg Jahnel. Cubic surfaces with a Galois invariant double-six. *Cent. Eur. J. Math.*, 8(4):646–661, 2010.

[**Els**] Andreas-Stephan Elsenhans. Good models for cubic surfaces. (To appear).

[**GSHPBS12**] J. Gonzalez-Sanchez, M. Harrison, I. Polo-Blanco, and J. Schicho. Algorithms For Del Pezzo Surfaces of Degree 5 (Construction, Parametrization). 2012.

[**Har77**] Robin Hartshorne. *Algebraic Geometry, GTM 52*. Springer, ASpringer, 1977.

[**Hor76**] E. Horikawa. Algebraic Surfaces of General Type with Small c_1^2. II. *Invent. Math.*, 37:121–155, 1976.

[**HS06**] M.C. Harrison and J. Schicho. Rational Parametrisation for Degree 6 Del Pezzo Surfaces using Lie Algebras. In *Proceedings ISSAC'06*, 2006.

[**Hun96**] Bruce Hunt. *The geometry of some special arithmetic quotients*, volume 1637 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1996.

[**Man86**] Yu. I. Manin. *Cubic Forms (2nd ed.)*. North-Holland Publishing Co., Amsterdam, 1986. Vol. 4 of North-Holland Mathematical Library.

[**RS00**] Kristian Ranestad and Frank-Olaf Schreyer. Varieties of sums of powers. *J. Reine Angew. Math.*, 525:147–181, 2000.

[**Sal58**] George Salmon. *A treatise on the analytic geometry of three dimensions.* Revised by R. A. P. Rogers. 7th ed. Vol. 1. Edited by C. H. Rowe. Chelsea Publishing Company, New York, 1958.

[**Sch98**] Josef Schicho. Rational parametrization of surfaces. *J. Symbolic Comput.*, 26(1):1–29, 1998.

[**Sch00**] Josef Schicho. Proper parametrization of surfaces with a rational pencil. In *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation (St. Andrews)*, pages 292–300 (electronic), New York, 2000. ACM.

[**SvdV87**] A.J. Sommese and A. van der Ven. On the adjunction mapping. *Math. Ann.*, 278:593–603, 1987.

117 HILBERT SERIES OF POLARISED VARIETIES

Chapter 117
HILBERT SERIES OF POLARISED VARIETIES

117.1 Introduction

This chapter describes methods of generating graded rings that correspond to polarised algebraic varieties of various types. They can be used to generate examples of subcanonical curves—curves X polarised by a divisor D for which $kD = K_X$, the canonical class, for some integer k—K3 surfaces, Fano 3-folds and Calabi–Yau 3-folds. Of these, K3 surfaces are the best developed, and a database containing several thousand surfaces forms part of MAGMA.

117.1.1 Key Warning and Disclaimer

It is important to be aware of the nature and limitations of the output of the functions and databases described in this chapter. We list five of the issues in numbered points below, of which number 5 is the most important.

A typical example of a graded ring arises from a hyperelliptic curve C of genus g embedded in weighted projective space (wps):

$$C : (y^2 = f) \subset \mathbf{P}(1, 1, g + 1)$$

where the coordinates on $\mathbf{P}(1, 1, g + 1)$ are x_1, x_2, y of weights $1, 1, g + 1$ respectively, and $f = f_{2g+2}(x_1, x_2)$ is a homogeneous polynomial of degree $2g + 2$ in two variables having distinct roots. This embedded variety has homogeneous coordinate ring

$$R(C) = \frac{k[x_1, x_2, y]}{(y^2 - f)}$$

where k is the ground field. As a concise representation of this data, we record only a particular rational representation of the Hilbert series $P_R(t)$ of $R = R(C)$,

$$P_R(t) = 1 + 2t + 3t^2 + \cdots + (g + 1)t^g + (g + 3)t^{g+1} + \cdots = \frac{1 - t^{2g+2}}{(1 - t)(1 - t)(1 - t^{g+1})}$$

from which, as shown in Section 117.2.2, we deduce the fields

Weights = [1, 1, g+1], EquationDegrees = [2g+2].

Even though using this representation involves losing much information about the curve C, it has preserved enough detail so that it is still possible to do such things as create another

curve having the same basic invariants as C, or to recognise the family of hypersurfaces of degree 2 in $\mathbf{P}(1,1,g+1)$ to which C belongs.

This example illustrates several of the points one should keep in mind when interpreting the output of most functions in this chapter.

1. Weighted projective space (wps) The methods used here automatically generate examples of varieties defined by (weighted) homogeneous equations in wps, and so one must be familiar with wps from the outset. See Fletcher [IF00] for an accessible introduction to wps if necessary.

2. Field of definition Since the functions described in this chapter do not return literal equations of varieties, they do not assign a base field k. It is useful to have in mind $k = \mathbf{C}$, the complex numbers, but in many cases the base field is not relevant and one could work over any field. Having said that, there are cases where the base field is a crucial part of the problem.

3. Polarised varieties and their graded rings A *polarised variety* X, A is a variety X together with a divisor A that is ample on X; that is, there is a multiple kA of A which is a hyperplane section of X in some projective embedding $X \subset \mathbf{P}^N$. The homogeneous coordinate ring of X in this embedding is a graded ring which is generated in degree 1. But this embedding is not necessarily the one we want: the graded ring may be very large. Instead, we consider the *total graded ring* of A

$$R(X,A) = \oplus_{n\geq 0} H^0(X, \mathcal{O}_X(nA)),$$

which, with very few exceptions, is a much smaller ring. We do not define the terms precisely here, but suffice it to say that the Proj-correspondence between varieties in wps and the graded rings that are their homogeneous coordinate rings holds in this context between X, A and $R(X,A)$ just as it does for embeddings in ordinary projective space.

Thus we regard the following three pieces of data as being equivalent:

- a polarised variety X, A;
- the total graded ring $R(X,A)$ of a polarised variety X, A;
- the embedding $X \subset \mathbf{P}^N(w_0, \ldots, w_N)$ by all multiples of A, for some weights $w_0, \ldots, w_N$.

And so we use the words 'polarised variety' and 'graded ring' interchangeably. The fact that we also use 'variable', 'coordinate' and 'generator' synonymously is another reflection of this equivalence.

4. Numerical data of families of varieties In fact, we do not consider a single polarised variety X, A. Instead we record weaker information that characterises a family of varieties of which X, A is a particular member. The key piece of information that we work with is the Hilbert series $P_R(t)$ of the graded ring $R = R(X,A)$. This is calculated using the Riemann–Roch formula (RR) once we have decided which class of varieties we are concerned with. In favourable cases, RR takes as ingredients some discrete pieces of geometric data such as genus (which are invariant in flat families of suitably prescribed

varieties) and returns the dimension of the n-th graded piece R_n of the graded ring R. The algorithms work by taking such appropriate data as input, returning a Hilbert series (which is done by applying a formula that is hard-coded) and then analysing that series.

We can often produce extra information such as a prediction of weights $w_0, \ldots, w_N$ in which some suitable X is embedded. Some elementary examples of this are worked out Section 117.2.1.

5. Main problem In most cases, there are no criteria to determine whether a particular set of invariants for RR are actually the invariants of some polarised variety X, A. So even though the data that is then generated by MAGMA purports to be associated to some graded ring $R(X, A)$, there is no reason in any particular case why there really should exist such a polarised variety X, A. Fortunately, in many cases it is clear that there really is a variety that realises the output. In the example of the genus g hyperelliptic curve above, knowing the weights $(1, 1, g+1)$ and the degree of the equation, it is easy to see that there exists a nonsingular variety with these data and one could even write MAGMA routines to present an example using the scheme machinery of Chapter 117.10.2 and then attempt to construct a Weierstrass model of the hyperelliptic curve as described in Chapter 117.10.2 to obtain access to the specialist machinery provided for such curves.

As a rule, any polarised variety X, A that is described by the output of a function—even by the K3 database—cannot be assumed to exist, or if it does exist, it might not take exactly the form described. To prove that such a variety exists as described, it is sufficient to show that there is a quasi-smooth variety in the given wps having the given Hilbert series (or Hilbert numerator).

117.1.2 Overview of the Chapter

Graded ring calculations can be carried out in many different contexts. Included here are functions that work with subcanonical curves, K3 surfaces, Fano 3-folds and Calabi–Yau 3-folds. The latter three have appeared recently as parts of PhD theses, by Altinok [Alt98], Suzuki [Suz] and Buckley [Buc03] respectively. Other references for some of this material are [ABR02], [Rei00], [Pap03], [Bro03].

Section 117.2 gives a sketch of the theory of Hilbert series and describes functions that compute Hilbert series from Hilbert polynomials. It includes worked out examples of the elementary calculations that are behind most of the chapter. It contains the important definition of *Hilbert numerator* with respect to a collection of weights, which turns out to be the key point when we try to describe graded rings as the coordinate rings of polarised varieties.

Singularities are a main ingredient of RR in many applications, and Section 117.3 describes their construction and properties.

Five of the next six sections are devoted to different classes of polarised varieties. In fact, there are four specific classes of polarised variety, and one general class which encompasses them. Section 117.4 contains functions that apply to the general class, and thus are inherited by all classes: when consulting later sections, one should bear in mind that most basic functions will be described in this section. Section 117.5 covers the first

and most elementary application of graded ring methods to studying subcanonical curves, that is, curves polarised by a divisor that divides their canonical class.

K3 surfaces are described in Section 117.6. Although one can construct a single graded ring in isolation, a benefit of the graded ring methods is that they can be used generate large lists in one go. One such application is the amplification of Magma's K3 database from the 391 K3 surfaces in codimension at most 4 to the 24,099 cases in the current version. This is discussed in Section 117.7 which includes a precise statement characterising which K3 surfaces are included in the database and a further severe disclaimer.

There are two classes of 3-fold available: Fano 3-folds and Calabi–Yau 3-folds. The former is covered in Section 117.8, the latter in Section 117.9. Each of these is in a fairly early stage of development, having only basic creation functions and no systematic means for generating the large lists similar to those that exist for K3 surfaces. Nevertheless, one can begin to write lists. Section 117.10 describes how one can assemble such lists into Magma databases, although the process is somewhat technical and has its own limitations. Anyone attempting such lists will be aware that, following results of Kawamata, there are only finitely many deformation families of Fano 3-folds—Suzuki [Suz] classifies those of high Fano index—while it is still unknown whether or not there are finitely many families of Calabi–Yau 3-folds, Kreuzer and Skarke's vast lists [KS00] notwithstanding.

117.2 Hilbert Series and Graded Rings

The theory of Hilbert series is standard so it is only briefly touched on here. See Matsumura [Mat89] Section 13 for more details (or other any standard textbook on algebra such as Zariski–Samuel or Eisenbud.)

117.2.1 Hilbert Series and Hilbert Polynomials

Let $R = \oplus R_n$, the sum taken over $n \geq 0$, be a finitely-generated graded k-algebra with $R_0 = k$ and grading given by n. The *Hilbert polynomial* of R is the numerical polynomial $p(t)$ such that $\dim R_n = p(n)$ for all n sufficiently large. The *Hilbert series* of R is the power series

$$P(R) = \sum_{n \geq 0} (\dim R_n) t^n.$$

For example, let $C \subset \mathbf{P}^3$ be the twisted cubic defined by the three equations

$$xz = y^2, \quad xt = yz, \quad yt = z^2$$

and let

$$R = \frac{k[x, y, z, t]}{xz - y^2, xt - yz, yt - z^2}$$

be the homogeneous polynomial ring of C. Then one computes that

$$P(R) = 1 + 4t + 7t^2 + 10t^3 + \cdots$$

as follows. Certainly $R_0 = k$, and there are no equations of degree 1 so R_1 is the 4-dimensional vector space spanned by x, y, z, t. Now R_2 is spanned by the 10 quadrics $x^2, xy, \dots, t^2$, but these are related by the three equations, so $\dim R_2 = 10 - 3 = 7$. And so on. If you are systematic about it, you will discover quickly that the dimension of R_n is $3n + 1$ for $n \geq 1$. This can be proved by induction. So the Hilbert polynomial of R is $p(t) = 3t + 1$, which determines every coefficient of the Hilbert series $P(R)$ except the constant term.

Consider a bigger example: let $C = C_{10} \subset \mathbf{P}(1, 1, 5)$ be a hyperelliptic curve of genus 4 and degree 2 given by an equation

$$y^2 = f_{10}(x_1, x_2)$$

in coordinates x_1, x_2, y on wps $\mathbf{P}(1, 1, 5)$ where f_{10} is some general polynomial of degree 10. Again one calculates the Hilbert polynomial as $p(t) = 2t - 3$ and Hilbert series as

$$P(t) = 1 + 2t + 3t^2 + 4t^3 + 5t^4 + 7t^5 + 9t^6 + \cdots.$$

This power series can also be described as a rational function

$$P(t) = \frac{1 - t^{10}}{(1 - t)^2(1 - t^5)}$$

from which one can formulate a rule of thumb for relating this expression to the description of the curve as $C_{10} \subset \mathbf{P}(1, 1, 5)$ (or see the answer in Section 117.10.2 or [Rei00], Section 3.2).

Now there are other curves in wps that have the same Hilbert polynomial p as this example, but which have a different Hilbert series. To find one, consider modifying early terms of the series

$$P(t) = 1 + t + 2t^2 + 4t^3 + 5t^4 + 7t^5 + 9t^6 + \dots$$

and attempt to build the curve B in some wps. The coefficient of the term t (which is 1 in this case) implies that there is one linear coordinate x. Then the $2t^2$ demands one degree 2 coordinate y so that the 2-dimensional space of degree 2 homogeneous functions is spanned by x^2, y. In degree 3, x, y generate only a 2-dimensional space $\langle x^3, xy \rangle$, but the graded piece in degree 3 is 4-dimensional, according to the term $4t^3$, so there must be two more generators z_1, z_2. In degree 4 we see $x^4, x^2y, y^2, xz_1, xz_2$ which is just right if we assume that these are linearly independent (which we do). Similarly in degree 5 we hit the right number on the nose. But in degree 6 we see 11 monomials:

$$x^6, x^4y, x^2y^2, y^3, x^3z_1, xyz_1, x^3z_2, xyz_2, z_1^2, z_1z_2, z_2^2.$$

These lie in a 9-dimensional vector space (9 being the coefficient of t^6 in $P(t)$) so there must be two relations between them. We could carry on in this vein, but in fact we have done enough now: it is an exercise in wps to show that the curve

$$B = B_{6,6} \subset \mathbf{P}(1, 2, 3, 3)$$

defined by two general polynomials of degree 6 has genus 4 and degree 2 just as in the first example. We will work this example in MAGMA in the next subsection.

Thus the Hilbert polynomial determines the coefficients of high powers of t in the Hilbert series. So in this case, and more generally for the subcanonical curves of Section 117.5, to determine a Hilbert series P one must produce both the Hilbert polynomial p and a sequence V of the early coefficients that are not determined by p.

There is one extra twist that can happen: it is possible that there is not a single Hilbert polynomial p that determines the higher coefficients, but a sequence of polynomials $p_0, \ldots, p_{r-1}$ so that the nth coefficient is given by $p_i(n)$ where i is the residue of n modulo r. It is easy to generate examples of this using the intrinsics described below. This phenomenon occurs naturally when the RR data includes quotient singularities or fractional divisors, since these tend to make periodic corrections to an otherwise well-determined Hilbert polynomial, and one could rephrase this as above by saying that there is a sequence of Hilbert polynomials. In practice, this is a small distraction that is completely bound up in the main creation intrinsics.

```
HilbertFunction(p,V)
```

```
HilbertFunction(Q,V)
```

The Hilbert function determined by a univariate polynomial p (the Hilbert polynomial), or a sequence Q of univariate polynomials, and a sequence of initial values V. The returned object is a function whose domain is the integers.

```
HilbertSeries(p,V)
```

```
HilbertSeries(Q,V)
```

The Hilbert series determined by a univariate polynomial p (the Hilbert polynomial), or a sequence Q of univariate polynomials, and a sequence of initial values V. The returned object is a rational function in one variable.

117.2.2 Interpreting the Hilbert Numerator

The Hilbert series of a variety $X \subset \mathbf{P}(w_0, \ldots, w_N)$ defined by equations of degrees $d_1, \ldots, d_r$ has the form

$$P(t) = \frac{1 - t^{d_1} - t^{d_2} - \cdots - t^{d_r} + t^e \pm \ldots \pm t^k}{(1 - t^{w_0}) \cdots (1 - t^{w_N})}.$$

We refer to the numerator of this expression as the *Hilbert numerator with respect to the weights* $w_0, \ldots, w_N$, or simply *Hilbert numerator* if the weights are clear.

The aim is to apply a converse statement: if we can manipulate the Hilbert series of a polarised variety into such a form, we guess that it is embedded in $\mathbf{P}(w_0, \ldots, w_N)$ by equations of degrees given by low degree terms with negative coefficients (where the absolute value of the coefficient determines the number of equations of that degree).

This can be taken further: the syzygies, and indeed a free resolution of the ideal of X, are also encoded weakly in the weights and the Hilbert numerator. For more details see [Rei00], Section 3, or [ABR02], Section 4.

`HilbertSeriesMultipliedByMinimalDenominator(p,V)`

`HilbertSeriesMultipliedByMinimalDenominator(Q,V)`

Let h be the Hilbert series determined by a univariate polynomial p, or a sequence Q of univariate polynomials, and a sequence of initial values V. This function produces a rational expression $g(t)/\Pi(1-t^d)$ for h, returning the numerator g as first value and a sequence containing the factors $1-t^d$ of the denominator as second value.

`HilbertNumerator(g, D)`

The product $g \times \Pi(1-t^d)$ taken over all integers $d \in D$, assuming the result is a polynomial. This function is typically used when g is a Hilbert series and D is a proposed collection of weights for a variety realising this series.

Example H117E1

We reproduce the example described in the introduction to this Section 117.2.

```
> T<t> := PolynomialRing(Rationals());
> p := 2*t - 3;
> V := [ 1, 2, 3, 4 ];
> h_fun := HilbertFunction(p,V);
> [ h_fun(n) : n in [0..7]];
[ 1, 2, 3, 4, 5, 7, 9, 11 ]
> h := HilbertSeries(p,V);
> h;
(t^5 + 1)/(t^2 - 2*t + 1)
```

The Hilbert series will be the Hilbert series of a polarised curve C (since the degree of the Hilbert polynomial—the order of growth of the coefficients of the Hilbert series—is one). It is already expressed as a rational function, so we apply a power series ring coercion to see the dimensions of the graded pieces—we could increase the precision if required.

```
> S<s> := PowerSeriesRing(Rationals(),8);
> S ! h;
1 + 2*s + 3*s^2 + 4*s^3 + 5*s^4 + 7*s^5 + 9*s^6 + 11*s^7 + O(s^8)
```

The rule of thumb for interpreting Hilbert series as varieties defined by homogeneous polynomials in wps requires one to write the Hilbert series in the form P/Q for polynomials P, Q, where Q is a product of terms $1-t^a$ (corresponding to the coordinates of weight a). The next line computes the Hilbert series together with the minimal such expression.

```
> HilbertSeriesMultipliedByMinimalDenominator(p,V);
t^5 + 1
[ -t + 1, -t + 1 ]
```

One could interpret this return value as a Noether normalisation of the graded ring of C. Instead, we try some values for a new weight a hoping to stumble on the weights of a wps in which C embeds. We start with $a = 4$, then try $a = 5$.

```
> HilbertNumerator(h,[1,1,4]);
-t^9 + t^5 - t^4 + 1
```

```
> HilbertNumerator(h,[1,1,5]);
-t^10 + 1
```

The first answer made little sense. Having included a generator in degree 4 it demanded an equation of the same degree. That is certainly possible, but assuming the equation involves only the weight 1 variables, it would factorise and the result would not be a variety. However, the final answer is just what we hope for: it suggests that C is realised by a degree 10 curve in $\mathbf{P}(1,1,5)$, which makes perfect sense and is what we expected.
Now consider the modified Hilbert series of the second example of Section 117.2 in which we changed the early coefficients from $V = [1,2,3,4]$ to $[1,1,2,4]$, but kept the Hilbert polynomial p.

```
> h1 := HilbertSeries(p,[1,1,2,4]);
> S ! h1;
1 + s + 2*s^2 + 4*s^3 + 5*s^4 + 7*s^5 + 9*s^6 + 11*s^7 + O(s^8)
> HilbertNumerator(h1,[1,1,5]);
-t^10 + t^9 - t^8 - t^7 + t^6 - t^4 + t^3 + t^2 - t + 1
```

This implies that there is an equation in degree 1. Such an equation immediately eliminates one of the two degree 1 coordinates which would be daft, so we eliminate this redundancy right at the beginning by trying instead

```
> HilbertNumerator(h1,[1,5]);
t^9 + t^7 + 2*t^6 + t^5 + t^4 + 2*t^3 + t^2 + 1
```

In short, this suggests a new variable in degree 2 and two new variables in degree 3: this could be done methodically by considering one generator at a time.

```
> HilbertNumerator(h1,[1,2,3,3,5]);
-t^17 + t^12 + 2*t^11 - 2*t^6 - t^5 + 1
```

Now examining this Hilbert numerator, we see that the first equation is in degree 5. But there is a variable of degree 5. Although that is not a problem, we could try to remove the degree 5 variable, as we would, in practice, if we knew that it appeared linearly in the equation.

```
> HilbertNumerator(h1,[1,2,3,3]);
t^12 - 2*t^6 + 1
```

Finally, this suggests $B_{6,6} \subset \mathbf{P}(1,2,3,3)$, and again it is an easy exercise in wps to confirm that such a curve really does exist with the predicted properties.

`FindFirstGenerators(g)`

This intrinsic function returns a sequence containing the results of a first attempt to deduce plausible weights of generators for the variety with Hilbert series g. The method used proceeds by advancing through the coefficients of g, in order of increasing degree, adding generators of that degree for each positive coefficient. The algorithm stops when it first finds a negative coefficient.

Example H117E2

We first make a Hilbert series which has initial terms $1 + 2t + 3t^2 + 4t^3$.

```
> T<t> := PolynomialRing(Rationals());
> p := 2*t - 3;
> V := [ 1, 2, 3, 4 ];
> h := HilbertSeries(p,V);
> h;
(t^5 + 1)/(t^2 - 2*t + 1)
> S<s> := PowerSeriesRing(Rationals(),4);
> S ! h;
1 + 2*s + 3*s^2 + 4*s^3 + O(s^4)
```

This h is, in fact, the Hilbert series of a graded ring generated by three elements in degrees 1,1,5.

```
> FindFirstGenerators(h);
[ 1, 1, 5 ]
```

There is no guarantee that this intrinsic will always find suitable weights, but it does return a subset of the weights that must occur for any variety that realises the given Hilbert series.

`ApparentCodimension(f)`

`ApparentEquationDegrees(f)`

`ApparentSyzygyDegrees(f)`

If $f = f(t)$ is a polynomial of the form

$$f = 1 - \sum_{i=1}^{N_0} a_{0,i}t^i + \sum_{k=N_0+1}^{N_1} a_{1,i}t^i - \ldots + (-1)^{k-1} \sum_{i=N_{k-2}+1}^{N_{k-1}} a_{k-1,i}t^i + (-1)^k t^{N_k}$$

then the apparent codimension is k, the apparent equation degrees are those i for which $a_{0,i}$ is nonzero (with $a_{0,i}$ equations having that degree i) and the apparent syzygy degrees are those i for which $a_{1,i}$ is nonzero.

117.3 Baskets of Singularities

We describe how to create the various point and curve singularities, together with collections or *baskets* of these singularities that are an ingredient of RR in higher dimensions.

Recall the basic principle of singularity contributions in RR: when we nominate a basket of singularities in the RR formula we do not necessarily expect the corresponding variety to have exactly those singularities associated with it (although that is very commonly the case), but rather to possess singularities which make exactly the same contribution to the RR formula as the singularities of the basket. Thus, we only allow the creation of a restricted class of singularities that (in good cases) allow us to realise all possible contributions.

117.3.1 Point Singularities

The point singularities that are allowed are always finite cyclic quotient singularities. In the surface case (over the complex numbers $\mathbf{C}$) they are always analytically isomorphic to a neighbourhood of the origin in the quotient $\mathbf{C}^2/G$, where G is the group of r-th roots of unity and the action is determined on eigencoordinates x, y of $\mathbf{C}^2$ by the action of a primitive root of unity $\lambda \in G$:

$$\lambda \cdot x = \lambda^a x, \quad \lambda \cdot y = \lambda^b y$$

for some $a, b \in \{0, \ldots, r-1\}$. Such a singularity is denoted by the symbol $\frac{1}{r}(a, b)$.

Similarly, one defines 3-dimensional quotient singularities, where the symbol $\frac{1}{r}(a, b, c)$ denotes a cyclic quotient singularity. Of course, one can go into yet higher dimensions, with symbol $\frac{1}{r}(a_1, \ldots, a_k)$, but although the functions of this chapter can create such singularities, they do not yet calculate RR for singularities of dimension higher than three. By definition, if $p = \frac{1}{r}(a_1, \ldots, a_k)$, the *index of* p is the positive integer r and the *polarisation of* p is the sequence $[a_1, \ldots, a_k]$.

The four main situations in which we use (quotient) point singularities are:

- Gorenstein surface singularities $\frac{1}{r}(a, r-a)$ with r coprime to a (for K3 surfaces)
- terminal 3-fold singularities $\frac{1}{r}(a, r-a, b)$ with r coprime to a, b (for Fano 3-folds)
- isolated canonical 3-fold singularities $\frac{1}{r}(a, b, c)$ with $a+b+c = 0 \bmod r$ and r coprime to each of a, b, c (for Calabi–Yau 3-folds)
- nonisolated canonical 3-fold singularities $\frac{1}{r}(a, b, c)$ with $a+b+c = 0 \bmod r$ and no three of r, a, b, c sharing a nontrivial common factor (for Calabi–Yau 3-folds).

In the final, nonisolated case, the points can be points at the intersection of two branches of the 1-dimensional singular loci, or other points on 1-dimensional singular loci that do not have the generic transverse behaviour. (This case is discussed further in Section 117.10.2 on curve singularities below.)

There is an additional key piece of data. The contribution of a point p to RR for $\chi(X, A)$ depends upon the eigenspace of the G-action in which A lies. In other words, the singularity is also polarised locally by A. This is called the *local polarisation* or the *eigenspace* of the singularity (the latter to distinguish it more clearly from the polarisation), and is another integer n in the range $\{0, \ldots, r-1\}$. When we need to include the eigenspace of A in the singularity, we use the symbol $\frac{1}{r}(a_1, \ldots, a_k)_n$.

Example H117E3

We make a point that can lie on a surface.

```
> p := Point(7,[3,4]);
> p;
1/7(3,4)
> IsGorensteinSurface(p);
true
```

Now we make a point of a 3-fold.

```
> q := Point(5,2,[1,2,3]);
```

```
> q;
1/5(1,2,3)_[2]
> IsTerminalThreefold(q);
true
> Eigenspace(q);
2
> p eq q;
false
```

We did not assign an eigenspace to the point p, so a default value was assigned.

```
> Eigenspace(p);
-1
```

117.3.1.1 Creation of Point Singularities

`Point(r,n,Q)`

`Point(r,Q)`

The point singularity with index a positive integer r, polarisation a sequence of positive integers Q and local polarisation an integer n. The group action should have no quasi-reflections, which means that no integer $k > 1$ is allowed to divide r and all but one of the elements of Q. The local polarisation n modulo r should be a unit modulo r. If it is not given as an argument, the default value is $n = -1$.

117.3.1.2 Accessing the Key Data and Testing Equality

`Dimension(p)`

Dimension of the variety on which the singularity p lies.

`Index(p)`

Index of the singularity p.

`Polarisation(p)`

Polarisation sequence of the singularity p.

`Eigenspace(p)`

Eigenspace of the polarising divisor of the point singularity p.

`p eq q`

Return `true` if and only if the two point singularities p, q have the following attributes equal: dimension, index, polarisation, eigenspace.

117.3.1.3 Identifying Special Types of Point Singularity

`IsIsolated(p)`

Return `true` if and only if the point singularity p is an isolated singularity; that is, if and only if every element of its polarisation is coprime to its index.

`IsGorensteinSurface(p)`

Return `true` if and only if the point singularity p is a Gorenstein surface point; that is, if and only if p is of type $\frac{1}{r}(a, r-a)$ for r, a coprime.

`IsTerminalThreefold(p)`

Return `true` if and only if the point singularity p is a terminal 3-fold point; that is, if and only if it is isolated, dimension 3 and the polarisation is of the form $a, b, r-b$, up to a permutation.

`TerminalIndex(p)`

The integer a when p is a singular point with polarisation sequence of the form $a, b, r-b$, up to a permutation, assuming that p is a terminal 3-fold point singularity.

`TerminalPolarisation(p)`

Polarisation sequence of the singularity p in the order $a, b, r-b$, where r is the index of p, and $b \leq r/2$. There will be an error if p is not a terminal 3-fold point singularity.

`IsCanonical(p)`

Return `true` if and only if the point singularity p is canonical; that is, if and only if the sum of the polarisation is trivial modulo the index.

117.3.2 Curve Singularities

We follow Buckley's analysis of curve singularities in RR [Buc03]. The first thing to note is that curve singularities in MAGMA are always 3-dimensional, that is, they are one-dimensional singular loci $C \subset X$ of polarised 3-folds X, A. The *degree* of C is the intersection number AC. At a general point of a curve singularity C, a transverse section is a Gorenstein surface quotient singularity, that is, a point of type $\frac{1}{r}(a, r-a)$ for coprime integers r, a. This is called the *transverse type* of C and r is called the *transverse index* of C.

There are two further key attributes of a curve singularity C: two integers N, t that carry information about the normal bundle of C and about possible special points on C. We do not discuss the meaning of N here except to say that it encodes the splitting type of the normal bundle (see [Buc03] and [SB] for further information), but in any case note that there are intrinsics described below that can identify appropriate values for N given some other invariants. This pair of invariants occur in RR only in the combination N/t, which itself appears only linearly. So given enough coefficients of the Hilbert series as

input, this value can be recovered. A curve, therefore, does have an attribute 'magic' that records this magic number N/t, and this can be set (and used in RR) even if N, t are not assigned.

We explain the attribute t. The transverse type of C gives a transverse section at any general point along C. But there may be special points, so-called *dissident points*, that do not have such a section. They certainly occur at an intersection point of two curve singularities, but may also be other points in C. They are quotient singularities $\{p_i\}$ whose indices are of the form rt_i for positive integers t_i, where r is the transverse index of C. The invariant t, called the *index* of C, is the GCD of the set of all such t_i.

Example H117E4

We create a curve singularity with given transverse type.

```
> p := Point(3,[1,2]);
> C := Curve(1/3,p,4,3);
> C;
Curve of degree 1/3, N = 4, t = 3 with transverse type 1/3(1,2)
```

We check some of the characteristics of C.

```
> TransverseIndex(C);
3
> IsCanonical(C);
true
> MagicNumber(C);
4/3
```

117.3.2.1 Creation of Curve Singularities

`Curve(d,p,m)`

`Curve(d,p,N)`

`Curve(d,p,N,t)`

The 3-fold curve singularity of degree d, transverse type p (a point surface singularity) and characteristic numbers N, t (which is 1 if not set) or magic number $m = N/t$.

117.3.2.2 Accessing the Key Data and Testing Equality

`Degree(C)`

The degree of the curve singularity C.

`TransverseType(C)`

The transverse type of the curve singularity C, that is, a point surface singularity that is the general transverse section of C.

`TransverseIndex(C)`

The transverse index of the curve singularity C, that is, the index of the point surface singularity that is a general transverse section of C.

`NormalNumber(C)`

The invariant N of the curve singularity C.

`Index(C)`

The invariant t (sometimes called τ) of the curve singularity C that is determined by the indices of dissident points on C.

`MagicNumber(C)`

The number N/t of the curve singularity C, where N is the normal number of C and t is the index of C.

`Dimension(C)`

The dimension of the curve singularity C (currently always 3).

`IsCanonical(C)`

Return `true` if and only if the transverse type of the curve singularity C is a canonical (or Du Val) surface singularity.

`C eq D`

Return `true` if and only if the two curve singularities C, D have the following attributes equal: dimension, degree, transverse type, magic number (or, equivalently, the pair of invariants N, t).

117.3.3 Baskets of Singularities

A *basket of singularities*, or simply *basket*, is a collection of point and curve singularities. One constructs baskets in the hope of finding a variety that has exactly those singularities lying on it as its only singularities, and very often this is indeed the case. It is a marginal issue here, but worth noting that in fact baskets are collections of ideal singularities of a variety X that make exactly the same contributions to RR as the actual singularities of X. So as a matter of principle, one is not primarily seeking varieties with exactly the basket singularities, even though this is what happens in practice.

117.3.3.1 Creation and Modification of Baskets

`Basket(Q)`

`Basket(Q1,Q2)`

The basket of singularities where Q, Q_1, Q_2 are sequences of point or curve singularities.

`EmptyBasket()`

The basket of singularities containing no singularities.

`MakeBasket(Q)`

The basket of singularities containing point singularities that are encoded in the sequence Q. This may occur in different ways. If Q is a sequence of sequences of the form $[r, a]$ (that is, each having length 2) with r, a coprime, then the result will be a basket of points of the form $\frac{1}{r}(a, r-a)$. If Q is a sequence of sequences of some common length $N > 2$, then the result will be a basket of points of the form $\frac{1}{Q[1]}(Q[2], \ldots, Q[N])$. Note that a local polarisation n cannot be included in this constructor: the default value $n = -1$ is always assumed.

`Points(B)`

The sequence of point singularities of the basket B.

`Curves(B)`

The sequence of curve singularities of the basket B.

117.3.3.2 Tests for Baskets

`IsIsolated(B)`

Return `true` if and only if all singularities of the basket B are isolated (so, in particular, there are no curve singularities in B).

`IsGorensteinSurface(B)`

Return `true` if and only if all singularities of the basket B are Du Val singularities, that is, are of the form $\frac{1}{r}(a, r-a)$.

`IsTerminalThreefold(B)`

Return `true` if and only if all singularities of the basket B are terminal 3-fold singularities.

`IsCanonical(B)`

Return `true` if and only if all singularities of the basket B are canonical singularities.

117.3.4 Curves and Dissident Points

Curves of singularities may contain special points not of the typical transverse type; these are the dissident points. The following intrinsics generate sequences of possible dissident points for a particular curve singularity. The point is that some of the invariants of a curve singularity force a curve to have dissident points that together give a certain RR contribution. It is not always easy to see what these points should be, so constructing by hand baskets that include a particular curve singularity can be difficult. These intrinsics all give suggestions for possible sets of dissident points.

`CanonicalDissidentPoints(C)`

A sequence of sequences of points, each of which minimally accounts for the index of the curve singularity C (although further curves may be needed in order for it to make sense).

`SimpleCanonicalDissidentPoints(C)`

A sequence of sequences of points, each of which minimally accounts for the index of the curve singularity C and which does not allow further curves to meet C.

`PossibleCanonicalDissidentPoints(C)`

A sequence of points, each of may appear on the curve singularity C as a dissident point.

`PossibleSimpleCanonicalDissidentPoints(C)`

A sequence of points, each of may appear on the curve singularity C as a dissident point but which is not at the intersection of C with another curve.

117.4 Generic Polarised Varieties

Recall from Section 117.1.1 that, despite some ambiguity, we regard the following as being equivalent: polarised varieties X, A; schemes in wps $X \subset \mathbf{P}^N(w_0, \ldots, w_N)$ where A is a degree 1 hyperplane section; and data about the Hilbert series of the graded ring $R(X, A)$. Thus we constantly refer to a *polarised variety* X, and we expect to be able to retrieve its Hilbert series, its dimension, the codimension of its embedding and other such data.

With one exception, the intrinsics described in this section can be applied to all polarised varieties. The exception is the following little-used intrinsic that creates a polarised variety that is not of a specific type.

`PolarisedVariety(d,W,n)`

The polarised variety of dimension d, with weights given by the sequence W of positive integers and with Hilbert numerator the univariate polynomial n.

117.4.1 Accessing the Data

`Weights(X)`

The weights of the polarised variety X.

`Degree(X)`

The degree of the polarised variety X.

`Basket(X)`

The basket of singularities of the polarised variety X.

`RawBasket(X)`

The basket of singularities of the polarised variety X in sequence format, that is, the basket is a sequence of sequences, in which a singularity $\frac{1}{r}(a, b, c)$ is represented as a sequence $[r, a, b, c]$ of integers. (The Gorenstein surface singularity $\frac{1}{r}(a, r-a)$ admits further abbreviation to $[r, a]$.) Notice that the local polarisation n is not included in this raw basket data; its default value $n = -1$ is assumed.

`Dimension(X)`

The dimension of the polarised variety X.

`Codimension(X)`

The codimension of the polarised variety X.

`HilbertNumerator(X)`
`Numerator(X)`

The numerator $f(t)$ of the Hilbert series $P(t)$ of the polarised variety X when expressed as a rational function $P = f(t)/\&*[1 - t^w : w \in W]$ where the product in the denominator is taken over W, the sequence of weights of X.

`NoetherWeights(X)`

The weights corresponding to a Noether normalisation of the polarised variety X. In other words, these are the weights of polynomials in the graded ring of X that generate a polynomial subring of maximal dimension.

`NoetherNumerator(X)`

The numerator $n(t)$ of the Hilbert series $P(t)$ of the polarised variety X when expressed as a rational function $P = n(t)/\&*[1 - t^w : w \in N]$, where the product in the denominator is taken over N, the sequence of Noether weights of X.

`NoetherNormalisation(X)`

Given a polarised variety X return a pair, the first term of which is the sequence of Noether weights, the second the corresponding numerator.

`HilbertSeries(X)`

The Hilbert series of the polarised variety X expressed as a rational function.

`InitialCoefficients(X)`

The coefficients of the Hilbert series of the polarised variety X expressed as a power series. The number of coefficients returned is equal to the precision of the power series ring in which the Hilbert series was expanded.

`ApparentCodimension(X)`

`ApparentEquationDegrees(X)`

`ApparentSyzygyDegrees(X)`

`BettiNumbers(X)`

If $n(t)$ is the Hilbert numerator of X and is of the form

$$n = 1 - \sum_{i=1}^{N_0} a_{0,i}t^i + \sum_{k=N_0+1}^{N_1} a_{1,i}t^i - \ldots + (-1)^{k-1} \sum_{i=N_{k-2}+1}^{N_{k-1}} a_{k-1,i}t^i + (-1)^k t^{N_k}$$

then the apparent codimension of X is k, the apparent equation degrees are given by those i for which $a_{0,i}$ is nonzero (with $a_{0,i}$ equations of that degree i) and the apparent syzygy degrees are those integers i for which $a_{1,i}$ is nonzero. The Betti numbers are a sequence with first element the sum of all $a_{0,i}$, second element the sum of all $a_{1,i}$, and so on until $a_{k-1,i}$.

117.4.2 Generic Creation, Checking, Changing

Procedural versions of intrinsic functions modify polarised varieties at the generic level because they preserve any subtypes; functional versions exist for special types of polarised variety but not in general.

`X eq Y`

Return `true` if and only if the polarised varieties X and Y have the same dimension, weights, basket and Hilbert numerator. In particular, these conditions imply that X and Y have the same Hilbert series.

`CheckCodimension(X)`

Return `true` if and only if the codimension of X is equal to the apparent codimension of X determined by its Hilbert numerator.

`FirstWeights(X)`

These are weights assigned to the polarised variety X during its construction that carry some relevance; if no such weights were assigned, the usual weights of X will be returned.

`IncludeWeight(~X,w)`

Include the positive integer w among the weights of X, adjusting all other data associated to the embedding of X as required.

`RemoveWeight(~X,w)`

Remove the positive integer w from the weights of X, assuming it appears there and can be removed without destroying the property of the Hilbert numerator being a polynomial. All other data associated to the embedding of X is modified as required.

`MinimiseWeights(~X)`

Remove any weights from X whose presence is not required to keep the Hilbert numerator of X a polynomial.

117.5 Subcanonical Curves

A *subcanonical curve* is a polarised variety C, D where C is a nonsingular curve of genus $g \geq 2$ and D is a divisor on C such that $K_C = kD$ for some positive integer k.

117.5.1 Creation of Subcanonical Curves

`SubcanonicalCurve(g,d,Q)`

The subcanonical curve C, D of genus g, degree d and initial Hilbert series coefficients Q.

`IsSubcanonicalCurve(g,d,Q)`

Return `true` if and only if the data g, d, Q passes some basic checks that there is a subcanonical curve C, D of genus g, degree d and initial Hilbert series coefficients Q. In that case, the second return value is such a curve.

`HilbertPolynomialOfCurve(g,m)`

The Hilbert polynomial $mt + 1 - g$ of a divisor of degree m on a curve of genus g.

`IsEffective(C)`

Return `true` if and only if the polarising divisor of the subcanonical curve C is effective; that is, if and only if the Hilbert series has the form $1 + p_1 t + \cdots$ with $p_1 > 0$.

117.5.2 Catalogue of Subcanonical Curves

This section describes intrinsics that allow the user to generate many examples of Hilbert series of subcanonical curves and attempt to interpret them as curves embedded in wps.

`EffectiveSubcanonicalCurves(g)`

`EffectiveSubcanonicalCurves(g,d)`

A sequence containing data for effective subcanonical curves of genus $g \geq 3$ (polarised by a divisor of degree d if the second argument is given).

`IneffectiveSubcanonicalCurves(g)`

`IneffectiveSubcanonicalCurves(g,d)`

A sequence containing data for ineffective subcanonical curves of genus $g \geq 3$ (polarised by a divisor of degree d if the second argument is given).

117.6 K3 Surfaces

This section describes intrinsics that construct K3 surfaces. It also describes a few intrinsics that can be used to study them, but see Section 117.4 for the general intrinsics that apply to all polarised varieties.

The calculations are based on Altinok's Riemann–Roch formula [Alt98] for polarised K3 surfaces with Du Val singularities.

117.6.1 Creating and Comparing K3 Surfaces

The basic RR data from which a K3 surface can be created comprises an integer, the *genus*, $g \geq -1$ and a basket of (Gorenstein surface) point singularities B. The basket B can be created explicitly as a basket using the functions of Section 117.10.2, but a convenient shortcut is provided whereby the basket argument may be given 'in raw basket format', that is, as a sequence B of length two sequences, each of the form $[r, a]$, denoting the singularity $\frac{1}{r}(a, r-a)$.

`K3Surface(g,B)`

A K3 surface with genus g and basket of singularities B (which may be a basket type or in raw basket format $[[r, a], ...]$).

`K3Copy(X)`

A new K3 surface that carries exactly the same data as the K3 surface X.

117.6.2 Accessing the Key Data

`Genus(X)`

The genus of the K3 surface X; that is, $p_1 - 1$ where p_1 is the coefficient of t in the Hilbert series of X.

`TwoGenus(X)`

The 2-genus of the K3 surface X; that is, $p_2 - 1$ where p_2 is the coefficient of t^2 in the Hilbert series of X.

`SingularRank(X)`

The sum $\sum(r-1)$ taken over the singularities $\frac{1}{r}(a, r-a)$ given in the basket of singularities of the K3 surface X.

`AFRNumber(X)`

The number assigned to the K3 surface X in the low codimension lists of Altinok–Fletcher–Reid.

117.6.3 Modifying K3 Surfaces

Sometimes it is desirable to add or remove weights from a given K3 surface. There are two intrinsics that allow this to be done (and check that a weight really can be removed). These intrinsics are used systematically in the construction of the K3 database.

`IncludeWeight(X,w)`

Return a new K3 surface that is the same as X but with the positive integer w included among the weights and all other data associated to the embedding adjusted as required.

`RemoveWeight(X,w)`

Return a new K3 surface that is the same as X but with the positive integer w removed from the weights, assuming it appears there and can be removed without destroying the property of the Hilbert numerator being a polynomial. All other data associated to the embedding is adjusted as required.

117.7 The K3 Database

The K3 database in MAGMA is a collection of 24,099 K3 surfaces. Recall from Section 117.6 the meaning of K3 surface in this context, and from Section 117.1.1 the relationship between the Hilbert series, the weights and the (Hilbert) numerator.

We describe the set of K3 surfaces selected for inclusion in the database. For $g = -1, 0, 1, 2$, all K3 surfaces of genus g are included, there being 4281, 6479, 6627 and 6628 surfaces, respectively. For higher genus, the data associated to the 6628 K3 surfaces of genus 2 propagates in a predictable way, so only those K3 surfaces with codimension at most 7 and genus in the range 3 to 9 have been included.

Data is held in blocks of surfaces indexed by the first five coefficients of their Hilbert series (excluding the constant term). Note that the t-coefficient of the Hilbert series is one more that the genus, and this defect holds for all genera. To determine the number of surfaces of genus 1, the intrinsics described below may be used. Note the genus argument is a sequence beginning with the integer 2: the sequence is arranged so that the user can ask a more precise question by including other leading genera (up to the first five), and the value 2 is to account for the genus–Hilbert coefficient defect.

```
> D := K3Database();
> NumberOfK3Surfaces(D,[2]);
6627
```

The database is fairly large, so naive searches take time. Specialised tools, described below, support much more efficient searches and should be used wherever possible. We demonstrate this point with timings for a typical search. The first searches the entire database for all K3 surfaces of genus 3 and takes over 2 minutes. It is much more efficient to use a function that looks up curves according to their genus, since this is the primary indexing property used by the database. The second search does this, and takes only a fraction of the time.

```
> time [ X : X in D | Genus(X) eq 3 ];
Time: 139.510
> time [K3SurfaceK(D,[4],i) :i in [1..NumberOfK3Surfaces(D,[4])]];
Time: 0.500
```

117.7.1 Searching the K3 Database

In this section a simple example is presented of extracting a K3 surface with particular properties from the K3 database. Section 117.10.2 provides much greater details and more examples: note, in particular, that only a few hundred of the surfaces that occur in small codimension have been confirmed to exist (even though the vast majority are believed to exist).

Example H117E5

We begin by defining D to be the K3 database.

```
> D := K3Database();
```

```
> D;
The database of K3 surfaces
```

It contains data associated to 24099 (families of) K3 surfaces.

```
> #D;
24099
```

There are several ways to access the K3 surfaces in the database. In the first place, the database is organised into blocks of K3 surfaces that have a common genus. These blocks are then subdivided into K3 surfaces that have a common 2-genus. The blocks having a common 2-genus are further subdivided right down to 5-genus, that is, the coefficient of t^5 in the Hilbert series. These subdivisions are the natural indexing units of the database. One gets the third surface with genus 0 by

```
> X := K3Surface(D,0,3);
> X;
K3 surface no.3, genus 0, in codimension 1 with data
  Weights: [ 1, 6, 8, 9 ]
  Basket: 1/2(1,1), 1/3(1,2), 1/9(1,8)
  Degree: 1/18          Singular rank: 11
  Numerator: -t^24 + 1
  Projection to codim 1 K3 no.2 -- type I from 1/9(1,8)
  Unproj'n from codim 2 K3 no.4 -- type I from 1/10(1,9)
  Unproj'n from codim 2 K3 no.15 -- type IV from 1/5(2,3)
  Unproj'n from codim 3 K3 no.28 -- type II_1 from 1/4(1,3)
  Unproj'n from codim 4 K3 no.84 -- type II_2 from 1/3(1,2)
  Unproj'n from codim 6 K3 no.280 -- type II_5 from 1/2(1,1)
```

This printout displays a lot of information about this surface and its relationship to other surfaces. The minimal printing option may be use to obtain a concise description of this surface alone.

```
> X:Minimal;
K3 surface (g=0, no.3) in P^3(1,6,8,9)
  Basket: 1/2(1,1), 1/3(1,2), 1/9(1,8)
  Numerator: -t^24 + 1
```

When using several genera to access a surface, the genus arguments must be collected together in a sequence. For example, there are 282 K3 surfaces whose first three genera are $p_1 = 0$, $p_2 = 1$, $p_3 = 3$; that is, have weights that are of the form $[2, 3, 3, \ldots]$.

```
> NumberOfK3Surfaces(D,[0,1,3]);
282
```

We get the first of these as follows. The arguments inside the sequence brackets are coefficients of the Hilbert polynomial, while the corresponding genus is one less than the coefficient. N.B. Note the offset by -1 between these arguments and the genera.

```
> K3Surface(D,[0,1,3],1);
K3 surface no.1130, genus -1, in codimension 4 with data
  Weights: [ 2, 3, 3, 3, 4, 4, 5 ]
  Basket: 2 x 1/2(1,1), 5 x 1/3(1,2)
```

```
  Degree: 1/3           Singular rank: 12
  Numerator: t^24 - ... + t^10 - t^9 - 2*t^8 - t^7 - t^6 + 1
  Projection to codim 1 K3 no.820 -- type II_2 from 1/3(1,2)
  Unproj'n from codim 5 K3 no.1131 -- type I from 1/5(2,3)
  Unproj'n from codim 6 K3 no.1145 -- type II_1 from 1/4(1,3)
  Unproj'n from codim 7 K3 no.1412 -- type II_2 from 1/3(1,2)
  Unproj'n from codim 8 K3 no.2176 -- type IV from 1/2(1,1)
```

The genus and number of a K3 surface identifies it uniquely in the database, so the same function may be used to see surface number 1131 which has projection to X.

```
> K3Surface(D,-1,1131) : Minimal;
K3 surface (g=-1, no.1131) in P^7(2,3,3,3,4,4,5,5)
  Basket: 1/2(1,1), 4 x 1/3(1,2), 1/5(2,3)
  Numerator: -t^29 + ... + 6*t^11 - 3*t^9 - 4*t^8 - t^7 - t^6 + 1
```

The projection is from the $\frac{1}{5}(2,3)$ singularity, resulting in the extra $\frac{1}{2}(1,1)$ and $\frac{1}{3}(1,2)$ points. There are also searches that do not use the primary indexing directly. For example, the following variation of `K3Surface` searches for a K3 surface with weights $2,2,3,5,7,9,11$.

```
> K3Surface(D,[2,2,3,5,7,9,11]) : Minimal;
K3 surface (g=-1, no.1615) in P^6(2,2,3,5,7,9,11)
  Basket: 3 x 1/2(1,1), 1/11(2,9)
  Numerator: t^39 - ... + t^16 - t^13 - t^11 - t^9 + 1
```

`K3Database()`

The database of K3 surfaces.

`Number(D,X)`

The integer n such that the K3 surface `Y := K3Surface(D,Genus(X)+1,n)` in the database D has the same Hilbert series as the K3 surface X. The second return value is the K3 surface Y. If there is no such K3 surface, the returned index value is zero.

`Index(D,X)`

The integer i such that the K3 surface `Y := K3Surface(D,i)` in the database D has the same Hilbert series as the K3 surface X. The second return value is the K3 surface Y. If there is no such K3 surface, the returned index value is zero.

Example H117E6

The 'Number' of a K3 surface in the database and its 'Index' may differ: the K3 surfaces of any fixed genus are numbered separately, while the index runs over the whole database.
To illustrate this, consider the following K3 surface.

```
> X := K3Surface(1,[[2,1],[3,1],[4,1],[7,1],[8,1]]);
> X;
K3 surface in codimension 11 with data
  Weights: [ 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 6, 7, 8 ]
  Basket: 1/2(1,1), 1/3(1,2), 1/4(1,3), 1/7(1,6), 1/8(1,7)
  Degree: 613/168               Singular rank: 19
  Numerator: -t^47 + ... + 38*t^7 - 2*t^6 - 13*t^5 - 7*t^4 + 1
```

This surface has been calculated in isolation. The weights that have been assigned to it are just enough to make sense of the initial terms of the Hilbert series, and to make the singularities. By construction, the K3 database may have added weights to make simple projections work. So we search for X in the database using either 'Index' or 'Number'.

```
> D := K3Database();
> n,Y := Number(D,X);
> i,Y1 := Index(D,X);
> n,i;
1474 12234
```

This result means that X has the same Hilbert series as the 1474-th K3 surface in D of genus 1, which is the same as the 12234-th K3 surface in D.

```
> Y eq Y1;
true
> Y1 eq K3Surface(D,i);
true
```

One can see that Y is in much higher codimension than X, so that extra weights have been assigned to Y.

```
> Codimension(Y);
17
```

117.7.2 Working with the K3 Database

`K3Surface(D,i)`

The ith K3 surface in the K3 database D.

`K3Surface(D,Q,i)`

The ith K3 surface in the K3 database D among those with index suite $Q = [g1 + 1, g2 + 1, ...]$.

`K3Surface(D,g,i)`

The ith K3 surface in the K3 database D among those with genus $g \geq -1$.

`K3Surface(D,g1,g2,i)`

The ith K3 surface in the K3 database D among those with genus $g1 \geq -1$ and 2-genus $g2 \geq -1$.

`K3Surface(D,W)`

The K3 surface in the K3 database D having the weights specified in the sequence W.

`K3Surface(D,g,B)`

The K3 surface in the K3 database D with genus $g \geq -1$ and basket of singularities B (as a basket type or in raw sequence format).

117.8 Fano 3-folds

This section describes intrinsics that construct Fano 3-folds. It also describes a few intrinsics that can be used to study them, but see Section 117.4 for the general intrinsics that apply to all polarised varieties.

The calculations are based on Suzuki's Riemann–Roch formula [Suz] for polarised Fano 3-folds with terminal singularities.

Example H117E7

We make two Fano 3-folds having the same basket but different genus.

```
> X := Fano(2,MakeBasket([[3,1,2,2]]),2);
> X;
Fano 3-fold X,A of Fano index 2, Fano genus 2, in codimension 1 with data
  Weights: [ 1, 1, 2, 3, 5 ]
  Basket: 1/3(1,2,2)
  Degrees: A^3 = 1/3,   (1/12)Ac_2(X) = 8/9
  Numerator: -t^10 + 1
> FanoGenus(X);
2
> FanoBaseGenus(X);
2
> Fano(2,MakeBasket([[3,1,2,2]]),3);
Fano 3-fold X,A of Fano index 2, Fano genus 3, in codimension 2 with data
  Weights: [ 1, 1, 1, 2, 2, 3 ]
  Basket: 1/3(1,2,2)
  Degrees: A^3 = 4/3,   (1/12)Ac_2(X) = 8/9
  Numerator: t^8 - 2*t^4 + 1
```

In this example, the smallest possible genus—the Fano base genus—is 2, and an error will be reported if a smaller value is requested.

Note that the singularities used must be polarised by fA, where f is the Fano index: in practice, this means their index r must be coprime to f and their weights must be of the form $f, a, r-a$.

117.8.1 Creation: $f = 1, 2$ or ≥ 3

`Fano(f,B,g)`

A Fano 3-fold with Fano index $f \geq 1$, Fano genus $g \geq 0$ and basket of singularities B. The singularities must be terminal singularities. The basket can also be presented in raw sequence format: in this case, B is a sequence containing terms such as $[r, a, b, c]$ which denotes the singularity $\frac{1}{r}(a, b, c)$.

`Fano(f,B)`

A Fano 3-fold with Fano index $f \geq 3$ and basket of singularities B. The singularities must be terminal singularities. The basket can also be presented in raw sequence format: in this case, B is a sequence containing terms such as $[r, a, b, c]$ which denotes the singularity $\frac{1}{r}(a, b, c)$.

`FanoIndex(X)`

The Fano index f of the Fano 3-fold X.

`FanoGenus(X)`

The Fano genus of the Fano 3-fold X, an integer ≥ 0 equal to the dimension of the space of sections of the polarising divisor. (The term *genus* often refers to two less that this number).

`FanoBaseGenus(X)`

The smallest possible value for the Fano genus of the Fano 3-fold X.

`BogomolovNumber(X)`

The intersection number $A(c_1(X)^2 - 3c_2(X))$ for the polarised Fano 3-fold X, A.

`IsBogomolovUnstable(X)`

Return `true` if and only if the Bogomolov number $A(c_1(X)^2 - 3c_2(X))$ for the polarised Fano 3-fold X, A is strictly positive.

117.8.2 A Preliminary Fano Database

`FanoDatabase()`

The database of Fano 3-folds.

`Fano(D,i)`

The ith Fano 3-fold in the Fano database D.

`Fano(D,f,i)`

The ith Fano 3-fold in the Fano database D that has Fano index f.

`Fano(D,f,Q,i)`

The ith Fano 3-fold in the Fano database D that has Fano index f and initial plurigenera as specified by the sequence Q (up to the first four plurigenera).

117.9 Calabi–Yau 3-folds

This section describes intrinsics that construct Calabi–Yau 3-folds. It also describes a few intrinsics that can be used to study them, but see Section 117.4 for the general intrinsics that apply to all polarised varieties.

The calculations are based on Buckley's Riemann–Roch formula [Buc03] for polarised Calabi–Yau 3-folds with canonical singularities.

`CalabiYau(p1,p2,B)`

The Calabi–Yau 3-fold X, A with $h^0(X, A) = p_1$, $h^0(X, 2A) = p_2$ and basket of singularities B.

`FindN(X)`

`MaximumN` RngIntElt *Default :* 100

The first nonnegative value, for the Calabi–Yau 3-fold X, of N_{C_i} (where C_i is the ith curve of the basket of X) together with the distance between successive values of N_{C_i}. (The pair $0, 0$ is returned if no solutions below the parameter `MaximumN` are found).

`FindN(p1,p2,B)`

`MaximumN` RngIntElt *Default :* 100

The first nonnegative value of N_{C_i} (where C_i is the ith curve of the basket B of 3-fold points and curves and p_1, p_2 are the first two genera) together with the distance between successive values of N_{C_i}. (The pair $0, 0$ is returned if no solutions below the parameter `MaximumN` are found).

117.10 Building Databases

This section contains a sketch of how databases of examples can be built. This is an aside from the rest of the chapter, and is only of interest if one either wants to understand the construction of the K3 database for its own sake, or wishes to create other similar lists that take too long to be regenerated on demand. Note, however, that the database facility described here is not suitable for very large databases: roughly speaking, it is intended for those containing a few tens of thousands of elements, not millions, and certainly not in the Kreuzer–Skarke [KS00] range.

117.10.1 The K3 Database

This section describes briefly the method of construction used for the K3 database. The K3 database is fully installed in MAGMA—see Section 117.7 for instructions—and there is no need to use the functions described here to rebuild it unless you intend to modify the code to incorporate new information in the database.

The construction is in two steps. First we create all the K3 surfaces we require and write them as abbreviated records to a series of files. Then we load these files one at a time, writing their contents in further abbreviated form to a binary data file, `K3S.dat`. The writing functions keep track of key indexing information and write this to an index file, `K3S.ind`. Functions to read from these files are already installed, so the process is completed by copying these two files to the standard data directory in the MAGMA libraries.

We now go through the process in detail.

117.10.1.1 Creating Many K3 Surfaces

`CreateK3Data(g)`

`CreateK3Data(g,r)`

`CreateK3Data(g,B)`

Create a sequence containing all K3 surfaces of genus $g \geq -1$; restrict to those with singular rank at most r, if given as an argument, or having baskets in the sequence B of baskets, if given as an argument.

The return sequence is ordered according to the natural numerical order on the coefficients of Hilbert series. Analysis of projections of Types I and II is made to modify weights of K3 surfaces, and any inconsistencies between different predictions of weights coming from different projections are reported.

117.10.1.2 K3 Surfaces as Records

`K3SurfaceToRecord(X)`

A record in K3 record format that contains a subset of the data associated to the K3 surface X from which all attributes of X can be computed.

`K3Surface(x)`

The K3 surface with the same data as that of the record x in K3 record format.

117.10.1.3 Writing K3 Surfaces to a File

`WriteK3Data(Q,F)`

Write the K3 surfaces in the sequence Q to the file with name given by the string F. The resulting file F is a text file containing MAGMA code which, when loaded into a MAGMA session, generates a sequence called `data` of records in K3 record format that corresponds to Q.

117.10.1.4 Writing the Data and Index Files

With the data for the K3 database saved as a series of files "k3data-1", "k3data0" and so on, the command `load "writek3db.m";` (or `load "PATH/writek3db.m";`) loads them in turn, writes the two binary files and then deletes the data from the MAGMA session. MAGMA must be running in the directory containing all the data files "k3data...".

117.10.1.5 Reading the Raw Data

In normal MAGMA use, even though the data in the K3 database is in coded form (as a tuple, in fact), when returned to the user it is expressed as a K3 surface. Of course, this final translation step takes a little time, insignificant in most use, but very significant when searching through the whole database. So there are functions to access the raw data, and then to translate it into a K3 surface. The search intrinsics installed in the MAGMA packages use these functions, as should any new searches that need only modest increase in speed.

`K3SurfaceRaw(D,i)`

`K3SurfaceRaw(D,Q,i)`

The i-th element of the K3 database D expressed as a tuple of data (or the i-th element whose Hilbert series has coefficients of t^i given by the integers in the sequence Q, which may have length at most 5).

`K3Surface(x)`

The K3 surface with the same data as that of the tuple x that is in the raw K3 database format.

117.10.2 Making New Databases

Here we explain in general terms how to write new databases of polarised varieties. We give complete instructions for writing the data and index files, but are more sketchy on the other steps: you will have to assemble the data, write translation functions and write any cosmetic wrapping functions for yourself.

Step 1: Prepare the directory. Make a new directory, 'NewDB' say, and copy the files `data_spec.m`, `write.m`, `init_info.m`, `write_tools.m`, `write_func.m`, `create_ind_func.m` into NewDB from the MAGMA package directories. Of these files, the first two will have to be edited to conform to the required database, while the remaining four are common to all databases.

Step 2: Decide record format. Edit the file `data_spec.m`. This determines which data is stored, and places unbreakable restrictions on that data. In particular, the choice of up to 5 indexing parameters is made here. Write intrinsics that translate between the data in its original representation and in its record representation.

Step 3: Update the writing functions. Edit the file `write.m`.

Step 4: Collect the data. Write all required data as records in one or more files. It is essential that the data is written to these files so that it can be read into a single MAGMA session in increasing order (with respect to the chosen indexes).

Step 5: Build the binary data and index files.

Step 6: Move the binaries to the data libraries.

Step 7: Final cosmetics. Write as MAGMA package code any cosmetic wrappers or search routines that are to be used with the new database. Document these intrinsics.

117.11 Bibliography

[**ABR02**] Selma Altinok, Gavin Brown, and Miles Reid. Fano 3-folds, $K3$ surfaces and graded rings. In *Topology and geometry: commemorating SISTAG*, volume 314 of *Contemp. Math.*, pages 25–53. Amer. Math. Soc., Providence, RI, 2002.

[**Alt98**] Selma Altinok. *Graded Rings Corresponding to Polarised K3 Surfaces and Q-Fano 3-folds*. PhD thesis, University of Warwick, 1998. URL:http://www.maths.warwick.ac.uk/~miles/students/Selma/.

[**Bro03**] Gavin Brown. Datagraphs in algebraic geometry. In *Symbolic and Numerical Scientific Computing – Proc. of SNSC'01*, volume 2630 of *LNCS*, Heidelberg, 2003. Springer-Verlag. F. Winkler and U. Langer (eds.).

[**Buc03**] Anita Buckley. *Orbifold Riemann-Roch for 3-folds and applications to Calabi-Yaus*. Phd thesis, Warwick University, 2003.

[**IF00**] Anthony Iano-Fletcher. Working with weighted complete intersections. In A. Corti and M. Reid, editors, *Explicit birational geometry of 3-folds*, pages 101 – 173. CUP, Cambridge, 2000.

[**KS00**] Maximilian Kreuzer and Harald Skarke. Complete classification of reflexive polyhedra in four dimensions. *Adv. Theor. Math. Phys.*, 4(6):1209–1230, 2000.

[**Mat89**] H. Matsumura. *Commutative ring theory*. CUP, ACUP, 1989.

[**Pap03**] S. Papadakis. Kustin–Miller unprojection with complexes. *J. Algebraic Geometry*, to appear, 2003.

[**Rei00**] Miles Reid. Graded rings and birational geometry. In K. Ohno, editor, *Proc. of algebraic geometry symposium (Kinosaki)*, pages 1 – 72. 2000. URL:http://www.maths.warwick.ac.uk/~miles/3folds/.

[**SB**] B. Szendrői and A. Buckley. Orbifold Riemann-Roch for threefolds with an application to Calabi-Yau geometry. math.AG/0309033.

[**Suz**] K. Suzuki. On Fano indices of Q-Fano 3-folds. Avaialable as math.AG/0210309 on arXiv.

118 TORIC VARIETIES

Chapter 118
TORIC VARIETIES

118.1 Introduction and First Examples

We describe a package to work quite generally with toric geometry within the scope of MAGMA's scheme machinery. From that point of view, we regard toric varieties as a vast array of possible ambient spaces for schemes, generalising the affine and projective spaces already available. But of course toric geometry is a large topic in its own right, and in the first place we present it for its own sake.

There are many different points of view on toric geometry. We model our approach on several sources: the primary ones are Danilov [Dan78] and Cox [Cox95], but also Fulton [Ful93] and Oda [Oda88].

One of the primary points of view, exemplified by Danilov [Dan78], is to regard the two-way relationship between a part of scheme theory and the combinatorics of polyhedra as foundational. This package can operate from that point of view.

Another, more recent, point of view is that of the Cox ring, introduced by Cox [Cox95], a (multi-)graded polynomial ring that works as a homogeneous coordinate ring for a toric variety (or indeed any variety with finitely-generated Picard group) in much the same way as the coordinate ring of affine or projective space. This package can also operate from this point of view, and indeed this is the central object in its design. Again, this is bound to impose practical restrictions on the dimensions in which calculations can reasonably be expected to work: we use these 'Cox coordinates' to define subschemes of toric varieties, which often leads to Gröbner basis calculations, and these are famously intolerant of the number of variables involved.

118.1.1 The Projective Plane as a Toric Variety

The projective plane is the toric variety corresponding to the fan with three maximal cones lying in the Euclidean plane:

$$\langle (0,1),(1,0)\rangle\,, \quad \langle (1,0),(-1,-1)\rangle\,, \quad \langle (-1,-1),(0,1)\rangle\,.$$

Example H118E1

We build the projective plane (defined over the rational field) as a toric variety. There are various simple constructors for this, but we do it slowly by constructing its fan first: we list the one-dimensional rays of the fan, and then describe each maximal cone by listing the sequence of indices of the rays that generate it.

```
> rays := [ [0,1], [1,0], [-1,-1] ];
> cones := [ [1,2], [1,3], [2,3] ];
> F := Fan(rays,cones);
> F;
```

```
Fan F with 3 rays:
    ( 0,  1),
    ( 1,  0),
    (-1, -1)
and 3 cones with indices:
    [ 1, 2 ],
    [ 1, 3 ],
    [ 2, 3 ]
```

This is enough to determine a toric variety; we name its natural (Cox) coordinates as x, y, z during the definition. (Since this is a scheme, it needs to have a base ring assigned.)

```
> X<x,y,z> := ToricVariety(Rationals(),F);
> X;
Toric variety of dimension 2
Variables: x, y, z
The irrelevant ideal is:
    (z, y, x)
The grading is:
    1, 1, 1
```

As with all schemes in MAGMA, points can be created by coercing sequences of coefficients (either from the given base field or from an extension of that) into the scheme.

```
> X ! [1,2,3];
(1 : 2 : 3)
```

When working over an extension k of the base field, one must coerce into the point set (of k-valued points of X).

```
> k<i> := QuadraticField(-1);
> X(k);
Set of points of X with coordinates in k
> X(k) ! [1,i,2*i];
(1 : i : 2*i)
```

The irrelevant ideal (described in the display of X above) describes the locus of coordinates that do not represent points; in this case, the irrelevant ideal records the fact that $(0 : 0 : 0)$ is not a point of projective space.

```
>  X ! [0,0,0];

>> X ! [0,0,0];
     ^
Runtime error in '!': Illegal coercion
> Y<u,v,w> := ProjectiveSpace(Rationals(),2);
> Y;
Projective Space of dimension 2
Variables: u, v, w
> IrrelevantIdeal(Y);
Ideal of Polynomial ring of rank 3 over Rational Field
```

```
Order: Graded Reverse Lexicographical
Variables: u, v, w
Homogeneous
Basis:
[
    u,
    v,
    w
]
> Gradings(Y);
[
    [ 1, 1, 1 ]
]
> Fan(Y);
Fan with 3 rays:
    ( 1,  0),
    ( 0,  1),
    (-1, -1)
and 3 cones with indices:
    [ 1, 2 ],
    [ 1, 3 ],
    [ 2, 3 ]
```

118.1.2 Resolution of a Nonprojective Toric Variety

Danilov translated the projectivity of a toric variety into properties of the fan. In geometry, projectivity is equivalent to the existence of an ample line bundle. In the combinatorics of the fan, the requirement is the existence of positive linear functions on each cone that agree on the boundaries and are strictly convex across boundaries. This is one small part of the theory of divisors on toric varieties that we cover in much greater detail in Section 118.6.

Example H118E2

We construct an example of a complete but non-projective toric variety following Danilov's original example of how a fan can fail to admit a strictly convex piece-wise linear support function. We will build the fan by hand, first specifying exactly the (one-dimensional) rays that it contains.

```
> rays := [ [0,0,1], [4,0,1], [0,4,1], [1,1,1], [2,1,1], [1,2,1], [-1,-1,-1] ];
```

Now we specify the top-dimensional cones that the fan contains. This is done by naming the rays that generate each cone from the list of rays above. So, for example, the sequence `[1,3,6]` below refers to the cone generated by the 1st, 3rd and 6th rays, that is by $[0, 0, 1]$, $[0, 4, 1]$ and $[1, 2, 1]$.

```
> cones := [ [1,3,6], [1,4,6], [1,2,4], [2,4,5], [2,3,5], [3,5,6], [4,5,6],
```

```
>             [1,3,7], [1,2,7], [2,3,7] ];
```

This is enough information to determine a fan. The command below to construct the fan F with this data takes a little time, since it checks that the maximal cones we nominated are indeed maximal and intersect correctly to lie in a fan.

```
> F := Fan(rays,cones);
```

Finally we move away from the lattice world to the geometry by creating the toric variety corresponding to the fan F.

```
> X := ToricVariety(Rationals(),F);
```

We can now ask questions of X as we would with any other variety or ambient space.

```
> Dimension(X);
3
> IsComplete(X);
true
> IsProjective(X);
false
```

Unfortunately our specimen is flawed: it has singularities.

```
> IsNonsingular(X);
false
> Y := Resolution(X);
> IsProjective(Y);
false
```

The last line takes a little time: MAGMA computes the ample cone of Y and then determines whether or not it is empty. In this case, Y remains non-projective, although there is no reason to expect that: Danilov's original example can be made projective by blowing up a line (or simply flopping a line).

118.1.3 The Cox Ring of a Toric Variety

The Cox ring of a toric variety is its natural homogeneous coordinate ring. It is a multigraded ring (with other data besides). It is often easier to construct the gradings for a Cox ring than it is to describe a fan.

A Cox ring requires four pieces of data:

- a polynomial ring R (a ring of coordinates for an affine space)
- a sequence B of 'irrelevant' ideals (the loci defined by these are discarded from the affine space)
- a sequence Z of sequences of integral weights for R; each element has length the number of indeterminates of R
- a sequence Q of sequences of quotient gradings for R.

Example H118E3

Cox's original construction determines a Cox ring from the fan of a toric variety, and then realises the toric variety as the quotient of the affine space (with the irrelevant locus discarded) by the action of a torus (determined by the sequences of weights). But it is also possible simply to specify this data independently of a fan.

```
> R<u,v,x,y,z> := PolynomialRing(Rationals(),5);
> irrel1 := ideal< R | u,v >;
> irrel2 := ideal< R | x,y,z >;
> B := [ irrel1, irrel2 ];
> Zwts := [
>             [ 1, 1, 0, -1, -3 ],
>             [ 0, 0, 1,  2,  3 ] ];
> Qwts := [];
> C := CoxRing(R,B,Zwts,Qwts);
> C;
Cox ring C with underlying Polynomial ring of rank 5 over Rational Field
Order: Lexicographical
Variables: u, v, x, y, z
The components of the irrelevant ideal are:
    (z, y, x), (v, u)
The 2 gradings are:
    1,  1,  0, -1, -3,
    0,  0,  1,  2,  3
```

The pieces of data can be retrieved by `Gradings(C)`, `QuotientGradings(C)` and so on.
In favourable cases, a Cox ring does indeed arise from a fan using Cox's construction. This fan can be recovered.

```
> F := Fan(C);
> F;
Fan F with 5 rays:
    ( 1,  0,  0),
    ( 0,  1,  0),
    ( 1,  1,  3),
    (-2, -2, -3),
    ( 1,  1,  1)
and 6 cones with indices:
    [ 1, 3, 4 ],
    [ 1, 3, 5 ],
    [ 1, 4, 5 ],
    [ 2, 3, 4 ],
    [ 2, 3, 5 ],
    [ 2, 4, 5 ]
```

Alternatively, one can construct a toric variety from a Cox ring—as in Cox's construction, the Cox ring contains exactly the data required to construct a variety as a torus quotient.

```
> X := ToricVariety(C);
> Dimension(X);
```

```
3
```

Sequences of coefficients not lying in the locus of the irrelevant ideal can be interpreted as closed points of X as usual.

```
> X ! [1,0,1,0,0];
(1 : 0 : 1 : 0 : 0)
>  X ! [1,0,0,0,0];

>> X ! [1,0,0,0,0];
     ^
Runtime error in '!': Illegal coercion
```

The attempted coercion of the second point fails because its x, y and z coordinates (the 3rd, 4th and 5th coefficients of the vector) are all zero, but the locus $x = y = z = 0$ is defined by a component of the irrelevant ideal and so has been discarded: points of X do not have all three of those coordinates simultaneously equal to zero.
The homogeneous coordinates that the Cox ring provides for X behave in a very similar way to the homogeneous coordinates on projective space. In particular, one can define subschemes of X by the vanishing of polynomials in the Cox ring that are homogeneous with respect to all the $\mathbf{Z}$-gradings (and the quotient gradings too).

```
> f := x^4*y + u^2*y^3 + v^5*z^2;
> Multidegree(X,f);
[ -1, 6 ]
[]
```

The multidegree is returned as two sequences: the first is the sequence of degrees of f with respect to each of the $\mathbf{Z}$-gradings in turn, and the second is that with respect to the quotient gradings (in this case there are none). One defines a scheme as usual in Magma.

```
> V := Scheme(X, f);
> V;
Scheme over Rational Field defined by
u^2*y^3 + v^5*z^2 + x^4*y
> Dimension(V);
2
```

Much of Magma's scheme machinery works for schemes inside toric varieties, although at this stage some does not. (Some functions are missing because of the absence of standard nonsingular affine patches on toric varieties. In some cases these will be replaced by their orbifold analogues in due course.)

118.2 Fans in Toric Lattices

A *fan* F (in a toric lattice L) is a collection of cones in L satisfying typical conditions of cell decompositions (the inclusion of faces of cones is part of the data, and the requirement that any two cones of the fan intersect in a common face is enforced, for instance). The *support* of F is the union of all the cones as a subset of L. The main two cases are when either the support of F is the support of a single cone (which is regarded as being subdivided by the cones of the fan) or it is the whole of L. Other cases do occur and are allowed by our package—even having cones of different dimensions lying in complementary linear subspaces of L, although that is not commonly interpreted geometrically.

There are several constructors for well-known fans, and also standard methods for modifying fans. If these are not enough, then one can simply list the top-dimensional cones of a fan; MAGMA will check that they intersect correctly and will add the lower-dimensional cones as necessary.

Fans are of type `TorFan`.

118.2.1 Construction of Fans

We first list the comprehensive constructors for fans. After that, we have a collection of constructors for well-known fans, and there also are methods for modifying fans.

`Fan(Q)`

`define_fan`	BOOLELT	*Default :* `false`
`max_cones`	BOOLELT	*Default :* `false`

The fan generated by the cones in the sequence Q.

The optional parameter `define_fan` (by default `false`) can be set to `true`, so that the verification whether the input data is correct is skipped. The optional parameter `max_cones` (by default `false`) can be set to `true` to skip the verification whether the cones are maximal. Both will lead to errors later if the cones do not in fact determine a fan as claimed.

`Fan(R,S)`

`define_fan`	BOOLELT	*Default :* `false`

The fan whose rays are the sequence of toric lattice points R (or sequences of sequences of integer coefficients of such points) and whose maximal cones correspond to the sequence S of sequences of integers: each such sequence is interpreted as the indices of a collection of rays of R, and these rays are used to generate a cone.

This constructor checks that the given data does indeed define a fan. This check can be lengthy. It can be omitted by setting the parameter `define_fan` to be `false`, although this should be used with extreme care: there is no telling what might go wrong if incorrect data is assumed to be a fan.

`Fan(C)`

The fan comprising all faces of the cone C.

`FanOfAffineSpace(n)`

The standard fan of affine space of dimension n, where n is a positive integer.

`FanOfWPS(W)`

A standard fan for the weighted projective space with weights W, a sequence of positive integers.

`#FanOfProjectiveSpace(n)`

The standard fan of projective space of dimension n, a positive integer.

`FanOfFakeProjectiveSpace(W,Q)`

A standard fan for the fake weighted projective space with weights W, a sequence of positive integers, and finite cyclic group actions given Q, a sequence of sequences of rational numbers. The finite cyclic actions are determined as follows: for $\mathbf{Z}/r$ to act diagonally with weights $(a_1, \ldots, a_n)$, include the sequence `[a1/r,...,an/r]` as an element of Q.

`ZeroFan(L)`

The fan in the toric lattice L supported at the origin.

`NormalFan(F,C)`

The normal fan to a cone C in a fan F in the toric lattice that is the quotient of the ambient lattice of F by the span of C; the quotient map of lattices is the second return value.

`SpanningFan(P)`

The toric fan that spans the polyhedron P; in particular, the rays of the fan are generated by the vertices of P.

`DualFan(P)`

The toric fan dual to the polyhedron P.

Example H118E4

The spanning fan of a polytope P is a fan in the same lattice as P with rays passing through its vertices. Here we make the fan of the Hirzebruch surface $\mathbf{F}_1$ as a spanning fan.

```
> P := Polytope([ [0,1], [1,0], [1,-1], [-1,0] ]);
> SpanningFan(P);
Fan with 4 rays:
    ( 0,  1),
    ( 1,  0),
    ( 1, -1),
    (-1,  0)
and 4 cones with indices:
    [ 1, 4 ],
```

```
    [ 1, 2 ],
    [ 2, 3 ],
    [ 3, 4 ]
```

The dual fan of a polytope P is a fan in the dual lattice to that of P whose rays are constant when evaluated on some facet of P. Here we make the fan of projective 3-space as a dual fan.

```
> Q := StandardSimplex(3);
> DualFan(Q);
Fan with 4 rays:
    (-1, -1, -1),
    ( 1,  0,  0),
    ( 0,  1,  0),
    ( 0,  0,  1)
and 4 cones with indices:
    [ 2, 3, 4 ],
    [ 1, 3, 4 ],
    [ 1, 2, 4 ],
    [ 1, 2, 3 ]
Mapping from: 3-dimensional toric lattice (Z^3)^* to 3-dimensional toric lattice
(Z^3)^* given by a rule
```

`Blowup(F,v)`

The blowup of the toric fan F at the point v that is an element of the ambient toric lattice of F.

`IsInSupport(v,F)`

Return `true` if and only if the support of the fan F contains the element v of its ambient toric lattice. In this case, the index of the first cone of F that contains v is also returned.

`Fan(F1,F2)`

`F1 * F2`

`Fan(Q)`

`F ^ n`

The nth Cartesian product of the fan F.

`F eq G`

Return `true` if and only if the two fans F and G are equal as objects in MAGMA (not simply isomorphic as fans).

118.2.2 Components of Fans

One can retrieve all the usual components of fans: their cones, rays and other dimensional skeletons, and so on.

Note the novelty of 'virtual' rays. Virtual rays occur when the rays of the fan in question do not span the ambient lattice, and so one may easily never encounter them. They occur if and only if the underlying toric variety is a product of $\mathbf{C}^*$ times X for a smaller toric variety X. If this the case, the fan is contained in a hyperplane (or smaller linear space, if there are more factors of $\mathbf{C}^*$). Virtual rays will be in the direction transversal to this hyperplane. Thus a virtual ray is not an honest ray, but a formal object introduced to allow $\mathbf{C}^*$ as a toric variety and give meaning to its coordinate. If virtual rays are not specified on creation, then MAGMA will decide where to put them (later, when it needs them).

`Skeleton(F,n)`

The fan generated by cones of the fan F of dimension at most the integer n.

`C in F`

Return `true` if and only if the cone C is one of the cones of the fan F.

`Cones(F)`

A sequence of the maximal cones of the fan F.

`Cones(F,i)`

The sequence of all i-dimensional cones in the fan F.

`ConesOfCodimension(F,i)`

The sequence of all codimension i cones in the fan F.

`AllCones(F)`

A sequence of all the cones of the fan F.

`Cone(F,i)`

The ith cone of the sequence of maximal cones of the fan F.

`Cone(F,S)`

The cone of the fan F spanned by the rays of indices given by the sequence S of integers. (If the optional parameter `extend` is set to `true`, then the smallest cone containing the given rays will be returned.)

`SingularCones(F)`

A sequence containing the minimal singular cones of the fan F. A second (parallel) sequence contains sets of the indices of the rays that generate these cones.

Example H118E5

The weighted projective space $\mathbf{P}(1,2,2,3)$ has a singular line (with stabiliser $\mathbf{Z}/2$) and a singular point (with stabiliser $\mathbf{Z}/3$).

```
> F := FanOfWPS([1,2,2,3]);
> SingularCones(F);
[
    2-dimensional simplicial cone with 2 minimal generators:
        (-1, -1,  0),
        (-1,  1,  0),
    3-dimensional simplicial cone with 3 minimal generators:
        (-1, -1,  0),
        ( 1, -1, -1),
        ( 1,  0,  1)
]
[
    { 1, 4 },
    { 1, 2, 3 }
]
```

The two cones correspond to these two singular strata; the point strata lying on the line are not returned, since they can be recovered, if needed, as the cones having this 2-dimensional cone in their boundary.

`ConesOfMaximalDimension(F)`

A sequence of maximal cones of the fan F when ordered by inclusion. (This is the same as `Cones(F)` if the support of F is equidimensional.)

`ConeIndices(F)`

The sequence S of sets of integers, such that ith cone of the fan F is generated by rays with indices $S[i]$.

`ConeIndices(F,C)`

The sequence of integers that are the indices of the rays which generate the cone C of the fan F.

`ConeIntersection(F,C1,C2)`

The intersection of the two cones C_1 and C_2, both of which are members of the fan F. (This is usually more efficient than `C1 meet C2` given that the fan F exists.)

`Face(F,C)`

The smallest cone in the fan F which contains the cone C. (An error is returned if there is no such cone in the fan.)

`DualFaceInDualFan(P,Q)`

The cone in the toric fan dual to the polyhedron P which is dual to the face of P determined by the sequence of integers Q.

`Rays(F)`

A sequence containing the rays of the fan F (as a sequence of primitive lattice points on each ray).

`Ray(F,i)`

The ith ray of the fan F (regarded as the primitive ambient toric lattice point on the ray).

`AllRays(F)`

A sequence of the rays of the fan F (including all virtual rays).

`PureRays(F)`

A sequence of the (non-virtual) rays of the fan F.

`PureRayIndices(F)`

A sequence of the indices of the non-virtual rays of the fan F among all its rays.

`VirtualRays(F)`

A sequence of the virtual rays of the fan F.

`VirtualRayIndices(F)`

A sequence of the indices of the virtual rays of the fan F among all its rays.

118.2.3 Properties of Fans

`Ambient(F)`

The ambient toric lattice of the toric fan F.

`IsComplete(F)`

Return `true` if and only if the toric fan F has its entire ambient toric lattice; that is, the cones of F cover the whole ambient space.

`IsSingular(F)`

Return `false` if and only if all the cones of the fan F are nonsingular.

`IsNonsingular(F)`

Return `true` if and only if all the cones of the fan F are nonsingular.

`IsQFactorial(F)`

Return `true` if and only if all the cones of the fan F are **Q**-factorial.

`IsTerminal(F)`

Return `true` if and only if all the cones of the fan F are terminal.

`IsCanonical(F)`

Return `true` if and only if all the cones of the fan F are canonical.

`IsGorenstein(F)`

Return `true` if and only if all the cones of the fan F are Gorenstein.

`IsQGorenstein(F)`

Return `true` if and only if all the cones of the fan F are **Q**-Gorenstein.

118.2.4 Maps of Fans

`F @ f`

The image of the fan F by the map f of toric lattices.

`SimplicialSubdivision(F)`
`SimplicialSubdivision(C)`

A toric fan that is a simplicial subdivision of the toric fan F (or of the toric cone C).

Example H118E6

If C is a cone on a square, then it has two small simplicial subdivisions—the two sides of a standard flop, in fact. The simplicial subdivision intrinsic selects one of these.

```
> L := ToricLattice(3);
> C := Cone([L| [1,0,0], [0,1,0], [0,0,1],[1,-1,1]]);
> SiC := SimplicialSubdivision(C);
> #Cones(SiC);
2
> [ ZGenerators(B) : B in Cones(SiC) ];
[
    [
        (0, 1, 0),
        (1, -1, 1),
        (1, 0, 0)
    ],
    [
        (0, 1, 0),
        (1, -1, 1),
        (0, 0, 1)
```

```
    ]
]
```

`IsFanMap(F1,F2)`

Return `true` if and only if the two fans F_1 and F_2 lie in the same toric lattice, and each cone of F_1 is a subcone of some cone of F_2.

`IsFanMap(F1,F2,f)`

Return `true` if and only if the toric lattice map f between the ambient toric lattices of the two fans F_1 and F_2 maps every cone of F_1 into a cone of F_2.

`ResolveFanMap(F1,F2)`

A toric fan F that resolves the identity map of lattices restricted to the toric fans F_1 and F_2. The two fans F_i are expected to lie in the same lattice and to have the same support, and the resulting toric fan F gives a common refinement of them; in particular, F will admit a fan map into each of the F_i. (If the F_i have different supports, the fan F will be supported on their intersection and will only refine this part of each of them. Geometrically speaking, this will produce a non-proper resolution.)

118.3 Geometrical Properties of Cones and Polyhedra

`IsSingular(C)`

Return `true` if and only if the affine variety associated with the cone C is singular.

`IsNonsingular(C)`

Return `true` if and only if the affine variety associated with the cone C is nonsingular.

`IsSmooth(P)`

Return `true` if and only if the polyhedron P is a smooth polytope.

`IsGorenstein(C)`

Return `true` if and only if the cone C has (the primitive points on its) rays contained in an affine hyperplane that is defined by an integral equation.

`IsReflexive(P)`

Return `true` if and only if the polyhedron P is reflexive; i.e. P and its dual $P^\vee$ are both integral polytopes.

`IsQGorenstein(C)`

Return `true` if and only if the cone C has (the primitive points on its) rays contained in an affine hyperplane.

`GorensteinIndex(C)`

The Gorenstein index of the affine variety corresponding to the cone C together with the dual vector determining the equation of the hyperplane. (It is an error if C is not $\mathbf{Q}$-Gorenstein.)

`GorensteinIndex(P)`

The Gorenstein index of the lattice polytope P; i.e. the smallest positive integer k such that $kP^\vee$ is an integral polytope.

`IsQFactorial(C)`
`IsSimplicial(P)`

Return `true` if and only if the cone C or polytope P is simplicial.

`IsTerminal(C)`

Return `true` if and only if the singularity of the affine variety associated to the cone C is (at worst) terminal.

`IsCanonical(C)`

Return `true` if and only if the singularity of the affine variety associated to the cone C is (at worst) canonical.

`IsFano(P)`

Return `true` if and only if the polyhedron P is a Fano polytope (i.e. of maximum dimension in the ambient lattice, containing the origin strictly in its interior, with primitive lattice vertices).

Example H118E7

We make the cone corresponding to the (affine) terminal quotient singularity $\mathbf{C}^3/(\mathbf{Z}/5)$ where $\mathbf{Z}/5$ acts as the 5th roots of unity in the diagonal representation $\mathrm{diag}(1,2,3)$.

```
> L := ToricLattice(3);
> v := L ! [1/5,2/5,3/5];
> LL,emb := AddVectorToLattice(v);
> C := PositiveQuadrant(L);
> CC := Image(emb,C);
> CC;
Cone CC with 3 generators:
    (1, 0, 0),
    (0, 1, 0),
```

```
    (3, 1, 5)
```

We can check that this really is terminal and compute its Gorenstein index, the least positive multiple of the canonical class that is Cartier.

```
> IsTerminal(CC);
true
> GorensteinIndex(CC);
5 (1, 1, -3/5)
```

We can compute a resolution of singularities of this cone, the analogue of a simplicial subdivision for cones, although we must treat it as a fan to do so.

```
> F := Fan(CC);
> F;
Fan F with 3 rays:
    (0, 1, 0),
    (1, 0, 0),
    (3, 1, 5)
and one cone with indices:
    [ 1, 2, 3 ]
> Resolution(F);
Fan with 8 rays:
    (0, 1, 0),
    (1, 0, 0),
    (3, 1, 5),
    (2, 1, 2),
    (1, 1, 1),
    (3, 1, 3),
    (3, 1, 4),
    (2, 1, 3)
and 11 cones
```

Note that this is not a minimal resolution: such a resolution would only need to subdivide at the four additional rays at the (original) lattice points $1/5(1,2,3)$, $1/5(2,4,1)$, $1/5(3,1,4)$ and $1/5(4,3,2)$.

118.4 Toric Varieties

118.4.1 Constructors for Toric Varieties

We list some simple constructors for simple toric varieties. There are more general constructors for toric varieties (either from their fans or their Cox rings) in other sections.

`ToricVariety(k,n)`

Projective n-space $\mathbf{P}^n$ defined over the field k as a toric variety.

`ToricVariety(k,Z)`

The (weighted) projective space $\mathbf{P}(Z)$ defined over the field k with weights the positive integer sequence Z as a toric variety.

`ToricVariety(k,Z,Q)`

The fake weighted projective space defined over the field k with weights the positive integer sequence Z and a single sequence of quotient weights the sequence Q of rational numbers.

`ToricVariety(k,M,v)`

The n-dimensional toric variety $n \geq 2$ defined over the field k with weights begin the two sequences of integers (of the same length $n+2$) that comprise M and linearisation the length 2 integer sequence v. (This toric variety is the GIT quotient of k^{n+2} by a 2-dimensional torus acting with weights M and linearisation v. To get a toric variety of the right dimension, v must lie in the mobile cone implicit in the notation. In practice, this means that the columns of M must generate a cone with vertex in a 2-dimensional toric lattice and v must lie in the 'very-interior' of that cone, in the sense that it must lie in the strict interior of C and in the subcone generated by all columns of M except the two most extreme.)

Example H118E8

We build a Hirzebruch surface as a GIT quotient.

```
> X<u,v,x,y> := ToricVariety(Rationals(),[[1,1,0,-1],[0,0,1,1]],[1,1]);
> X;
Toric variety of dimension 2
Variables: u, v, x, y
The components of the irrelevant ideal are:
    (y, x), (v, u)
The 2 gradings are:
    1,  1,  0, -1,
    0,  0,  1,  1
```

The polarisation $(1,1)$ that we used is forgotten—all that is left is X.

`ToricVariety(k)`

The zero-dimensional point over the field k defined as a toric variety.

`ProjectiveSpace(k,n)`

Projective n-space $\mathbf{P}^n$ defined over the field k.

`ProjectiveSpace(k,W)`

`FakeProjectiveSpace(k,W,Q)`

The (fake) weighted projective space over the field k with weights the sequence of integers W (and quotient weights the sequence of sequences of rational numbers, if provided).

118.4.2 Toric Varieties and Their Fans

`ToricVariety(k,F)`

The toric variety (defined over the field k) corresponding to the toric fan F.

`Fan(X)`

The toric fan corresponding to the toric variety X.

`Rays(X)`

The rays of the fan of the toric variety X.

`OneParameterSubgroupsLattice(X)`

The lattice of weights of the toric variety X; this is the lattice which supports the toric fan of X.

`MonomialLattice(X)`

The monomial lattice of the toric variety X, namely the toric lattice dual to that containing the fan of X.

`CoxMonomialLattice(X)`

The lattice whose elements represent Weil divisors on the toric variety X; it is dual to ray lattice of X.

`DivisorClassLattice(X)`

The divisor class lattice of the toric variety X.

`IrrelevantIdeal(X)`

A sequence of ideals that are the components of the irrelevant ideal of the toric variety X.

`QuotientGradings(X)`

A sequence of sequences of rational numbers describing the quotients by finite cyclic groups that arise in the construction of the toric variety X.

`NumberOfQuotientGradings(X)`

The number of sequences the generate the quotient gradings of the toric variety X.

118.4.3 Properties of Toric Varieties

`IsSingular(X)`

Return `false` if and only if the toric variety X is nonsingular.

`IsNonsingular(X)`

Return `true` if and only if the toric variety X is nonsingular.

`IsGorenstein(X)`

Return `true` if and only if the toric variety X is Gorenstein.

`IsQGorenstein(X)`

Return `true` if and only if the toric variety X is $\mathbf{Q}$-Gorenstein.

`IsQFactorial(X)`

Return `true` if and only if the toric variety X is $\mathbf{Q}$-factorial.

`IsTerminal(X)`

Return `true` if and only if the toric variety X has (at worst) terminal singularities.

`IsCanonical(X)`

Return `true` if and only if the toric variety X has (at worst) canonical singularities.

`IsComplete(X)`

Return `true` if and only if the toric variety X is complete.

`IsProjective(X)`

Return `true` if and only if the toric variety X is projective.

`IsFano(X)`

Return `true` if and only if the anticanonical divisor of the toric variety X is ample.

`IsFakeWeightedProjectiveSpace(X)`

Return `true` if and only if the toric variety X has exactly one $\mathbf{Z}$-grading.

`IsWeightedProjectiveSpace(X)`

Return `true` if and only if the toric variety X has exactly one $\mathbf{Z}$-grading and no quotient gradings.

118.4.4 Affine Patches on Toric Varieties

`ToricAffinePatch(X,i)`

The affine patch corresponding to i-th cone of fan of the toric variety X together with the inclusion map.

`ToricAffinePatch(X,S)`

`ToricAffinePatch(X,S)`

The toric variety, obtained from the toric variety X by set the monomials of the sequence S set to be non-zero (or alternatively the variables of X with indices from the sequence of integers S set non-zero). The inclusion map is returned as a second value.

118.5 Cox Rings

The Cox ring of a toric variety X is a polynomial ring whose variables are in bijection with the 1-skeleton of the fan of X together with three sequences of additional data:

(1) a sequence of the components of an ideal called *the irrelevant ideal*;

(2) a sequence of integral lattice points determining weights of $\mathbf{G}_m$ actions, called the $\mathbf{Z}$ *weights*;

(3) a sequence of rational lattice points determining weights of finite cyclic group actions, called the *quotient weights*.

When the Cox ring of a toric variety is displayed in MAGMA, all nontrivial data is also printed, but any sequences that are empty are omitted.

Cox rings provide a powerful way to construct toric varieties: under some mild conditions, specification of data of this nature determines a toric variety.

118.5.1 The Cox Ring of a Toric Variety

Cox [Cox95] associates a ring, now called the *Cox ring*, to a toric variety, and MAGMA allows exactly the same construction.

`CoxRing(X)`

The Cox ring of the toric variety X.

`CoxRing(k,F)`

The Cox ring of the toric variety defined over field k by the fan F.

Example H118E9

We build the weighted projective space $\mathbf{P}^2(1,2,3)$.

```
> P<x,y,z> := ProjectiveSpace(Rationals(),[1,2,3]);
```

The Cox ring of $\mathbf{P}^2(1,2,3)$ is the usual homogeneous coordinate ring, graded by the weights $1,2,3$ of the space—that is, x has weight 1, y has weight 2 and z has weight 3.

```
> CoxRing(P);
Cox ring with underlying Graded Polynomial ring of rank 3 over Rational Field
Order: Graded Reverse Lexicographical
Variables: x, y, z
Variable weights: [1, 2, 3]
The irrelevant ideal is:
    (x, y, z)
The grading is:
    1, 2, 3
```

The irrelevant ideal is the usual one for projective spaces: it decrees that $(0,0,0)$ is not a point of $\mathbf{P}^2$ since it lies in the locus defined by the irrelevant ideal.

Example H118E10

We build a toric variety X_2 whose fan resembles that of $\mathbf{P}^2$.

```
> F2 := Fan([[1,2],[-2,-1],[1,-1]], [[1,2],[1,3],[2,3]]);
> X2<u,v,w> := ToricVariety(Rationals(),F2);
```

However, X is not isomorphic to $\mathbf{P}^2$. The small catch is that the 1-skeleton of the fan F_2 that we defined (in other words, those three vectors $(1,2)$, $(-2,-1)$ and $(1,-1)$) does not generate the lattice N, but only a sublattice. So X will be the quotient of $\mathbf{P}^2$ by some finite group action.

```
> CoxRing(X2);
Cox ring with underlying Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: u, v, w
The irrelevant ideal is:
    (w, v, u)
The quotient grading is:
    1/3( 0, 2, 1 )
The integer grading is:
         1, 1, 1
```

The returned data are very similar to those for the Cox ring of $\mathbf{P}^2$. The difference is in the third piece of data: a sequence containing the single element $1/3(0,2,1)$. This indicates that X is the quotient of $\mathbf{P}^2$ by the action of $\mathbf{Z}/3$ given by

$$\varepsilon : (u,v,w) \mapsto (u, \varepsilon^2 v, \varepsilon w).$$

where ε is a cube-root of unity.

118.5.2 Cox Rings in Their Own Right

The introduction to this section describes Cox rings in abstract terms. It is possible to define them as polynomial rings plus additional data without naming a toric variety or a fan in the first place.

`CoxRing(R,B,Z,Q)`

The Cox ring with polynomial ring R of rank n (that is, having n variables) and additional data as follows: B is a sequence of ideals (or of sequences of elements of R, each of which will be interpreted as the generators of ideals); Z is a sequence of sequences of integers, each one of length n; Q is a sequence of sequences of rationals, each one of length n.

The sequence B is regarded as the components of the irrelevant ideal, and Z and Q are the $\mathbf{Z}$ weights and quotient weights respectively.

`C1 eq C2`

Return `true` if and only if the two Cox rings C_1 and C_2 have the same underlying polynomial rings and are defined by the same combinatorial data.

`BaseRing(C)`

`CoefficientRing(C)`

The coefficient field of the Cox ring C.

`UnderlyingRing(C)`

The underlying polynomial ring of the Cox ring C.

`Length(C)`

The rank of the underlying polynomial ring of the Cox ring C, that is, the number of polynomial variables of C.

`IrrelevantIdeal(C)`

The irrelevant ideal of the Cox ring C.

`IrrelevantComponents(C)`

A sequence containing the components of the irrelevant ideal of the Cox ring C.

`IrrelevantGenerators(C)`

A sequence of sequences, each containing the generators of the components of the irrelevant ideal of the Cox ring C.

`Gradings(C)`

The $\mathbf{Z}$ gradings of the Cox ring C, that is, a sequence of sequences of integers.

`NumberOfGradings(C)`

The number of **Z** gradings of the Cox ring C.

`QuotientGradings(C)`

The quotient gradings of the Cox ring C, that is, a sequence of sequences of rational numbers.

`NumberOfQuotientGradings(C)`

The number of quotient gradings of the Cox ring C.

`C . i`

The i-th indeterminate for the underlying polynomial ring of the Cox ring C.

`AssignNames(~C, S)`

Procedure to change the printed names of the indeterminates of the Cox ring C. The ith indeterminate will be given name the ith element of the sequence S of strings (which has length at most the number of indeterminates of C). This does not change the names of the indeterminates for calling—this must be done with an explicit assignment or with the angle bracket notation when defining the Cox ring in the first place.

`Name(C,i)`

The ith variable of the underlying polynomial ring of the Cox ring C.

118.5.3 Recovering a Toric Variety From a Cox Ring

It is simple either to recover a toric variety from a Cox ring (if one exists at all) or the fan and associated lattice machinery corresponding to the toric variety. The algorithm is straightforward: use the **Z** weights to determine the rays of a fan and then the irrelevant ideal to construct the rest of the cone structure of the fan. The quotient weights may then require the lattice to be extended to a larger lattice but containing the same fan. With this point of view, the Cox ring can be regarded as the primary collection of data of a toric variety.

`ToricVariety(C)`

The toric variety whose Cox ring is C. It is not checked whether the Cox data defines a toric variety; if you are unsure, you should ask for the fan.

Example H118E11

Sometimes, rather than defining a fan, it is easier to construct a Cox ring first and build a toric variety from that.

```
> R<x1,x2,x3,y1,y2,y3,y4> := PolynomialRing(Rationals(),7);
> I := [ ideal<R|x1,x2,x3>, ideal<R|y1,y2,y3,y4> ];
> Z := [ [1,1,1,0,-3,-5,-5], [0,0,0,1,1,1,1] ];
> Q := [];
> C := CoxRing(R,I,Z,Q);
> C;
Cox ring C with underlying Polynomial ring of rank 7 over Rational Field
Order: Lexicographical
Variables: x1, x2, x3, y1, y2, y3, y4
The components of the irrelevant ideal are:
    (y4, y3, y2, y1), (x3, x2, x1)
The 2 gradings are:
    1,  1,  1,  0, -3, -5, -5,
    0,  0,  0,  1,  1,  1,  1
> X := ToricVariety(C);
> X;
Toric variety of dimension 5
Variables: x1, x2, x3, y1, y2, y3, y4
The components of the irrelevant ideal are:
    (y4, y3, y2, y1), (x3, x2, x1)
The 2 gradings are:
    1,  1,  1,  0, -3, -5, -5,
    0,  0,  0,  1,  1,  1,  1
```

Now MAGMA can compute the fan of X if we really want it.

```
> Fan(X);
Fan with 7 rays:
    ( 1,  0,  0,  0,  0),
    ( 0,  1,  0,  0,  0),
    ( 0,  0,  1,  0,  0),
    ( 1,  1,  1,  2,  0),
    (-3, -3, -3, -5,  0),
    ( 0,  0,  0,  0,  1),
    ( 2,  2,  2,  3, -1)
and 12 cones
```

`Fan(C)`

The fan associated to the Cox ring is C; an error is reported if there is no such fan.

`CoxMonomialLattice(C)`

The Cox monomial lattice of the Cox ring C.

`DivisorClassLattice(C)`

The divisor class lattice of the Cox ring C.

`MonomialLattice(C)`

The monomial lattice of the Cox ring C.

`OneParameterSubgroupsLattice(C)`

The one-parameter subgroups lattice of the Cox ring C.

`RayLattice(C)`

The ray lattice of the Cox ring C.

`DivisorClassGroup(C)`

The divisor class group of the Cox ring C.

`RayLatticeMap(C)`

The map from the ray lattice of the Cox ring C to the ambient lattice of its fan.

`WeilToClassGroupsMap(C)`
`WeilToClassLatticesMap(C)`

Comparison maps between toric lattices related to the Cox ring C of a toric variety.

118.6 Invariant Divisors and Riemann-Roch Spaces

Divisors on toric varieties work in the same way as on any other varieties, except that within each linear equivalence class it is possible to choose torus invariant representatives. These invariant divisors are composed of toric strata, and so in raw combinatorial terms one can regard divisors as being integer (or rational) labels on the rays of the fan. This is a convenient way to construct divisors, but there are many other methods.

As for other schemes on which divisor calculations are defined in MAGMA, divisors on a toric variety have a single divisor group as their parent. Divisors can be constructed by coercing appropriate data into this group, but this is not the only method so it can be ignored for most purposes. (This group is, however, the connection between divisors and the toric variety, so it is always alive in the background.)

118.6.1 Divisor Group

`DivisorGroup(X)`

The divisor group of the toric variety X. This is simply a parent object for divisors on X, and it is not computed as an abstract group.

`ToricVariety(G)`

The toric variety of which G is the divisor group.

`G1 eq G2`

Return `true` if and only if the divisor groups G_1 and G_2 are those of the same toric variety.

`Divisor(G,S)`
`Divisor(G,S)`

The divisor on the toric variety X associated to the divisor group G with coefficients given by the sequence S of integers or rationals with respect to the rays of the fan of X.

`Divisor(G,i)`

The divisor on the toric variety X associated to the divisor group G given by the vanishing of the ith coordinate of X.

118.6.2 Constructing Invariant Divisors

`Divisor(X,S)`

The Weil divisor (respectively **Q**-Weil divisor) on the toric variety X whose multiplicity on the ith coordinate divisor is the ith element of the sequence S of integers (respectively, rational numbers).

`Divisor(X,i)`

The divisor on the toric variety X given by the vanishing of the ith coordinate of X.

`Divisor(X,f)`

The divisor on the toric variety X defined by the polynomial f of the Cox ring of X.

`Divisor(X,m)`

If m is in the monomial lattice of the toric variety X, this gives the principal divisor on X corresponding to the monomial m. If m is a form on the ray lattice of X, then this gives the Weil divisor corresponding to m.

`ZeroDivisor(X)`

The zero divisor on the toric variety X.

`Representative(X,m)`

`effective` BoolElt *Default :* `true`

A divisor D on the toric variety X whose class modulo linear equivalence equals m, an element of the divisor class group of X. Unless the parameter `effective` is set to `false`, D will be chosen to be effective if possible.

`Representative(X,m)`

`effective` BoolElt *Default :* `true`

A divisor D on the toric variety X whose class modulo linear equivalence equals m, an element of the Picard lattice or divisor class lattice of X. Unless the parameter `effective` is set to `false`, D will be chosen to be effective if possible.

`CanonicalDivisor(X)`

The canonical divisor of the toric variety X.

`CanonicalClass(X)`

`group` MonStgElt *Default :* "*Pic*"

The class of canonical divisor of the toric variety X. By default this is returned as an element of the Picard lattice of X (and X must be $\mathbf{Q}$-Gorenstein for this to make sense). However, the parameter `group` can be changed to `Cl` to return the divisor in the divisor class lattice or `ClZ` to return the divisor in the divisor class group.

`D1 + D2` `n * D` `- D` `D1 - D2` `D * v`

Standard arithmetic operations for divisors D, D_1, D_2 on a toric variety, where $n \in \mathbf{Q}$ and v is a point of the ambient toric lattice of the corresponding fan.

Example H118E12

We compute the Kawamata blowup of $1/7(1, 2, 5)$. First we construct the singular cone by hand:

```
> L := ToricLattice(3);
> C := PositiveQuadrant(L);
> v := L![1/7,2/7,5/7];
> LL,phi := AddVectorToLattice(v);
> CC := Cone(phi(Rays(C)));
> CC;
3-dimensional simplicial cone CC with 3 minimal generators:
    (1, 0, 0),
    (0, 1, 0),
    (4, 1, 7)
```

Now we compute the blowup.

```
> FF := Fan(CC);
```

```
> vv := phi(v);
> vv;
(3, 1, 5)
> GG := Blowup(Fan(CC),vv);
```

The blowup map is easy to recover:

```
> X := ToricVariety(Rationals(),FF);
> Y<x,y,z,t> := ToricVariety(Rationals(),GG);
> f := ToricVarietyMap(Y,X);
> f;
A map between toric varieties described by:
    (t)^(2/7)*(x),
    (t)^(1/7)*(y),
    (t)^(5/7)*(z)
```

Finally we shall compute the discrepancy of this Kawamata blowup. It should be 1/7.

```
> KX := CanonicalDivisor(X);
> KY := CanonicalDivisor(Y);
> KY - Pullback(f,KX);
Q-Weil divisor with coefficients:
    0, 0, 0, 1/7
```

118.6.3 Properties of Divisors

`Variety(D)`

The toric variety on which the divisor D is defined.

`Parent(D)`

The divisor group of a toric variety in which the divisor D lies.

`Weil(D)`

The multiplicities on rays of the fan of X that determine the invariant divisor D, where X is the toric variety on which D is defined.

`Cartier(D)`

The sequence of toric lattice elements (of the monomial lattice of X) that determine the divisor D on the toric affine patches of the toric variety X on which D lies. This requires that D be $\mathbf{Q}$-Cartier.

`IsQCartier(D)`

Return `true` if and only if some integer multiple of the divisor D on a toric variety is Cartier.

`IsCartier(D)`

Return `true` if and only if the divisor D on a toric variety is Cartier.

`IsWeil(D)`

Return `true` if and only if the divisor D on a toric variety is a Weil divisor (that is, its coefficients are integers rather than rational numbers).

`IsAmple(D)`

Returns `true` if and only if the divisor D on a toric variety is ample.

`IsNef(D)`

Return `true` if and only if the divisor D on a toric variety is nef.

`IsBig(D)`

Return `true` if and only if the divisor D on a toric variety is big.

`PicardClass(D)`

The class in the Picard lattice corresponding to the $\mathbf{Q}$-Cartier divisor D.

`MovablePart(D)`

The movable part (or mobile part) of the divisor D on a toric variety.

Example H118E13

We compute a variety as a blowup of the projective plane.

```
> X := ProjectiveSpace(Rationals(),[1,1,1]);
> Y<u,v,x,y> := Blowup(X, &+Rays(Fan(X))[1..2]);
> Y;
Toric variety of dimension 2
Variables: u, v, x, y
The components of the irrelevant ideal are:
    (y, x), (v, u)
The 2 gradings are:
    0, 0, 1, 1,
    1, 1, 1, 0
```

We consider a (toric coordinate) divisor on Y.

```
> D := Divisor(Y,4);
> MovablePart(D);
Weil divisor with coefficients:
    0, 0, 0, 0
> MovablePart(D) eq ZeroDivisor(Y);
```

```
true
```

The movable part of this divisor is the zero divisor. Adding a little bit of another effective divisor doesn't yet make a mobile divisor, but it has made it stably mobile: some multiple now has a movable part.

```
> E := D + (1/2)*Divisor(Y,u);
> MovablePart(E);
Weil divisor with coefficients:
    0, 0, 0, 0
>   MovablePart(2*E);
Weil divisor with coefficients:
    1, 0, 0, 1
>   MovablePart(2*E) eq (D + Divisor(Y,u));
true
```

`ImageFan(D)`

The dual fan to the rational polyhedron of sections of the divisor D on a toric variety X. If X is a complete variety, this will give the fan of Proj of the ring of sections of positive powers of D.

`Proj(D)`

Proj (as a toric variety) of the ring of sections of the divisor D on a toric variety. The map of underlying lattices which determines the map `Variety(D)` $\rightarrow$ `Proj(D)` is also returned.

`RelativeProj(D)`

The relative (sheaf) Proj of sections of the divisor D on a toric variety. If D is **Q**-Cartier, then the identity will be constructed; for non **Q**-Cartier divisors, a partial **Q**-factorialisation will be given.

`IntersectionForm(X,C)`

If the cone C is a codimension 1 face of the fan of the toric variety X, return the dual toric lattice vector that represents intersection of the toric subvariety corresponding to C.

118.6.4 Linear Equivalence of Divisors

`IsQPrincipal(D)`

Return `true` if and only if some integer multiple of the divisor D on a toric variety is principal.

`IsPrincipal(D)`

Return `true` if and only if the divisor D on a toric variety is principal.

`IsLinearlyEquivalentToCartier(D)`

Return `true` if and only if the divisor D on a toric variety is linearly equivalent to a Cartier divisor; if so, then a representative Cartier divisor is also returned.

`AreLinearlyEquivalent(D,E)`

`IsLinearlyEquivalent(D,E)`

Return `true` if and only if the divisors D and E are linearly equivalent.

`DefiningMonomial(D)`

The monomial (if D is effective) or rational monomial defining the divisor D on a toric variety.

`LatticeElementToMonomial(D,v)`

The monomial in the Cox ring that corresponds to the monomial lattice element v when regarded as a section of the divisor D.

118.6.5 Riemann–Roch Spaces of Invariant Divisors

`RiemannRochPolytope(D)`

The Riemann–Roch space of the divisor D as a polytope in the monomial lattice of the underlying toric variety.

`RiemannRochBasis(D)`

A basis of the Riemann–Roch space of the divisor D on a toric variety X: this is a sequence of rational functions on X.

`RiemannRochDimension(D)`

The dimension of the Riemann–Roch space of the divisor D on a toric variety.

`GradedCone(D)`

The graded cone of sections of multiples of the divisor D on a toric variety. In other words, the integral points of the ith graded piece of this cone represent sections of the divisor $i * D$.

`Polyhedron(D)`

The integral polyhedron whose integral points corresponds to sections of the divisor D.

Example H118E14

We make a simple toric variety Y by blowing up the plane.

```
> X := ProjectiveSpace(Rationals(),[1,1,1]);
> Y<u,v,x,y> := Blowup(X, &+Rays(Fan(X))[1..2]);
> Y;
Toric variety of dimension 2
Variables: u, v, x, y
The components of the irrelevant ideal are:
    (y, x), (v, u)
The 2 gradings are:
    0, 0, 1, 1,
    1, 1, 1, 0
```

We make a non-effective divisor as a difference of fibres of the natural map from Y to the line.

```
> D := 2*Divisor(Y,u) - Divisor(Y,v);
> IsEffective(D);
false
> P := Polyhedron(D);
> monos := [ LatticeElementToMonomial(D,v) : v in Points(P) ];
> monos;
[
    u,
    v
]
```

The polyhedron P is not quite the Riemann–Roch space of D, but it is for a divisor linearly equivalent to D.

```
> [ AreLinearlyEquivalent(Divisor(Y,m),D) : m in monos ];
[ true, true ]
```

`HilbertSeries(D)`

The Hilbert series of the divisor D on a toric variety X, namely

$$\Sigma_{m\geq 0} \dim H^0(X, mD).$$

This assumes that the spaces of sections $H^0(X, D)$ of D finite dimensional. This will be true if X is projective, for example, but it holds in other cases too.

`HilbertPolynomial(D)`

The Hilbert (quasi-)polynomial for the divisor D. The space of sections of D must be finite dimensional. That is, a sequence of polynomials $[p_0, \ldots, p_{r-1}]$ of length r, the quasi-period of the Hilbert polynomial, so that $\dim H^0(X, mD)$ is the value of $p_s(k)$ where $m = kr + s$ is the Euclidean division of m by r; in other words, s is the least residue of m modulo r. Note that since MAGMA indexes sequences from 1, we have that $p_i =$ `HilbertPolynomial(D)[i+1]`.

`HilbertCoefficients(D,l)`

The first $l+1$ coefficients of the Hilbert series of the divisor D on a toric variety (starting with $0D$ up to and including lD).

`HilbertCoefficient(D,i)`

Thefirst ith coefficient of the Hilbert series of the divisor D on a toric variety.

Example H118E15

One can recreate the Cech cohomology calculation of the Riemann–Roch space of a divisor D on a toric variety X:

$$H^0(X, \mathcal{O}_X(D)) = \cap_\sigma (m_\sigma + \check{\sigma})$$

where the sum is taken over top-dimensional cones σ in the fan of X, and m_σ is the monomial in the dual lattice which determines D on the affine patch X_σ; $\check{\sigma}$ is the cone dual to σ.

```
> X<x,y,z> := ProjectiveSpace(Rationals(),[1,3,5]);
> D := Divisor(X,[2,3,1]);
> cones := Cones(Fan(X));
> RRD := &meet [ Polytope([Cartier(D)[i]]) + Dual(cones[i]) : i in [1..3]];
> IsPolytope(RRD);
true
```

We could compute the number of points of this cone RRD by saying `#Points(RRD)`. This should be regarded as a slow method to determine the number of points—after all, it requires us to find all the points before counting them. There is a specialised point-counting intrinsic `NumberOfPoints` which does not find the points first. The latter should be used for point counting, although in relatively small examples the former can be a little faster.

```
> NumberOfPoints(RRD);
14
```

To compute also the number of points of integral dilations of this polytope, we revert to using the divisor and computing its Hilbert series (or a few coefficients if that's all we need).

```
> time HilbertCoefficients(D,10);
[ 1, 14, 44, 92, 156, 238, 337, 452, 585, 735, 902 ]
> h<t> := HilbertSeries(D);
> h;
(-4*t^8 - 21*t^7 - 38*t^6 - 51*t^5 - 51*t^4 - 47*t^3 - 30*t^2 - 13*t - 1)/(t^9 -
    t^8 - t^6 + t^5 - t^4 + t^3 + t - 1)
> h * (1 - t) * (1 - t^3) * (1 - t^5);
4*t^8 + 21*t^7 + 38*t^6 + 51*t^5 + 51*t^4 + 47*t^3 + 30*t^2 + 13*t + 1
```

118.7 Maps of Toric Varieties

The initial method of expressing some maps between toric varieties is to derive them from maps between their associated lattices. These can also then be presented in terms of the variables of the Cox ring; this is the usual method for describing toric maps between projective spaces, for instance. This often results in radical expressions such as

$$(u, v) \mapsto (\sqrt{u}, v, v\sqrt{u}).$$

This example could be describing a map from $\mathbf{P}^1$ to $\mathbf{P}(1, 2, 3)$, for example: it is a 'monomial' map which observes the gradings. In that case, we could represent the same map by $(u, v) \mapsto (u, uv, u^2v)$ or $(u, v) \mapsto (1, v/u, v/u)$, or a host of other expressions. These two expressions have the benefit that they are polynomial or rational functions in u and v, and so they automatically define rational maps, but they have disadvantages too: for example, evaluating the map at the point $(0, 1)$ is not defined for these expressions, whereas it gives image $(1, 0, 1)$ in the original radical expression. (Notice that the choice of root does not matter, as long as it is assumed that the same choice $a = \sqrt{u}$ is made at each coordinate.)

More generally, one can define all maps between toric varieties (not just those arising from maps of lattices) using an appropriate notion of 'rational radical function', defined in terms of the polynomial Cox coordinates. This is very common when describing maps between standard projective spaces: one writes down a sequence of homogeneous polynomials of the same degree, without demanding that they are monomials.

The key point is that a rational map between varieties pulls rational functions (that are defined on the image) back to rational functions. It is enough to test this on a basis of rational functions. In the example above, if x, y, z are the coordinates on $\mathbf{P}(1, 2, 3)$ then y/x^2 and z/x^3 form a basis, and these both pull back to v/u, which is a rational function on $\mathbf{P}^1$.

We allow maps to be constructed from maps of the underlying toric lattices of fans. When displayed, they are described in these radical polynomial terms.

118.7.1 Maps from Lattice Maps

`ToricVarietyMap(X,Y,f)`

`ToricVarietyMap(X,Y)`

The rational map between toric varieties X and Y determined by the map f between their respective toric lattices (that is, the lattices underlying their respective fans). If the map f is not specified, it is assumed to be the identity map (and X and Y are assumed to have the same toric lattice).

`Blowup(X,v)`

The blowup of the toric variety X at the toric lattice point v of the toric lattice containing the fan of X; the natural map from the blowup to X is also returned.

`IdentityMap(X)`

The identity map on the toric variety X.

118.7.2 Properties of Toric Maps

`IsRegular(f)`

Return `true` if and only if the map f between toric varieties is regular.

`IndeterminacyLocus(f)`

A sequence of subschemes of the toric variety that is the domain of the map f between toric varieties at which f is not defined. (Note that these subschemes may in fact be empty.)

Example H118E16

We build a map from a Hirzebruch surface using the complete linear system of divisor.

```
> F2<u,v,x,y> := HirzebruchSurface(Rationals(),2);
> D := Divisor(F2,x);
> Y,f := Proj(D);
> Y;
Toric variety of dimension 2
Variables: $.1, $.2, $.3
The irrelevant ideal is:
    ($.3, $.2, $.1)
The grading is:
    1, 1, 2
> f;
Mapping from: 2-dimensional toric lattice N to 2-dimensional toric lattice N
given by a rule
```

The image variety Y is clearly the weighted projective space $\mathbf{P}(1,1,2)$. The map f returned is a map of underlying lattices. We can convert it into a map of the toric varieties, after which it will be presented in Cox coordinates.

```
> F := ToricVarietyMap(F2,Y,f);
> F;
A map between toric varieties described by:
    1,
    (v)*(u)^(-1),
    (x)*(y)^(-1)*(u)^(-2)
```

Now we can ask whether this map F is a morphism.

```
> IsRegular(F);
true
```

118.8 The Geometry of Toric Varieties

118.8.1 Resolution of Singularities and Linear Systems

`Resolution(X)`

A resolution of singularities of the toric variety X together with the natural morphism from the resolution to X. The resolution is not necessarily minimal.

`ResolveLinearSystem(D)`

The toric variety Y whose fan lives in the same lattice as the fan of the toric variety X on which the divisor D is defined, such that Y resolves the map given by the linear system of D.

118.8.2 Mori Theory of Toric Varieties

`MoriCone(X)`

The Mori Cone of toric variety X (as an abstract cone), that is, the cone generated by numerical classes of torus invariant curves on X.

`NefCone(X)`

The nef Cone of toric variety X (as an abstract cone).

`ExtremalRays(X)`

`ExtremalRayContractions(X)`

The images of extremal contractions of rays in the nef-cone of the toric variety X.

`ExtremalRayContraction(X,i)`

The toric variety that is the image of the ith extremal contraction of the toric variety X; the contraction morphism is returned as a second value.

`ExtremalRayContractionDivisor(X,i)`

The toric divisor that gives the ith extremal contraction of the toric variety X (that is, the divisor is the pullback of an ample divisor on the image).

`TypeOfContraction(X,i)`

`TypesOfContractions(X)`

A string describing the ith extremal contraction of the toric variety X (or all together in a sequence if i is not specified).

`IsMoriFibreSpace(X,i)`

Return `true` if and only if the ith extremal ray of the toric variety X gives an extremal contraction to a variety of lower dimension than X.

`IsDivisiorialContraction(X,i)`

Return `true` if and only if the ith extremal ray of the toric variety X gives a divisorial contraction of X.

`IsFlipping(X,i)`

Return `true` if and only if the ith extremal ray of the toric variety X gives a small contraction of X; in this case the intrinsic `Flip(X,i)` provides the flipped toric variety.

`Flip(X,i)`

The flipped variety of the ith extremal contraction of the toric variety X, assuming that this extremal contraction is of flipping type.

`Flip(D)`

The (generalised) flip of the morphism given by the $\mathbf{Q}$-Cartier divisor, assuming that this morphism is small.

`WeightsOfFlip(X,i)`

The weights of a $\mathbf{G}_{\mathrm{m}}$ action whose variation would give the flip of the ith extremal contraction of the toric variety X, assuming that this extremal contraction is of flipping type.

Example H118E17

```
> F0 := FanOfWPS([1,1,1,1]);
> L3 := Ambient(F0);
> F := Blowup(F0,L3 ! [2,-5,3]);
> X := ToricVariety(Rationals(),F);
> ExtremalRays(X);
[
    (0, -1),
    (1, 56)
]
> TypeOfContraction(X,1);
divisorial (K.C<0)
> TypeOfContraction(X,2);
flip
> WeightsOfFlip(X,2);
[
    [ 3, 2, -5, -1 ]
]
```

Example H118E18

We build a (nonsingular) variety X that is the projectivisation of the direct sum of line bundles $\mathcal{O}(0,0,0,1,1,1,1,2)$ on the projective line $\mathbf{P}^1$.

```
> X<[x]> := RationalScroll(Rationals(),1,[0,0,0,1,1,1,1,2]);
> X;
Toric variety of dimension 8
Variables: x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8], x[9], x[10]
The components of the irrelevant ideal are:
    (x[10], x[9], x[8], x[7], x[6], x[5], x[4], x[3]), (x[2], x[1])
The 2 gradings are:
    1,  1,  0,  0,  0, -1, -1, -1, -1, -2,
    0,  0,  1,  1,  1,  1,  1,  1,  1,  1
```

We compute its nef cone simply to initiate all its Mori theoretic data.

```
> _ := NefCone(X);
```

We can consider various extreme rays of the Mori cone of X. (Our choice of parameters mean we consider rays $[C]$ for which $DC \leq 0$, where D is the zero divisor: that is, we consider all rays.)

```
> IsFlipping(X,1: divisor:=ZeroDivisor(X), inequality:="weak");
false
> IsFlipping(X,2: divisor:=ZeroDivisor(X), inequality:="weak");
true
```

One of the rays corresponds to the fibration of X onto $\mathbf{P}^1$. The other is a ray of (anti-)flipping type. We can make the antiflip.

```
> Y<[y]> := Flip(X,2: divisor:=ZeroDivisor(X), inequality:="weak");
> Y;
Toric variety of dimension 8
Variables: y[1], y[2], y[3], y[4], y[5], y[6], y[7], y[8], y[9], y[10]
The components of the irrelevant ideal are:
    (y[10], y[9], y[8], y[7], y[6]), (y[5], y[4], y[3], y[2], y[1])
The 2 gradings are:
    0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 2, 2, 2, 1, 1, 1, 1, 0
> IsNonsingular(Y);
false
> IsTerminal(Y);
true
```

This antiflip can be regarded as coming from a change in the linearisation of a geometric invariant theory quotient: from a linearisation like $(1,1)$ (between variables 2 and 3 in the given order) to one like $(2,3)$ (between variables 5 and 6). To understand the antiflip better, sometimes it helps to consider its weights (the relation between vertices on the star of the flipping locus, or, in geometric invariant theory terms, the weights of the local $\mathbf{G}_{\mathrm{m}}$ action that determine the flip).

```
> WeightsOfFlip(X,2: divisor:=ZeroDivisor(X), inequality:="weak");
[
```

```
    [ 1, 1, 0, 0, 0, -1, -1, -1, -1, -2 ],
    [ 0, 0, 1, 1, 1, 1, 1, 1, 1, 1 ]
]
```

Here we see that a $\mathbf{P}^1$ has been antiflipped in favour of a weighted $\mathbf{P}^4(1,1,1,1,2)$, which is the source of the singularity on Y.

`MMP(X)`

`type` MonStgElt *Default : "terminal"*

A sequence of toric varieties that are all the varieties visited by making any sequence of extremal contractions from the toric variety X (and, if necessary, making the corresponding flip). A second sequence records the maps, in each case first by a sequence of the indices of the domain and codomain, and second by a string that describes the map.

The parameter `type` indicates which extremal rays are considered. It can be `terminal`, `canonical` or `all`. In each case, only toric varieties having these singularities will be allowed as images of the maps. (In particular, if the default value `terminal` is chosen, then only true K_X-negative extremal contractions will be followed.)

Example H118E19

First make a toric variety X, in this case some blowup of $\mathbf{P}^3$.

```
> F := FanOfWPS([1,1,1,1]);
> G := Blowup(F, Ambient(F) ! [1,-1,1]);
> X := ToricVariety(Rationals(),G);
```

We compute all minimal model programs from X. There are two outputs: first a sequence containing those toric varieties encountered during these processes, and second a sequence containing all the maps encountered.

```
> models,mmp := MMP(X);
> #models;
3
> mmp;
[ [*
    [ 1, 2 ],
    divisorial (K.C<0)
*], [*
    [ 2, 3 ],
    map to point
*] ]
```

In this case there are three varieties. We could check that they are: (1) X itself, (2) $\mathbf{P}^3$, and (3) a point. We also see two maps: the first, labelled $[1,2]$, is the contraction of X back down to $\mathbf{P}^3$, and the second is the extremal contraction of $\mathbf{P}^3$ to a point.

The intrinsic call `MMP(X)` has a parameter, and the default is to search only for true absolute minimal model programs: those which proceed only by contracting extremal rays that are negative against the canonical class (and flipping them if necessary). To allow contractions of other extremal rays (and the possibility of requiring antiflips), we can set the parameter to `type:="all"`.

```
> models,mmp := MMP(X : type:="all");
> models;
[
    Toric variety of dimension 3
    Variables: $.1, $.2, $.3, $.4, $.5
    The components of the irrelevant ideal are:
        ($.5, $.4), ($.3, $.2, $.1)
    The 2 gradings are:
        1, 0, 0, 2, 1,
        1, 1, 1, 1, 0,
    Toric variety of dimension 3
    Variables: $.1, $.2, $.3, $.4
    The irrelevant ideal is:
        ($.4, $.3, $.2, $.1)
    The grading is:
        1, 1, 1, 1,
    Toric variety of dimension 3
    Variables: $.1, $.2, $.3, $.4, $.5
    The components of the irrelevant ideal are:
        ($.3, $.2), ($.5, $.4, $.1)
    The 2 gradings are:
        1, 0, 0, 2, 1,
        1, 1, 1, 1, 0,
    Toric variety of dimension 0,
    Toric variety of dimension 1
    Variables: $.1, $.2
    The irrelevant ideal is:
        ($.2, $.1)
    The grading is:
        1, 1
]
> mmp;
[ [*
    [ 1, 2 ],
    divisorial (K.C<0)
*], [*
    [ 1, 3 ],
    flop
*], [*
    [ 2, 4 ],
    map to point
*], [*
    [ 3, 1 ],
```

```
    flop
*], [*
    [ 3, 5 ],
    fibration (K.C<0)
*] ]
```

Now we also see a flop from X to a new toric variety Y (model number 3), and then Y admits Mori fibration to $\mathbf{P}^1$ (which we could check is model number 5).

118.8.3 Decomposition of Toric Morphisms

The title of this section is that of a well-known paper of Reid [Rei83]. It applies Mori theory to toric varieties relatively over a base to compute a relative minimal model and the relative canonical model of a toric variety. Rather than wrap this up in intrinsics, we show how to apply the various components of this package to realise Reid's result.

Example H118E20

We will compute a relative minimal model and a relative canonical model of the weighted projective space $\mathbf{P}^3(1,2,5,6)$, a 3-fold that does not have canonical (nor terminal) singularities.

```
> A := ProjectiveSpace(Rationals(),[1,2,5,6]);
> IsCanonical(A);
false
```

We find the relative models by running a minimal model program on a resolution of A relative to A itself (that is, we only allow morphisms $V \to W$ if they factor the given morphism $V \to A$). So we start by constructing a resolution.

```
> V0,f0 := Resolution(A);
> V0;
Toric variety of dimension 3
Variables: $.1, $.2, $.3, $.4, $.5, $.6, $.7, $.8, $.9, $.10, $.11, $.12
The irrelevant ideal is:
   [... omitted... ]
The 9 gradings are:
    0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
    0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0,
    0, 0, 1, 2, 0, 0, 0, 1, 0, 0, 0, 0,
    0, 0, 2, 3, 0, 0, 0, 0, 0, 1, 0, 0,
    0, 1, 2, 3, 0, 0, 1, 0, 0, 0, 0, 0,
    0, 1, 3, 3, 0, 0, 0, 0, 0, 0, 0, 1,
    0, 1, 3, 4, 1, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 4, 5, 0, 1, 0, 0, 0, 0, 0, 0,
    1, 2, 5, 6, 0, 0, 0, 0, 0, 0, 0, 0
```

We see that the resolution made $8 = 9 - 1$ blowups. To compute everything, we would keep track of the morphisms (as we have started with f_0 above); for brevity, we forego that.

We build a function which detects which of the extremal contractions from a given V (starting with V_0) that factor $V \to A$. This code also reports whether the contraction is divisorial or of flipping type so that we can make the next step accordingly.

```
> raysOverA := func< W |
>     [ <i,TypeOfContraction(W,i)> :
>               i in [1..#ExtremalRays(W)] |
>     IsRegular(ToricVarietyMap(ExtremalRayContraction(W,i),A)) ] >;
```

And so we look for extremal rays over A that we can contract.

```
> raysOverA(V0);
[ <1, "divisorial (K.C<0)">, <2, "divisorial (K.C<0)">,
<3, "divisorial (K.C<0)"> ]
```

In other words, the extremal rays 1, 2 and 3 all determine extremal divisorial contractions. (If we were writing a faster routine, we would probably stop as soon as we'd found that ray 1 worked.) We can pick any of these: we choose ray 1 and repeat the process.

```
> V1 := ExtremalRayContraction(V0,1);
> raysOverA(V1);
[ <1, "divisorial (K.C<0)">, <2, "divisorial (K.C<0)">, <4, "divisorial
(K.C<0)"> ]
```

This time rays 1, 2, and 4 work; again these lead to divisorial contractions. We follow ray 1.

```
> V2 := ExtremalRayContraction(V1,1);
> raysOverA(V2);
[ <2, "divisorial (K.C<0)">, <3, "divisorial (K.C<0)"> ]
> V3 := ExtremalRayContraction(V2,2);
> raysOverA(V3);
[ <2, "divisorial (K.C<0)"> ]
> V4 := ExtremalRayContraction(V3,2);
> raysOverA(V4);
[ <2, "divisorial (K.C<0)"> ]
> V5<[w]> := ExtremalRayContraction(V4,2);
> assert #raysOverA(V5) eq 0;
```

There are no extremal rays over A to contract. We have reached a relatively minimal model, V_5; we check its singularities, although there is no need.

```
> V5;
Toric variety of dimension 3
Variables: w[1], w[2], w[3], w[4], w[5], w[6]
The components of the irrelevant ideal are:
    (w[6], w[5]), (w[6], w[3]), (w[5], w[4], w[2]), (w[3], w[1]), (w[4], w[2],
        w[1])
The 3 gradings are:
    0, 0, 1, 1, 0, 1,
    0, 1, 2, 3, 1, 0,
    1, 2, 5, 6, 0, 0
> IsTerminal(V5);
```

```
true
```

To continue on to a relatively canonical model we must consider rays that are trivial against the canonical class, not negative as we have so far. We need to modify the relative rays function; we simply weaken the inequality used when evaluating against the canonical class.

```
> weakraysOverA := func< W |
>     [ <i,TypeOfContraction(W,i : inequality:="weak")> :
>                   i in [1..#ExtremalRays(W:inequality:="weak")] |
>  IsRegular(ToricVarietyMap(ExtremalRayContraction(W,i:inequality:="weak"),A))]>;
```

And so we continue.

```
> weakraysOverA(V5);
[ <2, "divisorial (K.C=0)"> ]
> V6<[u]> := ExtremalRayContraction(V5,2 : inequality:="weak");
> assert #weakraysOverA(V6) eq 0;
```

After one canonically-trivial divisorial contraction there are no rays left to contract. We have reached the relative canonical model.

```
> V6;
Toric variety of dimension 3
Variables: u[1], u[2], u[3], u[4], u[5]
The components of the irrelevant ideal are:
    (u[5], u[3]), (u[4], u[2], u[1])
The 2 gradings are:
    0, 0, 1, 1, 1,
    1, 2, 5, 6, 0
> IsTerminal(V6);
false
> IsCanonical(V6);
true
```

Of course, in this toric setting, it would have been simpler to find the single short vector in the fan of A that was causing all the trouble and blow that up.

118.9 Schemes in Toric Varieties

The polynomials of the Cox ring of a toric variety X provide homogeneous coordinates on X that can be used to define subschemes of X. These subschemes are true MAGMA schemes, and so the usual scheme machinery works for them. However, there is a substantial caveat to this for the first version of the toric geometry package: affine patches have not been installed systematically, and so scheme machinery that uses affine patches of schemes will not work.

118.9.1 Construction of Subschemes

`Scheme(X,f)`

The subscheme of the toric variety X defined by the polynomial f from the Cox ring of X.

`Scheme(X,Q)`

The subscheme of the toric variety X defined by the sequence Q of polynomials from the Cox ring of X.

Example H118E21

Toric varieties are the natural ambient space for many varieties. Here we review the example of a trigonal curve from the Schemes chapter (it is self-contained here).
First make a curve. (This curve is in fact trigonal—it admits a 3-to-1 cover of the projective line. Once you've had that thought, it's actually pretty clear: the defining equation is a cubic in y. But there's more to it than just being trigonal, as we will see.)

```
> P<x,y,z> := ProjectiveSpace(Rationals(),2);
> C := Curve(P,x^8 + x^4*y^3*z + z^8);
> Genus(C);
8
```

This curve is of general type (that is, its genus is at least 2), so we can consider the canonical map: that will either be an embedding or a 2-to-1 map to a projective line.
We make the canonical map take its image in a toric variety.

```
> eqns := Sections(CanonicalLinearSystem(C));
> X<[a]> := ProjectiveSpace(Rationals(),7);
> f := map< P -> X | eqns >;
> V := f(C);
> V;
Curve over Rational Field defined by
a[1]^3 + a[2]^2*a[4] + a[1]*a[8]^2,
a[1]^2*a[3] + a[2]^2*a[6] + a[3]*a[8]^2,
a[1]^2*a[5] + a[2]*a[4]*a[6] + a[5]*a[8]^2,
a[1]*a[4]*a[6] - a[2]^2*a[7],
a[1]*a[6]^2 - a[2]^2*a[8],
a[2]*a[6]^2 + a[1]^2*a[7] + a[7]*a[8]^2,
a[4]*a[6]^2 + a[1]^2*a[8] + a[8]^3,
a[2]*a[3] - a[1]*a[4],
a[3]^2 - a[1]*a[5],
a[3]*a[4] - a[1]*a[6],
a[4]^2 - a[2]*a[6],
a[2]*a[5] - a[1]*a[6],
a[3]*a[5] - a[1]*a[7],
a[4]*a[5] - a[2]*a[7],
a[5]^2 - a[1]*a[8],
a[3]*a[6] - a[2]*a[7],
```

```
a[5]*a[6] - a[2]*a[8],
a[3]*a[7] - a[1]*a[8],
a[4]*a[7] - a[2]*a[8],
a[5]*a[7] - a[3]*a[8],
a[6]*a[7] - a[4]*a[8],
a[7]^2 - a[5]*a[8]
```

All those binomial equations suggest that V lies on a toric variety embedded in $X = \mathbf{P}^7$. We can recover this toric variety and its map to X.

```
> W,g := BinomialToricEmbedding(V);
> Y<[b]> := Domain(g);
> Y;
Toric variety of dimension 2
Variables: b[1], b[2], b[3], b[4]
The components of the irrelevant ideal are:
    (b[3], b[2]), (b[4], b[1])
The 2 gradings are:
    0, 1, 1, 0,
    1, 0, 2, 1
```

It is a well-known consequence of (geometric) Riemann–Roch that trigonal curves lie on scrolls in their canonical embeddings. Exactly which scroll is an intrinsic property of the particular curve: the Maroni invariant of a trigonal curve can be realised as the twist that occurs in the scroll, in this case 2 (visible in the last line of output above).
This makes good sense: the scroll Y has a natural map to $\mathbf{P}^1$, and the equation of the curve W is a cubic in the fibre variables $b[2], b[3]$ so defines a 3-to-1 cover of the base.

```
> I := Saturation(DefiningIdeal(W),IrrelevantIdeal(Y));
> Basis(I);
[
     b[1]^8*b[2]^3 + b[1]*b[3]^3*b[4] + b[2]^3*b[4]^8
]
```

The need for saturation is already visible in the equations of V: all those cubics are really multiples of a single cubic on the scroll by irrelevant ideals, but written in the coordinates of the projective space.

118.10 Bibliography

[**Cox95**] David A. Cox. The homogeneous coordinate ring of a toric variety. *J. Algebraic Geom.*, 4(1):17–50, 1995.

[**Dan78**] V. I. Danilov. The geometry of toric varieties. *Uspekhi Mat. Nauk*, 33(2(200)): 85–134, 247, 1978.

[**Ful93**] William Fulton. *Introduction to toric varieties*, volume 131 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, NJ, 1993. The William H. Roever Lectures in Geometry.

[**Oda88**] Tadao Oda. *Convex bodies and algebraic geometry*, volume 15 of *Ergebnisse der Mathematik und ihrer Grenzgebiete (3) [Results in Mathematics and Related Areas (3)]*. Springer-Verlag, Berlin, 1988. An introduction to the theory of toric varieties, Translated from the Japanese.

[**Rei83**] Miles Reid. Decomposition of toric morphisms. In *Arithmetic and geometry, Vol. II*, volume 36 of *Progr. Math.*, pages 395–418. Birkhäuser Boston, Boston, MA, 1983.

PART XVI
ARITHMETIC GEOMETRY

119 RATIONAL CURVES AND CONICS

Chapter 119
RATIONAL CURVES AND CONICS

119.1 Introduction

This chapter describes the specialised categories of nonsingular plane curves of genus zero: The rational plane curves and conics. Rational curves and conics in MAGMA are nonsingular plane curves of degree 1 and 2, respectively. The central functionality for conics concerns the existence of points over the rationals. If a point is known to exist then a conic can be parametrised by a projective line or a rational curve. In addition, several algorithms are implemented to convert conics to standard Legendre, or diagonal, models and, for curves over $\mathbf{Q}$, to a reduced Legendre model or a minimal model. A rational curve in MAGMA is a linearly embedded image of the projective line to which the full machinery of algebraic plane curves can be applied. The special category of conics is called `CrvCon` and that of rational curves is `CrvRat`.

The central algorithms of this chapter deal with the classification and reduction of genus zero curves to one of the standard models to which efficient algorithms can be applied. These special types serve to classify all curves up to birational isomorphism. Since the canonical divisor K_C of a genus zero curve is of degree -2, a basis for the Riemann–Roch space of the effective divisor $-K_C$ has dimension 3 and gives an anti-canonical embedding in the projective plane. The homogeneous quadratic relations between the functions define a conic model for any genus zero curve. If the curve has a rational point then a similar construction with the divisor of this point gives a birational isomorphism with the projective line. For conics over the rationals, efficient algorithms of Simon [Sim05] allow one to first find a point, if one exists, and then to reduce to simpler models. The existence of a point is easily determined by local conditions, and this local data is carried by the data of the ramified or bad primes of reduction; if such a point exists then the existence can be certified by a certificate. Simon's algorithm in fact parametrises the curve (by the projective line) which gives a birational isomorphism with the curve.

Not every curve of genus zero can be "trivialised" by reduction to a rational curve in this way; the obstruction to having a rational point, and therefore to being parametrised by a projective line, is measured by the primes of bad reduction and also by the automorphism group, both of which are closely associated to an isomorphism class of quaternion algebras. The final algorithms of this chapter make use of this connection to compute the automorphism group of a curve and to find isomorphisms between conics.

119.2 Rational Curves and Conics

The general tools for constructing and analysing curves are described in Chapter 114. We do not repeat them here, but rather give some examples in Section 119.2.3 to demonstrate those basics that the user will need. In this section we describe the main parametrisation function for rational curves and functions which enable type change from a curve of genus zero to a rational curve.

119.2.1 Rational Curve and Conic Creation

Rational curves and conics are the specialised types for nonsingular plane curves of genus zero defined by polynomials of degree 1 and 2, respectively. The condition of nonsingularity is equivalent to that of absolute irreducibility for conics, and imposes no condition on a linear equation in the plane.

`Conic(coeffs)`

`Ambient` SCH

This creates a conic curve with the given sequence of coefficients (which should have length 3 or 6). A sequence $[a, b, c]$ designates the conic $aX^2 + bY^2 + cZ^2$, while a sequence $[a, b, c, d, e, f]$ designates the conic $aX^2 + bY^2 + cZ^2 + dXY + eYZ + fXZ$.

The optional parameter `Ambient` may be used to give the specific ambient projective space in which to create the conic; otherwise a new ambient will be created for it to lie in.

`Conic(M)`

`Ambient` SCH

This creates a conic curve associated to M, which must be a symmetric 3×3 matrix. Explicitly, the equation of the conic is $[X, Y, Z]M[X, Y, Z]^{tr}$.

The optional parameter `Ambient` may be used to give the specific ambient projective space in which to create the conic; otherwise a new ambient will be created for it to lie in.

`Conic(X, f)`

Returns the conic defined by the polynomial f in the projective plane X.

`IsConic(S)`

Returns `true` if and only if the scheme S is a nonsingular plane curve of degree 2, in which case it also returns a curve (of type `CrvCon`) with the same defining polynomial as S.

`RationalCurve(X, f)`

Returns the rational curve defined by the linear polynomial f in the projective plane X.

`IsRationalCurve(S)`

Returns `true` if and only if the scheme S is defined by a linear polynomial in some projective plane $\mathbf{P}^2$; if so, it also returns a curve of type `CrvRat` with the same defining polynomial as S in $\mathbf{P}^2$.

Example H119E1

In the following example we create a degree two curve over the rational field, then create a new curve of conic type using `IsConic`.

```
> P2<x,y,z> := ProjectivePlane(Rationals());
> C0 := Curve(P2, x^2 + 3*x*y + 2*y^2 - z^2);
> C0;
Curve over Rational Field defined by
x^2 + 3*x*y + 2*x^2 - z^2
> bool, C1 := IsConic(C0);
```

Clearly this is a nonsingular degree two curve, so `bool` must be `true` and we have created a new curve C_1 in the same ambient space P_2 but of conic type.

```
> C1;
Conic over Rational Field defined by
x^2 + 3*x*y + 2*y^2 - z^2
> AmbientSpace(C0) eq AmbientSpace(C1);
true
> DefiningIdeal(C0) eq DefiningIdeal(C1);
true
> C0 eq C1;
false
> Type(C0);
CrvPln
> Type(C1);
CrvCon
```

The equality test return `false` because the two objects are of different MAGMA type.

119.2.2 Access Functions

The basic access functions for rational curves and conics are inherited from the general machinery for plane curves and hypersurface schemes.

`DefiningPolynomial(C)`

Returns the defining polynomial of the conic or rational curve C.

`DefiningIdeal(C)`

Returns the defining ideal of the conic or rational curve C.

`BaseRing(C)` `BaseField(C)`

Returns the base ring of the curve C.

`Category(C)` `Type(C)`

Returns the category of rational curves `CrvRat` or of conics `CrvCon`; these are special subtypes of planes curves (type `CrvPln`), which are themselves subtypes of general curves (type `Crv`).

119.2.3 Rational Curve and Conic Examples

These examples illustrate how to obtain standard models of a curve of genus zero, either as a conic or as a parametrisation by the projective line.

Example H119E2

We begin with an example of a singular curve of geometric genus zero.

```
> P2<x,y,z> := ProjectivePlane(FiniteField(71));
> C := Curve(P2, (x^3 + y^2*z)^2 - x^5*z);
> C;
Curve over GF(71) defined by
x^6 + 70*x^5*z + 2*x^3*y^2*z + y^4*z^2
> ArithmeticGenus(C);
10
> Genus(C);
0
> #RationalPoints(C);
73
> Z := SingularSubscheme(C);
> Degree(Z);
18
```

We see that C is highly singular and that its desingularisation has genus zero. At most 18 of 73 points are singular; note that a nonsingular curve of genus zero would have 72 points. We now investigate the source of the extra points.

```
> cmps := IrreducibleComponents(Z);
> [ Degree(X) : X in cmps ];
[ 11, 7 ]
> [ Degree(ReducedSubscheme(X)) : X in cmps ];
[ 1, 1 ]
> [ RationalPoints(X) : X in cmps ];
[
{@ (0 : 0 : 1) @},
{@ (0 : 1 : 0) @}
]
```

Since the only singular rational points on C are $(0 : 0 : 1)$ and $(0 : 1 : 0)$, the "obvious" point $(1 : 0 : 1)$ must be nonsingular and we can use it to obtain a rational parametrisation of the curve as explained in Section 119.6.

```
> P1<u,v> := ProjectiveSpace(FiniteField(71), 1);
> p := C![1, 0, 1];
> m := Parametrization(C, Place(p), Curve(P1));
> S1 := {@ m(q) : q in RationalPoints(P1) @};
> #S1;
72
> [ q : q in RationalPoints(C) | q notin S1 ];
```

```
[ (0 : 1 : 0) ]
```

We conclude that the extra point comes from a singularity whose resolution does not have any degree one places over it (see Section 114.9 of Chapter 114 for background on places of curves). We can verify this explicitly.

```
> [ Degree(p) : p in Places(C![0, 1, 0]) ];
[ 2 ]
```

Example H119E3

In this example we start by defining a projective curve and we check that it is rational; that is, that it has genus zero.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C0 := Curve(P2, x^2 - 54321*x*y + y^2 - 97531*z^2);
> IsNonsingular(C0);
true
```

The curve C_0 is defined as a degree 2 curve over the rationals. By making a preliminary type change to the type of conics, `CrvCon`, we can test whether there exists a rational point over $\mathbf{Q}$ and use efficient algorithms of Section 119.5.1 for finding rational points on curves in conic form. The existence of a point (defined over the base field) is equivalent to the existence of a parametrisation (defined over the base field) of the curve by the projective line.

```
> bool, C1 := IsConic(C0);
> bool;
true
> C1;
Conic over Rational Field defined by
x^2 - 54321*x*y + y^2 - 97531*z^2
> HasRationalPoint(C1);
true (398469/162001 : -118246/162001 : 1)
> RationalPoint(C1);
(398469/162001 : -118246/162001 : 1)
```

The parametrisation intrinsic requires a one-dimensional ambient space as one of the arguments. This space will be used as the domain of the parametrisation map.

```
> P1<u,v> := ProjectiveSpace(Rationals(), 1);
> phi := Parametrization(C1, Curve(P1));
> phi;
Mapping from: Prj: P1 to CrvCon: C1
with equations :
398469*u^2 + 944072*u*v + 559185*v^2
-118246*u^2 - 200850*u*v - 85289*v^2
162001*u^2 + 329499*u*v + 162991*v^2
and inverse
-4634102139*x + 30375658963*y + 31793350442*z
5456130497*x - 25641668406*y - 32136366969*z
and alternative inverse equations :
```

```
5456130497*x - 25641668406*y - 32136366969*z
-6424037904*x + 21645471041*y + 31600239062*z
```

The defining functions for the parametrisation may look large, but they are defined simply by a linear change of variables from the 2-uple embedding of the projective line in the projective plane. We now do a naive search for rational points on the curve.

```
> time RationalPoints(C1 : Bound := 100000);
{@ @}
Time: 2.420
```

Although there were no points with small coefficients, the parametrisation provides us with any number of rational points:

```
> phi(P1![0, 1]);
(559185/162991 : -85289/162991 : 1)
> phi(P1![1, 1]);
(1901726/654491 : -404385/654491 : 1)
> phi(P1![1, 0]);
(398469/162001 : -118246/162001 : 1)
```

Example H119E4

The first part of this example illustrates how to obtain diagonal equations for conics.

```
> P2<x,y,z> := ProjectiveSpace(RationalField(), 2);
> f := 1134*x^2 - 28523*x*y - 541003*x*z - 953*y^2 - 3347*y*z - 245*z^2;
> C := Conic(P2, f);
> LegendrePolynomial(C);
1134*x^2 - 927480838158*y^2 - 18604260593650520388494 1*z^2
> ReducedLegendrePolynomial(C);
817884337*x^2 - y^2 - 353839285266278*z^2
```

Now we demonstrate how to extend the base field of the conic; that is, to create the conic with the same coefficients but in a larger field. This can be done by calling `BaseExtend`; in this instance, we move from the rationals to a particular number field.
(Note the assignment of names to C in order to make the printing nicer.)

```
> P<t> := PolynomialRing(RationalField());
> K := NumberField(t^2 - t + 723);
> C<u,v,w> := BaseExtend(C, K);
> C;
Conic over K defined by
1134*u^2 - 28523*u*v - 953*v^2 - 541003*u*w - 3347*v*w - 245*w^2
```

119.3 Conics

In this section we discuss the creation and basic attributes of conics, particularly the standard models for them. In subsequent sections we treat the local-global theory and existence of points on conics, the efficient algorithms for finding rational points, parametrisations and isomorphisms of genus zero curves with standard models, and finally the automorphism group of conics.

119.3.1 Elementary Invariants

`Discriminant(C)`

Given a conic C, returns the discriminant of C. The discriminant of a conic with defining equation

$$a_{11}x^2 + a_{12}xy + a_{13}xz + a_{22}y^2 + a_{23}yz + a_{33}z^2 = 0$$

is defined to be the value of the degree 3 form

$$4a_{11}a_{22}a_{33} - a_{11}a_{23}^2 - a_{12}^2a_{33} + a_{12}a_{13}a_{23} - a_{13}^2a_{22}.$$

Over any ring in which 2 is invertible this is just $1/2$ times the determinant of the matrix

$$\begin{pmatrix} 2a_{11} & a_{12} & a_{13} \\ a_{12} & 2a_{22} & a_{23} \\ a_{13} & a_{23} & 2a_{33} \end{pmatrix}.$$

119.3.2 Alternative Defining Polynomials

The functions described here provide access to basic information stored for a conic C. In addition to the defining polynomial, curves over the rationals compute and store a diagonalised *Legendre* model for the curve, whose defining polynomial can be accessed.

`LegendrePolynomial(C)`

Returns the Legendre polynomial of the conic C, a diagonalised defining polynomial of the form $ax^2+by^2+cz^2$. Once computed, this polynomial is stored as an attribute. The transformation matrix defining the isomorphism from C to the Legendre model is returned as the second value.

`ReducedLegendrePolynomial(C)`

Returns the reduced Legendre polynomial of the conic C, which must be defined over $\mathbf{Q}$ or $\mathbf{Z}$; that is, a diagonalised integral polynomial whose coefficients are pairwise coprime and square-free. The transformation matrix defining the isomorphism from C to this reduced Legendre model is returned as the second value.

119.3.3 Alternative Models

`LegendreModel(C)`

Returns the Legendre model of the conic C — an isomorphic curve of the form

$$ax^2 + by^2 + cz^2 = 0,$$

together with an isomorphism to this model.

`ReducedLegendreModel(C)`

Returns the reduced Legendre model of the conic C, which must be defined over $\mathbf{Q}$ or $\mathbf{Z}$; that is, a curve in the diagonal form $ax^2 + by^2 + cz^2 = 0$ whose coefficients are pairwise coprime and square-free. The isomorphism from C to this model is returned as a second value.

119.3.4 Other Functions on Conics

`MinimalModel(C)`

Returns a conic, the matrix of whose defining polynomial has smaller discriminant than that of the conic C (where possible). The algorithm used is the minimisation stage of Simon's algorithm [Sim05], as used in `HasRationalPoint`. A map from the conic to C is also returned.

Example H119E5

In the following example we are able to reduce the conic at 13.

```
> P2<x,y,z> := ProjectiveSpace(RationalField(), 2);
> f := 123*x^2 + 974*x*y - 417*x*z + 654*y^2 + 113*y*z - 65*z^2;
> C := Conic(P2, f);
> BadPrimes(C);
[ 491, 18869 ]
> [ x[1] : x in Factorization(Integers()!Discriminant(C)) ];
[ 13, 491, 18869 ]
> MinimalModel(C);
Conic over Rational Field defined by
-9*x^2 + 4*x*y + 6*x*z + 564*y^2 + 178*y*z + 1837*z^2
Mapping from: Conic over Rational Field defined by
-9*x^2 + 4*x*y + 6*x*z + 564*y^2 + 178*y*z + 1837*z^2 to CrvCon: C
with equations :
x + 6*y - 10*z
-x - 8*y + 4*z
-8*x - 50*y + 61*z
and inverse
-144/13*x + 67/13*y - 28/13*z
29/26*x - 19/26*y + 3/13*z
-7/13*x + 1/13*y - 1/13*z
```

119.4 Local-Global Correspondence

The Hasse–Minkowski principle for quadratic forms implies that a conic has a point over a number field if and only if it has a point over its completion at every finite and infinite prime. This provides an effective set of conditions for determining whether a conic has a point: Only the finite number of primes dividing the discriminant of the curve need to be checked, and by Hensel's lemma it is only necessary to check this condition to finite precision. The theory holds over any global field (a number field or the function field of a curve over a finite field) but the algorithms implemented at present only treat the field $\mathbf{Q}$.

119.4.1 Local Conditions for Conics

We say that p is a ramified or bad prime for a conic C if there exists no p-integral model for C with nonsingular reduction. Every such prime is a divisor of the coefficients of the Legendre polynomial. However, in general it is not possible to have a Legendre model whose coefficients are divisible only by the ramified primes. We use the term ramified or bad prime for a conic to refer to the local properties of C which are independent of the models.

`BadPrimes(C)`

Given a conic C over the rationals, returns the sequence of the finite ramified primes of C: Those at which C has intrinsic locally singular reduction. This uses quaternion algebras.

N.B. Although the infinite prime is not included, the data of whether or not $C/\mathbf{Q}$ is ramified at infinity is carried by the length of this sequence. The number of bad primes, including infinity, must be even so the parity of the sequence length specifies the ramification information at infinity.

119.4.2 Norm Residue Symbol

Hilbert's norm residue symbol logically pertains to the theory of quadratic forms and lattices, and to that of quadratic fields and their norm equations. We express it here for its relevance to determining conditions for the existence of local points on conics over $\mathbf{Q}$. The norm residue symbol gives a precise condition for a quadratic form to represent zero over $\mathbf{Q}_p$, or more generally over any local field; this is equivalent to the condition that a conic has a point over that field.

An explicit treatment of the properties and theory of the norm residue symbol can be found in Cassels [Cas78]; the Hilbert symbol is treated by Lam [Lam73].

`NormResidueSymbol(a, b, p)`

Given two rational numbers or integers a and b, and a prime p, returns 1 if the quadratic form $ax^2 + by^2 - z^2$ represents zero over $\mathbf{Q}_p$ and returns -1 otherwise.

`HilbertSymbol(a, b, p : parameters)`

`Al` MonStgElt *Default :* `"NormResidueSymbol"`

Computes the Hilbert symbol $(a, b)_p$, where a and b are elements of a number field and p is either a prime number (if $a, b \in \mathbf{Q}$) or a prime ideal. For $a, b \in \mathbf{Q}$, by default Magma uses `NormResidueSymbol` to compute the Hilbert symbol; one may insist on instead using the same algorithm as for number fields by setting the optional argument `Al` to `"Evaluate"`.

Example H119E6

In the following example we show how the norm residue symbols can be used to determine the bad primes of a conic.

```
> P2<x,y,z> := ProjectiveSpace(RationalField(), 2);
> a := 234;
> b := -33709;
> c := 127;
> C := Conic(P2, a*x^2 + b*y^2 + c*z^2);
> HasRationalPoint(C);
false
> fac := Factorization(Integers()!Discriminant(C));
> fac;
[ <2, 3>, <3, 2>, <13, 2>, <127, 1>, <2593, 1> ]
```

So we only need to test the primes 2, 3, 13, 127 and 2593. By scaling the defining polynomial of the curve $ax^2 + by^2 + cz^2 = 0$ by $-1/c$, we obtain the quadratic form $-a/cx^2 - b/cy^2 - z^2$; this means that we want to check the Hilbert symbols for the coefficient pair $(-a/c, -b/c)$.

```
> [ NormResidueSymbol(-a/c, -b/c, p[1]) : p in fac ];
[ -1, 1, 1, -1, 1 ];
```

The norm residue symbol indicates that only 2 and 127 have no local p-adic solutions, which confirms the bad primes as reported by Magma:

```
> BadPrimes(C);
[ 2, 127 ]
```

119.5 Rational Points on Conics

This section contains functions for deciding solubility and finding points on conics over the following base fields: the rationals, finite fields, number fields, and rational function fields in odd characteristic.

A point on a conic C is given as a point in the pointset $C(K)$ where K is the base ring of the conic, unless that base ring is the integers in which case the returned point belongs to the pointset $C(\mathbf{Q})$.

Over the rationals (or integers), the algorithm used to find rational points is due to D. Simon [Sim05]. Simon's algorithm works with the symmetric matrix of the defining polynomial of the conic. It computes transformations reducing the determinant of the matrix by each of the primes which divide the discriminant of the conic until it has absolute value 1. After this it does an indefinite LLL which reduces the matrix to a unimodular integral diagonal matrix, which is equivalent to the conic $x^2 + y^2 - z^2 = 0$, and then the Pythagorean parametrisation of this can be pulled back to the original conic.

The existence of a point on a conic $C/\mathbf{Q}$ is determined entirely by local solubility conditions, which are encapsulated in a solubility certificate. The algorithm of Simon works prime-by-prime, determining local solubility as it goes through its calculation of square roots modulo primes. In theory the existence of a point can be determined using Legendre symbols instead of computing these square roots. This is not implemented because the running time is dominated by the time needed to factorise the discriminant.

Over number fields the algorithm for finding points is as follows. One first reduces to diagonal form, which is done with care to ensure that one can factor the coefficients (assuming that one can factor the discriminant of the original conic). The algorithm for diagonal conics is a variant of Lagrange's method (which was once the standard method for solving diagonal conics over $\mathbf{Q}$). It involves a series of reduction steps in each of which the conic is replaced by a simpler conic. Reduction is achieved by finding a short vector in a suitable lattice sitting inside two copies of the base field. (This lattice is defined by congruence conditions arising from local solutions of the conic.)

After several reduction steps one is often able to find a solution via an easy search. In other cases, one is unable to reduce further and must call `NormEquation` to solve the reduced conic. (This is still vastly superior to calling `NormEquation` on the original conic.) The basic reduction loop is enhanced with several tricks; in particular, when it is not too large the class group of the base field may be used to reduce much further than would otherwise be feasible.

Over rational function fields the algorithm for solving conics is due to Cremona and van Hoeij [CR06]. The implementation was contributed by John Cremona and David Roberts.

Over finite fields the algorithm used for finding a point $(x : y : 1)$ on a conic is to solve for y at random values of x.

119.5.1 Finding Points

The main function is `HasRationalPoint`, which also returns a point when one exists. This (and also `RationalPoint`) works over various kinds of fields; the other functions work only

over the rationals (or integers) and finite fields.

When a point is found it is also cached for later use.

`HasRationalPoint(C)`

The conic C should be defined over the integers, rationals, a finite field, a number field, or a rational function field over a finite field of odd characteristic. The function returns `true` if and only if there exists a point on the conic C, and if so also returns one such point.

`RationalPoint(C)`

The conic C should be defined over the integers, rationals, a finite field, a number field, or a rational function field over a finite field of odd characteristic. If there exists a rational point on C over its base ring then a representative such point is returned; otherwise an error results.

`Random(C : parameters)`

`Bound`	RngIntElt	*Default :* 10^9
`Reduce`	BoolElt	*Default :* `false`

Returns a randomly selected rational point of the conic C. Such a solution is obtained by choosing random integers up to the bound specified by the parameter `Bound`, followed by evaluation at a parametrisation of C, then by point reduction if the parameter `Reduce` is set to `true`. (See Section 119.5.2 for more information about point reduction.)

`Points(C : parameters)`

`RationalPoints(C : parameters)`

`Bound`	RngIntElt

Given a conic over the rationals or a finite field, returns an indexed set of the rational points of the conic. If the curve is defined over the rationals then a positive value for the parameter `Bound` *must* be given; this function then returns those points whose integral coordinates, on the reduced Legendre model, are bounded by the value of `Bound`.

Example H119E7

We define three conics in Legendre form, having the same prime divisors in their discriminants, which differ only by a sign change (twisting by $\sqrt{-1}$) of one of the coefficients.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C1 := Conic(P2, 23*x^2 + 19*y^2 - 71*z^2);
> RationalPoints(C1 : Bound := 32);
{@ @}
> C2 := Conic(P2, 23*x^2 - 19*y^2 + 71*z^2);
> RationalPoints(C2 : Bound := 32);
{@ @}
```

```
> C3 := Conic(P2, 23*x^2 - 17*y^2 - 71*z^2);
> RationalPoints(C3 : Bound := 32);
{@ (-31 : 36 : 1), (-31 : -36 : 1), (31 : 36 : 1), (31 : -36 : 1),
(-8/3 : 7/3 : 1), (-8/3 : -7/3 : 1), (8/3 : 7/3 : 1), (8/3 : -7/3 : 1),
(-28/15 : 11/15 : 1), (-28/15 : -11/15 : 1), (28/15 : 11/15 : 1),
(28/15 : -11/15 : 1) @}
```

This naive search yields no points on the first two curves and numerous points on the third one. A guess that there are no points on either of the first two curves is proved by a call to `BadPrimes`, which finds that 19 and 23 are ramified primes for the first curve and 23 and 71 are ramified primes for the second.

```
> BadPrimes(C1);
[ 19, 23 ]
> BadPrimes(C2);
[ 23, 71 ]
> BadPrimes(C3);
[]
```

The fact that there are no ramified primes for the third curve is equivalent to a `true` return value from the function `HasRationalPoint`. As we will see in Section 119.6, an alternative approach which is guaranteed to find points on the third curve would be to construct a rational parametrisation.

119.5.2 Point Reduction

If a conic $C/\mathbf{Q}$ in Legendre form $ax^2 + by^2 + cz^2 = 0$ has a point, then Holzer's theorem tells us that there exists a point $(x : y : z)$ satisfying

$$|x| \leq \sqrt{|bc|}, \quad |y| \leq \sqrt{|ac|}, \quad |z| \leq \sqrt{|ab|},$$

with x, y, and z in $\mathbf{Z}$, or equivalently $\max(|ax^2|, |by^2|, |cz^2|) \leq |abc|$. Such a point is said to be Holzer-reduced. There exist constructive algorithms to find a Holzer-reduced point from a given one; the current MAGMA implementation uses a variant of Mordell's reduction due to Cremona [CR03].

`IsReduced(p)`

Returns `true` if and only if the projective point p on a conic in reduced Legendre form satisfies Holzer's bounds. If the curve is not a reduced Legendre model then the test is done after passing to this model.

`Reduction(p)`

Returns a Holzer-reduced point derived from p.

Example H119E8

We provide a tiny example to illustrate reduction and reduction testing on conics.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C1 := Conic(P2, x^2 + 3*x*y + 2*y^2 - 2*z^2);
> p := C1![0, 1, 1];
> IsReduced(p);
false
```

The fact that this point is not reduced is due to the size of the coefficients on the reduced Legendre model.

```
> C0, m := ReducedLegendreModel(C1);
> C0;
Conic over Rational Field defined by
X^2 - Y^2 - 2*Z^2
> m(p);
(3/2 : 1/2 : 1)
> IsReduced(m(p));
false
> Reduction(m(p));
(-1 : 1 : 0)
> Reduction(m(p)) @@ m;
(-2 : 1 : 0)
> IsReduced($1);
true
```

Example H119E9

In this example we illustrate the intrinsics which apply to finding points on conics.

```
> P2<x,y,z> := ProjectiveSpace(RationalField(), 2);
> f := 9220*x^2 + 97821*x*y + 498122*y^2 + 8007887*y*z - 3773857*z^2;
> C := Conic(P2, f);
```

We will now see that C has a reduced solution; indeed, we find two different reduced solutions. For comparison we also find a non-reduced solution.

```
> HasRationalPoint(C);
true (157010/741 : -19213/741 : 1)
> p := RationalPoint(C);
> p;
(157010/741 : -19213/741 : 1)
> IsReduced(p);
true
> q := Random(C : Reduce := true);
> q;
(-221060094911776/6522127367379 : -49731359955013/6522127367379 : 1)
> IsReduced(q);
true
```

```
> q := Random(C : Bound := 10^5);
> q;
(4457174194952129/84200926607090 : -1038901067062816/42100463303545 : 1)
> IsReduced(q);
false
> Reduction(q);
(-221060094911776/6522127367379 : -49731359955013/6522127367379 : 1)
```

To make a parametrisation of C one can use the intrinsic `Parametrization` to create a map of schemes from a line to C.

```
> phi := Parametrization(C);
> P1<u,v> := Domain(phi);
> q0 := P1![0, 1];  q1 := P1![1, 1];  q2 := P1![1, 0];
> phi(q0);
(7946776/559407 : -21332771/1118814 : 1)
> phi(q1);
(26443817/1154900 : -5909829/288725 : 1)
> phi(q2);
(157010/741 : -19213/741 : 1)
> C1, psi := ReducedLegendreModel(C);
> psi(phi(q0));
(-1793715893111/4475256 : -22556441171843029/4475256 : 1)
> p0 := Reduction($1);
> p0;
(1015829527/2964 : -59688728328467/2964 : 1)
> IsReduced(p0);
true
> p0 @@ psi;
(157010/741 : -19213/741 : 1)
```

119.6 Isomorphisms

119.6.1 Isomorphisms with Standard Models

In this section we discuss isomorphisms between heterogeneous types — isomorphisms between combinations of curves of type `Crv`, `CrvCon`, and `CrvRat`, and their parametrisations by a projective line.

The first function which we treat here is `Conic`, which is as much a constructor as an isomorphism. It takes an arbitrary genus zero curve C and uses the anti-canonical divisor $-K_C$ of degree 2 to construct the Riemann–Roch space. For a genus zero curve this is a dimension 3 space of degree 2 functions and gives a projective embedding of C in $\mathbf{P}^2$ as a conic. This provides the starting point to make any genus zero curve amenable to the powerful machinery for point finding and isomorphism classification of conics.

An isomorphism — provided that one exists — of the projective line with a conic can be described as follows. The 2-uple embedding $\phi : \mathbf{P}^1 \to \mathbf{P}^2$ defined by $(u : v) \mapsto (u^2 : uv : v^2)$

gives an isomorphism of $\mathbf{P}^1$ with the conic C_0 with defining equation $y^2 = xz$. The inverse isomorphism $C_0 \to \mathbf{P}^1$ is defined by the maps

$$(x : y : z) \mapsto (x : y) \text{ on } x \neq 0,$$
$$(x : y : z) \mapsto (y : z) \text{ on } z \neq 0,$$

respectively. Since these open sets cover the conic C_0 this defines an isomorphism and not just a birational map. In order to describe an isomorphism of a conic C_1 with $\mathbf{P}^1$ it is then necessary and sufficient to give a change of variables which maps $C_0 = \phi(\mathbf{P}^1)$ onto the conic C_1. This matrix is called the *parametrisation matrix* and is stored with C_1 once a rational point is found.

`Conic(C)`

Given a curve of genus zero, returns a conic determined by the anti-canonical embedding of C.

Example H119E10

We demonstrate the function `Conic` on the curve of Example H119E2 to find a conic model, even though we know that it admits a rational parametrisation.

```
> P2<x,y,z> := ProjectivePlane(FiniteField(71));
> C0 := Curve(P2, (x^3 + y^2*z)^2 - x^5*z);
> C1, m := Conic(C0);
> C1;
Conic over GF(71) defined by
x^2 + 70*x*z + y^2
> m : Minimal;
(x : y : z) -> (x^3*y*z : 70*x^5 + x^4*z + 70*x^2*y^2*z : x^3*y*z + y^3*z^2)
```

`ParametrizationMatrix(C)`

This function is an optimised routine for parametrising a conic C defined over $\mathbf{Z}$ or $\mathbf{Q}$. It returns a 3×3 matrix M which defines a parametrisation of C as a projective change of variables from the 2-uple embedding of a projective line in the projective plane; i.e., for a point $(x_0 : y_0 : z_0)$ on C, the point

$$(x_1 : y_1 : z_1) = (x_0 : y_0 : z_0)M$$

satisfies the equation $y_1^2 = x_1 z_1$. Note that as usual in MAGMA the action of M is on the right and, consistently, the action of scheme maps is also on the right.

Example H119E11

In this example we demonstrate that the parametrisation matrix determines the precise change of variables to transform the conic equation into the equation $y^2 = xz$. We begin with a singular plane curve C_0 of genus zero and construct a nonsingular conic model in the plane.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C0 := Curve(P2, (x^3 + y^2*z)^2 - x^5*z);
> C1, m := Conic(C0);
> C1;
Conic over Rational Field defined by
x^2 - x*z + y^2
```

The curve C_1 has obvious points, such as $(1 : 0 : 1)$, which MAGMA internally verifies without requiring an explicit user call to `HasRationalPoint`.

```
> ParametrizationMatrix(C1);
[1 0 1]
[0 1 0]
[0 0 1]
> Evaluate(DefiningPolynomial(C1), [x, y, x+z]);
-x*z + y^2
```

We note (as is standard in MAGMA) that the action of matrices, as with maps of schemes, is a right action on coordinates $(x : y : z)$.

`Parametrization(C)`

`Parametrization(C, P)`

`Parametrization(C, p)`

`Parametrization(C, p, P)`

Given a conic curve C over a general field, these functions return a parametrisation as an isomorphism of schemes $P \to C$. Here P is a copy of a projective line; it may be specified as one of the arguments or a new projective line will be created. Note that it is now required that P is given (or created) as a curve rather than as an ambient space (as used to be permitted). This allows the immediate use of pullback/push-forward functionality for the parametrisation map.

When a rational point or place is not specified as one of the arguments then the base field of C must be one of the kinds allowed in `HasRationalPoint`. If the conic has no rational points then an error results.

`ParametrizeOrdinaryCurve(C)`

`ParametrizeOrdinaryCurve(C, p)`

`ParametrizeOrdinaryCurve(C, p, I)`

`ParametrizeRationalNormalCurve(C)`

These functions are as above (see `Parametrization`), but use different algorithms.

When C is a plane curve with only ordinary singularities (see subsection 114.3.6) then a slightly different procedure is followed that relies less on the general function field machinery and tends to be faster and can produce nicer parametrisations. The variants `ParametrizeOrdinaryCurve` allow direct calls to these more specialised procedures. The I argument is the adjoint ideal of C (*loc. cit.*), which may be passed in if already computed.

The final function listed is slightly different; it applies only to rational normal curves. i.e., non-singular rational curves of degree d in ordinary d-dimensional projective space for $d \geq 1$. For the sake of speed the irreducibility of C is not checked. The function uses adjoint maps to find either a line or conic parametrisation of C: If d is odd then an isomorphism from the projective line to C is returned, and if d is even then an isomorphism from a plane conic is returned. The method uses no function field machinery and can be much faster than the general function.

Example H119E12

In this example we show how to parametrise a projective rational curve with a map from the one-dimensional projective space. First we construct a singular plane curve and verify that it has geometric genus zero.

```
> k := FiniteField(101);
> P2<x,y,z> := ProjectiveSpace(k, 2);
> f := x^7 + 3*x^3*y^2*z^2 + 5*y^4*z^3;
> C := Curve(P2, f);
> Genus(C);
0
```

In order to parametrise the curve C we need to find a nonsingular point on it, or at least a point of C over which there exists a unique degree one place; geometrically, such a point is one at which C has a cusp. To find such a point we invoke the intrinsic `RationalPoints` on C; since C is defined over a finite field this call returns an indexed set of all the points of C that are rational over its base field.
Having done the previous in the background, we demonstrate that the particular point $(2 : 33 : 1)$ is such a nonsingular rational point.

```
> p := C![2,33,1];
> p;
(2 : 33 : 1)
> IsNonsingular(C, p);
```

```
true
```

The parametrisation function takes a projective line as the third argument; this will be used as the domain of the parametrisation map.

```
> P1<u,v> := ProjectiveSpace(k, 1);
> phi := Parametrization(C, Place(p), Curve(P1));
> phi;
Mapping from: Prj: P1 to Prj: P2
with equations :
2*u^7 + 5*u^6*v + 81*u^5*v^2 + 80*u^4*v^3 + 13*u^3*v^4
33*u^7 + 88*u^6*v + 90*u^5*v^2 + 73*u^4*v^3 + 25*u^3*v^4 +
    83*u^2*v^5 + 72*u*v^6 + 24*v^7
u^7
```

Finally we confirm that the map really does parametrise the curve C. Note that the map is normalised so that the point at infinity on the projective line P_1 maps to the prescribed point p.

```
> Image(phi);
Scheme over GF(101) defined by
x^7 + 3*x^3*y^2*z^2 + 5*y^4*z^3
> Image(phi) eq C;
false
> DefiningIdeal(Image(phi)) eq DefiningIdeal(C);
true
> phi(P1![1, 0]);
(2 : 33 : 1)
```

119.7 Automorphisms

119.7.1 Automorphisms of Rational Curves

Automorphisms of the projective line $\mathbf{P}^1$ are well-known to be 3-transitive — for any three distinct points p_0, p_1, and p_∞, there exists an automorphism taking $0 = (0 : 1)$, $1 = (1 : 1)$, and $\infty = (1 : 0)$ to p_0, p_1, and p_∞. Conversely, the images of 0, 1, and ∞ uniquely characterise an automorphism. We use this characterisation of isomorphisms to define automorphisms of rational curves as bijections of three element sequences of points over the base ring.

`Automorphism(C, S, T)`

Given a rational curve C and two indexed sets $S = \{p_0, p_1, p_\infty\}$ and $T = \{q_0, q_1, q_\infty\}$, each of distinct points over the base ring of C, returns the unique automorphism of C taking p_0, p_1, p_∞ to q_0, q_1, q_∞.

119.7.2 Automorphisms of Conics

The automorphism group of a conic, and indeed the conics themselves, have a special relationship with quaternion algebras (see Lam [Lam73] or Vignéras [Vig80]). For the sake of exposition we focus on conics C/K defined by a Legendre equation

$$ax^2 + by^2 + cz^2 = 0$$

where $abc \neq 0$. We define a quaternion algebra A over K with anticommuting generators i, j, and $k = c^{-1}ij$ satisfying relations

$$i^2 = -bc, \quad j^2 = -ac, \quad k^2 = -ab,$$

and setting $I = ai$, $J = bj$, and $K = ck$. Then for any extension L/K we may identify the ambient projective pointset $\mathbf{P}^2(L)$ with the projectivisation of the trace zero part

$$A_L^0 = \{\alpha \in A_L \mid \mathrm{Tr}(\alpha) = 0\} = LI + LJ + LK$$

of the quaternion algebra $A_L = A \otimes_K L$ via $(x : y : z) \mapsto xI + yJ + zK$. Under this map, the pointsets $C(L)$ are identified with the norm zero elements

$$\mathrm{N}(xI + yJ + zK) = abc(ax^2 + by^2 + cz^2) = 0$$

in $\mathbf{P}^2(L)$. This allows us to identify the automorphism group $\mathrm{Aut}_K(C)$ with the quotient unit group A^*/K^* acting on each A_L^0 by conjugation.

In the MAGMA implementation of this correspondence, the quaternion algebra A is computed and stored as an attribute of the curve C. Automorphisms of C can be created from any invertible element of the quaternion algebra. We note that the isomorphism of two conics is equivalent to the isomorphism of their quaternion algebras, and that a rational parametrisation of a conic is equivalent to an isomorphism $A \cong \mathrm{M}_2(K)$.

We note that while the advanced features of finding rational points (and hence finding parametrisations) and determining isomorphism only exist over $\mathbf{Q}$, it is possible to represent the automorphism group and find automorphisms of the curve independently of these other problems. The current implementation works only in characteristic different from 2, as otherwise a Legendre model does not exist.

`QuaternionAlgebra(C)`

Returns the quaternion algebra in which automorphisms of the conic C can be represented.

`Automorphism(C, a)`

Given a conic C and a unit a of the quaternion algebra associated to C, returns the automorphism of C corresponding to a.

Example H119E13

In this example we demonstrate how to use the quaternion algebra of a conic to construct nontrivial automorphisms of the curve when no rational points exist over the base ring. (We use the printing level `Minimal` to produce human readable output for scheme maps.)

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C0 := Conic(P2, 2*x^2 + x*y + 5*y^2 - 37*z^2);
> HasRationalPoint(C0);
false
> BadPrimes(C0);
[ 3, 37 ]
> A := QuaternionAlgebra(C0);
> RamifiedPrimes(A);
[ 3, 37 ]
> B := Basis(MaximalOrder(A));
> B;
[ 1, 1/52*i + 1/4*j + 2/481*k, j, 1/2 + 1/148*k ]
> m1 := Automorphism(C0, A!B[2]);
> m1 : Minimal;
(x : y : z) -> (-5/8*x + 259/48*y - 37/3*z : 37/12*x + 69/8*y - 74/3*z :
    x + 7/2*y - 61/6*z)
> m2 := Automorphism(C0, A!B[3]);
> m2 : Minimal;
(x : y : z) -> (x + 1/2*y : -y : z)
> m3 := Automorphism(C0, A!B[4]);
> m3 : Minimal;
(x : y : z) -> (-x - 1/2*y : 1/5*x - 9/10*y : z)
> [ Trace(B[2]), Trace(B[3]), Trace(B[4]) ];
[ 0, 0, 1 ]
> m1 * m1 : Minimal;
(x : y : z) -> (x : y : z)
> m2 * m2 : Minimal;
(x : y : z) -> (x : y : z)
> m3 * m3 : Minimal;
(x : y : z) -> (9/10*x + 19/20*y : -19/50*x + 71/100*y : z)
```

The last example points out that *pure* quaternions τ in A_0 give rise to involutions — a reflection of the projective plane about the point defined by τ in $\mathbf{P}^2(K)$.

One of the strengths of the scheme model in Magma is the ability to work with points of a curve over any extension. Provided that we have a rational point over some extension field L/K, we are able to apply automorphisms of the curve to generate an unbounded number of new points over that extension by means of the action of the automorphism group. We demonstrate this by looking at quadratic twists of the curve to find a point over $\mathbf{Q}$, which identifies a point on the original curve over a quadratic extension.

```
> C1 := Conic(P2, 2*x^2 + x*y + 5*y^2 - z^2);
> HasRationalPoint(C1);
false
```

```
> C2 := Conic(P2, 2*x^2 + x*y + 5*y^2 - 2*z^2);
> HasRationalPoint(C2);
true
> RationalPoint(C2);
(-1 : 0 : 1)
```

This gives rise to the obvious points $(\pm 1 : 0 : \sqrt{74}/37)$ on the original curve.

```
> P<t> := PolynomialRing(RationalField());
> L<a> := NumberField(t^2 - 74);
> p := C0(L)![1, 0, a/37];
> m1(p);
(1/1880*(207*a + 3034) : 1/940*(-37*a + 2146) : 1)
> m2(p);
(1/2*a : 0 : 1)
> m3(p);
(-1/2*a : 1/10*a : 1)
```

We could alternatively have formed the base extension of the curve to the new field L and used the known point over L to find a parametrisation of the curve by the projective line. This approach demonstrates that it is possible to work with points and curve automorphisms without passing to a new curve.

119.8 Bibliography

[**Cas78**] J. W. S. Cassels. *Rational Quadratic Forms.* Academic Press, London–New York–San Francisco, 1978.

[**CR03**] J. E. Cremona and D. Rusin. Efficient solution of rational conics. *Mathematics of Computation*, 72(243):1417–1441, 2003.

[**CR06**] J. E. Cremona and D. Rusin. Solving conics over function fields. *Journal de Theorie des Nombres de Bordeaux*, 18:595–606, 2006.

[**Lam73**] T. Y. Lam. *The Algebraic Theory of Quadratic Forms.* W. A. Benjamin, Inc., Reading, MA, 1973.

[**Sim05**] Denis Simon. Solving quadratic equations using reduced unimodular quadratic forms. *Math. Comp.*, 74(251):1531–1543 (electronic), 2005.

[**Vig80**] M.-F. Vignéras. *Arithmétique des Algèbres de Quaternions*, volume 800 of *Lecture Notes in Mathematics.* Springer-Verlag, Berlin, 1980.

120 ELLIPTIC CURVES

Chapter 120
ELLIPTIC CURVES

120.1 Introduction

This chapter describes features for working with elliptic curves in MAGMA. It contains basic functionality that is applicable to curves over fairly general fields. There are separate chapters describing features that are specific to

- curves over finite fields (Chapter 121),
- curves over the rationals or number fields (Chapter 122),
- curves over univariate function fields (Chapter 123).

An elliptic curve E is the projective closure of the curve given by the generalised Weierstrass equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

The curve is specified by the sequence of coefficients $[a_1, a_2, a_3, a_4, a_6]$; the two element sequence $[a_4, a_6]$ may be used instead when $a_1 = a_2 = a_3 = 0$.

Elliptic curve functionality covers both elementary invariants of curves and arithmetic in the group of rational points, as well as higher level features for computing local invariants, heights, and Mordell–Weil groups for curves over $\mathbf{Q}$, and for point counting and the determination of the group structure over $\mathbf{F}_q$. The base ring of elliptic curves is currently restricted to fields. Curves over the rationals have special features to allow the construction of integral and minimal models, and for base change to finite fields, in acknowledgement of the integral structure over $\mathbf{Z}$ or $\mathbf{Z}_p$.

For curves over the rationals or over number fields there are routines for determining minimal models and an implementation of Tate's algorithm for determining Kodaira symbols and various local invariants. Algorithms for the computation of the Mordell–Weil group are heavily based on publications of John Cremona; see [Cre97] for details. There are also separate implementations of 2-descent for curves over number fields, and 3- and 4-descent for curves over the rationals. Additionally, several aspects of the analytic theory (including modular parametrisations and Heegner points) are implemented for curves over the rationals.

Elliptic curves are specialised forms of the more general curve and scheme types, and as such all functions which apply to these general types work on elliptic curves (although a few of them behave differently for elliptic curves). Some of these functions are described here, but not all of them — refer to chapters 112 (Schemes) and 114 (Curves) for descriptions of these functions, as well as an explanation of the relationships between points, point sets, and schemes. In particular, note that the parent of a point is a point set, and *not* the curve.

The name of the category of elliptic curves is `CrvEll`, with points of type `PtEll` lying in point sets of type `SetPtEll`. There is also the category `SchGrpEll` for subgroup schemes of elliptic curves, and a special category `SymKod` exists for the datatype of Kodaira symbols, which classify the local structure of the special fibre at p of the Néron model of an elliptic curve $E/\mathbf{Q}$.

This chapter, the first of four on elliptic curves, contains a treatment of the basics for curves over general fields: their construction, their arithmetic, and their basic properties. Specialised machinery provided for elliptic curves over finite fields is described in Chapter 121; Chapter 122 presents the wide range of techniques available for determining information about the group of rational points for curves over $\mathbf{Q}$ and over number fields, while elliptic curves over function fields are discussed in Chapter 123.

120.2 Creation Functions

120.2.1 Creation of an Elliptic Curve

An elliptic curve E may be created by specifying Weierstrass coordinates for the curve over a field K, where integer coordinates are interpreted as elements of $\mathbf{Q}$. Note that the coordinate ring K defines the base point set of E, and points defined over an extension field of K must be created in the appropriate point set.

`EllipticCurve([a, b])`

`EllipticCurve([a1, a2, a3, a4, a6])`

Given a sequence of elements of a ring K, this function creates the elliptic curve E over K defined by taking the elements of the sequence as Weierstrass coefficients. The length of the sequence must either be two, that is $s = [a, b]$, in which case E is defined by $y^2 = x^3 + ax + b$, or five, that is $s = [a_1, a_2, a_3, a_4, a_6]$, in which case E is given by $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$. Currently K must be field; if integers are given then they will be coerced into $\mathbf{Q}$. The given coefficients must define a nonsingular curve; that is, the discriminant of the curve must be nonzero.

`EllipticCurve(f)`

`EllipticCurve(f, h)`

Given univariate polynomials f and h, this function creates the elliptic curve defined by $y^2 + h(x)y = f(x)$, or by $y^2 = f(x)$ if h is not given. The polynomial f must be monic of degree 3 and h must have degree at most 1.

`EllipticCurveFromjInvariant(j)`

`EllipticCurveWithjInvariant(j)`

Given a ring element j, this function creates an elliptic curve E over K with j-invariant equal to j defined as follows: If $j = 0$ and K does not have characteristic 3 then E is defined by $y^2 + y = x^3$; if $j = 1728$ then E is defined by $y^2 = x^3 + x$ (which covers the case where $j = 0$ and K has characteristic 3); otherwise $j - 1728$ is invertible in K and E is defined by $y^2 + xy = x^3 - (36/(j - 1728))x - 1/(j - 1728)$.

Example H120E1

We create one particular elliptic curve over **Q** in three different ways:

```
> EllipticCurve([1, 0]);
Elliptic Curve defined by y^2 = x^3 + x over Rational Field
> Qx<x> := PolynomialRing(Rationals());
> EllipticCurve(x^3 + x);
Elliptic Curve defined by y^2 = x^3 + x over Rational Field
> EllipticCurveWithjInvariant(1728);
Elliptic Curve defined by y^2 = x^3 + x over Rational Field
```

`EllipticCurve(C)`

Verbose `EllModel` *Maximum :* 3

Given a scheme C describing a curve of genus 1 with an easily recognised rational point, this function returns an elliptic curve E together with a birational map from C to E. If there is no "obvious" rational point then this routine will fail. C must belong to one of the following classes:

(i) Hyperelliptic curves of genus 1 of the form

$$C : y^2 + h(x)y = f(x)$$

with f of degree 3 or 4 and h of degree at most 1. If the function x on C has a rational branch point then that point is sent to the origin on E. Otherwise, if C has a rational point at $x = \infty$ then that point is used.

(ii) Nonsingular plane curves of degree 3. If the curve is already in general Weierstrass form up to a permutation of the variables then this is recognised and used as a model for the elliptic curve. Otherwise the base field of the curve must have characteristic different from 2 and 3; in this case, the curve is tested for having a rational flex. If it has then a linear transformation suffices to get the curve into general Weierstrass form, and this is used.

(iii) Singular plane curves of degree 4 over a base field of characteristic different from 2 with a unique cusp, with the tangent cone meeting the curve only at that point. Up to linear transformation, these are curves of type

$$y^2 = f(x),$$

with f of degree 4. Such curves are brought into the standard form above. If either a rational point exists with $x = 0$ or the curve intersects the line at infinity in a rational point then that point is used to put the curve in general Weierstrass form.

`EllipticCurve(C, P)`

Verbose `EllModel` *Maximum :* 3

Given a scheme C describing a curve of genus 1 with a nonsingular rational point P, this function returns an elliptic curve E together with a birational map sending the supplied point to the origin on E.

If C is a plane curve of degree 3 over a base field having characteristic different from 2 and 3 then particular tricks are tried. If the supplied point is a flex then a linear transformation is used. Otherwise, Nagell's algorithm [Nag28], also described in [Cas91], is used.

If C is a plane curve of degree 4 over a base field of characteristic different from 2 with a unique cusp and a tangent cone that meets the curve only in that cusp, then a construction in [Cas91] is used.

In all other cases, C has to be a plane curve and a Riemann–Roch computation is used. This is potentially very expensive.

`EllipticCurve(C, pl)`

Verbose `EllModel` *Maximum :* 3

Given a plane curve C of genus 1 and a place pl of degree 1, this function returns an elliptic curve E together with a birational map from C to E.

This routine uses a (potentially expensive) Riemann–Roch computation. This is different to the routine that takes a point instead of a place, which uses special methods when the curve has degree 3 or 4.

`SupersingularEllipticCurve(K)`

Given a finite field K, this function returns a representative supersingular elliptic curve over K.

Example H120E2

In the following example Nagell's algorithm is applied.

```
> P2<X, Y, Z> := ProjectiveSpace(Rationals(), 2);
> C := Curve(P2, X^3 + Y^2*Z - X*Y*Z - Z^3);
> pt := C![0, 1, 1];
> time E1, phi1 := EllipticCurve(C, pt);
Time: 0.070
> E1;
Elliptic Curve defined by y^2 = x^3 - 5/16*x^2 + 1/32*x + 15/1024 over Rational
Field
```

Whereas this next call (given a place) is forced to do the Riemann–Roch computation.

```
> time E2, phi2 := EllipticCurve(C, Place(pt));
Time: 0.120
> E2;
Elliptic Curve defined by y^2 + 1024*y = x^3 + 16*x^2 over Rational Field
```

```
> phi1;
Mapping from: CrvPln: C to CrvEll: E1
with equations :
9/8*X^2 - 1/4*X*Y
-1/8*X^2 + 1/4*X*Y + 13/16*X*Z - 1/8*Y*Z - 1/8*Z^2
X^2 - 2*X*Y + 2*X*Z
and inverse
$.1^2 - 3/16*$.1*$.3 + 1/128*$.3^2
1/2*$.1^2 + $.1*$.2 - 15/32*$.1*$.3 + 9/256*$.3^2
$.1*$.2 + 1/8*$.1*$.3 - 1/8*$.2*$.3 - 1/64*$.3^2
> phi2;
Mapping from: CrvPln: C to CrvEll: E2
with equations :
-64*X^2*Y + 64*X^2*Z - 128*X*Y^2 + 256*X*Y*Z - 128*X*Z^2 + 64*Y^2*Z - 64*Y*Z^2
256*X^2*Y + 256*X^2*Z + 1024*X*Y*Z - 1024*X*Z^2 + 1792*Y^2*Z - 4352*Y*Z^2 +
    2048*Z^3
Y^3 - 3*Y^2*Z + 3*Y*Z^2 - Z^3
```

Example H120E3

In this example the starting curve is less clearly elliptic. Since this curve does not match any of the special forms described the Riemann–Roch computation will be used.

```
> P2<X, Y, Z> := ProjectiveSpace(Rationals(), 2);
> C := Curve(P2, X^3*Y^2 + X^3*Z^2 - Z^5);
> Genus(C);
1
> pt := C![1, 0, 1];
> E, toE := EllipticCurve(C, pt);
> E;
Elliptic Curve defined by y^2 = x^3 + 3*x^2 + 3*x over Rational Field
> toE;
Mapping from: CrvPln: C to CrvEll: E
```

120.2.2 Creation Predicates

`IsEllipticCurve([a, b])`

`IsEllipticCurve([a1, a2, a3, a4, a6])`

The function returns `true` if the given sequence of ring elements defines an elliptic curve (in other words, if the discriminant is nonzero). When true, the elliptic curve is also returned.

`IsEllipticCurve(C)`

Given a hyperelliptic curve, the function returns `true` if C has degree 3. When true, the function also returns the elliptic curve, and isomorphisms to and from it. This function is deprecated and will be removed in a later release. Instead, use `Degree(C) eq 3` and `EllipticCurve(C)`.

Example H120E4

We check a few small primes to see which ones make certain coefficients define an elliptic curve.

```
> S := [ p : p in [1..20] | IsPrime(p) ];
> for p in S do
>     ok, E := IsEllipticCurve([GF(p) | 1, 1, 0, -3, -17 ]);
>     if ok then print E; end if;
> end for;
Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 1 over GF(3)
Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 10*x + 9 over GF(13)
Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 14*x over GF(17)
```

120.2.3 Changing the Base Ring

The following general scheme functions provide a convenient way to generate a new elliptic curve similar to an old one but defined over a different field. Some care must be taken, however: If the result is not a valid elliptic curve then the functions will still succeed but the object returned will be a general curve rather than an elliptic curve.

`BaseChange(E, K)`

`BaseExtend(E, K)`

Given an elliptic curve E defined over a field k, and a field K which is an extension of k, returns an elliptic curve E' over K using the natural inclusion of k in K to map the coefficients of E into K.

`ChangeRing(E, K)`

Given an elliptic curve E defined over a field k, and a field K, returns an elliptic curve E' over K by using the standard coercion from k to K to map the coefficients of E into K. This is useful when there is no appropriate ring homomorphism between k and K (e.g., when $k = \mathbf{Q}$ and K is a finite field).

`BaseChange(E, h)`

`BaseExtend(E, h)`

Given an elliptic curve E over a field k and a ring map $h : k \to K$, returns an elliptic curve E' over K by applying h to the coefficients of E.

`BaseChange(E, n)`

`BaseExtend(E, n)`

Given an elliptic curve E defined over a finite field K, returns the base extension of E to the degree n extension of K.

Example H120E5

```
> K1 := GF(23);
> K2 := GF(23, 2);
> f := hom<K1 -> K2 | >;
> E1 := EllipticCurve([K1 | 1, 1]);
> E2 := EllipticCurve([K2 | 1, 1]);
> assert E2 eq BaseExtend(E1, K2);
> assert E2 eq BaseExtend(E1, f);
> assert E2 eq BaseExtend(E1, 2);
> // this is illegal, since K1 is not an extension of K2
> BaseExtend(E2, K1);

>> BaseExtend(E2, K1);
              ^
Runtime error in 'BaseExtend': Coercion of equations is not possible
> assert E1 eq ChangeRing(E2, K1);  // but this is OK
```

120.2.4 Alternative Models

Given an elliptic curve E, there are standard alternative models for E that may be of interest. Each of the functions in this section returns an isomorphic curve E' that is the desired model, together with the isomorphisms $E \to E'$ and $E' \to E$.

Note: The second isomorphism is now completely redundant as it is the inverse of the first; in a later release only the first will be returned.

`WeierstrassModel(E)`

Given an elliptic curve E, this function returns an isomorphic elliptic curve E' in simplified Weierstrass form $y^2 = x^3 + ax + b$. It does not apply when the base ring of E has characteristic 2 or 3 (in which case such a simplified form may not exist).

`IntegralModel(E)`

Given an elliptic curve E defined over a number field K (which may be $\mathbf{Q}$), this function returns an isomorphic elliptic curve E' defined over K with integral coefficients.

`SimplifiedModel(E)`

A simplified model of the elliptic curve E is returned. If E is defined over $\mathbf{Q}$ this has the same effect as `MinimalModel` below. Otherwise, if the characteristic of the base ring is different from 2 and 3, the simplified model agrees with the `WeierstrassModel`. For characteristics 2 and 3 the situation is more complicated; see chapter 4 of [Con99] for the definition for curves defined over finite fields.

`MinimalModel(E)`

Given an elliptic curve E defined over $\mathbf{Q}$ or a number field K with class number one, returns a global minimal model E' for E. By definition, the global minimal model E' is an integral model isomorphic to E over K such that the discriminant of E' has minimal valuation at all non-zero prime ideals π of K. (For $\mathbf{Q}$, this means that it has minimal p-adic valuation at all primes p.)

Such a global minimal model is only guaranteed to exist when the class number of K is 1 (and so one always exists when $K = \mathbf{Q}$); depending on the curve, however, such a model may exist even when this is not the case. If no such minimal model exists then a runtime error will occur.

`MinimalModel(E, p)`

`MinimalModel(E, P : parameters)`

`UseGeneratorAsUniformiser` BoolElt *Default :* `false`

Given an elliptic curve E defined over a number field K, and a prime ideal P, returns a curve isomorphic to E which is minimal at P. If K is $\mathbf{Q}$ then the ideal may be specified by the appropriate (integer) prime p.

For curves over number fields, when the parameter `UseGeneratorAsUniformiser` is set to `true` then Magma will check whether the ideal is principal; if so, an ideal generator will be used as the uniformising element. This means that at other primes the returned model will remain integral, or minimal, if the given model was.

120.2.5 Predicates on Curve Models

`IsWeierstrassModel(E)`

Returns `true` if and only if the elliptic curve E is in simplified Weierstrass form.

`IsIntegralModel(E)`

Returns `true` if and only if the elliptic curve E is an integral model.

`IsSimplifiedModel(E)`

Returns `true` if and only if the elliptic curve E is a simplified model.

`IsMinimalModel(E)`

Returns `true` if and only if the elliptic curve E is a minimal model.

`IsIntegralModel(E, P)`

Returns `true` if the defining coefficients of the elliptic curve E, a curve over a number field, have non-negative valuation at the prime ideal P, which must be an ideal of an order of the coefficient ring of E.

Example H120E6

We define an elliptic curve over the rationals and then find an integral model, a minimal model, and an integral model for the short Weierstrass form.

```
> E := EllipticCurve([1/2, 1/2, 1, 1/3, 4]);
> E;
Elliptic Curve defined by y^2 + 1/2*x*y + y = x^3 + 1/2*x^2 + 1/3*x + 4
over Rational Field
> IE := IntegralModel(E);
> IE;
Elliptic Curve defined by y^2 + 3*x*y + 216*y = x^3 + 18*x^2 + 432*x +
186624 over Rational Field
> ME := MinimalModel(IE);
> ME;
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 + 619*x + 193645
over Rational Field
> WE := WeierstrassModel(E);
> WE;
Elliptic Curve defined by y^2 = x^3 + 9909/16*x + 6201603/32 over
Rational Field
> IWE := IntegralModel(WE);
> IWE;
Elliptic Curve defined by y^2 = x^3 + 649396224*x + 208091266154496
over Rational Field
> IsIsomorphic(IWE, ME);
true
```

120.2.6 Twists of Elliptic Curves

In the following, all twists will be returned in the form of simplified models.

`QuadraticTwist(E, d)`

Given an elliptic curve E and an element d of the base ring, returns the quadratic twist by d. This is isomorphic to E if and only if either the characteristic is 2 and the trace of d is 0, or the characteristic is not 2 and d is a square. The routine does not always work in characteristic 2.

`QuadraticTwist(E)`

Given an elliptic curve E over a finite field, returns a quadratic twist; that is, a nonisomorphic curve whose trace is the negation of `Trace(E)`.

`QuadraticTwists(E)`

Given an elliptic curve E over a finite field, returns the sequence of nonisomorphic quadratic twists. The first of these curves is isomorphic to E.

`Twists(E)`

Given an elliptic curve over a finite field K, returns the sequence of all nonisomorphic elliptic curves over K which are isomorphic over an extension field. The first of these curves is isomorphic to E.

Example H120E7

In this example we compute the quadratic twists of an elliptic curve over the finite field $\mathbf{F}_{13}$ and verify that they fall in two isomorphism classes over the base field.

```
> E1 := EllipticCurve([GF(13) | 3, 1]);
> E5 := QuadraticTwist(E1, 5);
> E5;
Elliptic Curve defined by y^2 = x^3 + 10*x + 8 over GF(13)
> S := QuadraticTwists(E1);
> S;
[
    Elliptic Curve defined by y^2 = x^3 + 3*x + 1 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 12*x + 5 over GF(13)
]
> [ IsIsomorphic(E1, E) : E in S ];
[ true, false ]
> [ IsIsomorphic(E5, E) : E in S ];
[ false, true ]
```

`IsTwist(E, F)`

Returns `true` if and only if the elliptic curves E and F are isomorphic over an extension field — this function only tests the j-invariants of the curves.

`IsQuadraticTwist(E, F)`

Returns `true` if and only if the elliptic curves E and F are isomorphic over a quadratic extension field; if so, returns an element d of the base field such that F is isomorphic to `QuadraticTwist(E, d)`.

Example H120E8

In this example we take curves of j-invariant 0 and $12^3 = 12$ over the finite field $\mathbf{F}_{13}$, and compute all twists. Since the automorphism groups are nontrivial the number of twists is larger than the number of quadratic twists. By computing the numbers of points we see that the curves are indeed pairwise nonisomorphic over the base field.

```
> E3 := EllipticCurve([GF(13) | 0, 1]);
> jInvariant(E3);
0
> E4 := EllipticCurve([GF(13) | 1, 0]);
> jInvariant(E4);
12
> T3 := Twists(E3);
> T3;
[
    Elliptic Curve defined by y^2 = x^3 + 1 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 2 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 4 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 8 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 3 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 6 over GF(13)
]
> [ #E : E in T3 ];
[ 12, 19, 21, 16, 9, 7 ]
> T4 := Twists(E4);
> T4;
[
    Elliptic Curve defined by y^2 = x^3 + x over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 2*x over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 4*x over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 8*x over GF(13)
]
> [ #E : E in T4 ];
[ 20, 10, 8, 18 ]
```

Observe that exactly two of the twists are quadratic twists (as must always be the case).

```
> [ IsQuadraticTwist(E3, E) : E in T3 ];
[ true, false, false, true, false, false ]
> [ IsQuadraticTwist(E4, E) : E in T4 ];
[ true, false, true, false ]
```

`MinimalQuadraticTwist(E)`

Determine the minimal twist of the rational elliptic curve E. This is defined locally prime-by-prime, and the algorithm for finding it is based on this observation. For each odd prime p of bad reduction the algorithm iteratively replaces the curve by its twist by p if the latter has smaller discriminant.

For $p = 2$ there is no accepted definition of minimal, and the prime at infinity also plays a role. After having handled all the odd primes there are only twists by -1, 2, and -2 left to consider. The algorithm implemented in MAGMA first chooses a twist with minimal 2-valuation of the conductor; this is unique precisely when this 2-valuation is less than 5. [Note that others have chosen instead to minimise the 2-valuation of Δ.]

The prime 2 can be eliminated at this point as was done in the case of the odd primes, leaving only the consideration of twisting the curve by -1. Here the algorithm arbitrarily chooses the curve that has $(-1)^{v_2(c_6)}\text{odd}(c_6)$ congruent to 3 mod 4. The function returns as its second argument the integer d by which the curve was twisted to obtain the minimal twist.

Example H120E9

```
> E:=EllipticCurve([0, 1, 0, 135641131, 1568699095683]);
> M, tw := MinimalQuadraticTwist(E);
> M, tw;
Elliptic Curve defined by y^2 + y = x^3 + x^2 + 3*x + 5 over Rational Field
7132
> MinimalModel(QuadraticTwist(M, tw));
Elliptic Curve defined by y^2 = x^3 + x^2 + 135641131*x + 1568699095683 over
Rational Field
```

120.3 Operations on Curves

120.3.1 Elementary Invariants

`aInvariants(E)`

`Coefficients(E)`

`ElementToSequence(E)`

`Eltseq(E)`

Given an elliptic curve E, this function returns a sequence consisting of the Weierstrass coefficients of E; this is the sequence $[a_1, a_2, a_3, a_4, a_6]$ such that E is defined by $y^2z + a_1xyz + a_3yz^2 = x^3 + a_2x^2z + a_4xz^2 + a_6z^3$. Note that this function returns the five coefficients even if E was defined by a sequence $[a, b]$ of length two (the first three coefficients are zero in such a case).

`bInvariants(E)`

This function returns a sequence of length 4 containing the b-invariants of the elliptic curve E, namely $[b_2, b_4, b_6, b_8]$. In terms of the coefficients of E these are defined by

$$\begin{aligned} b_2 &= a_1^2 + 4a_2 \\ b_4 &= a_1a_3 + 2a_4 \\ b_6 &= a_3^2 + 4a_6 \\ b_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2. \end{aligned}$$

`cInvariants(E)`

This function returns a sequence of length 2 containing the c-invariants of the elliptic curve E, namely $[c_4, c_6]$. In terms of the b-invariants of E these are defined by

$$\begin{aligned} c_4 &= b_2^2 - 24b_4 \\ c_6 &= -b_2^3 + 36b_2b_4 - 216b_6. \end{aligned}$$

`Discriminant(E)`

This function returns the discriminant Δ of the elliptic curve E. In terms of the b-invariants of E it is defined by

$$\Delta = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6$$

and there is also the relationship $1728\Delta = c_4^3 - c_6^2$.

`jInvariant(E)`

Return the j-invariant of the elliptic curve E. In terms of the c-invariants and the discriminant of E it is defined by $j = c_4^3/\Delta$. Two elliptic curves defined over the same base field are isomorphic over some extension field exactly when their j-invariants are equal.

`HyperellipticPolynomials(E)`

Returns polynomials $x^3 + a_2x^2 + a_4x + a_6$ and $a_1x + a_3$, formed from the invariants of the elliptic curve E.

Example H120E10

Here are a few simple uses of the above functions.

```
> E := EllipticCurve([0, -1, 1, 1, 0]);
> E;
Elliptic Curve defined by y^2 + y = x^3 - x^2 + x over Rational Field
> aInvariants(E);
[ 0, -1, 1, 1, 0 ]
> Discriminant(E);
```

```
-131
> c4, c6 := Explode(cInvariants(E));
> jInvariant(E) eq c4^3 / Discriminant(E);
true
```

Example H120E11

By constructing a generic elliptic curve we can see that the relationships described above hold.

```
> F<a1, a2, a3, a4, a6> := FunctionField(Rationals(), 5);
> E := EllipticCurve([a1, a2, a3, a4, a6]);
> E;
Elliptic Curve defined by y^2 + a1*x*y + a3*y = x^3 + a2*x^2 + a4*x + a6
over F
> aInvariants(E);
[
    a1,
    a2,
    a3,
    a4,
    a6
]
> bInvariants(E);
[
    a1^2 + 4*a2,
    a1*a3 + 2*a4,
    a3^2 + 4*a6,
    a1^2*a6 - a1*a3*a4 + a2*a3^2 + 4*a2*a6 - a4^2
]
> b2,b4,b6,b8 := Explode(bInvariants(E));
> cInvariants(E);
[
    a1^4 + 8*a1^2*a2 - 24*a1*a3 + 16*a2^2 - 48*a4,
    -a1^6 - 12*a1^4*a2 + 36*a1^3*a3 - 48*a1^2*a2^2 + 72*a1^2*a4 +
        144*a1*a2*a3 - 64*a2^3 + 288*a2*a4 - 216*a3^2 - 864*a6
]
> c4,c6 := Explode(cInvariants(E));
> c4 eq b2^2 - 24*b4;
true
> c6 eq -b2^3 + 36*b2*b4 - 216*b6;
true
> d := Discriminant(E);
> d;
-a1^6*a6 + a1^5*a3*a4 - a1^4*a2*a3^2 - 12*a1^4*a2*a6 + a1^4*a4^2 +
    8*a1^3*a2*a3*a4 + a1^3*a3^3 + 36*a1^3*a3*a6 - 8*a1^2*a2^2*a3^2 -
    48*a1^2*a2^2*a6 + 8*a1^2*a2*a4^2 - 30*a1^2*a3^2*a4 + 72*a1^2*a4*a6 +
    16*a1*a2^2*a3*a4 + 36*a1*a2*a3^3 + 144*a1*a2*a3*a6 - 96*a1*a3*a4^2 -
    16*a2^3*a3^2 - 64*a2^3*a6 + 16*a2^2*a4^2 + 72*a2*a3^2*a4 +
```

```
    288*a2*a4*a6 - 27*a3^4 - 216*a3^2*a6 - 64*a4^3 - 432*a6^2
> d eq -b2^2*b8 - 8*b4^3 - 27*b6^2 + 9*b2*b4*b6;
true
> 1728*d eq c4^3 - c6^2;
true
```

120.3.2 Associated Structures

`Category(E)` `Type(E)`

Returns the category of elliptic curves, `CrvEll`.

`BaseRing(E)` `CoefficientRing(E)`

The base ring of the elliptic curve E; that is, the parent of its coefficients and the coefficient ring of the default point set of E.

120.3.3 Predicates on Elliptic Curves

`E eq F`

Returns `true` if and only if the elliptic curves E and F are defined over the same ring and have the same coefficients.

`E ne F`

The logical negation of `eq`.

`IsIsomorphic(E, F)`

Given two elliptic curves E and F this function returns `true` if there exists an isomorphism between E and F over the base field, and `false` otherwise. If E and F are isomorphic then the isomorphism is returned as a second value. This function requires being able to take roots in the base field.

`IsIsogenous(E, F)`

Given two elliptic curves E and F defined over the rationals or a finite field, this function returns `true` if the curves E and F are isogenous over this field and `false` otherwise. In the rational case, if the curves are isogenous then the isogeny will be returned as the second value. For finite fields the isogeny computation operates via point counting and thus no isogeny is returned.

Example H120E12

We return to the curves in the earlier quadratic twist example. By definition, these curves are not isomorphic over their base field, but are isomorphic over a quadratic extension.

```
> K := GF(13);
> E := EllipticCurve([K | 3, 1]);
> E5 := QuadraticTwist(E, 5);
> IsIsomorphic(E, E5);
false
> IsIsomorphic(BaseExtend(E, 2), BaseExtend(E5, 2));
true
```

Since they are isomorphic over an extension, their j-invariants must be the same.

```
> jInvariant(E) eq jInvariant(E5);
true
```

120.4 Polynomials

The *torsion* or *division* polynomials are the polynomials defining the subschemes of n-torsion points on the elliptic curve.

`DefiningPolynomial(E)`

Returns the homogeneous defining polynomial for the elliptic curve E.

`DivisionPolynomial(E, n)`

`DivisionPolynomial(E, n, g)`

Given an elliptic curve E and an integer n, returns the n-th division polynomial as a univariate polynomial over the base ring of E. If n is even this polynomial has multiplicity two at the nonzero 2-torsion points of the curve. The second return value is the n-th division polynomial, divided by the univariate 2-torsion polynomial if n is even. The third argument is the cofactor, equal to the univariate 2-torsion polynomial if n is even, and 1 otherwise.

If a polynomial is passed as a third argument then the division polynomial is computed efficiently modulo that polynomial.

`TwoTorsionPolynomial(E)`

Returns the multivariate 2-torsion polynomial $2y + a_1x + a_3$ of the elliptic curve E, as a bivariate polynomial.

Example H120E13

Let E be an elliptic curve over a finite field K. The following code fragment illustrates the relationship between the roots in K of the n-th division polynomial for E, and the x-coordinates of the points of n-torsion on E.

```
> K := GF(101);
> E := EllipticCurve([ K | 1, 1]);
> Roots(DivisionPolynomial(E, 5));
[ <86, 1>, <46, 1> ]
> [ P : P in RationalPoints(E) | 5*P eq E!0 ];
[ (86 : 34 : 1), (0 : 1 : 0), (46 : 25 : 1), (86 : 67 : 1), (46 : 76 : 1) ]
```

It is worth noting that even if the roots of the division polynomial lie in the field, the corresponding points may not, lying instead in a quadratic extension.

```
> Roots(DivisionPolynomial(E, 9));
[ <4, 1>, <17, 1>, <28, 1>, <34, 1>, <77, 1> ]
> Points(E, 4);
[]
> K2<w> := ext<K | 2>;
> Points(E(K2), 4);
[ (4 : 82*w + 57 : 1), (4 : 19*w + 44 : 1) ]
> Order($1[1]);
9
```

120.5 Subgroup Schemes

A subgroup scheme G of an elliptic curve E is a subscheme of E defined by a univariate polynomial ψ and closed under the group law on E. The points of G are those points of E whose x-coordinate is a root of ψ. All elliptic curves are considered to be subgroup schemes with defining polynomial $\psi = 0$.

120.5.1 Creation of Subgroup Schemes

`SubgroupScheme(G, f)`

Creates the subgroup scheme of the subgroup scheme G defined by the univariate polynomial f. No checking is done to ensure that the rational points of the result actually do form a group under the addition law. Note that G can be an elliptic curve.

`TorsionSubgroupScheme(G, n)`

Returns the subgroup scheme of n-torsion points of the subgroup scheme G. Note that G can be an elliptic curve.

120.5.2 Associated Structures

`Category(G)`

`Type(G)`

Returns the category of elliptic curve subgroup schemes, `SchGrpEll`.

`Curve(G)`

`Generic(G)`

Returns the elliptic curve E of which G is a subgroup scheme.

`BaseRing(G)`

`CoefficientRing(G)`

Returns the base ring of the subgroup scheme G; this is the same as the base ring of its curve.

`DefiningSubschemePolynomial(G)`

Returns the univariate polynomial that defines G as a subscheme of its curve.

120.5.3 Predicates on Subgroup Schemes

`G1 eq G2`

Returns `true` if and only if $G1$ and $G2$ are subgroup schemes of the same elliptic curve and are defined by equal polynomials.

`G1 ne G2`

The logical negation of `eq`.

120.5.4 Points of Subgroup Schemes

`#G`

`Order(G)`

The order of the group of rational points on the subgroup scheme G.

`FactoredOrder(G)`

The factorisation of the order of the group of rational points on the subgroup scheme G.

`Points(G)`

`RationalPoints(G)`

The indexed set of rational points of the subgroup scheme G over its base ring.

Example H120E14

We construct a curve over $\mathbf{F}_{49}$ and form several subgroup schemes from it.

```
> K<w> := GF(7, 2);
> P<t> := PolynomialRing(K);
> E := EllipticCurve([K | 1, 3]);
> G := SubgroupScheme(E, (t-4)*(t-5)*(t-6));
> G;
Subgroup scheme of E defined by x^3 + 6*x^2 + 4*x + 6
> Points(G);
{@ (0 : 1 : 0), (6 : 1 : 1), (6 : 6 : 1), (4 : 1 : 1), (4 : 6 : 1),
(5 : 0 : 1) @}
```

The points of order 3 form a further subgroup.

```
> [ Order(P) : P in $1 ];
[ 1, 3, 3, 6, 6, 2 ]
> G2 := SubgroupScheme(G, t - 6);
> G2;
Subgroup scheme of E defined by x + 1
> Points(G2);
{@ (0 : 1 : 0), (6 : 1 : 1), (6 : 6 : 1) @}
```

We can find this subgroup another way, as the intersection of the 15-torsion points of E and G.

```
> G3 := TorsionSubgroupScheme(E, 15);
> #G3;
15
> G4 := SubgroupScheme(G3, DefiningSubschemePolynomial(G));
> G4;
Subgroup scheme of E defined by x + 1
> G2 eq G4;
true
```

120.6 The Formal Group

These functions are for elliptic curves over any (exact) field.

`FormalGroupLaw(E, prec)`

This function returns a polynomial in two variables, $T_1 + T_2 + \ldots$, which expresses the formal group law associated to addition on E, up to precision `prec`. (More precisely, it contains the terms which have total degree less than or equal to `prec`.)

The formal variables T_1 and T_2 may be identified with the function $-x/y$ on E, where x and y are the standard affine coordinates on E. (Note that this function is a local parameter in a neighbourhood of O_E.)

`FormalGroupHomomorphism(phi, prec)`

This function returns the homomomorphism of formal groups associated to the isogeny `phi`, represented as a power series in one variable up to precision `prec`. As in `FormalGroupLaw`, this is in terms of the parameter $-x/y$ on each curve.

`FormalLog(E)`

`Precision` RngIntElt *Default :* 10

This function returns the formal logarithm for the elliptic curve E as a power series $f(T)$, where the parameter T is the function $-x/y$ on E. (This is the same parameter used in `FormalGroupLaw`).

The function also returns a point $P(T)$ on E with coordinates in a Laurent series ring with generator T, which again corresponds to $-x/y$. Thus $P(T)$ is a formal parametrisation of E in a neighbourhood of O_E.

120.7 Operations on Point Sets

Each elliptic curve E has associated with it a family of *point sets* of E indexed by coefficient rings. These point sets, not E, are the objects in which points lie. If K is the base ring of E and L is some extension of K then the elements of the point set $E(L)$ comprise all points lying on E whose coordinates are in L.

There is a distinguished point set $E(K)$ of E which is called the *base point set* of E. Many intrinsics (such as `#`, or `TorsionSubgroup`), strictly speaking, only make sense when applied to point sets; as a convenience, when E is passed to these functions the behaviour is the same as if the base point set $E(K)$ were passed instead. It is important to remember, however, that E and $E(K)$ are different objects and will not always behave in the same manner.

The above statements are equally valid if the elliptic curve E is replaced by some subgroup scheme G. Moreover, the types of the point sets of G and E are the same, and similarly for points. (They may be distinguished by checking the type of the scheme of which they are point sets.)

120.7.1 Creation of Point Sets

`E(L)`

`PointSet(E, L)`

Given an elliptic curve E (or a subgroup scheme thereof) and an extension L of its base ring, this function returns a point set whose elements are points on E with coefficients in L.

`E(m)`

`PointSet(E, m)`

Given an elliptic curve E (or a subgroup scheme thereof) and a map m from the base ring of E to a field L, this function returns a point set whose elements are points on E with coefficients in L. The map is retained to permit coercions between point sets.

120.7.2 Associated Structures

`Category(H)`

`Type(H)`

Given a point set H of an elliptic curve, returns the category `SetPtEll` of point sets of elliptic curves.

`Scheme(H)`

Returns the associated scheme (either an elliptic curve or a subgroup scheme) of which H is a point set.

`Curve(H)`

Returns the associated elliptic curve that contains `Scheme(H)`.

`Ring(H)`

Returns the ring that contains the coordinates of points in H.

120.7.3 Predicates on Point Sets

`H1 eq H2`

Returns whether the two point sets are equal. That is, whether the point sets have equal coefficient rings and elliptic curves (subgroup schemes).

`H1 ne H2`

The logical negation of `eq`.

Example H120E15

We create an elliptic curve E over GF(5) and then construct two associated point sets:

```
> K := GF(5);
> E := EllipticCurve([K | 1, 0]);
> H := E(K);
> H;
Set of points of E with coordinates in GF(5)
> H2 := E(GF(5, 2));
> H2;
Set of points of E with coordinates in GF(5^2)
```

We note that although these are point sets of the same curve, they are not equal because the rings are not equal.

```
> Scheme(H) eq Scheme(H2);
true
> Ring(H) eq Ring(H2);
false
> H eq H2;
```

```
false
```

Similarly, we see that a point set of a subgroup scheme is not the same object as the point set of the curve because the schemes are different.

```
> P<t> := PolynomialRing(K);
> G := SubgroupScheme(E, t - 2);
> HG := G(K);
> Scheme(HG) eq Scheme(H);
false
> Ring(HG) eq Ring(H);
true
> HG eq H;
false
```

Also note that the scheme and the parent curve of point sets of G are different:

```
> Scheme(HG);
Subgroup scheme of E defined by x + 3
> Curve(HG);
Elliptic Curve defined by y^2 = x^3 + x over GF(5)
```

120.8 Morphisms

Four types of maps between elliptic curves may be constructed: isogenies, isomorphisms, translations, and rational maps. Isogenies and isomorphisms are by far the most important and have the most functions associated to them. Isogenies are always surjective as scheme maps, even though the MAGMA parlance of a map from $E(\mathbf{Q}) \to F(\mathbf{Q})$ may seem to indicate that (for instance) the multiplication-by-two map is not surjective. There is an internal limit that the degrees of the polynomials defining an isogeny cannot be more than 10^7.

120.8.1 Creation Functions

Example H120E16

The following example gives a construction of a 2-isogeny between elliptic curves. This example follows Example III 4.5 of Silverman [Sil86], and demonstrates a parametrised family of 2-isogenies of elliptic curves together with its dual.

```
> FF := FiniteField(167);
> a := FF!2; b := FF!3; r := a^2 - 4*b;
> E1 := EllipticCurve([0, a, 0, b, 0]);
> E2 := EllipticCurve([0, -2*a, 0, r, 0]);
> _<x> := PolynomialRing(BaseRing(E1));
> Ff, f := IsogenyFromKernel(E1, x);
> Fg, g := IsogenyFromKernel(E2, x);
> b, m1 := IsIsomorphic(Ff, E2); assert b;
```

```
> b, m2 := IsIsomorphic(Fg, E1); assert b;
> // Verify that f and g are dual isogenies of degree 2
> &and[ m2(g(m1(f(P)))) eq 2*P : P in RationalPoints(E1) ];
true
> &and[ m1(f(m2(g(Q)))) eq 2*Q : Q in RationalPoints(E2) ];
true
```

`Isomorphism(E, F, [r, s, t, u])`

Given elliptic curves E and F defined over the same field K, and four elements r, s, t, u of K with $u \neq 0$, this function returns the isomorphism $E \to F$ mapping $O_E \mapsto O_F$ and mapping $(x, y) \mapsto (u^2x + r, u^3y + su^2x + t)$. This function returns an error if the values passed do not define such an isomorphism.

`Isomorphism(E, F)`

Given elliptic curves E and F defined over the same field K, this function computes and returns an isomorphism from E to F where such an isomorphism exists. The map returned will be the same as the second return value of `IsIsomorphic`.

`Automorphism(E, [r, s, t, u])`

Given an elliptic curve E defined over a field K, and four elements r, s, t, u of K with $u \neq 0$, this function returns the automorphism $E \to E$ mapping $O_E \mapsto O_E$ and mapping $(x, y) \mapsto (u^2x + r, u^3y + su^2x + t)$. This function returns an error if the values passed do not define such an automorphism.

`IsomorphismData(I)`

The sequence $[r, s, t, u]$ of elements defining the isomorphism I.

Example H120E17

We illustrate the isomorphism routines with some simple examples.

```
> K := GF(73);
> E1 := EllipticCurve([K | 3, 4, 2, 5, 1]);
> E2 := EllipticCurve([K | 8, 2, 29, 45, 28]);
> IsIsomorphic(E1, E2);
true
> m := Isomorphism(E1, E2, [3, 2, 1, 4]);
> m;
Elliptic curve isomorphism from: CrvEll: E1 to CrvEll: E2
Taking (x : y : 1) to (16*x + 3 : 64*y + 32*x + 1 : 1)
> P1 := Random(E1);
> P2 := Random(E1);
> m(P1 + P2) eq m(P1) + m(P2);
```

```
true
```

From the isomorphism data we can apply the map by hand if desired:

```
> r, s, t, u := Explode(IsomorphismData(Inverse(m)));
> P3 := E2![ 69, 64 ];
> x, y := Explode(Eltseq(P3));
> E1 ! [ u^2*x + r, u^3*y + s*u^2*x + t ];
(68 : 32 : 1)
> m($1) eq P3;
true
```

`IsIsomorphism(I)`

This function returns `true` if and only if the isogeny I has the same action as some isomorphism; if so, the isomorphism is also returned.

`IsomorphismToIsogeny(I)`

This function takes a map I of type isomorphism and returns an equivalent map with type isogeny.

Example H120E18

An example of how to use the previous two functions to transform isogenies to isomorphisms and vice versa.

```
> FF := FiniteField(23);
> E0 := EllipticCurve([FF | 1, 1]);
> E1 := EllipticCurve([FF | 3, 2]);
> b, iso := IsIsomorphic(E0, E1);
> b;
true
> iso;
Elliptic curve isomorphism from: CrvEll: E0 to CrvEll: E1
Taking (x : y : 1) to (16*x : 18*y : 1)
> isog := IsomorphismToIsogeny(iso);
> isog;
Elliptic curve isogeny from: CrvEll: E0 to CrvEll: E1
taking (x : y : 1) to (16*x : 18*y : 1)
> b, new_iso := IsIsomorphism(isog);
> b;
true
> inv := Inverse(new_iso);
> P := Random(E0);
> inv(isog(P)) eq P;
true
```

`TranslationMap(E, P)`

Given a rational point P on the elliptic curve E, this function returns the morphism $t_P : E \to E$ defined by $t_P(Q) = P + Q$ for all rational points Q of E.

`RationalMap(i, t)`

Let i be an isogeny and t be a translation map $t_P : E \to E$ where $t_P(Q) = P + Q$ for some rational point $P \in E$. This function returns the rational map $\phi : E \to F$ obtained by composing i and t (applying t first). Any rational map $E \to F$ can be represented in this form.

`TwoIsogeny(P)`

Given a 2-torsion point P of the elliptic curve E, this function returns a 2-isogeny on E with P as its kernel.

Example H120E19

One may, of course, also define maps between elliptic curves using the generic map constructors, as the following examples show.

```
> E1 := EllipticCurve([ GF(23) | 1, 1 ]);
> E2 := EllipticCurve([ GF(23, 2) | 1, 1 ]);
```

The doubling map on E_1 lifted to E_2:

```
> f := map<E1 -> E2 | P :-> 2*P>;
```

Two slightly different ways to define the negation map on E_1 lifted to E_2:

```
> f := map<E1 -> E2 | P :-> E2![ P[1], -P[2], P[3] ]>;
> f := map<E1 -> E2 | P :-> (P eq E1!0) select E2!0
>                           else E2![ P[1], -P[2], 1 ]>;
```

`IsogenyFromKernel(G)`

Let G be a subgroup scheme of an elliptic curve E. There is a separable isogeny $f : E \to E_f$ to some other elliptic curve E_f, which has kernel G. This function returns the curve E_f and the map f using Velu's formulae.

`IsogenyFromKernelFactored(G)`

Returns a sequence of isogenies whose product is the isogeny returned by the invocation `IsogenyFromKernel(G)`. These isogenies have either degree 2 or odd degree. This function was introduced because composing isogenies can be computationally expensive. The generic `Expand` function on a composition of maps can then be used if desired.

`IsogenyFromKernel(E, psi)`

Given an elliptic curve E and a univariate polynomial *psi* which defines a subgroup scheme of E, compute an isogeny using Velu's formulae as above.

`IsogenyFromKernelFactored(E, psi)`

Returns a sequence of isogenies whose product is the isogeny returned by the invocation `IsogenyFromKernel(E, psi)`. These isogenies have either degree 2 or odd degree. This function was introduced because composing isogenies can be computationally expensive. The generic `Expand` function on a composition of maps can then be used if desired.

`PushThroughIsogeny(I, v)`

`PushThroughIsogeny(I, G)`

Given an isogeny I and a subgroup G which contains the kernel of I, find the image of G under the action of I. The subgroup G may be replaced by its defining polynomial v.

`DualIsogeny(phi)`

Given an isogeny $\phi : E_1 \to E_2$, the function returns another isogeny $\phi^* : E_2 \to E_1$ such that $\phi^* \circ \phi$ is multiplication by the degree of ϕ. The result is remembered and `DualIsogeny(DualIsogeny(phi))` returns `phi`.

Example H120E20

We exhibit one way of calculating the dual of an isogeny. First we create a curve and an isogeny f with kernel equal to the full 5-torsion polynomial.

```
> E := EllipticCurve([GF(97) | 2, 3]);
> E1, f := IsogenyFromKernel(E, DivisionPolynomial(E, 5));
```

The image curve E_1 is isomorphic, but not equal, to the original curve E. We proceed to find the dual of f.

```
> deg := Degree(f);
> psi := DivisionPolynomial(E, deg);
> f1 := PushThroughIsogeny(f, psi);
> E2, g := IsogenyFromKernel(E1, f1);
> // Velu's formulae give an isomorphic curve, not the curve itself.
> IsIsomorphic(E2, E);
true
> h := Isomorphism(E2, E);
> f_dual := g*IsomorphismToIsogeny(h);
```

The latter isogeny is the dual of f, as we verify:

```
> &and [ f_dual(f(P)) eq deg*P : P in RationalPoints(E) ];
```

```
true
```

Of course, a simpler way to verify this is just to check equality:

```
> f_dual eq DualIsogeny(f);
true
```

120.8.2 Predicates on Isogenies

`IsZero(I)`

`IsConstant(I)`

Returns `true` if and only if the image of the isogeny I is the zero element of its codomain.

`I eq J`

Returns `true` if and only if the isogenies I and J are equal.

120.8.3 Structure Operations

`IsogenyMapPsi(I)`

Returns the univariate polynomial ψ used in defining the isogeny I. The roots of ψ determine the kernel of I.

`IsogenyMapPsiMulti(I)`

Returns the polynomial ψ used in defining the isogeny I as a multivariate polynomial.

`IsogenyMapPsiSquared(I)`

Returns the denominator of the fraction giving the image of the x-coordinate of a point under the isogeny I (the numerator is returned by `IsogenyMapPhi(I)`).

`IsogenyMapPhi(I)`

Returns the univariate polynomial ϕ used in defining the isogeny I.

`IsogenyMapPhiMulti(I)`

Returns the polynomial ϕ used in defining the isogeny I as a multivariate polynomial.

`IsogenyMapOmega(I)`

Returns the multivariate polynomial ω used in defining the isogeny I.

`Kernel(I)`

Returns the subgroup of the domain consisting of the elements that map to zero under the isogeny I.

`Degree(I)`

Returns the degree of the morphism I.

120.8.4 Endomorphisms

Let E be an elliptic curve defined over the field K. The set of endomorphisms $\mathrm{End}(E)$ of E to itself forms a ring under composition and addition of maps.

The endomorphism ring of E always contains a subring isomorphic to $\mathbf{Z}$. A curve E whose endomorphism ring contains additional isogenies is said to have *complex multiplication*. If E is defined over a finite field then E always has complex multiplication. The endomorphism ring of E is isomorphic to an order in either a quadratic number field or a quaternion algebra.

`MultiplicationByMMap(E, m)`

Returns the multiplication-by-m endomorphism $[m] : E \to E$ of the elliptic curve E, such that $[m](P) = m * P$.

`IdentityIsogeny(E)`

Returns the identity map $E \to E$ of the elliptic curve E as an isogeny.

`IdentityMap(E)`

Returns the identity map $E \to E$ of the elliptic curve E as an isomorphism. The defining coefficients are $[r, s, t, u] = [0, 0, 0, 1]$.

`NegationMap(E)`

Returns the isomorphism of the elliptic curve E which maps P to $-P$, for all $P \in E$.

`FrobeniusMap(E, i)`

Returns the Frobenius isogeny $(x : y : 1) \mapsto (x^{p^i} : y^{p^i} : 1)$ of an elliptic curve E defined over a finite field of characteristic p.

`FrobeniusMap(E)`

Equivalent to `FrobeniusMap(E, 1)`.

Example H120E21

We check the action of the `FrobeniusMap` function.

```
> p := 23;
> FF1 := FiniteField(p);
> FF2 := FiniteField(p, 2);
> E1 := EllipticCurve([FF1 | 1, 3]);
> E2 := BaseExtend(E1, FF2);
> frob := FrobeniusMap(E2, 1);
> #{ E1!P : P in RationalPoints(E2) | P eq frob(P) } eq #E1;
true
```

120.8.5 Automorphisms

The functions in this section deal with automorphisms in the category of elliptic curves. For an elliptic curve E these are the automorphisms of the underlying curve that also preserve the group structure (equivalently, that send the identity O_E to itself).

Warning: The behaviour of these functions depends on the type of the input curve. For a general curve in MAGMA (an object that is a `Crv` but not a `CrvEll`) over a finite field, `AutomorphismGroup` and `Automorphisms` give automorphisms in the category of curves.

`AutomorphismGroup(E)`

For an elliptic curve E over a general field K, this function determines the group of automorphisms of E that are defined over K. It returns an abstract group A (an abelian group or a polycyclic group), together with a map which sends elements of A to actual automorphisms of E (maps of schemes with domain and codomain E).

`Automorphisms(E)`

For an elliptic curve E over a general field K, this function returns a sequence containing the elements of the `AutomorphismGroup` of E.

120.9 Operations on Points

Points on an elliptic curve over a field are given in terms of projective coordinates: A point (a, b, c) is equivalent to (x, y, z) if and only if there exists an element u (in the field of definition) such that $ua = x$, $ub = y$, and $uc = z$. The equivalence class of (x, y, z) is denoted by the projective point $(x : y : z)$. At least one of the projective coordinates must be nonzero. We call the coefficients normalised if either $z = 1$, or $z = 0$ and $y = 1$.

120.9.1 Creation of Points

In this section the descriptions will refer to a point set H, which is either the H in the signature or the base point set of the elliptic curve E in the signature.

`H ! [x, y, z]` `elt< H | x, y, z >`

`E ! [x, y, z]` `elt< E | x, y, z >`

Given a point set $H = E(R)$ and coefficients x, y, z in R satisfying the equation for E, return the normalised point $P = (x : y : z)$ in H. If z is not specified it is assumed to be 1.

`H ! 0` `Id(H)` `Identity(H)`

`E ! 0` `Id(E)` `Identity(E)`

Returns the normalised identity point $(0 : 1 : 0)$ of the point set H on the elliptic curve E.

`Points(H, x)`

`Points(E, x)`

Returns the sequence of points in the pointset H on the elliptic curve E whose x-coordinate is x.

`Points(H)` `RationalPoints(H)`

`Points(E)` `RationalPoints(E)`

`Bound`	RngIntElt	*Default :* 0
`DenominatorBound`	RngIntElt	*Default :* Bound
`Denominators`	SetQ	*Default :*
`Max`	RngIntElt	*Default :*
`NPrimes`	RngIntElt	*Default :* 30

Given an elliptic curve E (or associated point set) over the rationals, a number field, or a finite field, this function returns an indexed set of points on the curve (or in the point set). When over a finite field this will contain all the rational points.

When over **Q** or a number field, a positive value for the parameter `Bound` must be given. Over **Q** this refers to the height of the x-coordinate of the points. Over number fields the algorithm searches x-coordinates in some chosen box and with some chosen denominators depending on the `Bound` (so here, too, there is loosely a linear relationship between the `Bound` and the heights of the x-coordinates searched).

The other optional parameters are only for the number field case. The denominators of x-coordinates to be searched can be specified as `Denominators` (a set of integral elements in the field) or by setting `DenominatorBound` (an integer). If an integer `Max` is specified then the routine returns as soon as this many points are found. The parameter `NPrimes` is an internal parameter that controls the number of primes used in the sieving.

The algorithm uses a sieve method; the number field case is described in Appendix A of [Bru02]. In both cases the implementation of the sieving is reasonably fast.

`PointsAtInfinity(H)`

`PointsAtInfinity(E)`

Returns the indexed set containing the identity point of the pointset H or on the elliptic curve E.

120.9.2 Creation Predicates

`IsPoint(H, S)`

`IsPoint(E, S)`

Returns `true` if the sequence of values in S are the coordinates of a point in the pointset H or on the elliptic curve E, `false` otherwise. If this is `true` then the corresponding point is returned as the second value.

`IsPoint(H, x)`
`IsPoint(E, x)`

Returns `true` if x is the x-coordinate of a point in the pointset H or on the elliptic curve E, `false` otherwise. If this is `true` then a corresponding point is returned as the second value. Note that the point at infinity of H will never be returned.

120.9.3 Access Operations

`P[i]`

Returns the i-th coefficient for an elliptic curve point P, for $1 \leq i \leq 3$.

`ElementToSequence(P)`
`Eltseq(P)`

Given a point P on an elliptic curve, this function returns a sequence of length 3 consisting of its coefficients (normalised).

120.9.4 Associated Structures

`Category(P)`
`Type(P)`

Given a point P on an elliptic curve, this function returns `PtEll`, the category of elliptic curve points.

`Parent(P)`

Given a point P on an elliptic curve, this function returns the parent point set for P.

`Scheme(P)`
`Curve(P)`

Given a point P on an elliptic curve, this function returns the corresponding scheme or elliptic curve for the parent point set of P.

120.9.5 Arithmetic

The points on an elliptic curve over a field form an abelian group, for which we use the additive notation. The identity element is the point $O = (0 : 1 : 0)$.

`-P`

Returns the additive inverse of the point P on an elliptic curve E.

`P + Q`

Returns the sum $P + Q$ of two points P and Q on the same elliptic curve.

`P +:= Q`

Given two points P and Q on the same elliptic curve, sets P equal to their sum.

`P - Q`

Returns the difference $P - Q$ of two points P and Q on the same elliptic curve.

`P -:= Q`

Given two points P and Q on the same elliptic curve, sets P equal to their difference.

`n * P`

Returns the n-th multiple of the point P on an elliptic curve.

`P *:= n`

Sets the point P equal to the n-th multiple of itself.

120.9.6 Division Points

`P / n`

Given a point P on an elliptic curve E and an integer n, this function returns a point Q on E such that $P = nQ$, if such a point exists. If no such point exists then a runtime error results.

`P /:= n`

Given a point P on an elliptic curve E and an integer n, this function sets P equal to a point Q on E such that $P = nQ$, if such a point exists. If no such point exists then a runtime error results.

`DivisionPoints(P, n)`

Given a point P on an elliptic curve E and an integer n, this function returns the sequence of all points Q on E such that $P = nQ$ holds. If there are no such points then an empty sequence is returned.

`IsDivisibleBy(P, n)`

Given a point P on an elliptic curve E and an integer n, this function returns `true` if P is n-divisible, and if so, an n-division point. Otherwise `false` is returned.

Example H120E22

```
> E := EllipticCurve([1, 0, 0, 12948, 421776]);
> P := E![ -65498304*1567, -872115836268, 1567^3 ];
> DivisionPoints(P, 3);
[ (312 : -6060 : 1), (-30 : -66 : 1), (216 : 3540 : 1) ]
```

Note that P has three three-division points — this tells us that there are three 3-torsion points in E. In fact, there are 9 points in the torsion subgroup.

```
> DivisionPoints(E!0, 9);
[ (0 : 1 : 0), (24 : 852 : 1), (24 : -876 : 1), (-24 : -300 : 1), (-24 : 324 : 1),
(600 : 14676 : 1), (600 : -15276 : 1), (132 : 2040 : 1), (132 : -2172 : 1) ]
```

Example H120E23

We construct some points in a certain elliptic curve over $\mathbf{Q}$ and try by hand to find a "smaller" set of points that generate the same group.

```
> E := EllipticCurve([0, 0, 1, -7, 6]);
> P1 := E![ 175912024457 * 278846, -41450244419357361, 278846^3 ];
> P2 := E![ -151 * 8, -1845, 8^3 ];
> P3 := E![ 36773 * 41, -7036512, 41^3 ];
> P1; P2; P3;
(175912024457/77755091716 : -41450244419357361/21681696304639736 : 1)
(-151/64 : -1845/512 : 1)
(36773/1681 : -7036512/68921 : 1)
```

Now we try small linear combinations in the hope of finding nicer looking points. We shall omit the bad guesses and just show the good ones.

```
> P1 + P2;
(777/3364 : 322977/195112 : 1)
```

Success! We replace P1 with this new point and keep going.

```
> P1 +:= P2;
> P2 + P3;
(-3 : 0 : 1)
> P2 +:= P3;
> P3 - P1;
(-1 : -4 : 1)
> P3 -:= P1;
> P1 - 2*P2;
(0 : 2 : 1)
> P1 -:= 2*P2;
> [ P1, P2, P3 ];
[ (0 : 2 : 1), (-3 : 0 : 1), (-1 : -4 : 1) ]
```

The pairwise reductions no longer help, but there is a smaller point with x-coordinate 1:

```
> IsPoint(E, 1);
```

```
true (1 : 0 : 1)
```

After a small search we find:

```
> P1 - P2 - P3;
(1 : 0 : 1)
> P2 := P1 - P2 - P3;
> [ P1, P2, P3 ];
[ (0 : 2 : 1), (1 : 0 : 1), (-1 : -4 : 1) ]
```

Using a naive definition of "small" these are the smallest possible points. (Note that there are points of smaller canonical height.) These points are in fact the generators of the Mordell–Weil group for this particular elliptic curve. Since none of the transformations changed the size of the space spanned by the points it follows that the original set of points are also generators of E. However, the reduced points form a much more convenient basis.

Example H120E24

We construct an elliptic curve over a function field (hence an elliptic surface) and form a "generic" point on it. First, we construct the function field.

```
> E := EllipticCurve([GF(97) | 1, 2]);
> K<x, y> := FunctionField(E);
```

Now we lift the curve to be over its own function field and form a generic point on E.

```
> EK := BaseChange(E, K);
> P := EK![x, y, 1];
> P;
(x : y : 1)
> 2*P;
((73*x^4 + 48*x^2 + 93*x + 73)/(x^3 + x + 2) : (85*x^6 + 37*x^4 + 5*x^3
    + 60*x^2 + 96*x + 8)/(x^6 + 2*x^4 + 4*x^3 + x^2 + 4*x + 4)*y : 1)
```

Finally, we verify that addition of the generic point defines the addition law on the curve.

```
> m2 := MultiplicationByMMap(E, 2);
> P := E![ 32, 93, 1 ];
> m2(P);
(95 : 63 : 1)
> 2*P;
(95 : 63 : 1)
```

120.9.7 Point Order

`Order(P)`

Given a point on an elliptic curve defined over **Q** or a finite field, this function computes the order of P; that is, the smallest positive integer n such that $n \cdot P = O$ on the curve. If no such positive n exists then 0 is returned to indicate infinite order. If the curve is defined over a finite field then the order of the curve will first be computed.

`FactoredOrder(P)`

Given a point on an elliptic curve defined over **Q** or over a finite field, this function returns the factorisation of the order of P. If the curve is over a finite field then on repeated applications this is generally much faster than factorising `Order(P)` because the factorisation of the order of the curve will be computed and stored. An error ensues if the curve is defined over **Q** and P has infinite order.

Example H120E25

We show a few simple operations with points on an elliptic curve over a large finite field.

```
> E := EllipticCurve([GF(NextPrime(10^12)) | 1, 1]);
> Order(E);
1000001795702
> FactoredOrder(E);
[ <2, 1>, <7, 1>, <13, 1>, <19, 1>, <31, 1>, <43, 1>, <59, 1>, <3677, 1> ]
> P := E ! [652834414164, 320964687531, 1];
> P;
(652834414164 : 320964687531 : 1)
> IsOrder(P, Order(E));
true
> FactoredOrder(P);
[ <2, 1>, <7, 1>, <13, 1>, <19, 1>, <31, 1>, <43, 1>, <59, 1>, <3677, 1> ]
> FactoredOrder(3677 * 59 * P);
[ <2, 1>, <7, 1>, <13, 1>, <19, 1>, <31, 1>, <43, 1> ]
```

120.9.8 Predicates on Points

`IsId(P)`

`IsIdentity(P)`

`IsZero(P)`

Returns `true` if and only if the point P is the identity point of its point set, `false` otherwise.

`P eq Q`

Returns `true` if and only if P and Q are points on the same elliptic curve and have the same normalised coordinates.

`P ne Q`

The logical negation of `eq`.

`P in H`

Given a point P, return `true` if and only if P is in the point set H. That is, it satisfies the equation of E and its coordinates lie in R, where $H = E(R)$.

`P in E`

Given a point P, return `true` if and only if P is on the elliptic curve E (i.e., satisfies its defining equation). Note that this is an exception to the general rule, in that P does not have to lie in the base point set of E for this to be true.

`IsOrder(P, m)`

Returns `true` if and only if the point P has order m. If you believe that you know the order of the point then this intrinsic is likely to be much faster than just calling `Order`.

`IsIntegral(P)`

Given a point P on an elliptic curve defined over $\mathbf{Q}$, this function returns `true` if and only if the coordinates of the (normalisation of) P are integers.

`IsSIntegral(P, S)`

Given a point P on an elliptic curve defined over $\mathbf{Q}$ and a sequence S of primes, this function returns `true` if and only if the coordinates of the (normalisation of) P are S-integers. That is, the denominators of $x(P)$ and $y(P)$ are only supported by primes of S.

Example H120E26

We find some integral and S-integral points on a well-known elliptic curve.

```
> E := EllipticCurve([0, 17]);
> P1 := E![ -2, 3 ];
> P2 := E![ -1, 4 ];
> S := [ a*P1 + b*P2 : a,b in [-3..3] ];
> #S;
49
> [ P : P in S | IsIntegral(P) ];
[ (43 : -282 : 1), (5234 : -378661 : 1), (2 : -5 : 1), (8 : 23 : 1),
(4 : 9 : 1), (-2 : -3 : 1), (52 : -375 : 1), (-1 : -4 : 1), (-1 : 4 : 1),
(52 : 375 : 1), (-2 : 3 : 1), (4 : -9 : 1), (8 : -23 : 1), (2 : 5 : 1),
(5234 : 378661 : 1), (43 : 282 : 1) ]
> [ P : P in S | IsSIntegral(P, [2, 3]) and not IsIntegral(P) ];
```

```
[ (1/4 : 33/8 : 1), (-8/9 : 109/27 : 1), (-206/81 : -541/729 : 1),
(137/64 : 2651/512 : 1), (137/64 : -2651/512 : 1), (-206/81 : 541/729 : 1),
(-8/9 : -109/27 : 1), (1/4 : -33/8 : 1) ]
```

120.9.9 Weil Pairing

MAGMA contains an optimised implementation of the Weil pairing on an elliptic curve. This function is used in the computation of the group structure of elliptic curves over finite fields, making the determination of the group structure efficient.

`WeilPairing(P, Q, n)`

Given n-torsion points P and Q of an elliptic curve E, this function computes the Weil pairing of P and Q.

`IsLinearlyIndependent(S, n)`

Given a sequence S of points of an elliptic curve E such that each point has order dividing n, this function returns `true` if and only if the points in S are linearly independent over $\mathbf{Z}/n\mathbf{Z}$.

`IsLinearlyIndependent(P, Q, n)`

Given points P and Q of an elliptic curve E, this function returns `true` if and only if P and Q form a basis of the n-torsion points of E.

Example H120E27

We compute the Weil pairing of two 3-torsion points on the curve $y^2 = x^3 - 3$ over $\mathbf{Q}$.

```
> E := EllipticCurve([0, -3]);
> E;
> P1, P2, P3 := Explode(ThreeTorsionPoints(E));
> P1;
(0 : 2*zeta_3 + 1 : 1)
> Parent(P1);
Set of points of E with coordinates in Cyclotomic Field
  of order 3 and degree 2
> Parent(P2);
Set of points of E with coordinates in Number Field with
  defining polynomial X^3 - 18*X - 30 over the Rational Field
```

In order to take the Weil pairing of two points we need to coerce them into the same point set. This turns out to be a point set over a number field K of degree 6.

```
> Qmu3 := Ring(Parent(P1));
> K<w> := CompositeFields(Ring(Parent(P1)), Ring(Parent(P2)))[1];
> wp := WeilPairing( E(K)!P1, E(K)!P2, 3 );
> wp;
1/975*(14*w^5 + 5*w^4 - 410*w^3 - 620*w^2 + 3964*w + 8744)
```

```
> Qmu3!wp;
zeta_3
```

120.10 Bibliography

[**Bru02**] N. R. Bruin. *Chabauty methods and covering techniques applied to generalized Fermat equations*, volume 133 of *CWI Tract*. Stichting Mathematisch Centrum Centrum voor Wiskunde en Informatica, Amsterdam, 2002. Dissertation, University of Leiden, Leiden, 1999.

[**Cas91**] J. W. S. Cassels. *Lectures on elliptic curves*, volume 24 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, 1991.

[**Con99**] I. Connell. *Elliptic Curve Handbook*. McGill University, 1999.

[**Cre97**] J. E. Cremona. *Algorithms for modular elliptic curves*. Cambridge University Press, Cambridge, second edition, 1997.

[**Nag28**] Trygve Nagell. Sur les propriétés arithmétiques des cubiques planes du premier genre. *Acta Math.*, 52:93–126, 1928.

[**Sil86**] J. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1986.

121 ELLIPTIC CURVES OVER FINITE FIELDS

Chapter 121
ELLIPTIC CURVES OVER FINITE FIELDS

This chapter describes the specialised facilities for elliptic curves defined over finite fields. Details concerning their construction, arithmetic, and basic properties may be found in Chapter 120. Most of the machinery has been constructed with Elliptic Curve Cryptography in mind.

The first major group of intrinsics relate to the determination of the order of the group of rational points of an elliptic curve over a large finite field. A variety of canonical lift algorithms are provided for characteristic 2 fields while the SEA algorithm is used for fields having characteristic greater than 2. These tools are used as the basis for functions that search for curves suitable for cryptographic applications.

A function for computing the Weil pairing forms the basis of the MOV reduction of the discrete logarithm problem (DLP) for a supersingular elliptic curve to a DLP in a finite field. A second type of attack on the DLP is based on the use of Weil descent. Tools implementing a generalisation of the GHS attack for ordinary curves in characteristic 2 are provided.

Finally, for a direct attack on the DLP for elliptic curves, a parallel collision search version of the Pollard rho algorithm is available.

121.1 Supersingular Curves

`IsSupersingular(E : parameters)`

`Proof` BoolElt *Default :* `true`

Given an elliptic curve E over a finite field, this function returns `false` if E is ordinary, otherwise proves that E is supersingular and returns `true`. If the parameter `Proof` is set to `false` then the effect of the function is the same as that of `IsProbablySupersingular`.

`SupersingularPolynomial(p)`

Given a prime p, returns the separable monic polynomial over $\mathbf{F}_p$ whose roots are precisely the j invariants of supersingular elliptic curves in characteristic p. The polynomial is computed by a formula; ignoring factors corresponding to $j = 0$ or 1728, it is the partial power-series expansion of a certain hypergeometric function reduced mod p. For p of moderate size this is a very fast method.

`IsOrdinary(E)`

Given an elliptic curve E over a finite field, this function returns `true` if the elliptic curve E is ordinary, otherwise `false` (i.e., if it is supersingular). Thus, this function is the logical negation of `IsSupersingular`.

`IsProbablySupersingular(E)`

Given an elliptic curve E over a finite field, this function returns returns `false` if the elliptic curve E is proved to be ordinary, otherwise `true`. The algorithm is nondeterministic and repeated tests are independent.

121.2 The Order of the Group of Points

121.2.1 Point Counting

MAGMA contains an efficient implementation of the Schoof–Elkies–Atkin (SEA) algorithm, with Lercier's extension to base fields of characteristic 2, for computing the number of points on an elliptic curve over a finite field. There are also implementations of the faster p-adic canonical lift methods and the p-adic deformation method in small characteristic. Calculations are performed in the smallest field over which the curve is defined, and the result is lifted to the original field.

Like all group functions on elliptic curves, these intrinsics really apply to a particular point set; the curve is identified with its base point set for the purposes of these functions. To aid exposition only the versions that take the curves are shown but an appropriate point set (over a finite field) may be used instead.

`#E`

`Order(E)`

The order of the group of K-rational points of E, where E is an elliptic curve defined over the finite field K.

`FactoredOrder(E)`

This function returns the factorisation of the order of the group of K-rational points of E, where E is an elliptic curve defined over the finite field K. This factorisation is stored on E and is reused when computing the order of points on E.

`SEA(H :` *parameters*`)`

`SEA(E :` *parameters*`)`

`Al`	MONSTGELT	*Default :* `"Default"`
`MaxSmooth`	RNGINTELT	*Default :* ∞
`AbortLevel`	RNGINTELT	*Default :* 0
`UseSEA`	BOOLELT	*Default :* `false`

This is the internal algorithm used by the point counting routines, but it can be invoked directly on an elliptic curve E or a point set H if desired. The most obvious

reason to do this is because of the parameters `AbortLevel` and `MaxSmooth`, which allow the computation to be stopped early if it can be shown not to satisfy certain conditions (such as the order of the group being prime).

Warning: Unlike the external functions which call it (such as `Order` and `FactoredOrder`), `SEA` does not store the computed group order back on the curve itself and so functions which need the group order will have to recompute it.

The parameter `Al` can be set to one of `"Enumerate"`, `"BSGS"`, or `"Default"`. Setting `Al` to `"Enumerate"` causes the group order to be found by enumeration of all points on the curve and hence is only practical for small orders. Setting `Al` to `"BSGS"` is currently equivalent to setting `Al` to `"Default"`, which uses point enumeration for suitably small finite fields, the full Schoof–Elkies–Atkin algorithm if the characteristic is not small, and canonical lift methods or deformation if it is. Currently, small characteristic means $p < 1000$. For $p < 37$ or $p \in \{41, 47, 59, 71\}$, canonical lift is used; deformation is the chosen method for other small p.

The canonical lift methods use a range of techniques to lift a modular parameter of the curve to an unramified p-adic ring. If a Gaussian Normal Basis of type ≤ 2 exists for that ring (see `HasGNB` on page 1270) then the method of Lercier and Lubicz from [LL03] is used with some local adaptations by Michael Harrison to increase the speed for fields of moderate size. Otherwise, the MSST algorithm as described in [Gau02] is used for fields up to a certain size with Harley's recursive version ([Har]) taking over for larger fields.

For $p < 10$ and $p = 13$ the modular parameter used is a generator of the function field of a genus 0 modular curve $X_0(p^r)$. The parameter gives a particularly nice modular polynomial Φ for use in the above-mentioned iterative lifting processes (see [Koh03]). In these cases Φ has a relatively simple factored form and evaluations are hand-coded for extra efficiency.

For p where $X_0(p)$ is hyperelliptic (excepting the anomalous case $p = 37$) we use a new method, developed by Michael Harrison, which lifts a *pair* of modular parameters corresponding to the x and y coordinates in a simple Weierstrass model of $X_0(p)$.

The deformation method uses code designed and implemented by Hendrik Hubrechts. It is based on ideas of Dwork, first used in the design of explicit computational algorithms by Alan Lauder, and works by computing the deformation over a family of curves of the Frobenius action on rigid cohomology by solving a certain differential equation p-adically. Details can be found in [Hub07]. The method is used for E over $\mathbf{F}_{p^n}$ with $p < 1000$ for which the canonical lift method is not used and for n greater than a smallish bound that increases slowly with p. For such p and smaller n, either enumeration or small characteristic SEA is used.

To use Schoof–Elkies–Atkin in small characteristic (with Lercier's improvements) instead of canonical lift or deformation, set `UseSEA` to `true`.

The parameter `MaxSmooth` can be used to specify a limit on the number of small primes that divide the group order. That is, we will consider the group order to be "smooth" if it is divisible by the product of small primes and this product is

larger than the value of `MaxSmooth`. Then the possible values of `AbortLevel` have the following meanings:

0 : Abort if the curve has smooth order.

1 : Abort if both the curve and its twist have smooth order.

2 : Abort if either the curve or its twist have smooth order.

If the early abort is triggered then the returned group order is 0.

One common use of these parameters is when searching for a curve with prime order. Setting `MaxSmooth := 1` will cause the early termination of the algorithm if a small factor of the order is found. If, however, the algorithm did not abort then the group order is not necessarily prime — this just means that no small prime dividing the order was encountered during the computation.

Note that the canonical lift methods give no intermediate data on the order of E, so `MaxSmooth` and `AbortLevel` have no effect when these methods are used.

Example H121E1

The first examples in characteristic 2 illustrate the increased speed when we work with fields that have GNBs available:

```
> //example with no Gaussian Normal Basis (GNB)
> K := FiniteField(2, 160); // finite field of size 2^160
> E := EllipticCurve([K!1, 0, 0, 0, K.1]);
> time #E;
1461501637330902918203686141511652467686942715904
Time: 0.100
> //example with a GNB of Type 1
> HasGNB(pAdicRing(2, 1), 162, 1);
true
> K := FiniteField(2, 162); // finite field of size 2^162
> E := EllipticCurve([K!1, 0, 0, 0, K.1]);
> time #E;
5846006549323611672814738806268278193573757976576
Time: 0.020
> //example with a GNB of Type 2
> K := FiniteField(2, 173); // finite field of size 2^173
> E := EllipticCurve([K!1, 0, 0, 0, K.1]);
> time #E;
11972621413014756705924586057649971923911742202056704
Time: 0.090
```

and here is an example in characteristic 3 (with type 1 GNB)

```
> F := FiniteField(3, 100);
> j := Random(F);
> E := EllipticCurveWithjInvariant(j);
> time #E;
515377520732011331036461505619702343613256090042
```

```
Time: 0.020
```

Finally (for small characteristic), a case with $p = 37$ where the deformation method is used.

```
> F := FiniteField(37, 30);
> j := Random(F);
> E := EllipticCurveWithjInvariant(j);
> time #E;
1111864136109938119501862966932866499557824177 67
Time: 1.580
```

Here we can see the early abort feature in action:

```
> p := NextPrime(10^9);
> p;
1000000007
> K := GF(p);
> E := EllipticCurve([K | -1, 1]);
> time SEA(E : MaxSmooth := 1);
0
Time: 0.010
> time #E;
1000009476
Time: 0.070
```

For curves this small the amount of time saved by an early abort is fairly small, but for curves over large fields the savings can be quite significant. As mentioned above, even when SEA does not abort there is no guarantee that the curve has prime order:

```
> E := EllipticCurve([K | 0, 1]);
> time SEA(E : MaxSmooth := 1);
1000000008
> IsSupersingular(E);
true
> E := EllipticCurve([K | -30, 1]);
> time SEA(E : MaxSmooth := 1);
1000035283
> Factorization($1);
[ <13, 1>, <709, 1>, <108499, 1> ]
```

In the first case the curve was supersingular and so the order was easily computed directly; thus no early abort was attempted. The latter case shows that small primes may not be looked at even when the curve is ordinary.
Next we find a curve with prime order (see also CryptographicCurve on page 3987):

```
> for k in [1..p] do
>   E := EllipticCurve([K | k, 1]);
>   n := SEA(E : MaxSmooth := 1);
>   if IsPrime(n) then
>     printf "Found curve of prime order %o for k = %o\n", n, k;
>     break;
```

```
>   end if;
> end for;
Found curve of prime order 999998501 for k = 29
> E;
Elliptic Curve defined by y^2 = x^3 + 29*x + 1 over GF(1000000007)
> #E;
999998501
```

`SetVerbose("SEA", v)`

This procedure changes the verbose printing level for the SEA algorithm which counts the number of points on an elliptic curve. Currently the legal values for v are `false`, `true`, or an integer between 0 and 5 (inclusive), where `false` is equivalent to 0 and `true` is equivalent to 1. A nonzero value gives information during the course of the algorithm, increasing in verbosity for higher values of v.

N.B. The previous name for this verbose flag, `"ECPointCount"` is now deprecated and will be removed in a future release. It should continue to function in this release, but its verbose levels are different to those of `"SEA"`.

`Order(E, r)`

Returns the order of the elliptic curve E over the extension field of K of degree r, where K is the base field of E. The order is found by calculating the order of E over K and lifting.

`Trace(E)`

`TraceOfFrobenius(E)`

Returns the trace of the Frobenius endomorphism on the elliptic curve E, equal to $q + 1 - n$ where q is the cardinality of the coefficient field and n is the order of the group of rational points of E.

`Trace(E, r)`

`TraceOfFrobenius(E, r)`

Returns the trace of the r-th power Frobenius endomorphism, where E is an elliptic curve over a finite field.

Example H121E2

The computation of the order of a curve over a finite field invokes the SEA algorithm. This data determines the number of points over all finite extensions, and is equivalent to the data for the trace of Frobenius. The values of the trace of Frobenius and group order over all extensions are therefore a trivial computation.

```
> FF<w> := GF(2^133);
> E := EllipticCurve([1, 0, 0, 0, w]);
> time #E;
10889035741470030830941160923712974513152
Time: 8.830
> FactoredOrder(E);
[ <2, 10>, <3, 2>, <150959, 1>, <654749, 1>, <1195399591723677421740186 7, 1> ]
> time TraceOfFrobenius(E);
-113173485896391746559
Time: 0.000
> time TraceOfFrobenius(E, 3);
22474974662793793921125259142328990600010427887259242963159 05
Time: 0.000
```

Example H121E3

The following code demonstrates the computation of twists of an elliptic curve over a finite field, and the relationship with the trace of Frobenius.

```
> E := EllipticCurveFromjInvariant(GF(101^2)!0);
> Twists(E);
[
   Elliptic Curve defined by y^2 = x^3 + 1 over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (61*w + 91) over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (98*w + 16) over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (9*w + 66) over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (59*w + 76) over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (87*w + 81) over GF(101^2)
]
> IsSupersingular(E);
true
```

Since E is supersingular, so are the twists of E. Hence the traces are divisible by the prime:

```
> [ TraceOfFrobenius(F) : F in Twists(E) ];
[ -202, -101, 101, 202, 101, -101 ]
> [ TraceOfFrobenius(F) : F in QuadraticTwists(E) ];
[ -202, 202 ]
```

We see that the traces come in pairs; the trace of a curve is the negative of the trace of its quadratic twist. This is not just true of the supersingular curves:

```
> E := EllipticCurveFromjInvariant(GF(101^2)!12^3);
> Twists(E);
```

```
[
   Elliptic Curve defined by y^2 = x^3 + x over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (40*w + 53)*x over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (3*w + 99)*x over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (92*w + 19)*x over GF(101^2)
]
> IsSupersingular(E);
false
> [ TraceOfFrobenius(F) : F in Twists(E) ];
[ -198, 40, 198, -40 ]
> [ TraceOfFrobenius(F) : F in QuadraticTwists(E) ];
[ -198, 198 ]
```

121.2.2 Zeta Functions

`ZetaFunction(E)`

Given an elliptic curve E over a finite field, the function returns the zeta function of E as a rational function in one variable.

Note that each of the function calls `Order(E)`, `Trace(E)`, and `ZetaFunction(E)` invoke the same call to the SEA algorithm and determine equivalent data.

Example H121E4

The zeta function $\zeta_E(t)$ of an elliptic curve E over a finite field $\mathbf{F}_q$ is a rational function whose logarithmic derivative has the power series expansion:

$$\frac{d \log \zeta_E(t)}{d \log t} = \sum_{n=1}^{\infty} |E(F_{q^n})|\, t^n$$

The following example illustrates this property of $\zeta_E(t)$.

```
> p := 11;
> E := EllipticCurve([GF(p) | 1, 2]);
> Z := ZetaFunction(E);
> Q<t> := LaurentSeriesRing(Rationals());
> t*Derivative(Log(Evaluate(Z, t))) + O(t^7);
16*t + 128*t^2 + 1264*t^3 + 14848*t^4 + 160976*t^5 + 1769600*t^6 + O(t^7)
> [ Order(E, n) : n in [1..6] ];
[ 16, 128, 1264, 14848, 160976, 1769600 ]
```

121.2.3 Cryptographic Elliptic Curve Domains

This section describes functions for generating/validating good cryptographic Elliptic Curve Domains. The basic data of such a domain is an elliptic curve E over a finite field together with a point P on E such that the order of P is a large prime and the pair (E, P) satisfies the standard security conditions for being resistant to MOV and Anomalous attacks (see [JM00]).

`CryptographicCurve(F)`

`ValidateCryptographicCurve(E, P, ordP, h)`

`OrderBound`	RngIntElt	*Default :* 2^{160}
`Proof`	BoolElt	*Default :* `true`
`UseSEA`	BoolElt	*Default :* `false`

Given a finite field F, the first function finds an Elliptic Curve Domain over F; it generates random elliptic curves over F until a suitable one is found. For each generated curve E, the order $\#E$ is calculated and a check is made as to whether sieving $\#E$ by small primes leaves a strong pseudoprime

$$p \geq \max(\texttt{OrderBound}, 4\sqrt{(\#F)}).$$

If so, the security conditions (which only depend on the order of P) are checked for p. Finally, if `Proof` is `true`, p is proven to be prime. If all of the above conditions hold true then a random point P on E of order p is found and $E, P, p, \#E/p$ are returned.

If the Schoof–Elkies–Atkins method is used for computing $\#E$ then the early abort mechanism (see SEA on page 3980) can help to shortcut the process when `OrderBound` is close to $\#F$. However, when they apply, it is generally still quicker overall to use the much faster p-adic point-counting methods for large fields. To force the use of the SEA method for point-counting set `UseSEA` to `true`. This is not recommended!

Given data of the form returned by the first function as input, the second function verifies that it is valid data for a secure EC Domain with $ordP$ satisfying the above inequality for p. If `Proof` is `true` (*resp.* `false`), $ordP$ is proven to be prime (*resp.* subjected to a strong pseudoprimality test).

`NB:` For the first function, it is required that $\#F \geq$ `OrderBound` (or $2 *$ `OrderBound` if F has characteristic 2 when all ordinary elliptic curves have even order).

`SetVerbose("ECDom", v)`

Set the verbose printing level for progress output when running the above functions. Currently the legal values for v are `true`, `false`, 0, 1, and 2 (`false` is the same as 0, and `true` is the same as 1).

Example H121E5

For a bit of extra speed, we look for curves over $F = GF(2^{196})$ where a Type 1 GNB exists for p-adic point-counting (see SEA on page 3980). We begin by looking for a curve over F with a point whose order is greater than the default `OrderBound`.

```
> SetVerbose("ECDom", 1);
> F := FiniteField(2, 196);
> E,P,ord,h := CryptographicCurve(F);
Finished! Tried 2 curves.
Total time: 1.721
> ord; h;
1255420347077336152767157884637073187358718656499286956542 1
8
```

Now we search for a curve whose order is twice a prime (the best we can do in characteristic 2!). This can be accomplished by setting `OrderBound` to half the field size.

```
> E,P,ord,h := CryptographicCurve(F : OrderBound := 2^195);
Finished! Tried 102 curves.
Total time: 22.049
> ord; h;
5021681388309344611068631538579735994185285216292918566831 9
2
> ValidateCryptographicCurve(E, P, ord, h);
Verifying primality of the order of P...
true
```

121.3 Enumeration of Points

In this section the descriptions will refer to a point set H, which is either the H in the signature or the base point set of the elliptic curve E in the signature.

`Points(E)`
`Points(H)`
`RationalPoints(E)`
`RationalPoints(H)`

Returns the set of rational points of the point set H or of the base point set of the elliptic curve E, including the point at infinity.

`Random(E)`
`Random(H)`

Returns a random rational point of the point set H or of the base point set of the elliptic curve E. Every rational point has a roughly equal chance of being selected, including the point at infinity.

121.4 Abelian Group Structure

Like all group functions on elliptic curves, these intrinsics really apply to a particular point set; the curve is identified with its base point set for the purposes of these functions. To aid exposition only the versions that take the curves are shown but an appropriate point set over a finite field may be used instead.

`AbelianGroup(E)`

`TorsionSubgroup(E)`

Computes the abelian group isomorphic to the group of rational points on the elliptic curve E over a finite field. The function returns two values: an abelian group A and a map m from A to E. The map m provides an isomorphism between the abstract group A and the group of rational points on the curve.

`Generators(H)`

`Generators(E)`

Given an elliptic curve E defined over a finite field or a pointset H of E, this function returns generators for the group of points of E. The i-th element of the sequence corresponds to the i-th generator of the group as returned by the function `AbelianGroup`.

`NumberOfGenerators(H)`

`NumberOfGenerators(E)`

`Ngens(H)`

`Ngens(E)`

The number of generators of the group of rational points of (the point set H of) the elliptic curve E; this is simply the length of the sequence returned by `Generators(E)`.

Example H121E6

A simple example of some of the above calls in action:

```
> FF<w> := GF(1048583, 2);
> E := EllipticCurve([ 1016345*w + 272405, 660960*w + 830962 ]);
> A, m := AbelianGroup(E);
> A;
Abelian Group isomorphic to Z/3 + Z/366508289334
Defined on 2 generators
Relations:
  3*A.1 = 0
  366508289334*A.2 = 0
> S := Generators(E);
> S;
[ (191389*w + 49138 : 878749*w + 1008891 : 1), (852793*w + 24192 :
```

```
    376202*w + 48552 : 1) ]
> S eq [ m(A.1), m(A.2) ];
true
```

121.5 Pairings on Elliptic Curves

Pairings on elliptic curves over finite fields have many uses, ranging from checking independence of torsion points to more practical applications in cryptography. Both destructive applications (such as the MOV attack) and constructive applications (such as pairing-based cryptography) are worth mentioning. MAGMA now contains an implementation of the most popular pairings on elliptic curves over finite fields, including the Weil pairing, the Tate pairing, and various versions of the Eta and Ate pairings.

All pairings evaluate what is now called a Miller function, denoted by $f_{n,P}$ and defined as any function on E with divisor $n(P) - ([n]P) - (n-1)\infty$, where ∞ is the point at infinity of the curve.

121.5.1 Weil Pairing

The Weil pairing is a non-degenerate bilinear map from $E[n] \times E[n]$ to μ_n, the n-th roots of unity. The Weil pairing $w_n(P,Q)$ is computed as $(-1)^n f_{n,P}(Q)/f_{n,Q}(P)$.

`WeilPairing(P, Q, n)`

Given n-torsion points P and Q on an elliptic curve E, this function computes the Weil pairing of P and Q. Both points need to be in the same point set.

121.5.2 Tate Pairing

The Tate pairing (sometimes called the Tate–Lichtenbaum pairing) is a non-degenerate bilinear map from $E[n] \times E/nE$ into $K^*/(K^*)^n$, where K is a finite field containing the n-th roots of unity. In practice one often works with the reduced version of the pairing by mapping into μ_n using the final exponentiation; i.e., powering by $\#K^*/n$. The Tate pairing $t_n(P,Q)$ is computed as $f_{n,P}(Q)$.

`TatePairing(P, Q, n)`

Given an n-torsion point P and a point Q on an elliptic curve, this function computes the Tate pairing $t_n(P,Q)$ by returning a random representative of the coset. Both points need to be in the same point set; the field K is the field of definition of this point set.

`ReducedTatePairing(P, Q, n)`

Given an n-torsion point P and a point Q on an elliptic curve, this function computes the reduced Tate pairing $e_n(P,Q) = t_n(P,Q)^{(\#K-1)/n}$. Both points need to be in the same point set; the field K is the field of definition of this point set and n should divide $\#K - 1$.

121.5.3 Eta Pairing

The Eta pairing is only defined on supersingular curves and can be considered as an optimised version of the Tate pairing. It works as follows.

Let E be a supersingular elliptic curve defined over $\mathbf{F}_q$ and $n|\#E(\mathbf{F}_q)$. The security multiplier k is the smallest positive integer such that $q^k \equiv 1 \bmod n$. Here, in the supersingular case, k divides 6. Now, $E[n] \subseteq E(\mathbf{F}_{q^k})$ is a free $\mathbf{Z}/n\mathbf{Z}$-module of rank 2 which splits into the direct sum of 2 rank one eigenspaces under the action of q-Frobenius (denoted by F in the following), with eigenvalues 1 and q (as long as $(n, k) = 1$ and $k > 1$).

The Eta pairing is defined on the subgroup of $E[n] \times E[n]$ given by the product of the two F-eigenspaces $G_1 \times G_2$, where $P \in G_1$ iff $F(P) = P$ and $Q \in G_2$ iff $F(Q) = [q]Q$. G_1 is just $E(\mathbf{F}_q)[n]$. If R is an arbitrary n-torsion point in $E(\mathbf{F}_{q^k})$, k times its G_1-component is just the F-trace of R. This can be used to find non-trivial points in G_2 (see the example below).

There are three versions of the Eta pairing: one unreduced and two reduced versions. The Eta pairing $e_T(P, Q)$ is defined as $f_{T,P}(Q)$ where $T = t - 1$ and $\#E(\mathbf{F}_q) = q + 1 - t$ or $T = q$.

As for the Tate pairing, the unreduced version takes values in $K^*/(K^*)^n$ and the reduced versions in $\mu_n(K)$ where K is $\mathbf{F}_{q^k}$.

`EtaTPairing(P, Q, n, q)`

`CheckCurve`	BoolElt	*Default :* `false`
`CheckPoints`	BoolElt	*Default :* `false`

Given a supersingular elliptic curve $E/\mathbf{F}_q$ and two points P, Q in $E[n]$ such that $P = F(P)$ and $F(Q) = [q]Q$ with F the q-power Frobenius, this function computes the Eta pairing with $T = t-1$ where $\#E(\mathbf{F}_q) = q+1-t$. The result is non-reduced and thus a random representative of a whole coset. Both points need to be in the same point set $E(\mathbf{F}_{q^k})$ and q should be the size of the base field.

`ReducedEtaTPairing(P, Q, n, q)`

`CheckCurve`	BoolElt	*Default :* `false`
`CheckPoints`	BoolElt	*Default :* `false`

The reduced version of the `EtaTPairing` function.

`EtaqPairing(P, Q, n, q)`

`CheckCurve`	BoolElt	*Default :* `false`
`CheckPoints`	BoolElt	*Default :* `false`

Given a supersingular elliptic curve $E/\mathbf{F}_q$, two points P, Q in $E[n]$ such that $P = F(P)$ and $F(Q) = [q]Q$ with F the q-power Frobenius, this function computes the Eta pairing with $T = q$. This pairing is automatically reduced: It maps directly into n-th roots of unity. Both points need to be in the same point set $E(\mathbf{F}_{q^k})$ and q should be the size of the base field.

In the previous intrinsics one can explicitly check that the curve is supersingular by setting the parameter `CheckCurve` to `true`, and that the input points are indeed in the eigenspaces of Frobenius by setting the parameter `CheckPoints` to `true`. By default MAGMA does not perform these checks for efficiency reasons.

121.5.4 Ate Pairing

The Ate pairing generalises the Eta pairing to all elliptic curves, but is defined on $G_2 \times G_1$. i.e., the arguments are swapped with respect to the Eta pairing. Like the Eta pairing there are three versions, one unreduced and two reduced. The Ate pairing $a_T(Q, P)$ with $Q \in G_2$ and $P \in G_1$ is defined as $f_{T,Q}(P)$, where $T = t - 1$ and $\#E(\mathbf{F}_q) = q + 1 - t$ or $T = q$.

`AteTPairing(Q, P, n, q)`

`CheckPoints` BOOLELT *Default :* `false`

Given two points Q, P in $E[n]$ such that $F(Q) = [q]Q$ and $P = F(P)$ with F the q-power Frobenius, this function computes the Ate pairing with $T = t - 1$ where $\#E(\mathbf{F}_q) = q + 1 - t$. The result is non-reduced and thus a random representative of a whole coset. Both points need to be in the same point set $E(\mathbf{F}_{q^k})$ and q should be the size of the base field.

`ReducedAteTPairing(Q, P, n, q)`

`CheckPoints` BOOLELT *Default :* `false`

The reduced version of the `AteTPairing` function.

`AteqPairing(P, Q, m, q)`

`CheckPoints` BOOLELT *Default :* `false`

Given two points Q, P in $E[n]$ such that $F(Q) = [q]Q$ and $P = F(P)$ with F the q-power Frobenius, this function computes the Ate pairing with $T = q$. This pairing is automatically reduced: It maps directly into n-th roots of unity. Both points need to be in the same point set $E(\mathbf{F}_{q^k})$ and q should be the size of the base field.

In the previous intrinsics one can explicitly check that the input points are indeed in the eigenspaces of Frobenius by setting the parameter `CheckPoints` to `true`. By default MAGMA does not perform these checks for efficiency reasons.

Example H121E7

In this example, we first create a so-called BN-curve, which is an elliptic curve of the form $y^2 = x^3 + b$ defined over $\mathbf{F}_p$ of prime order n and embedding degree $k = 12$. These curves can be found easily by testing for which s both $p(s) = 36s^4 + 36s^3 + 24s^2 + 6s + 1$ and $n(s) = 36s^4 + 36s^3 + 18s^2 + 6s + 1$ are prime. Finding the parameter b is equally easy by testing a few random values, since there are only 6 isogeny classes. The point $G = (1, y)$ can be used as base point.

```
> Zs<s> := PolynomialRing(Integers());
```

```
> ps := 36*s^4 + 36*s^3 + 24*s^2 + 6*s + 1;
> ns := 36*s^4 + 36*s^3 + 18*s^2 + 6*s + 1;
> z := 513235038556;  // some random start value
> repeat
>   z := z+1;
>   p := Evaluate(ps, z);
>   n := Evaluate(ns, z);
> until IsProbablePrime(p) and IsProbablePrime(n);
> p;
24978577110957807134030566063991512750990207247 23
> n;
24978577110957807134030550259379176186344734948 29
> Fp := FiniteField(p);
> b := Fp!0;
> repeat
>   repeat b := b + 1; until IsSquare(b + 1);
>   y := SquareRoot(b + 1);
>   E := EllipticCurve([Fp!0, b]);
>   G := E![1, y];
> until IsZero(n*G);
> E;
Elliptic Curve defined by y^2 = x^3 + 18 over
  GF(24978577110957807134030566063991512750990207247 23)
> #E eq n;  // just to verify
true
> t := p + 1 - n;
> t;
158046123365646454722 9895
> k := 12; // security multiplier
> (p^k - 1) mod n; // check on p, n and k
0
> Fpk := GF(p, k);
> N := Resultant(s^k - 1, s^2 - t*s + p);  // number of points over big field
> Cofac := N div n^2;
> P := E(Fpk)!G;
> Q := Cofac*Random(E(Fpk));  // Q has order n now
```

Up to this point we have constructed a BN-curve and two points of order n. As such we can test, for instance, that P and $2P$ are linearly dependent:

```
> WeilPairing(P, 2*P, n) eq 1;
true
```

and also that the Weil pairing can be obtained from 2 Tate pairing computations:

```
> WeilPairing(P, Q, n) eq (-1)^n*TatePairing(P, Q, n)/TatePairing(Q, P, n);
true
```

or test the bilinearity of the reduced Tate pairing

```
> ReducedTatePairing(P, 2*Q, n) eq ReducedTatePairing(P, Q, n)^2;
```

```
true
```

and that the corresponding test for the Tate pairing would fail since the result is a random representative of the coset:

```
> TatePairing(P, 2*Q, n) eq TatePairing(P, Q, n)^2;
false
```

Since the curve is ordinary we cannot use the Eta pairing on this curve. We can use the Ate pairing, but this is defined on $G_2 \times G_1$ and up to this point we only have a generator of G_1. To find a generator of G_2 we need to remove the component in Q that lies in G_1. This can be done easily by using the trace of the point Q: $\mathrm{Tr}(Q) = \sum_{i=0}^{k-1} F(Q)$ with F the p-power Frobenius. The trace equals k times the component of Q in G_1.

```
> TrQ := &+[ E(Fpk) ! [Frobenius(Q[1], Fp, i), Frobenius(Q[2], Fp, i)]  :
>                       i in [0..k-1]];
> Q := k*Q - TrQ;
```

At this point Q is in G_2 and we can compute the Ate pairing of Q and P. For instance, we can test compatibility of the reduced Ate pairing with the reduced Tate pairing:

```
> T := t - 1;
> L := (T^k - 1) div n;
> c := &+[ T^(k - 1 - i)*p^i : i in [0..k-1] ] mod n;
> ReducedAteTPairing(Q, P, n, p)^c eq ReducedTatePairing(Q, P, n)^L;
true
> Frobenius(AteqPairing(Q, P, n, p)^k, Fp, k - 1) eq ReducedTatePairing(Q, P, n);
true
```

To test the Eta pairing computations we will use one of the supersingular elliptic curves $y^2 + y = x^3 + x + b$ with $b = 0$ or 1 over a finite field $\mathbf{F}_{2^m}$ and m odd. These curves have security parameter $k = 4$.

```
> F2m := GF(2, 163);
> q := #F2m;
> E := EllipticCurve([F2m!0, 0, 1, 1, 1]);
> #E;
11692013098647223345629483497433542615764159168513
> IsPrime($1);
true
> n := #E;
> t := TraceOfFrobenius(E);
> P := Random(E);
> k := 4;
> F2m4 := ExtensionField<F2m, X | X^4 + X + 1>;
> N := Resultant(s^k - 1, s^2 - t*s + q);  // number of points over big field
> Cofac := N div n^2;
> P := E(F2m4) ! P;
> Q := Cofac*Random(E(F2m4));  // Q has order n now
> TrQ := &+[ E(F2m4) ! [Frobenius(Q[1], F2m, i), Frobenius(Q[2], F2m, i)]  :
>                       i in [0..k-1]];
```

```
> Q := k*Q - TrQ;
```

After this setup we can now compute the Eta pairing and test compatibility with the Tate pairing.

```
> d := GCD(k, q^k - 1);
> Frobenius(EtaqPairing(P, Q, n, q)^(k div d), F2m, k - 1) eq
>                                         ReducedTatePairing(P, Q, n);
true
```

Example H121E8

The following example demonstrates the Menezes, Okamoto, and Vanstone (MOV) reduction of the discrete logarithm on a supersingular elliptic curve to a discrete logarithm in a finite field using the Weil Pairing. The group structure of a supersingular curve E over a finite prime field $\mathbf{F}_p$ for $p > 3$ can be $\mathbf{Z}/n\mathbf{Z}$ or $\mathbf{Z}/2\mathbf{Z} \times \mathbf{Z}/(n/2)\mathbf{Z}$, where $n = p + 1$, and the group structure over a degree 2 extension is $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$. The nontrivial Weil pairing on this is the basis for this reduction.

```
> p := NextPrime(2^131);
> n := p + 1;
> n;
272225893536750770770699685945414569168 8
> Factorization(n);
[ <2, 3>, <3, 2>, <37809151880104273718152734159085356829, 1> ]
> E0 := SupersingularEllipticCurve(GF(p));
> G<x>, f := AbelianGroup(E0);
> G;
Abelian Group isomorphic to Z/272225893536750770770699685945414569168 8
Defined on 1 generator
Relations:
    272225893536750770770699685945414569168 8*x = 0
> n eq #G;
true
> P0 := f(x);
> E1 := BaseExtend(E0, GF(p^2));
> P1 := E1!P0;
> repeat
>    Q1 := Random(E1);
>    z1 := WeilPairing(P1, Q1, n);
> until Order(z1) eq n;
> IsOrder(Q1, n);
true
> r := 1234567;
> z2 := WeilPairing(r*P1, Q1, n);
> z1^r eq z2;
true
> WeilPairing(P1, r*P1, n);
1
```

121.6 Weil Descent in Characteristic Two

One approach to attacking the discrete logarithm problem for elliptic curves over finite fields is through Weil descent. Let E have base field K and let k be a subfield of K. The basic idea is to find a higher genus curve C/k along with a non-trivial homomorphism from $E(K)$ to $\mathrm{Jac}(C)(k)$ [the Jacobian of C]. This transfers the problem to one over a smaller field and Index Calculus methods over C can then be applied.

Magma now contains an implementation, due to Florian Heß, of the GHS Weil Descent for ordinary (i.e., $j(E) \neq 0$) elliptic curves in characteristic 2 (see [Gau00] or the chapter by F. Heß in [Bla05]), which constructs such a C along with the divisor map from E to C.

`WeilDescent(E, k, c)`

`HyperellipticImage` BoolElt *Default :* `true`

The curve E/K is an ordinary elliptic curve over a finite field K of characteristic 2. The argument k is a subfield of K and c is a non-zero element of K.

The function uses GHS descent to construct a plane curve C/k and a map from $E(K)$ to k-divisors of C. There are two possibilities:

i) The curve C is hyperelliptic and Magma implements the group law on its Jacobian, $\mathrm{Jac}(C)$. In this case, C will be returned as a hyperelliptic curve (type `CrvHyp`) and the map returned will actually be the divisor class homomorphism from $E(K)$ to $\mathrm{Jac}(C)(k)$ with points of $E(K)$ being identified with divisor classes in the usual way: $P \leftrightarrow (P) - (0)$.

ii) Otherwise, C is returned as a general plane curve and the map is from points of $E(K)$, considered as divisors of degree 1, to the corresponding k-divisor of C. In this case, where specialised Jacobian arithmetic is not available, it may be more convenient to work with the effective divisors rather than degree 0 divisors. The user may readily convert to the degree 0 case by getting $h(0)$ initially and then using $h((P) - (0)) = h(P) - h(0)$, where h is the divisor map.

In case ii), no reduction of the image divisor is performed by the divisor map and if $[K : k]$ is large then this divisor will have high degree and the computation may be quite slow. h is defined by divisor pullback corresponding to the function field inclusion $K(E) \hookrightarrow K(C)$ followed by the trace from K down to k.

If the user prefers to have C and h returned as in case ii), even when C is hyperelliptic, the parameter `HyperellipticImage` should be set to `false`.

The third argument c is a parameter to specify the precise data for the descent. If $r = \sqrt{(1/j(E))}$ and $b = r/c$, the function field $K(E)$ is isomorphic to the degree 2 extension of the rational function field $K(x)$ with equation

$$y^2 + y = c/x + a_2(E) + bx$$

and it is this Artin–Schreier extension that is used for the GHS descent as explained in the above references.

It is possible to work directly with function fields, if preferred, using the function `WeilDescent`. However this function produces a divisor map between function fields rather than curves and does not give the divisor class map in case i) above.

The genus of C and its degree are very dependent on the choice of c. For a description of how to choose a good c for given E and k, see sections VIII.2.4 and VIII.2.5 in the chapter by Heß cited in the introduction to this section.

We merely note here that $c \in k$ or $b \in k$ leads to a hyperelliptic C, though this may have very large genus and other choices, giving non-hyperelliptic curves, may be better.

`WeilDescentGenus(E, k, c)`

Returns the genus of the Weil descent curve C produced by `WeilDescent` for input E, k, c.

`WeilDescentDegree(E, k, c)`

Returns the degree in the second variable of the plane Weil descent curve C produced by `WeilDescent` for input E, k, c.

Example H121E9

The following example is similar to one from the article of F. Heß referred to above. E is defined over $\mathbf{F}_{2^{155}}$ and C is a hyperelliptic curve of genus 31 over $\mathbf{F}_{2^5}$.

```
> K<w> := FiniteField(2, 155);
> k := FiniteField(2, 5);
> Embed(k, K);
> b :=  w^154 + w^152 + w^150 + w^146 + w^143 + w^142 + w^141 +
>    w^139 + w^138 + w^137 + w^136 + w^134 + w^133 + w^132 + w^131 +
>    w^127 + w^125 + w^123 + w^121 + w^117 + w^116 + w^115 + w^112 +
>    w^111 + w^109 + w^108 + w^107 + w^105 + w^104 + w^102 + w^101 +
>    w^100 + w^99 + w^98 + w^97 + w^95 + w^90 + w^89 + w^88 + w^85 +
>    w^83 + w^81 + w^80 + w^79 + w^78 + w^76 + w^75 + w^73 + w^72 +
>    w^69 + w^68 + w^62 + w^61 + w^59 + w^54 + w^52 + w^47 + w^45 +
>    w^44 + w^43 + w^40 + w^39 + w^37 + w^36 + w^34 + w^32 + w^31 +
>    w^25 + w^15 + w^13 + w^10 + w^8 + w^7 + w^6;
> E := EllipticCurve([K| 1, 0, 0, b, 0]);
> C,div_map := WeilDescent(E, k, K!1);
> C;
Hyperelliptic Curve defined by y^2 + (k.1^16*x^32 + k.1^27*x^8 + k.1^2*x^4 +
    k.1^29*x^2 + k.1^30*x + k.1^20)*y = k.1^30*x^32 + k.1^10*x^8 + k.1^16*x^4 +
    k.1^12*x^2 + k.1^13*x + k.1^3 over GF(2^5)
> ptE := Points(E, w^2)[1];
> ord := Order(ptE);
> ord;
1427247692705959881058290915150890193303910 26
> ptJ := div_map(ptE); // point on Jacobian(C)
> // check that order ptJ on J = order ptE on E
> J := Jacobian(C);
> ord*ptJ eq J!0;
true
> [ (ord div p[1])*ptJ eq J!0 : p in Factorization(ord) ];
```

```
[ false, false, false, false, false, false, false, false ]
```

121.7 Discrete Logarithms

Computing discrete logarithms on elliptic curves over finite fields is considered to be a very difficult problem. The best algorithms for general elliptic curves take exponential time, and do not take much advantage of properties of the curve. Solving such problems can thus be computationally infeasible for large curves. Indeed, elliptic curves are becoming increasingly appealing for applications in cryptography.

Although hard instances of the elliptic curve discrete logarithm may be impossible to solve, MAGMA is able to efficiently solve reasonable sized instances, or instances where the large prime factor of the order of the base point is not too big. MAGMA does this as follows: The first step is to compute the factorisation of the order of the base point, which may require calling SEA if the order of the curve is not already known to MAGMA. Then it checks that the order of the base point is a multiple of the order of the other point. If this is not true then there is no solution. It next breaks the problem down to solving the discrete logarithm modulo the prime power factors of the order using the Pohlig–Hellman algorithm.

For very small primes it will try to solve this by exhaustive search. If a solution does not exist then MAGMA might determine this during the exhaustive search and abort the remaining computations. For the larger prime power factors, MAGMA uses the parallel collision search version of Pollard's rho algorithm. The implementation includes Edlyn Teske's idea of r-adding walks and Michael Wiener's and Robert Zuccherato's idea of treating the point P the same as $-P$ (for curves of the form $y^2 = x^3 + ax + b$) so as to reduce the search space by a factor of $1/\sqrt{2}$ (this idea was independently discovered by other researchers). It should be noted that MAGMA is not always able to determine the nonexistence of a solution and therefore may run forever if given very bad parameters. For this reason, the user has the option of setting a time limit on the calculation.

`Log(Q, P)`

The discrete log of P to the base Q (an integer n satisfying $n * Q = P$ where $0 \leq n <$ `Order`(Q)). The arguments Q and P must be points on the same elliptic curve, which must be defined over a finite field. The function returns -1 if it is determined that no solution exists.

`Log(Q, P, t)`

The discrete log of P to the base Q (an integer n satisfying $n * Q = P$ where $0 \leq n <$ `Order`(Q)). The arguments Q and P must be points on the same elliptic curve, which must be defined over a finite field. The argument t is a time limit in seconds on the calculation; if this limit is exceeded then the calculation is aborted. The time limit t must be a small positive integer. This function returns -1 if it is determined that no solution exists; if the calculation is aborted due to the time limit being exceeded then -2 is returned.

Example H121E10

In the example below we create a random elliptic curve over a randomly chosen 40-bit prime field. Our base point Q is chosen randomly on the curve, and the other point P is selected as a random multiple of Q. Using the `Log` function, we recover that multiplier (m) and finally we verify that the solution is correct.

```
> GetSeed();
1020 132
> K := GF(RandomPrime(40));
> E := EllipticCurve([Random(K), Random(K)]);
> E;
Elliptic Curve defined by y^2 = x^3 + 456271502613*x + 504334195864
over GF(742033232201)
> Q := Random(E);
> Q;
(174050269867 : 191768822966 : 1)
> FactoredOrder(Q);
[ <31, 1>, <7789, 1>, <3073121, 1> ]
> P := Random(Order(Q))*Q;
> P;
(495359429535 : 455525174166 : 1)
> m := Log(Q, P);
> m;
597156621194
> m*Q - P;
(0 : 1 : 0)
```

121.8 Bibliography

[**Bla05**] I. Blake, G. Serrousi and N. Smart, editor. *Advances in Elliptic Curve Cryptography*, volume 317 of *LMS LNS*. Cambridge University Press, Cambridge, 2005.

[**Gau00**] P. Gaudry, F. Heß and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. Cryptology*, 15(1):19–46, 2000.

[**Gau02**] P. Gaudry. A Comparison and a Combination of SST and AGM Algorithms for Counting Points on Elliptic Curves in Characteristic 2. In Y. Zheng, editor, *Advances in Cryptology—AsiaCrypt 2002*, volume 2501 of *LNCS*, pages 311–327. Springer-Verlag, 2002. Proc. 8th International Conference on the Theory and Applications of Cryptology and Information Security, Dec. 1–5 2002, Queenstown, New Zealand.

[**Har**] R. Harley. Web posting. Under November 2002 entry at URL:http://listserv.nodak.edu/archives/nmbrthry.html.

[**Hub07**] Hendrik Hubrechts. Quasi-quadratic elliptic curve point counting using rigid cohomology. *to appear in the Journal of Symbolic Computation*, 2007. URL:http://wis.kuleuven.be/algebra/hubrechts/.

[**JM00**] D. Johnson and A. Menezes. The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical report, Univ. Waterloo, 2000. Available at URL:http://www.cacr.math.uwaterloo.ca/.

[**Koh03**] David R. Kohel. The AGM-$X_0(N)$ Heegner Point Lifting Algorithm and Elliptic Curve Point Counting. In *Advances in Cryptology—AsiaCrypt 2003*, number 2894 in LNCS, Berlin, 2003. Springer.

[**LL03**] R. Lercier and D. Lubicz. Counting Points on Elliptic Curves over Finite Fields of Small Characteristic in Quasi Quadratic Time. In *Advances in Cryptology—EuroCrypt 2003*, volume 2656 of *LNCS*, pages 360–373. Springer, 2003.

122 ELLIPTIC CURVES OVER Q AND NUMBER FIELDS

Chapter 122
ELLIPTIC CURVES OVER Q AND NUMBER FIELDS

122.1 Introduction

This chapter deals with functionality that is specific to elliptic curves defined over the rationals or over number fields. As a general goal, these functions are developed in parallel. Naturally however, the functionality available over $\mathbf{Q}$ is far ahead of that available for general number fields, both in the range of functions and in the efficiency of implementations.

Functions declared for number fields may also be used over $\mathbf{Q}$: to do so, it is usually necessary to construct $\mathbf{Q}$ as `RationalsAsNumberField()` rather than `Rationals()`.

122.2 Curves over the Rationals

122.2.1 Local Invariants

`Conductor(E)`

The conductor of the elliptic curve E defined over $\mathbf{Q}$.

`BadPrimes(E)`

Given an elliptic curve E defined over $\mathbf{Q}$, return the sequence of primes dividing the minimal discriminant of E. These are the primes at which the minimal model for E has bad reduction; note that there may be other primes dividing the discriminant of the given model of E.

`TamagawaNumber(E, p)`

Given an elliptic curve E defined over $\mathbf{Q}$ and a prime number p, this function returns the local Tamagawa number of E at p, which is the index in $E(\mathbf{Q}_p)$ of the subgroup $E^0(\mathbf{Q}_p)$ of points with nonsingular reduction modulo p. For any prime p that is of good reduction for E, this function returns 1.

`TamagawaNumbers(E)`

Given an elliptic curve E defined over $\mathbf{Q}$, this function returns the sequence of Tamagawa numbers at each of the bad primes of E, as defined above.

`LocalInformation(E, p)`

Given an elliptic curve E defined over $\mathbf{Q}$ and a prime number p, this function returns the local information at the prime p as a tuple of the form $< P, vpd, fp, c_p, K, split >$, consisting of p, its multiplicity in the discriminant, its multiplicity in the conductor, the Tamagawa number at p, the Kodaira symbol, and finally a boolean which is false iff the curve has nonsplit multiplicative reduction. The second object returned is a local minimal model for E at p.

`LocalInformation(E)`

Given an elliptic curve E this function returns a sequence of tuples of the kind described above, for the primes dividing the discriminant of E.

`ReductionType(E, p)`

Returns a string describing the reduction type of E at p; the possibilities are "Good", "Additive", "Split multiplicative" or "Nonsplit multiplicative". These correspond to the type of singularity (if any) on the reduced curve. This function is necessary as the Kodaira symbols (see below) do not distinguish between split and unsplit multiplicative reduction.

`FrobeniusTraceDirect(E, p)`

Given a rational elliptic curve E and a prime p, this function provides an efficient way to directly compute the trace of Frobenius (without having to create $GF(p)$, coercing E into this field, etc). The argument p is **not** checked to be prime.

`TracesOfFrobenius(E, B)`

The sequence of traces of Frobenius $a_p(E)$, of the reduction mod p of E, for all primes p up to B. This function is very carefully optimised.

Example H122E1

```
> E := EllipticCurve([ 0, 1, 1, 3, 5 ]);
> T := TracesOfFrobenius(E, 100); T;
[ 0, 1, -3, -2, -5, 1, -3, -1, 5, 6, 8, -1, -6, 7, -2, 1, 14, -1, -12,
16, -16, -7, 6, 12, 2 ]
> time T := TracesOfFrobenius(E, 10^6);
Time: 5.600
```

122.2.2 Kodaira Symbols

Kodaira symbols have their own type `SymKod`. Apart from the two functions that determine symbols for elliptic curves, there is a special creation function and a comparison operator to test Kodaira symbols.

`KodairaSymbol(E, p)`

Given an elliptic curve E defined over $\mathbf{Q}$ and a prime number p, this function returns the reduction type of E modulo p in the form of a Kodaira symbol.

`KodairaSymbols(E)`

Given an elliptic curve E defined over $\mathbf{Q}$, this function returns the reduction types of E modulo the bad primes in the form of a sequence of Kodaira symbols.

`KodairaSymbol(s)`

Given a string s, return the Kodaira symbol it represents. The values of s that are allowed are: `"I0"`, `"I1"`, `"I2"`, ..., `"In"`, `"II"`, `"III"`, `"IV"`, and `"I0*"`, `"I1*"`, `"I2*"`, ..., `"In*"`, `"II*"`, `"III*"`, `"IV*"`. The dots stand for `"I`k`"` with k a positive integer. The 'generic' type `"In"` allows the matching of types `"I`n`"` for any integer $n > 0$ (and similarly for `"In*"`).

`h eq k`

Given two Kodaira symbols h and k, this function returns `true` if and only if either both are identical, or one is generic (of the form `"In"`, or `"In*"`) and the other is specific of the same type: `"In"` will compare equal with any of `"I1"`, `"I2"`, `"I3"`, etc., and `"In*"` will compare equal with any of `"I1*"`, `"I2*"`, `"I3*"`, etc. Note however that `"In"` and `"I3"` are different from the point of view of set creation.

`h ne k`

The logical negation of `eq`.

Example H122E2

We search for curves with a particular reduction type '`I0*`' in a family of curves.

```
> S := [ ];
> for n := 2 to 100 do
>    E := EllipticCurve([n, 0]);
>    for p in BadPrimes(E) do
>       if KodairaSymbol(E, p) eq KodairaSymbol("I0*") then
>          Append(~S, <p, n>);
>       end if;
>    end for;
> end for;
> S;
[ <3, 9>, <3, 18>, <5, 25>, <3, 36>, <3, 45>, <7, 49>, <5, 50>,
<3, 63>, <3, 72>, <5, 75>, <3, 90>, <7, 98>, <3, 99>, <5, 100> ]
```

122.2.3 Complex Multiplication

`HasComplexMultiplication(E)`

Given an elliptic curve E over the rationals (or a number field), the function determines whether the curve has complex multiplication or not and, if so, also returns the discriminant of the CM quadratic order. The algorithm uses fairly straightforward analytic methods, which are not suited to very high degree j-invariants with CM by orders with discriminants more than a few thousand.

122.2.4 Isogenous Curves

`IsogenousCurves(E)`

The set of curves $\mathbf{Q}$-isogenous to a rational elliptic curve E. The method used is to proceed prime-by-prime. Mazur's Theorem restricts the possibilities to a short list, and j-invariant considerations leave only p-isogenies for $p = 2, 3, 5, 7, 13$ to be considered. For $p = 2$ or 3, the method proceeds by finding the rational roots of the p-th division polynomial for the curve E. From these roots, one can then recover the isogenous curves. The question as to whether or not there is a p-isogeny is entirely a function of the j-invariant of the curve, and this idea is used for $p = 5, 7, 13$. In these cases the algorithm first takes a minimal twist of the elliptic curve, and then finds rational roots of the polynomial of degree $(p + 1)$ that comes from a fibre of $X_0(p)$. The isogenous curves of the minimal twist corresponding to these roots are then computed, and then these curves are twisted back to get the isogenous curves of E. In all cases, if the conductor is squarefree, some small values of the Frobenius traces are checked mod p to ensure the feasibility of a p-isogeny. Other similar ideas involving checking congruences are also used to try to eliminate the possibility of a p-isogeny without finding roots. Tree-based methods are used to extend the isogeny tree from (say) 2-isogenies to 4-isogenies to 8-isogenies, etc. The integer returned as a second argument corresponds to the largest degree of a cyclic isogeny between two curves in the isogeny class. The ordering of the list of curves returned is well-defined; the first curve is always the curve of minimal Faltings height, and the heights generically increase upon going down the list. However, when there are isogenies of two different prime degrees, a different ordering is used. In the case where there is a 4-isogeny (or a certain type of 9-isogeny), there is an arbitrary choice made on the bottom tree leaves in order to make the ordering consistent.

`FaltingsHeight(E)`

`Precision` RngIntElt *Default :*

Compute the Faltings height of a rational elliptic curve. This is given by $-\frac{1}{2}\log \mathrm{Vol}(E)$ where $\mathrm{Vol}(E)$ is the volume of the fundamental parallelogram.

Example H122E3

First we compute isogenous curves using `DivisionPolynomial`; in this special case we obtain the entire isogeny class by considering only 3-isogenies directly from E.

```
> E:=EllipticCurve([1,-1,1,1,-1]);
> F:=Factorization(DivisionPolynomial(E,3));
> I:=IsogenyFromKernel(E,F[1][1]); I;
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 29*x - 53 over Rational
Field
> I:=IsogenyFromKernel(E,F[2][1]); I;
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 14*x + 29 over Rational
Field
```

Alternatively, we can get the `IsogenousCurves` directly. Note that `IsogenyFromKernel` also returns the map between the isogenous curves as a second argument.

```
> IsogenousCurves(E);
[
    Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 + x - 1 over Rational
    Field,
    Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 29*x - 53 over
    Rational Field,
    Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 14*x + 29 over
    Rational Field
]
9
```

122.2.5 Mordell–Weil Group

The Mordell–Weil theorem states that for an elliptic curve E defined over $\mathbf{Q}$, the set $E(\mathbf{Q})$ of points on E with $\mathbf{Q}$-rational coordinates forms a finitely generated abelian group, known as the *Mordell–Weil group*.

To compute Mordell–Weil groups in MAGMA, there are several ways to proceed:

(i) A large array of relevant tools (various descents, and other techniques) are described elsewhere in this chapter, and may be applied "by hand".

(ii) The routine `MordellWeilShaInformation` automates the process: it applies all relevant tools in a suitable order.

(iii) The older routines `Rank` and `MordellWeilGroup` use an (independent) implementation of an algorithm similar to the famous program `mwrank` by John Cremona. These routines do 2-descent or 2-isogeny descent, but use no additional techniques.

Warning: No algorithm has been implemented that is guaranteed to determine the Mordell-Weil rank! In some cases the functions `Rank` and `MordellWeilGroup` return a lower bound which is not known to be the true rank (or group). In cases where the rank is not proven, a warning message is printed the first time either `Rank` or `MordellWeilGroup` is called for a particular elliptic curve; on subsequent calls, the unproven result will be

returned without any warning. It is recommended to use `RankBounds` instead of `Rank`, to avoid confusion.

`MordellWeilShaInformation(E: parameters)`

`DescentInformation(E: parameters)`

`RankOnly`	BoolElt	*Default :* `false`
`ShaInfo`	BoolElt	*Default :* `false`
`Silent`	BoolElt	*Default :* `false`

This is a special function which uses all relevant Magma machinery to obtain as much information as possible about the Mordell-Weil group and the Tate-Shafarevich group of the given elliptic curve E over $\mathbf{Q}$. The tools used include 2-descent, 4-descent, Cassels-Tate pairings, 3-descent, and also analytic routines (rank and Heegner points) when the conductor of E is not too large. The information is progressively refined as the tools are applied (in order of their estimated cost), and a summary is printed at each stage. At the end, the collected information is returned in three sequences.

The first sequence returned contains lower and upper bounds on the Mordell-Weil rank of $E(\mathbf{Q})$, and the second is the sequence of independent generators of the Mordell-Weil group (modulo the torsion subgroup) that have been found. The third sequence returned contains the information obtained about the Tate-Shafarevich group $Sha(E)$: letting r_n denote the largest integer such that $\mathbf{Z}/n\mathbf{Z}$ is contained in $Sha(E)$, the tuple `<n, [l,u]>` would indicate that the computations prove $l \leq r_n \leq u$.

By default, the routine attempts to determine the rank and generators, and returns whenever it succeeds in determining them. When `RankOnly` is set to true, it returns as soon as the Mordell-Weil rank has been determined. When `ShaInfo` is set to true, it additionally probes the structure of $Sha(E)$ using all available tools.

Example H122E4

Conductor 389 is the smallest conductor of a curve with rank 2:

```
> E := EllipticCurve("389a1");
> time rank, gens, sha := MordellWeilShaInformation(E);
Torsion Subgroup is trivial
Analytic rank = 2
The 2-Selmer group has rank 2
Found a point of infinite order.
Found 2 independent points.
After 2-descent:
    2 <= Rank(E) <= 2
    Sha(E)[2] is trivial
(Searched up to height 100 on the 2-coverings.)
```

```
Time: 0.280
```

We now take a curve of conductor 571, which has rank 0 and nontrivial Tate-Shafarevich group:

```
> E := EllipticCurve("571a1");
> time rank, gens, sha :=MordellWeilShaInformation(E);
Torsion Subgroup is trivial
Analytic rank = 0
     ==> Rank(E) = 0
Time: 0.010
```

We must specify that we want information about the Tate-Shafarevich group:

```
> time rank, gens, sha :=MordellWeilShaInformation(E : ShaInfo);
Torsion Subgroup is trivial
Analytic rank = 0
     ==> Rank(E) = 0
The 2-Selmer group has rank 2
After 2-descent:
    0 <= Rank(E) <= 0
    (Z/2)^2 <= Sha(E)[2] <= (Z/2)^2
(Searched up to height 10000 on the 2-coverings.)
The Cassels-Tate pairing on Sel(2,E)/E[2] is
[0 1]
[1 0]
After using Cassels-Tate:
    0 <= Rank(E) <= 0
    (Z/2)^2 <= Sha(E)[4] <= (Z/2)^2
The 3-Selmer group has rank 0
After 3-descent:
    0 <= Rank(E) <= 0
    (Z/2)^2 <= Sha(E)[12] <= (Z/2)^2
Time: 0.840
```

`TorsionSubgroup(E)`

Given an elliptic curve E defined over $\mathbf{Q}$, this function returns an abelian group A isomorphic to the torsion subgroup of the Mordell–Weil group, and a map from this abstract group A to the elliptic curve providing the isomorphism. By a theorem of Mazur, A is either C_k (for k in $\{1..10\}$ or 12) or $C_2 \times C_{2k}$ (for k in $\{1..4\}$). If there are two generators then the first generator returned in this case will have order 2.

`Rank(E: parameters)`

`MordellWeilRank(E: parameters)`

`Bound` RngIntElt *Default :* 150

Given an elliptic curve E defined over $\mathbf{Q}$, this function returns the rank of the Mordell–Weil group of E.

The general procedure to calculate the rank involves searching for rational points on certain homogeneous spaces represented by equations of the form $y^2 = ax^4+bx^3+cx^2+dx+e$. The parameter `Bound` is a bound on the numerator and denominator of x that will be used when searching for such points.

`RankBounds(E: parameters)`

`MordellWeilRankBounds(E: parameters)`

`Bound` RngIntElt *Default :* 150

Given an elliptic curve E defined over $\mathbf{Q}$, this computes and returns lower and upper bounds on the rank of the Mordell–Weil group of E. The parameter `Bound` is as described in the intrinsic `Rank`. This function will not generate the usual warning message if the upper and lower bounds are not equal.

`MordellWeilGroup(E: parameters)`

`AbelianGroup(E: parameters)`

`Bound` RngIntElt *Default :* 150

`HeightBound` RngIntElt *Default :* 15

The Mordell–Weil group of an elliptic curve E defined over $\mathbf{Q}$. The function returns two values: an abelian group A and a map m from A to E. The map m provides an isomorphism between the abstract group A and the Mordell–Weil group.

The parameter `Bound` is used during the rank computation part of the algorithm, and has the same meaning as described in the intrinsic `Rank`, above. The group computation involves searching for points on E up to a certain (naive) height. The parameter `HeightBound` is used to limit this search as the rigorous bounds may not be feasible. When the `HeightBound` is less than the rigorous bound, a warning message is printed. In this situation, the user may wish to also use `Saturation` (see below).

As explained above for `Rank`, the computed rank may be less than the actual rank. In cases where the rank is not proven, a warning message is printed (the first time). The best way to check whether the rank has been proven is to use `RankBounds`.

`Generators(E)`

Given an elliptic curve E defined over $\mathbf{Q}$, this function returns generators for the Mordell–Weil group of E, in the form of a sequence of points of E. The i-th element of the sequence corresponds to the i-th generator of the group as returned by the function `MordellWeilGroup`; the generators of the torsion subgroup will come first (in the order described in `TorsionSubgroup`), followed by the points of infinite order.

`NumberOfGenerators(E)`

`Ngens(E)`

The number of generators of the group of rational points of the elliptic curve E; this is simply the length of the sequence returned by `Generators(E)`.

`Saturation(points, n)`

`TorsionFree`	BoolElt	*Default :* `false`
`OmitPrimes`	[RngIntElt]	*Default :* []
`Check`	BoolElt	*Default :* `true`

Given a sequence of points on an elliptic curve E over the rationals, and an integer n, this function returns a sequence of points generating a subgroup of $E(\mathbf{Q})$ which contains the given points and which is *p-saturated* for all primes p up to n. (A subgroup S is p-saturated in a group G if there is no intermediate subgroup H for which the index $[H : S]$ is finite and divisible by p.)

If `OmitPrimes` is set to be a sequence of primes, then the group is not checked to be p-saturated for those primes. If `TorsionFree` is set to tt true, torsion points are omitted from the result (that is, the result contains independent generators modulo torsion). If `Check` is set to `false`, the input sequence of points are assumed to be independent modulo torsion.

Example H122E5

```
> E := EllipticCurve([73, 0]);
> E;
Elliptic Curve defined by y^2 = x^3 + 73*x over Rational Field
> Factorization(Integers() ! Discriminant(E));
[ <2, 6>, <73, 3> ]
> BadPrimes(E);
[ 2, 73 ]
> LocalInformation(E);
[ <2, 6, 6, 1, II, true>, <73, 3, 2, 2, III, true> ]
> G, m := MordellWeilGroup(E);
> G;
Abelian Group isomorphic to Z/2 + Z + Z
Defined on 3 generators
Relations:
    2*G.1 = 0
```

```
> Generators(E);
[ (0 : 0 : 1), (36 : -222 : 1), (4/9 : 154/27 : 1) ]
> 2*m(G.1);
(0 : 1 : 0)
```

Example H122E6

Here is a curve with moderately large rank; we do not attempt to compute the full group since that would be quite time-consuming.

```
> E := EllipticCurve([0, 0, 0, -9217, 300985]);
> T, h := TorsionSubgroup(E);
> T;
Abelian Group of order 1
> time RankBounds(E);
7 7
Time: 0.070
```

This curve was well-behaved in that the computed lower and upper bounds on the rank are the same, and so we know that we have computed the rank exactly. Here is a curve where that is not the case:

```
> E := EllipticCurve([0, -1, 0, -140, -587]);
> time G, h := MordellWeilGroup(E);
Warning: rank computed (2) is only a lower bound
(It may still be correct, though)
Time: 0.250
> RankBounds(E);
2 3
```

The difficulty here is that the Tate–Shafarevich group of E is not trivial, and this blocks the 2-descent process used to compute the rank. We can compute the `AnalyticRank` of the curve to be 2, but this is only conjecturally equal to the rank. In cases like this, higher level descents may provide a more definitive answer; the machinery for two- and four-descents described in the next two sections can be used to confirm that the rank is indeed 2. In any case, we can certainly get the group with rank equal to the lower bound.

```
> G;
Abelian Group isomorphic to Z + Z
Defined on 2 generators (free)
> S := Generators(E);
> S;
[ (-6 : -1 : 1), (-7 : 1 : 1) ]
> [ Order(P) : P in S ];
[ 0, 0 ]
> h(G.1) eq S[1];
true
> h(G.2) eq S[2];
true
> h(2*G.1 + 3*G.2);
```

```
(-359741403/57729604 : 940675899883/438629531192 : 1)
> 2*S[1] + 3*S[2];
(-359741403/57729604 : 940675899883/438629531192 : 1)
```

As mentioned above, the rank of this curve is actually 2, so we have computed generators for the full group of E.

122.2.6 Heights and Height Pairing

These functions require that the corresponding elliptic curve has integral coefficients.

`NaiveHeight(P)`

`WeilHeight(P)`

Given a point $P = (a/b, c/d, 1)$ on an elliptic curve E defined over **Q** with integral coefficients, this function returns the naive (or Weil) height $h(P)$ whose definition is $h(P) = \log \max\{|a|, |b|\}$.

`Height(P: parameters)`

`CanonicalHeight(P: parameters)`

`Precision`	RngIntElt	*Default :*

Given a point P on an elliptic curve E defined over **Q** with integral coefficients, this function returns the canonical height $\hat{h}(P)$.

One definition of $\hat{h}(P)$ is $\hat{h}(P) = \lim_{n\to\infty} 4^{-n} h(2^n P)$, although this is of limited computational use. A more useful computational definition is as the sum of local heights: $\hat{h}(P) = \sum_{p\leq\infty} \hat{h}_p(P)$, where the sum ranges over each prime p and the so-called 'infinite prime'. Each of these local heights can be evaluated fairly simply, and in fact most of them are 0. The function *always* uses a minimal model for the elliptic curve internally, as otherwise the local height computation could fail. The computation at the infinite prime uses a slight improvement over the σ-function methods given in Cohen, with the implementation being based on an AGM-trick due to Mestre.

`LocalHeight(P, p)`

`Precision`	RngIntElt	*Default :*
`Check`	BoolElt	*Default :* `false`
`Renormalization`	BoolElt	*Default :* `false`

Given a point P on an elliptic curve E defined over **Q** with integral coefficients, this function returns $\hat{h}_p(P)$, the local height of P at p as described in `Height` above. The integer p must be either a prime number or 0; in the latter case the height at the infinite prime is returned. The `Check` vararg determines whether to check if the second argument is prime. The `Renormalization` vararg changes what definition of heights is used. Without this flag, a factor of $\frac{1}{6}\log \Delta_v$ is added at every place.

`HeightPairing(P, Q: parameters)`

`Precision` RngIntElt *Default :*

Given two points P, Q on the same elliptic curve defined over $\mathbf{Q}$ with integral coefficients, this function returns the height pairing of P and Q, which is defined by $\hat{h}(P,Q) = (\hat{h}(P+Q) - \hat{h}(P) - \hat{h}(Q))/2$.

`HeightPairingMatrix(S: parameters)`

`HeightPairingMatrix(E: parameters)`

`Precision` RngIntElt *Default :*

Given a sequence of points on an elliptic curve defined over $\mathbf{Q}$ with integral coefficients, this function returns the height pairing matrix. If an elliptic curve is passed to it, the corresponding matrix for the Mordell–Weil generators is returned.

`Regulator(S)`

`Precision` RngIntElt *Default :*

Given a sequence of points S on an elliptic curve E over $\mathbf{Q}$, this function returns the determinant of the Néron-Tate height pairing matrix of the sequence.

`Regulator(E)`

`Precision` RngIntElt *Default :*

Given an elliptic curve E this function returns the regulator of E; i.e., the determinant of the Néron-Tate height pairing matrix of a basis of the free quotient of the Mordell–Weil group.

Example H122E7

```
> E := EllipticCurve([0,0,1,-7,6]);
> P1, P2, P3 := Explode(Generators(E));
> Height(P1);
0.66820516565192793503314205 09
> IsZero(Abs(Height(2*P1) - 4*Height(P1)));
true
> BadPrimes(E);
[ 5077 ]
```

The local height of a point P at a prime is 0 except possibly at the bad primes of E, the 'infinite' prime, and those primes dividing the denominator of the x-coordinate of P. Since these generators have denominator 1 we see that only two local heights need to be computed to find the canonical heights of these points.

```
> P2;
(2 : 0 : 1)
> LocalHeight(P2, 0);
-0.65503594715968618249727806 9814
> LocalHeight(P2, 5077); // 0 + Log(5077)/6
```

```
1.42207930249123238829272871637
> LocalHeight(P2, 0 : Renormalization);
0.76704335533154620579545064655 2
> LocalHeight(P2, 5077 : Renormalization);
0.00000000000000000000000000000
> Height(P2);
0.7670433553315462057954506466
```

The above shows that the local height at a bad prime may still be zero, at least in the renormalised value.

`SilvermanBound(E)`

Given an elliptic curve E over $\mathbf{Q}$ with integral coefficients, this function returns the Silverman bound B of E. For any point P on E we will have $h(P) - \hat{h}(P) \leq B$.

`SiksekBound(E:` *parameters*`)`

`Torsion`	BoolElt	*Default :* `false`

Given an elliptic curve E over $\mathbf{Q}$ which is a minimal model, this function returns a real number B such that for any point P on E $h(P) - \hat{h}(P) \leq B$. In many cases, the Siksek bound is much better than the Silverman bound.

If the parameter `Torsion` is `true` then a potentially better bound B_{Tor} is computed, such that for any point P on E there exists a torsion point T so that $h(P+T) - \hat{h}(P) \leq B_{Tor}$. Note that $\hat{h}(P+T) = \hat{h}(P)$.

Example H122E8

We demonstrate the improvement of the Siksek bound over the Silverman bound for the three example curves from Siksek's paper [Sik95].

```
> E := EllipticCurve([0, 0, 0, -73705, -7526231]);
> SilvermanBound(E);
13.00022113685530200655193
> SiksekBound(E);
0.82150471924479497959096194911
> E := EllipticCurve([0, 0, 1, -6349808647, 193146346911036]);
> SilvermanBound(E);
21.75416864448061105008
> SiksekBound(E);
0.61777729068784838634233492172850957748 0
> E := EllipticCurve([1, 0, 0, -5818216808130, 5401285759982786436]);
> SilvermanBound(E);
27.56255914401769757660
> SiksekBound(E);
15.70818965430290161142481294545
```

This last curve has a torsion point, so we can further improve the bound:

```
> T := E![ 1402932, -701466 ];
```

```
> Order(T);
2
> SiksekBound(E : Torsion := true);
11.030987623117983983183295126526 88
```

Here is a point which demonstrates the applicability of the modified bound.

```
> P := E![ 14267166114 * 109, -495898392903126, 109^3 ];
> NaiveHeight(P) - Height(P);
12.193000709615680011116868901084
> NaiveHeight(P + T) - Height(P);
2.60218831527724007036
```

`IsLinearlyIndependent(P, Q)`

Given points P and Q on an elliptic curve E, this function returns `true` if and only if P and Q are independent free elements of the group of rational points of an elliptic curve (modulo torsion points). If `false`, the function returns a vector $v = (r, s)$ as a second value such that $rP + sQ$ is a torsion point.

`IsLinearlyIndependent(S)`

Given a sequence of points S belonging to an elliptic curve E, this function returns `true` if and only if the points in S are linearly independent (modulo torsion). If `false`, the function returns a vector v in the kernel of the height pairing matrix, i.e. giving a torsion point as a linear combination of the points in S.

`ReducedBasis(S)`

Given a sequence of points S belonging to an elliptic curve E over the rationals, the function returns a sequence of points that are independent modulo the torsion subgroup (equivalently, they have non-degenerate height pairing), and which generate the same subgroup of $E(\mathbf{Q})/E_{tors}(\mathbf{Q})$ as points of S.

Example H122E9

We demonstrate the linear independence test on an example curve with an 8-torsion point and four independent points. The torsion point is easily recognized and we establish the independence of the remaining points using the height function.

```
> E := EllipticCurve([0,1,0,-95549172512866864, 116909987427985538083349 00]);
> P0 := E![ 39860582, 2818809365988 ];
> P1 := E![ 144658748, -946639447182 ];
> P2 := E![ 180065822, 569437198932 ];
> P3 := E![ -339374593, 2242867099638 ];
> P4 := E![ -3492442669/25, 590454479818404/125 ];
> S0 := [P0, P1, P2, P3, P4];
> IsLinearlyIndependent(S0);
false (1 0 0 0 0)
> Order(P0);
```

```
8
> S1 := [P1, P2, P3, P4];
> IsLinearlyIndependent(S1);
true
```

We now demonstrate the process of solving for a nontrivial linear dependence among points on an elliptic curve by constructing a singular matrix, forming the corresponding linear combination of points, and establishing that the dependence is in the matrix kernel.

```
> M := Matrix(4, 4, [-3,-1,2,-1,3,3,3,-2,3,0,-1,-1,0,2,1,1]);
> Determinant(M);
0
> S2 := [ &+[ M[i, j]*S1[j] : j in [1..4] ] : i in [1..4] ];
> IsLinearlyIndependent(S2);
false ( 5 -3  8  7)
> Kernel(M);
RSpace of degree 4, dimension 1 over Integer Ring
Echelonized basis:
( 5 -3  8  7)
```

Despite the moderate size of the numbers which appear in the matrix M, we note that height is a quadratic function in the matrix coefficients. Since height is a logarithmic function of the coefficient size of the points, we see that the sizes of the points in this example are very large:

```
> [ RealField(16) | Height(P) : P in S2 ];
[ 137.376951049198, 446.51954933694, 52.724183282292, 59.091649046171 ]
> Q := S2[2];
> Log(Abs(Numerator(Q[1])));
467.65985870406594118081172 53
> Log(Abs(Denominator(Q[1])));
449.25545877278405839494427 65
```

`pAdicHeight(P, p)`

`Precision`	RngIntElt	*Default :* 0
E2	FldPadElt	*Default :* 0

Given a point P on an elliptic curve defined on the rationals and a prime $p \geq 5$ of good ordinary reduction, this function computes the p-adic height of P to the desired precision. The value of the `EisensteinTwo` function for the curve can be passed as a parameter. The algorithm dates back to [MT91] with improvements due to [MST06] and [Har08]. The normalization is that of the last-named paper and is $2p$ (or $-2p$ in some cases) as large as in other papers.

`pAdicRegulator(S, p)`

`Precision`	RngIntElt	*Default :* 0
`E2`	FldPadElt	*Default :* 0

Given a set of points S on an elliptic curve defined over the rationals and a prime $p \geq 5$ of good ordinary reduction, this function computes the p-adic height regulator of S to the desired precision. Here the normalization divides out a power of p from the individual heights.

`EisensteinTwo(E, p)`

`Precision`	RngIntElt	*Default :* 0

Given an elliptic curve over the rationals and a prime $p \geq 5$ of good ordinary reduction, this function computes the value of the Eisenstein series E_2 using the Monsky-Washnitzer techniques via Kedlaya's algorithm. See `pAdicHeight` above for references.

Example H122E10

We give examples for computing p-adic heights.

```
> E := EllipticCurve("5077a");
> P1 := E ! [2, 0];
> P2 := E ! [1, 0];
> P3 := E ! [-3, 0];
> assert P1+P2+P3 eq E!0; // the three points sum to zero
> for p in PrimesInInterval(5,30) do pAdicHeight(P1, p); end for;
4834874647277 + O(5^20)
5495832406058373*7 + O(7^20)
-1240398531370452467 4*11 + O(11^20)
6167277315947533603 89*13 + O(13^20)
5314497586743410875 4867*17 + O(17^20)
6514787540337334207 44924*19 + O(19^20)
1382894029404692415656094*23^2 + O(23^20)
-2615503441214747688800993518*29 + O(29^20)
>  pAdicHeight(P1, 37); // 37 is not ordinary for this curve
>> pAdicHeight(P1, 37); // 37 is not ordinary for this curve
Runtime error in 'pAdicHeight': p cannot be supersingular
> for p in PrimesInInterval(5,30) do pAdicRegulator([P1, P2], p); end for;
-263182161797834*5^-2 + O(5^20)
-35022392779944725 + O(7^20)
-331747503271490921584 + O(11^20)
246446095809126124462 + O(13^20)
-15285279157972275150672 70 + O(17^20)
33388427569565384672997 0*19 + O(19^20)
41297839835611557028076 0602 + O(23^20)
-45973362301048046561945193743 + O(29^20)
> pAdicRegulator([P1, P2, P3], 23); // dependent points
```

```
O(23^20)
> eisen_two := EisensteinTwo(E, 13 : Precision:=40); eisen_two;
25360291252705414983472710631099471724343012 + O(13^40)
> pAdicRegulator([P1, P2], 13 : Precision:=40, E2:=eisen_two);
85894629213918025547455609490608727790695215 + O(13^40)
```

122.2.7 Two-Descent and Two-Coverings

The two-descent process determines the locally soluble two-coverings of an elliptic curve defined over a number field K, giving them as hyperelliptic curves $C : y^2 = f(x)$ with f of degree four. To be practical, K must be of rather small degree, and for various parts of 2-descent to run (particularly reduction), K must have certain properties (such as being totally real). Also, the algorithms over the rationals have now been revisited, and are now faster than when using the general number field descent machinery.

A separate implementation is available for the case where E admits a 2-isogeny: this involves first computing the 2-coverings for the isogeny and its dual (in this case the covering maps have degree 2 instead of degree 4), and then lifting these to the level of a "full 2-descent".

This section also provides some tools for manipulating such curves, including generators for some rings of invariants of quartic forms, minimisation and reduction of such forms, and the maps back to the associated elliptic curve. This functionality overlaps with the package for genus one models (see 124). There is a straightforward interface to the 2-Selmer group machinery, which also works over number fields.

`TwoDescent(E:` *parameters*`)`

`RemoveTorsion`	BoolElt	*Default :* `false`
`RemoveGens`	{PtEll}	*Default :* {}
`WithMaps`	BoolElt	*Default :* `true`
Verbose	`TwoDescent`	*Maximum :* 1

Given an elliptic curve E over the rationals 2-descent can be used to construct a sequence of hyperelliptic curves (quartics) representing the elements of the 2-Selmer group. The group structure is not preserved by this function: the intrinsic `TwoSelmerGroup` should be used if this is desired. If `RemoveTorsion` is `true`, the generators of the torsion subgroup are factored out from the set. If `RemoveGens` is nonempty, the image of the specified points is factored out from the set. If `AppendMaps` is `true`, then the maps from the covers to the curve are appended as a second argument.

AssociatedEllipticCurve(f)

AssociatedEllipticCurve(C)

E CrvEll *Default :*

Gives the minimal model of the elliptic curve associated with a two-covering given as a polynomial f or a hyperelliptic curve C, along with a map from points $[x, y]$ on the two-covering to the curve.

If an elliptic curve is given as E, this must be isomorphic to the Jacobian of C, and then the map returned will be a map to the given E.

Example H122E11

```
> SetSeed(1); // results may depend slightly on the seed
> E := EllipticCurve([0, 1, 0, -7, 6]);
> S := TwoDescent(E);
> S;
[
    Hyperelliptic Curve defined by y^2 = x^4 + 4*x^3 - 2*x^2 - 20*x + 9 over
    Rational Field,
    Hyperelliptic Curve defined by y^2 = x^4 - x^3 - 2*x^2 + 2*x + 1 over
    Rational Field,
    Hyperelliptic Curve defined by y^2 = 2*x^4 - 4*x^3 - 8*x^2 + 4*x + 10 over
    Rational Field
]
```

E has three non-trivial two-descendants, hence its rank is at most 2. The first two curves yield obvious rational points, so we can find two independent points on E (and it has exact rank 2).

```
> pt_on_S1 := Points(S[1] : Bound:=10 )[1];
> pt_on_S1;
// We obtain the map from S[1] to E by
> _, phi := AssociatedEllipticCurve(S[1] : E:=E );
> phi( pt_on_S1 );
(1 : -1 : 1)
// Now do the same for the second curve.
> pt_on_S2 := Points(S[2] : Bound:=10 )[1];
> _, phi := AssociatedEllipticCurve(S[2] : E:=E );
> phi( pt_on_S2 );
(-3 : 3 : 1)
```

122.2.7.1 Two Descent Using Isogenies

`TwoIsogenyDescent(E : parameters)`

`Isogeny`	MapSch	*Default :*
`TwoTorsionPoint`	PtEll	*Default :*

Given an elliptic curve E over $\mathbf{Q}$ admitting a 2-isogeny $\phi : E' \to E$, this function computes 2-coverings representing the nontrivial elements of the Selmer groups of ϕ and of the dual isogeny $\phi' : E \to E'$. These coverings are given as hyperelliptic curves $C : y^2 = quartic(x)$. Six objects are returned: (i) the sequence of coverings C of E for ϕ; (ii) the corresponding list of maps $C \to E$; (iii) and (iv) the coverings C' and maps $C' \to E'$ for ϕ'; (v) and (vi) the isogenies ϕ and ϕ' that were used.

It's advisable to give a model for E that is close to minimal.

`LiftDescendant(C)`

This routine performs a higher descent on curves arising in `TwoIsogenyDescent(E)` on an elliptic curve E over $\mathbf{Q}$. The curves obtained are 2-coverings of E in the sense of ordinary (full) `TwoDescent`; more precisely, they are exactly the set of 2-coverings D for which the covering map $D \to E$ factors through C. Up to isomorphism, they are a subset of the 2-coverings returned by `TwoDescent(E)`. The advantage of this approach is that it works entirely over the base field of E, whereas `TwoDescent` will in general compute a class group over a quadratic extension of the base field. The example below explains how to recover all coverings produced by `TwoDescent(E)` using the 2-isogeny approach.

This function accepts any curve C in the first sequence of curves returned by `TwoIsogenyDescent(E)` (these are the 2-isogeny-coverings of E). More generally it accepts any hyperelliptic curve of the form $y^2 = d_1x^4 + cx^2 + d_2$. A model for the associated elliptic curve E is then $y^2 = x(x^2 + cx + d_1d_2)$.

The function returns three objects: a sequence containing the covering curves D, a list containing the corresponding maps $D \to C$, and lastly the covering map $C \to E$ from the given curve to some model of its associated elliptic curve.

122.2.7.2 Invariants

`QuarticIInvariant(q)` `QuarticJInvariant(q)`

`QuarticG4Covariant(q)` `QuarticG6Covariant(q)`

`QuarticHSeminvariant(q)` `QuarticPSeminvariant(q)`

`QuarticQSeminvariant(q)` `QuarticRSeminvariant(q)`

Compute invariants, semivariants, and covariants, as in paper [Cre01], of a given quartic polynomial q. The G4 and G6 covariants are polynomials of degrees four and six respectively; the I and J invariants are integers that generate the ring of integer invariants of the polynomial and satisfy $J^2 - 4I^3 = 27\Delta(f)$. The names H

and P have been used for essentially the same seminvariant in different papers; they are related by $H = -P$.

`QuarticNumberOfRealRoots(q)`

Using invariant theory, compute the number of real roots of a real quartic polynomial q.

`QuarticMinimise(q)`

This computes a minimal model of the quartic polynomial q over the rationals or a univariate rational function field.

Three objects are returned: the minimised quartic, the transformation matrix, and the scaling factor.

For further explanation see Chapter 124. The algorithm can be found in [CFS10].

`QuarticReduce(q)`

Given a quartic q, the algorithm of [Cre99] is applied to find the reduced quartic and the matrix that reduced it.

`IsEquivalent(f,g)`

Determines if the quartics f and g are equivalent.

122.2.8 The Cassels-Tate Pairing

The Tate–Shafarevich group of any elliptic curve E admits an alternating bilinear form on Ш(E) with values in $\mathbf{Q}/\mathbf{Z}$, known as the Cassels-Tate pairing. The key property is that if Ш(E) is finite (as conjectured), the Cassels-Tate pairing is non-degenerate. When restricted to the 2-torsion subgroup, one obtains a non-degenerate alternating bilinear form on Ш(E)[2]/2Ш(E)[4], or equivalently on $Sel^2(E)$ modulo the image of $Sel^4(E)$, with values in $\mathbf{Z}/2\mathbf{Z}$.

This means that if C and D are 2-coverings of E and the pairing (C, D) has value 1, then both C and D represent elements of order 2 in Ш(E), and moreover there are no locally solvable 4-coverings of E lying above them (in other words, `FourDescent(C)` and `FourDescent(D)` would both return an empty sequence). In this sense the Cassels-Tate pairing provides the same information as 4-descent, but is much easier to compute.

Similarly, for an element in Ш(E)[4]/2Ш(E)[8], the values of the pairing between this element and all elements in Ш(E)[2]/2Ш(E)[4] provides the same information as performing an 8-descent on C. These elements may be represented by a 4-covering $C \to E$ and a 2-covering $D \to E$ respectively.

In Magma the pairing between 2-coverings is implemented over $\mathbf{Q}$ number fields, and rational function fields $F(t)$ for F finite of odd characteristic. The pairing between a 2-covering and a 4-covering is implemented over $\mathbf{Q}$. A new, very efficient implementation of pairing on 2-coverings over $\mathbf{Q}$ was released in Magma V2.15.

The algorithms are due to Steve Donnelly and will be described in a forthcoming paper, a draft of which is available on request. For the pairing between 2-coverings, the only nontrivial computation is to solve a conic over the base field of E, so over $\mathbf{Q}$ the pairing

is easy to compute. For the pairing between 2- and 4-coverings, the key step is to solve a conic defined over a degree 4 field; this is also the case for performing 8-descent on the 4-covering, however the advantage here is that there is considerable freedom to choose the field to have small discriminant. Consequently it is more efficient to use the pairing than to apply `EightDescent`.

122.2.8.1 Verbose Information

To have information about the computation printed while it is running, one may use `SetVerbose("CasselsTate",n);` with $n = 1$ (for fairly concise information) or $n = 2$.

`CasselsTatePairing(C, D)`

Verbose `CasselsTate` *Maximum :* 2

This evaluates the Cassels-Tate pairing on 2-coverings of an elliptic curve over **Q**, a number field, or a function field $F(t)$ where F is a finite field of odd characteristic. The given curves C and D must be hyperelliptic curves of the form $y^2 = q(x)$ where $q(x)$ has degree 4, and they must admit 2-covering maps to the same elliptic curve. In addition, they must both be locally solvable over all completions of their base field (otherwise the pairing is not defined).

Typically the input curves C and D would be obtained using `TwoDescent(E)`.

The pairing takes values in $\mathbf{Z}/2\mathbf{Z}$ (returned as elements of **Z**).

`CasselsTatePairing(C, D)`

Verbose `CasselsTate` *Maximum :* 2

This evaluates the Cassels-Tate pairing between a 4-covering C and a 2-covering D of the same elliptic curve over **Q**. The arguments should be curves over **Q**, with C an intersection of two quadrics in $\mathbf{P}^3$ (for instance, a curve obtained from `FourDescent`), and D a hyperelliptic curve of the form $y^2 = q(x)$. In addition, they must both be locally solvable over all completions of **Q** (otherwise the pairing is not defined).

The pairing takes values in $\mathbf{Z}/2\mathbf{Z}$ (returned as elements of **Z**).

Example H122E12

We consider the first elliptic curve with trivial 2-torsion and nontrivial Tate-Shafarevich group.

```
> E := EllipticCurve("571a1"); E;
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 929*x - 10595 over Rational Field
> #TorsionSubgroup(E);
1
> time covers := TwoDescent(E); covers;
Time: 0.270
[
    Hyperelliptic Curve defined by y^2 = -11*x^4 - 68*x^3 - 52*x^2 + 164*x - 64
    over Rational Field,
    Hyperelliptic Curve defined by y^2 = -19*x^4 + 112*x^3 - 142*x^2 - 68*x - 7
    over Rational Field,
```

```
    Hyperelliptic Curve defined by y^2 = -4*x^4 + 60*x^3 - 232*x^2 + 52*x - 3
    over Rational Field
]
> time CasselsTatePairing(covers[1], covers[2]);
1
Time: 0.130
```

This proves that these two coverings both represent nontrivial elements in the Tate-Shafarevich group; in fact our computations show that the 2-primary part of Ш(E) is precisely $\mathbf{Z}/2 \times \mathbf{Z}/2$. We could have reached the same conclusion using 4-descent:

```
> time FourDescent(covers[1]);
[]
Time: 0.460
```

122.2.9 Four-Descent

In a 1996 paper [MSS96], Merriman, Siksek and Smart described a four-descent algorithm for elliptic curves over $\mathbf{Q}$. This section describes the MAGMA implementation of that algorithm. Another useful reference is Tom Womack's PhD thesis [Wom03]. Four-descent is performed on a two-cover, that is a hyperelliptic curve defined by a polynomial of degree four, with the assumption that the quartic of this descendant does not have a rational root. The terminology "four-descent" is thus slightly incorrect; MAGMA actually performs a second descent on a specific two-cover, and does not try to combine such information from all two-covers into the 4-Selmer group.

A *four-covering* F is a pair of symmetric 4×4 matrices, defining an intersection of two quadrics in P^3. Associated to F is an elliptic curve E; there is a rational map from F to E of degree 16. The four-descent process takes a two-covering curve C (something of the shape $y^2 = f(x)$ with f quartic, and possessing points over $\mathbf{Q}_p$ for all p), and returns a set of four-coverings that arise from C.

In particular, if C represents an element of order two in the Tate–Shafarevich group of E, then the four-descent process will return the empty set. If C represents an element of the Mordell–Weil group, at least one of the four-coverings arising from C will have a rational point — all of them will do so if the Tate–Shafarevich group of E is trivial — and once found this point can be lifted to a point on E.

`FourDescent(f : parameters)`

`FourDescent(C : parameters)`

`RemoveTorsion`	BOOLELT	*Default :* `false`
`IgnoreRealSolubility`	BOOLELT	*Default :* `false`
`RemoveTorsion`	BOOLELT	*Default :* `false`
`RemoveGensEC`	{PTELL}	*Default :* {}
`RemoveGensHC`	{PTHYP}	*Default :* {}

Verbose	`FourDescent`	*Maximum :* 3
Verbose	`LocalQuartic`	*Maximum :* 2
Verbose	`MinimiseFD`	*Maximum :* 2
Verbose	`QISearch`	*Maximum :* 1
Verbose	`ReduceFD`	*Maximum :* 2
Verbose	`QuotientFD`	*Maximum :* 2

Performs a four-descent on the curve $y^2 = f(x)$, where f is a quartic. Returns a set of four-coverings of size 2^{s-1}, where s is the Selmer 2-rank of the curve. If the verbose level of the main function is set to 3, then the auxiliary verbose levels are all set to at least 1. If `RemoveTorsion` is `true`, the generators of the torsion subgroup are factored out from the set. The optional argument `RemoveGensHC` performs a quotient by the images of given points that are on the input quartic to `FourDescent`. The optional argument `RemoveGensEC` forms a quotient by the images of given points; these can be on any elliptic curve that is isomorphic to the `AssociatedEllipticCurve` of the quartic (though all given points must be on the same elliptic curve). These options can be used together. It can be non-trivial to remove torsion and generators, as points on the elliptic curve need not pull-back to the given $y^2 = f(x)$ curve. The algorithm used exploits various primes of good reduction, and attempts to determine whether the images under the μ_p are the same. This in turn can be tricky, due to the Cassels kernel, and even more so when there are extra automorphisms (that is, when $j = 0, 1728$ over $\mathbf{F}_p$).

Example H122E13

This example shows that a well-known curve has rank 0 and that the 2-torsion subgroup of its Tate–Shafarevich group is isomorphic to $(\mathbf{Z}/2\mathbf{Z})^2$.

```
> D := CremonaDatabase();
> E := EllipticCurve(D, 571, 1, 1);
> time td := TwoDescent(E);
Time: 2.500
> #td;
3
```

There are three 2-covers, so the 2-Selmer group has order four (since `TwoDescent` elides the trivial element).

```
> time [ FourDescent(t) : t in td ];
[
    [],
    [],
    []
]
Time: 3.290
```

So none of the two-covers have four-covers lying over them; hence they all represent elements of Ш, and the Mordell–Weil rank must be zero.

`AssociatedEllipticCurve(qi)`

`AssociatedHyperellipticCurve(qi)`

`E`	CrvEll	*Default :*

Given an intersection of quadrics qi, return the associated elliptic and hyperelliptic curves, respectively, together with maps to them.

If an elliptic curve is given as E, this must be isomorphic to the Jacobian of the curve qi, and then the map returned will be a map to the given E.

`QuadricIntersection(F)`

`QuadricIntersection(P, F)`

Given a pair of symmetric 4×4 matrices F, this function returns the associated quadric intersection in $P = P^3$.

`QuadricIntersection(E)`

`QuadricIntersection(C)`

Given an elliptic curve E or a hyperelliptic curve C, write it as an intersection of quadrics. The inverse map for the hyperelliptic curve has problems due to difficulties with weighted projective space.

`IsQuadricIntersection(C)`

Given a curve C, determines if C is in P^3 and has two defining equations, both of which involve only quadrics. In the case where C is a quadric intersection, the associated pair of matrices are also returned.

`PointsQI(C, B :` *parameters*`)`

`OnlyOne`	BoolElt	*Default :* `false`
`ExactBound`	BoolElt	*Default :* `false`
Verbose	`QISearch`	*Maximum :* 1

Given a quadric intersection C, this function searches, by a reasonably efficient method due to Elkies [Elk00], for a point on C of naïve height up to B; the asymptotic running time is $O(B^{2/3})$.

If `OnlyOne` is set to `true`, the search stops as soon as it finds one point; however, the algorithm is p-adic and there is no guarantee that points with small coordinates in $\mathbf{Z}$ will be found first. If `ExactBound` is set to `true`, then points that are found with height larger than B will be ignored.

`TwoCoverPullback(H, pt)`

`TwoCoverPullback(f, pt)`

Given a two-covering of a rational elliptic curve (as either a hyperelliptic curve or a quartic) and a point on the elliptic curve, compute the pre-images on the two-covering. This is faster than using the generic machinery.

`FourCoverPullback(C, pt)`

Given a four-covering of a rational elliptic curve as an intersection of quadrics and a point either on the associated elliptic curve or the associated hyperelliptic curve, this function computes the pre-images on the covering. This is faster than using the generic machinery.

Example H122E14

This example exhibits a four-descent computation, and manipulation of points once they have been found, by mapping from the curve to its two- and four-covers.

```
> P<x> := PolynomialRing(Integers());
> E := EllipticCurve([0, -1, 0, 203, -93]);
> f := P!Reverse([-7, 12, 20, -120, 172]);
> f;
-7*x^4 + 12*x^3 + 20*x^2 - 120*x + 172
```

The quartic given was obtained with `mwrank`; the two-descent routine could have been used instead, although it provides a different (but equivalent) quartic.

```
> time S := FourDescent(f);
Time: 4.280
> #S;
1
```

The single cover indicates that the curve E has Selmer rank 1, though this was already known from the calculation that constructed f.

```
> _,m := AssociatedEllipticCurve(S[1] : E:=E );
> pts := PointsQI(S[1], 10^4);
> pts;
[ (-5/3 : 13/3 : -34/3 : 1) ]
```

We now map this point back to E.

```
> m(pts[1]);
(2346223045599488598/1033322524668523441 :
2032660922346093775326473 5467/1050397899358266605692672489 : 1)
> Height($1);
44.19679596739622477261983370
```

122.2.10 Eight-Descent

One may perform 8-descent (ie a further 2-descent) on curves of the kind produced by a 4-descent on an elliptic curve E over $\mathbf{Q}$. These are nonsingular intersections of two quadrics in $\mathbf{P}^3$ that are locally soluble. The 8-descent determines whether such a curve has any 2-coverings (in the sense of 2-descent) that are locally soluble everywhere.

The routine can therefore be used to prove, in many cases, that a given 4-covering of E is in fact an element of order 4 in the Tate–Shafarevich group of E. It can also be used to find 8-coverings of E, and to verify these are elements of the 8-Selmer group.

The 8-coverings are given as genus one normal curves of degree 8 in $\mathbf{P}^8$. They are minimised and reduced, so are useful in searching for points on E.

The algorithm and implementation (from MAGMA 2.17) are due to Tom Fisher; this implementation partly incorporates and partly replaces an earlier one by Sebastian Stamminger.

`EightDescent(C : parameters)`

`BadPrimesHypothesis`	BOOLELT	*Default :* `false`
`DontTestLocalSolvabilityAt`	{RNGINTELT}	*Default :* {}
`StopWhenFoundPoint`	BOOLELT	*Default :* `false`
Verbose	`EightDescent`	*Maximum :* 4
Verbose	`LegendresMethod`	*Maximum :* 3

For a curve C obtained from `FourDescent` on some elliptic curve E over $\mathbf{Q}$, this performs a further 2-descent on C. It returns a sequence of curves D, together with a sequence containing maps $D \to C$ from each of these curves to C. The curves returned are precisely the 8-descendants of E that lie above C and are locally soluble at all places.

When the optional argument `StopWhenFoundPoint` is set to `true`, the computation will stop if it happens to find a rational point on C, and immediately return the point instead of continuing to computing the 8-coverings.

In some cases local solubility testing (at large primes which may arise due to choices made in the algorithm) can be time-consuming. Testing at specified primes may be skipped by setting the optional argument `DontTestLocalSolvabilityAt` to the desired set of prime integers, or by setting `BadPrimesHypothesis` to be true. In that case, certain primes are omitted from the set of bad primes (namely those primes at which the intersection of C with an auxiliary third quadric has bad reduction, and which are not bad primes for any other reason).

The algorithm involves class group and unit computations in a number field of degree 4 (and sometimes also 8). It is recommended to set the bounds to be used in all such computations before calling EightDescent, via `SetClassGroupBounds`.

Verbose output: For a readable summary of the computation, set the verbose level to 1 by entering `SetVerbose("EightDescent",1)`. For full information about all the time-consuming steps in the process, set the verbose level to 3. For additional

information about solving the conic, set the verbose level for `"LegendresMethod"` to 2.

122.2.11 Three-Descent

Three descent is implemented for elliptic curves over the rationals. This involves computing the 3-Selmer group of the curve, and then representing the elements as plane cubics.

There is also a separate implementation of "descent by three isogenies", for elliptic curves over the rationals which admit a **Q**-rational isogeny of degree 3.

The main application of full three-descent is in studying elements of order 3 in Tate–Shafarevich groups. For the problem of determining the Mordell–Weil group, three-descent has no advantage over four descent except in special cases: the cost is greater (three-descent requires computing the class group and S-units in a degree 8 number field, compared with degree 4 for four-descent), and the reward is smaller (a point on a 3-coverings has height 1/6 as large as its image on the elliptic curve, while with four-descent the ratio is 1/8). However three-descent can be useful when there are elements of order 4 in the Tate–Shafarevich group, which four-descent cannot deal with.

On the other hand, for a curve with a **Q**-rational isogeny of degree 3, descent by 3-isogenies is likely to be the most efficient way to bound the Mordell–Weil rank, because it only requires class group and S-unit computations in quadratic fields.

There are two steps to the 3-descent process: firstly, computing the 3-Selmer group as a subgroup of $H^1(\mathbf{Q}, E[3])$ (explicitly, as a subgroup of $A^\times/A^{\times 3}$ for a suitable algebra A), and secondly, expressing the elements as genus one curves with covering maps to E. The elements of the 3-Selmer group are given as plane cubics, and the process of obtaining these cubics is far from trivial. (Note that in general, an element of $H^1(\mathbf{Q}, E[3])$ can only be given as a curve of degree 9 rather than degree 3). The main commands are `ThreeSelmerGroup(E)`, which performs the first step, and `ThreeDescent(E)`, which performs both steps together, while `ThreeDescentCubic` performs the second step for a given element of 3-Selmer group.

The algorithm for the first step is presented in [SS04], while the theory and algorithms for the second step are developed in the forthcoming series of papers [CFO+08], [CFO+09], [CFO+]. The bulk of the code was written by Michael Stoll and Tom Fisher (however, responsibility for the final version rests with Steve Donnelly).

The following verbose flags provide information about various stages of the three descent process: `Selmer, ThreeDescent, CSAMaximalOrder, Minimise` and `Reduce`. For instance, to see what `ThreeSelmerGroup` is doing while it is running, first enter `SetVerbose("Selmer",2);`. The verbose levels range from 0 to 3.

`ThreeDescent(E : parameters)`

`Method`	MonStgElt	*Default :* "*HessePencil*"
Verbose	`Selmer`	*Maximum :* 3
Verbose	`ThreeDescent`	*Maximum :* 3

Given an elliptic curve over the rationals, this function returns the elements of the 3-Selmer group as projective plane cubic curves C, together with covering maps

$C \to E$. Two objects are returned: a sequence containing one curve for each inverse pair of nontrivial elements in the 3-Selmer group, and a corresponding list of maps from these curves to E. (Note that a pair of inverse elements in the 3-Selmer group both correspond to the same cubic curve C, and their covering maps $C \to E$ differ by composition with the negation map $E \to E$.)

The function simply calls `ThreeSelmerGroup(E)`, and then `ThreeDescentCubic` on the Selmer elements. Note that if `ThreeSelmerGroup(E)` has already been computed, it will not be recomputed, because the results are stored in the attribute `E‘ThreeSelmerGroup`.

For more information see the description of `ThreeDescentCubic` below.

Example H122E15

Here is Selmer's famous example $3x^3 + 4y^3 + 5z^3 = 0$, which is an element of order 3 in the Tate–Shafarevich group of its Jacobian, the elliptic curve $x^3 + y^3 + 60z^3 = 0$.

```
> Pr2<x,y,z> := ProjectiveSpace(Rationals(),2);
> J := x^3 + y^3 + 60*z^3;
> E := MinimalModel(EllipticCurve(Curve(Pr2,J)));
> cubics, mapstoE := ThreeDescent(E);
> cubics;
[
    Curve over Rational Field defined by 2*x^3 + 30*y^3 - z^3,
    Curve over Rational Field defined by x^3 + 5*y^3 - 12*z^3,
    Curve over Rational Field defined by 6*x^3 + 5*y^3 - 2*z^3,
    Curve over Rational Field defined by 3*x^3 + 5*y^3 - 4*z^3
]
```

The 3-Selmer group of E is isomorphic to $\mathbf{Z}/3\mathbf{Z} \oplus \mathbf{Z}/3\mathbf{Z}$, and one element for each (nontrivial) inverse pair is returned. (Note: $3x^3+4y^3+5z^3$ will not necessarily appear; due to random choices in the program, an equivalent model may appear instead.)
The covering maps from these curves to E have degree 9, and are given by forms of degree 9 in x, y, z. For example, the map from the first curve $2x^3 - 3y^3 + 10z^3$ to E is given by:

```
> DefiningEquations(mapstoE[1]);
[
    -1377495072000*x*y^7*z + 4591650240000*x*y^4*z^4 - 15305500800000*x*y*z^7,
    24794911296000*y^9 - 123974556480000*y^6*z^3 - 413248521600000*y^3*z^6 +
    918330048000000*z^9,
    -4251528000*x^3*y^3*z^3
]
```

Since E has trivial Mordell–Weil group, we should not find any rational points on these curves! (In fact they are all nontrivial in the Tate–Shafarevich group.) Here we search for rational points on the first cubic, up to height roughly 10^4:

```
> time PointSearch( cubics[1], 10^4);
[]
```

```
Time: 0.490
```

An extended example, concerning "visible" 3-torsion in a Tate–Shafarevich group, can be found at the end of Chapter 124.

`ThreeSelmerGroup(E : parameters)`

`ThreeTorsPts`	TUP	*Default :*
`MethodForFinalStep`	MONSTGELT	*Default :* "*UseSUnits*"
`CompareMethods`	BOOLELT	*Default :* `false`
Verbose	`Selmer`	*Maximum :* 3

Given an elliptic curve E over the rationals, this function returns its 3-Selmer group as an abelian group, together with a map to the natural affine algebra.

The parameter `MethodForFinalStep` can be either "Heuristic" (which is usually much faster in large examples), "FindCubeRoots", or "UseSUnits" (the default, which has the advantage that it usually takes roughly the same amount of time as the S-unit computation that has already been performed). To try all three methods and compare the times, set `CompareMethods` to `true`.

Most of the computation time is usually spent on class group and unit group computations. These computations can be speeded up by using non-rigorous bounds, and there are two ways to control which bounds are used. The recommended way is to preset them using one of the intrinsics `SetClassGroupBounds` or `SetClassGroupBoundMaps` (see Section 37.6.1). The other way is to precompute the class groups of the fields involved, setting the optional parameters in `ClassGroup` as desired. (The relevant fields can be obtained as the fields over which the points in `ThreeTorsionPoints(E)` are defined.) The fields and their class group data would then be stored internally, and automatically used when `ThreeSelmerGroup(E)` is called (even when no optional parameters are provided).

When `ThreeTorsPts` is specified, it determines the fields used in the computation, and the algebra used to express the answers.

`ThreeDescentCubic(E, α : parameters)`

`ThreeTorsPts`	TUP	*Default :*
`Method`	MONSTGELT	*Default :* "*HessePencil*"
Verbose	`ThreeDescent`	*Maximum :* 3

Given an elliptic curve E over the rationals, and an element α in the 3-Selmer group of E, the function returns a projective plane cubic curve C, together with a map of schemes $C \to E$. The cubic is a principal homogeneous space for E, and the covering map $C \to E$ represents the same Selmer element as either α or $1/\alpha$ (and α can be recovered, up to inverse, by calling `ThreeSelmerElement` of the cubic).

The 3-Selmer element α is given as an element of the algebra associated to the 3-Selmer group (the algebra is the codomain of the map returned by `ThreeSelmerGroup`, as in `S3, S3toA := ThreeSelmerGroup(E)`). In this situation,

where α is `S3toA(s)` for some `s`, there is no need to specify the optional parameter `ThreeTorsPts`.

The algorithm comes from the series of papers cited in the introduction. There are three alternative ways to perform the final step of computing a ternary cubic. All three ways are implemented, and the optional parameter `Method` may be "HessePencil" (default), "FlexAlgebra" or "SegreEmbedding". However, the choice of `Method` is not expected to make a big difference to the running time.

The optional parameter `ThreeTorsPts` is a tuple containing one representative from each Galois orbit of $E[3] \setminus O$. Its purpose is to fix an embedding of the 3-Selmer group in $A^\times/A^{\times 3}$ (otherwise there is ambiguity when the fields involved have nontrivial automorphisms, or occur more than once). When `ThreeTorsPts` is not specified, `ThreeTorsionPoints(E)` is called (its value is stored internally and used throughout the MAGMA session).

`ThreeIsogenyDescent(E : parameters)`

`Isog`	MAPSCH	*Default :*
Verbose	`Selmer`	*Maximum :* 3
Verbose	`ThreeDescent`	*Maximum :* 3

Given an elliptic curve E over $\mathbf{Q}$ that admits a $\mathbf{Q}$-rational isogeny $E \to E_1$ of degree 3, the function performs "descent by 3-isogenies" on E. This involves computing the Selmer groups attached to the isogeny and its dual, and representing the elements of both Selmer groups as plane cubics, with covering maps of degree 3 to E_1 or E respectively. One cubic is given for each nontrivial pair of inverse Selmer elements, and one covering map is given for each cubic (the other covering map, which can be obtained by composing with the negation map on the elliptic curve, would correspond to the inverse Selmer element.)

There are five returned values, in the following order: a list of curves for $Sel(E \to E_1)$, a corresponding list of covering maps to E_1, a list of curves for the dual isogeny Selmer group $Sel(E_1 \to E)$, a corresponding list of covering maps to E, and finally the isogeny $E \to E_1$.

This function works simply by calling `ThreeIsogenySelmerGroups(E)`, and then calling `ThreeIsogenyDescentCubic` for each Selmer element.

The isogeny $E \to E_1$ may be passed in as `Isog`. If `Isog` is not specified, and E admits more than one such isogeny, then this is chosen at random.

`ThreeIsogenySelmerGroups(E : parameters)`

`Isog`	MAPSCH	*Default :*
Verbose	`Selmer`	*Maximum :* 3

Given an elliptic curve E over the rationals that admits a $\mathbf{Q}$-rational isogeny $E \to E_1$ of degree 3, the function computes the Selmer groups associated to the isogeny, and to its dual isogeny $E_1 \to E$. The Selmer groups are returned as abstract groups, together with maps to the relevant algebra. There are five returned values, in the

following order: the group, and the map, for $E \to E_1$, the group, and the map, for $E_1 \to E$, and finally the isogeny $E \to E_1$.

A bound for the rank of $E(\mathbf{Q})$ can be deduced, by taking the sum of the ranks of the Selmer groups for the two isogenies, and subtracting 1 if the kernel of one of the isogenies consists of rational points.

The isogeny $E \to E_1$ may be passed in as `Isog`. If `Isog` is not specified, and E admits more than one such isogeny, then this is chosen at random.

The algebra in which the Selmer group of a particular isogeny is exhibited is the etale algebra corresponding to the nontrivial points in the kernel of the dual isogeny; it is either a quadratic field, or a Cartesian product of two copies of **Q**.

`ThreeIsogenyDescentCubic(`ϕ`,` α`)`

Verbose `ThreeDescent` *Maximum :* 3

Given an isogeny ϕ of degree 3 between elliptic curves over **Q**, and any element α of $H^1(\mathbf{Q}, E[\phi])$, this function returns a plane cubic curve C representing α, together with a covering map $C \to E$ of degree 3.

The element α is given as an element in the algebra A associated to the Selmer group of ϕ; the algebra can be obtained from `ThreeIsogenySelmerGroups`. $H^1(\mathbf{Q}, E[\phi])$ is represented as the subgroup of $A^\times/(A^\times)^3$ consisting of elements whose norm is a cube.

`ThreeDescentByIsogeny(E)`

Verbose `Selmer` *Maximum :* 3

This performs a full 3-descent, returning models for the nontrivial inverse pairs in the 3-Selmer group and maps to E making them into 3-coverings. The only difference with `ThreeDescent` is that the computation is by first and second 3-isogeny descents. This restricts the number fields used to cubic extensions, rather than the sextic fields which may be needed for the generic `ThreeDescent` routine. The advantage is noticeable for larger examples.

Example H122E16

```
> E := EllipticCurve([0,0,0,0,131241]);
> G,m := TorsionSubgroup(E);
> #G;
1
> Rank(E);
2
> S1,_,S2 := ThreeIsogenyDescent(E);
> #S1,#S2;
0 4
```

This shows that there is nontrivial 3-torsion in Ш on some isogenous curve. Without computing the full 3-Selmer group of E it is impossible to tell if there is any nontrivial 3-torsion in Ш(E/**Q**).

```
> time Sel3 := ThreeDescentByIsogeny(E);
```

```
Time: 5.280
> #Sel3*2+1;
9
```

This shows that $Ш(E/\mathbf{Q})[3] = 0$ (since we know the rank is 2). Alternatively one could get the same by calling `ThreeDescent`. This would require computing class and unit group information in the sextic field below (and would be much slower).

```
> E3reps := ThreeTorsionPoints(E);
> Parent(E3reps[2,1]);
Number Field with defining polynomial x^6 + 14174028 over the Rational Field
>  time Sel3_slow := ThreeDescent(E);
forever!
```

`Jacobian(C)`

Given the equation of a nonsingular projective plane cubic curve C over the rationals, this function returns the Jacobian of C (over the rationals) as an elliptic curve. Note that C will be a principal homogeneous space of E.

`ThreeSelmerElement(E, C)`

`ThreeSelmerElement(C)`

Given an elliptic curve E over the rationals, and a plane cubic C with the same invariants as E (for instance, with E equal to `Jacobian(C)`) the function returns an element α in the algebra A associated to the 3-Selmer group of E. This α represents the same element of $H^1(\mathbf{Q}, E[3])$ that is represented by a covering $C \to E$ that takes the flex points of C to O. Note that α is only determined up to inverse in $H^1(\mathbf{Q}, E[3])$.

In particular, if we have computed `S3, S3toA := ThreeSelmerGroup(E)`, and if C is an everywhere locally soluble covering corresponding to the element `s` in `S3`, then α equals either `S3toA(s)` or `S3toA(-s)` in $A^\times/(A^\times)^3$.

`AddCubics(cubic1, cubic2 : parameters)`

`E`	CrvEll	*Default :*
`ReturnBoth`	BoolElt	*Default :* `false`

Given equations of two plane cubics over the rationals with the same invariants (in other words, they are homogeneous spaces for the same elliptic curve E, which is their Jacobian, over the rationals), the function computes the sum of the corresponding elements of $H^1(\mathbf{Q}, E[3])$. The sum is returned as another plane cubic, if possible (and otherwise an error results).

An element of $H^1(\mathbf{Q}, E[3])$ may be expressed as a plane cubic when it has index 3, equivalently when the so-called "obstruction" is trivial. This is always the case for elements that are everywhere locally soluble. In particular, the function will always succeed when the two given cubics are everywhere locally soluble (in other words, when they belong to the 3-Selmer group). The computation is done by converting

the cubics to elements of $H^1(\mathbf{Q}, E[3])$ as given by `ThreeSelmerElement`, adding the cocycles, and then converting the result to a cubic.

Note that a cubic only determines an element of $H^1(\mathbf{Q}, E[3])$ up to taking inverse, so the sum is not well defined; if the function is called with `ReturnBoth := true`, it returns both of the possible cubics.

`ThreeTorsionType(E)`

For an elliptic curve E over the rationals, this function classifies the Galois action on $E[3]$. The possibilities are "Generic", "2Sylow", "Dihedral", "Generic3Isogeny", "Z/3Z-nonsplit", "mu3-nonsplit", "Diagonal" and "mu3+Z/3Z".

`ThreeTorsionPoints(E : parameters)`

`OptimisedRep` BoolElt *Default :* `true`

For an elliptic curve E over the rationals, this function returns a tuple containing one representative point from each set of Galois conjugates in $E[3] \setminus O$.

Each point belongs to a point set $E(L)$, where L is the field generated by the coordinates of that point.

If `OptimisedRep` is set to `false`, then optimised representations of the fields will not be computed, in general.

`ThreeTorsionMatrices(E, C)`

Given an elliptic curve E over the rationals, and a plane cubic with the same invariants as E (in other words, a principal homogeneous space for E of index 3), the function returns a tuple of matrices. The matrices M_i correspond to the points T_i in `ThreeTorsionPoints(E)`, and have the corresponding base fields. Each matrix describes the action-by-translation on C of the corresponding point: the action of P_i on C is the restriction to C of the automorphism of the ambient projective space $\mathbf{P}^2$ given by the image of M_i in PGL_3.

Note that only the images in PGL_3 of the matrices are well-determined.

122.2.11.1 Six and Twelve Descent

If a 3-descent has been performed, the results can be used in conjunction with a 2-descent or a 4-descent to obtain coverings of degree 6 or 12 respectively. These "combined" coverings can be useful for finding Mordell-Weil generators of large height on the underlying elliptic curve.

The coverings are given as genus one normal curves (of degree 6 in $\mathbf{P}^5$, and of degree 12 in $\mathbf{P}^1 1$, respectively).

The algorithms are described in [Fis08].

`SixDescent(C2, C3)`

`SixDescent(model2, model3)`

Given a 2-covering and a 3-covering of an elliptic curve E (either as curves or as genus one models), this returns the 6-covering that represents their sum in the 6-Selmer group. The covering map $C6 \to C3$ is also returned.

`TwelveDescent(C3, C4)`

`TwelveDescent(model3, model4)`

Given a 3-covering and a 4-covering of an elliptic curve E (either as curves or as genus one models), this returns the two 12-coverings that represent their sum and difference in the 12-Selmer group. The covering maps to $C12 \to C4$ are also returned.

122.2.12 Nine-Descent

A nine-descent is performed on an everywhere locally solvable plane cubic curve. The cubic represents a class in the 3-Selmer group of its Jacobian (up to sign). A nine-descent computes the fibre above this class under the map from the 9-Selmer group to the 3-Selmer group induced by multiplication by 3. This fibre is the set of everywhere locally solvable 3-coverings of the cubic. These coverings are given as intersections of 27 quadrics in $\mathbf{P}^8$. As with four-descents the terminology "nine-descent" is slightly incorrect, as MAGMA performs a second 3-descent on a specific 3-covering and does not try to combine such information from all 3-coverings to form the 9-Selmer group.

The algorithm was developed in the PhD thesis of Brendan Creutz [Cre10]. Typically most of the computation time is spent on class group and unit group computations in the constituent fields of the degree 9 etale algebra associated to the flex points on the cubic. The primary use is to obtain information on the 3-primary part of the Shafarevich–Tate group. It is only in very rare circumstances that a nine descent is required (in addition to the other descent machinery) to determine the Mordell–Weil rank (e.g. if there were elements of order 24 in Ш).

`NineDescent(C : parameters)`

`ExtraReduction`	RNGINTELT	*Default :* 10
Verbose	`Selmer`	*Maximum :* 3
Verbose	`ComputeL`	*Maximum :* 2

Given a plane cubic curve C over $\mathbf{Q}$ this returns a sequence of curves in $\mathbf{P}^8$ defined by 27 quadrics and a list of degree 9 maps making these curves into 3-coverings of C. This is the 3-Selmer set of C (i.e. the set of everywhere locally solvable 3-coverings of C). If there are no coverings, then $C(\mathbf{Q}) = \emptyset$ and the class of C in the Shafarevich-Tate group of its Jacobian is not divisible by 3. Otherwise the coverings returned are lifts of C to the 9-Selmer group.

The coverings returned are minimised and reduced in an ad hoc fashion. If `ExtraReduction` is set to a larger value, more time will be spent reducing possibly resulting in smaller models. The current implementation requires that Galois act transitively on the flex points. If this is not the case, then one can use `pIsogenyDescent` instead.

`NineSelmerSet(C)`

Computes the 3-rank of the 3-Selmer set of plane cubic curve C defined over $\mathbf{Q}$. The value -1 is returned when the 3-Selmer set is empty. In this case $C(\mathbf{Q}) = \emptyset$ and the class of C in the Shafarevich-Tate group of its Jacobian is not divisible by 3. If the 3-Selmer set is nonempty, its 3-rank is the same as that of the 3-Selmer group of the Jacobian. In this case the class of C in the Shafarevich-Tate group of the Jacobian is divisible by 3.

122.2.13 p-Isogeny Descent

Given an isogeny $\phi : E_1 \to E_2$ of elliptic curves, one may define the ϕ-Selmer group of E_2 to be the set of everywhere locally solvable ϕ-coverings of E_2. Given a genus one curve C with Jacobian E_2, one may define its ϕ-Selmer set to be the set of everywhere locally solvable ϕ-coverings of C. A ϕ-isogeny descent computes the ϕ-Selmer group (or ϕ-Selmer set).

We denote the dual isogeny by $\phi^\vee$. In the case of elliptic curves, performing ϕ- and $\phi^\vee$-descents can be used to bound the Mordell-Weil rank and get information on Ш. For general genus one curves descent by isogeny can be used to rule out divisibility in Ш and consequently prove that there are no rational points.

In practice it is often easier to compute a 'fake' Selmer set. This is a set parameterising everywhere locally solvable unions of ϕ-coverings, with the property that every ϕ-covering lies in exactly one such union. Any locally soluble ϕ-covering gives rise to a locally soluble union, hence there is a map from the Selmer set to the fake Selmer set. In general this map may be neither surjective nor injective. However, if the fake Selmer set is empty, then this is also true of the Selmer set.

Current functionality allows one to compute ϕ-Selmer groups for an isogeny ϕ of prime degree in a number of situations. For isogenies of degree 2 or 3, they may be computed using `TwoIsogenyDescent` and `ThreeIsogenyDescent`. When ϕ is the quotient by a subgroup generated by a $\mathbf{Q}$-rational point of order $p \in \{5, 7\}$ the dimensions of the ϕ- and $\phi^\vee$-Selmer groups can be computed using an algorithm of Fisher [Fis00] and [Fis01]. This method also produces models for the everywhere locally solvable $\phi^\vee$-coverings of E.

Given a genus one normal curve C of prime degree p produced by a ϕ-isogeny descent on an elliptic curve, the $\phi^\vee$-Selmer set of C can be computed using an algorithm of Creutz described in [Cre10] and [CM12]. Computing the $\phi^\vee$-Selmer sets of all elements in the ϕ-Selmer group yields information that is equivalent to that given by the $\phi^\vee \circ \phi$-Selmer group.

Current functionality allows one to perform these second isogeny descents when ϕ has degree $p = 3$ or when $p \in \{5, 7\}$ and the flex points of C lie on the coordinate hyperplanes in $\mathbf{P}^{p-1}$. If the kernel of the isogeny is generated by a $\mathbf{Q}$-rational point of order p, then such a model always exists. Moreover, the models returned by the algorithm of Fisher have this property. For $p = 7$, computation of the coverings is not practical and only a 'fake'-Selmer set can be computed (see [CM12]).

`pIsogenyDescent(E,P)`

`pIsogneyDescent(E,p)`

`pIsogenyDescent(lambda,p)`

This performs a ϕ-descent on an elliptic curve over $\mathbf{Q}$ using the algorithm of Fisher. The descent requires an isogeny ϕ whose kernel is generated by a $\mathbf{Q}$-rational point of order $p \in \{5, 7\}$.

The input can be specified in one of three ways: (1) by giving an elliptic curve and a point of order p on the curve; (2) by giving an elliptic curve containing a point of order p and specifying p; or (3) by specifying p and a $\mathbf{Q}$-rational point on the modular curve $X_1(p)$. In the final case the choice for the coordinate λ on $X_1(p)(\mathbf{Q})$ is as described in [Fis00].

The return values are the p-ranks of the ϕ- and $\phi^\vee$-Selmer groups, a sequence of genus one normal curves of degree p representing the inverse pairs of nontrivial elements of the $\phi^\vee$-Selmer group modulo the image of the subgroup generated by the p-torsion point, and the isogenous elliptic curve. If the input is a coordinate on $X_1(p)$, the curve E is also returned.

`pIsogenyDescent(C,phi)`

`pIsogenyDescent(C,E1,E2)`

`pIsogenyDescent(C,P)`

Verbose `Selmer` *Maximum :* 3

This performs an isogeny descent on the genus one normal curve C of degree $p \in \{3, 5\}$ as described in [CM12]. C must be a projective plane cubic (in case $p = 3$) or an intersection of 5 quadrics in $\mathbf{P}^4$, with the additional requirement that the flex points of C lie on the coordinate hyperplanes. For the descent one must specify: (1) the isogeny ϕ; (2) the domain $E1$ and the codomain $E2$ of the isogeny; or (3) a $\mathbf{Q}$-rational point P of order p on an elliptic curve E which generates the kernel of the isogeny.

The return values are a sequence consisting of genus one normal curves of degree p representing the elements of the ϕ-Selmer set and a list of maps making these into ϕ-coverings of C.

`FakeIsogenySelmerSet(C,phi)`

`FakeIsogenySelmerSet(C,E1,E2)`

`FakeIsogenySelmerSet(C,P)`

Verbose `Selmer` *Maximum :* 3

This determines the $\mathbf{F}_p$-dimension of the 'fake'-ϕ-Selmer set of the genus one normal curve C of degree $p \in \{3, 5, 7\}$ (see [CM12] for the definition). By convention the empty set has dimension -1. The input is exactly as for `pIsogenyDescent`, however coverings are not produced. This makes the computations feasible as well for $p = 7$.

Example H122E17

We use a 5-isogeny descent to show that the elliptic curve with Cremona reference 570l3 has rank 0 and that there are nontrivial elements of order 5 in its Shafarevich-Tate group.

```
> lambda := 48/5;
> r_phiSel,r_phidualSel,Sha,E1,E2 := pIsogenyDescent(lambda,5);
> CremonaReference(E1);
570l1
> CremonaReference(E2);
570l3
> TorsionSubgroup(E1);
Abelian Group isomorphic to Z/10
Defined on 1 generator
Relations:
    10*$.1 = 0
> r_phiSel;
0
> r_phidualSel;
3
```

This shows that $E_2(\mathbf{Q})/\phi(E_1(\mathbf{Q})) \simeq 0$, while the $\phi^\vee$-Selmer group gives an exact sequence: $0 \to E_2(\mathbf{Q})/\phi^\vee(E_1(\mathbf{Q})) \subset (\mathbf{Z}/5\mathbf{Z})^3 \to Ш(\mathrm{E}_2/\mathbf{Q})[\phi^\vee] \to 0$. From this we conclude that the rank (for both curves) is 0, and so $Ш(\mathrm{E}_2/\mathbf{Q})[\phi^\vee]$ has $\mathbf{F}_5$-dimension 2. Representatives for the inverse pairs of nontrivial elements are given by the third return value.

```
> #Sha*2 + 1;
25
> C := Sha[1];
> CremonaReference(Jacobian(GenusOneModel(C)));
570l3
```

The above computation does not conclusively show that $Ш(\mathrm{E}_1/\mathbf{Q})$ has no nontrivial 5-torsion. For this we need a second isogeny descent. Namely we show that the C is not divisible by ϕ in Ш.

```
> time DimSelC := FakeIsogenySelmerSet(C,E1,E2);
Time: 2.950
> DimSelC;
-1
```

Example H122E18

Now we use a full 5-descent to find a large generator.

```
> r1,r2,SelModTors,E,EE := pIsogenyDescent(326/467,5);
> r1,r2;
0 2
> E5,m := TorsionSubgroup(E);
> E5;
Abelian Group isomorphic to Z/5
```

```
Defined on 1 generator
Relations:
    5*E5.1 = 0
>  time ConjecturalRegulator(E);
242.013813460415708000138268958 1
Time: 70.180
> Conductor(E);
271976526950
> ThreeTorsionType(E);
Generic
```

This shows that $E(\mathbf{Q}) \simeq \mathbf{Z} \times \mathbf{Z}/5\mathbf{Z}$ and that (assuming Ш(E/**Q**) is trivial) that the regulator is 242.01... This means one probably needs more than a 4-descent to find the generator. The conductor is too large for Heegner point techniques and the 3-torsion on E is generic, meaning 6- or 12-descent will be very slow (even if they are performed nonrigorously). We can do a further isogeny descent to obtain a full 5-covering.

```
> P := m(E5.1);
> C := Curve(Minimize(GenusOneModel(SelModTors[2])));
> time Ds,Pis := pIsogenyDescent(C,P);
7.880
> D := Ds[1];
> pi := Pis[1];
> time rp := PointSearch(D,10^10 :
>     Dimension := 1, OnlyOne := true, Nonsingular := true);
Time: 39.410
> Q := rp[1];
> Q;
(-11610740223/2573365369 : 1350220049/2573365369 :
-110993809/2573365369 : -3970329088/2573365369 : 1)
> piQ := pi(Q); // gives the point on C
> Dnew,Enew,FiveCovering := nCovering(GenusOneModel(D));
> Qnew := Dnew(Rationals())!Eltseq(Q);
> Ecan,EnewtoEcan := MinimalModel(Enew);
> P2 := EnewtoEcan(FiveCovering(Qnew));
> P2;
(187420468938753943863107148788058371189457516723324644302197836255922353361079466416310634589812396922966
1/9914667167291159058241314853870009454120074971804418128933406440554542707100308106416199970088431281 :
7124566254328843223085364743437761031161670909850807708287489814371506949369048945812470939251865125435294284174488796125360323570556418472505448076743676157
8/9872274204002120619421598520816609908998580640514378992620954314037347009617081244656735383786665018644683461670961010967761199736570470693302776180
71 : 1)
> CanonicalHeight(P2);
242.013813460415708000138268957
```

Example H122E19

Finally we give an example which shows that the map from the genuine Selmer set to the fake Selmer set need not be surjective.

```
> E1 := EllipticCurve("254a1");
> E2 := EllipticCurve("254a2");
> bool, phi := IsIsogenous(E1,E2);
> phicoveringsofE2,_,phidualcoveringsofE1,_,phi := ThreeIsogenyDescent(E1);
> C := phicoveringsofE2[1];
> C;
x^3 - x^2*y - x^2*z - 2*x*y^2 + 3*x*y*z - 2*x*z^2 + 2*y^2*z
> phidual := DualIsogeny(phi);
> MinimalModel(Codomain(phidual)) eq MinimalModel(Jacobian(C));
true
> time FakeIsogenySelmerSet(C,phidual);
2
Time: 0.620
> time SelC := pIsogenyDescent(C,phidual);
Time: 1.160
> Ilog(3,#SelC);
1
```

122.2.14 Heegner Points

For an elliptic curve of rank 1, it is possible to compute the generator by an analytic process; the elliptic logarithm of some multiple nP of the generator is the sum of the values of the modular parameterization at a series of points in the upper half-plane corresponding to a full set of class representatives for an appropriately-chosen quadratic field. There is no such thing as a free lunch, sadly; the calculation requires computing $O(hN)$ terms to a precision of $O(h)$ digits, so in practice it works best for curves contrived to have small conductors.

The calculation proceeds in three stages: choosing the quadratic field $\mathbf{Q}(\sqrt{-d})$, evaluating the modular parameterization, and recovering the generator from the elliptic logarithm value. Essentially, this is an implementation of the method of Gross and Zagier [GZ86]; Elkies performed some substantial computations using this method in 1994, and much of the work required to produce this implementation was done by Cremona and Womack. An array of tricks have been added, and are described to some extent in some notes of Watkins.

`HeegnerPoint(E : parameters)`

`NaiveSearch`	RngIntElt	*Default :* 1000
`Discriminant`	RngIntElt	*Default :*
`Cover`	Crv	*Default :*
`DescentPossible`	BoolElt	*Default :* `true`

`IsogenyPossible`	BoolElt	*Default :* `true`
`Traces`	SeqEnum	*Default :* []
Verbose	`Heegner`	*Maximum :* 1

Attempts to find a point on a rank 1 rational elliptic curve E using the method of Heegner points, returning `true` and the point if it finds one, otherwise `false`. The parameter `NaiveSearch` indicates to what height the algorithm will first do a search (using `Points`) before turning to the analytic method. The `Discriminant` parameter allows the user to specify an auxiliary discriminant; this must satisfy the Heegner hypothesis, and the corresponding quadratic twist must have rank 0. The `Cover` option allows the user to specify a 2-cover or 4-cover (perhaps obtained from `TwoDescent` or `FourDescent`) to speed the calculation. This should be either a hyperelliptic curve or a quadric intersection, and there must be a map from the cover to the given elliptic curve. If the `DescentPossible` option is true, then the algorithm might perform such a descent in any case. If the `IsogenyPossible` option is true, then the algorithm will first try to guess the best isogenous curve on which to do the calculation — if a `Cover` is passed to the algorithm, it will be ignored if E is not the best isogenous curve. The `Traces` option takes an array of integers which correspond to the first however-many traces of Frobenius for the elliptic curve, thus saving having to recompute them.

The algorithm assumes that the Manin constant of the given elliptic curve is 1 (which is conjectured in this case), and does not return a generator for the Mordell–Weil group, but a point whose height is that given by a conjectural extension of the Gross–Zagier formula. This should be $\sqrt{Sha}$ times a generator.

`HeegnerPoint(C :` *parameters*`)`

`HeegnerPoint(f :` *parameters*`)`

`NaiveSearch`	RngIntElt	*Default :* 10000
`Discriminant`	RngIntElt	*Default :*
`Traces`	SeqEnum	*Default :* []
Verbose	`Heegner`	*Maximum :* 3

These are utility functions for the above. The input is either a hyperelliptic curve given by a polynomial of degree 4, or this quartic polynomial — in both cases the quartic should have no rational roots — or a nonsingular intersection of two quadrics in $\mathbf{P}^3$. These functions call `HeegnerPoint` on the underlying elliptic curve. They then map the computed point back to the given covering curve. The rational reconstruction step of the `HeegnerPoint` algorithm can be quite time-consuming for some coverings, especially if the index is large. Also, if a cover corresponds to an element of the Tate–Shafarevich group, the algorithm will likely enter an infinite loop. These functions have not been as extensively tested as the ordinary `HeegnerPoint` function, and thus occasionally might fail due to unforeseen problems.

`ModularParametrization(E, z, B :` *parameters*`)`

`ModularParametrization(E, z :` *parameters*`)`

`ModularParametrization(E, Z, B :` *parameters*`)`

`ModularParametrization(E, Z :` *parameters*`)`

`Traces`	SeqEnum	*Default :* []

`ModularParametrization(E, f, B :` *parameters*`)`

`ModularParametrization(E, f :` *parameters*`)`

`ModularParametrization(E, F, B :` *parameters*`)`

`ModularParametrization(E, F :` *parameters*`)`

`Traces`	SeqEnum	*Default :* []
`Precision`	RngIntElt	*Default :*

Given a rational elliptic curve E and a point z in the upper-half-plane, compute the modular parametrization $\int_z^\infty f_E(\tau)\,d\tau$ where f_E is the modular form associated to E. The version with a bound B uses the first B terms of the q-expansion, while the other version determines how many terms are needed. The optional parameter `Traces` allows the user to pass the first however-many traces of Frobenius. There are also versions which take an array Z of complex points, and versions which take positive definite binary quadratic forms f (or an array F) rather than points in the upper-half-plane (a `Precision` can be specified with the latter).

`HeegnerDiscriminants(E,lo,hi)`

`Fundamental`	BoolElt	*Default :* `false`
`Strong`	BoolElt	*Default :* `false`

Given a rational elliptic curve and a range from lo to hi, compute the negative fundamental discriminant in this range that satisfies the Heegner hypothesis for the curve. The `Fundamental` option restricts to fundamental discriminants. The `Strong` option restricts to discriminants that satisfy a stronger Heegner hypothesis, namely that $a_p = -1$ for all primes p that divide $\gcd(D, N)$.

`HeegnerForms(E,D :` *parameters*`)`

`UsePairing`	BoolElt	*Default :* `false`
`UseAtkinLehner`	BoolElt	*Default :* `true`
`Use_wQ`	BoolElt	*Default :* `true`
`IgnoreTorsion`	BoolElt	*Default :* `false`

Given a rational elliptic curve of conductor N and a negative discriminant that meets the Heegner hypothesis for N, the function computes representatives for the complex multiplication points on $X_0(N)$.

In general the return value is a sequence of 3-tuples (Q, m, T) where Q is a quadratic form, m is a multiplicity, and T is a torsion point on E. Letting ϕ be

the modular parametrization map, the sum on E of the values $m(\phi(Q)+T)$ is a Heegner point in $E(\mathbf{Q})$. When the number of these values equals the class number $h = h(D)$ (and the multiplicities are all 1 or -1) then they are actually the images on E of the CM points on $X_0(N)$. (They all correspond to a single choice of square root of D modulo $4N$.)

If `UsePairing` is added, then CM points that give conjugate values under the modular parametrisation map are combined. If `UseAtkinLehner` is added, then more than one square root of D modulo $4N$ can be used. If `Use_wQ` is added, then forms can appear with extra multiplicity, due to primes that divide the GCD of the conductor and the fundamental discriminant. If `IgnoreTorsion` is added, then the fact that Atkin-Lehner can change the result by a torsion point will be ignored. The `UsePairing` option requires that `IgnoreTorsion` be true.

This function requires (so as not to confuse the user) that the ManinConstant of the curve be equal to 1.

`HeegnerForms(N,D : parameters)`

`AtkinLehner`	[RngIntElt]	*Default :* []

Given a level N and a discriminant that meets the Heegner hypothesis for the level, the function returns a sequence of binary quadratic forms which correspond to the Heegner points. The Atkin-Lehner option takes a sequence of q with $\gcd(q, N/q) = 1$ and has the effect of allowing more than one square root of D modulo $4N$ to be used.

`ManinConstant(E)`

Compute the Manin constant of a rational elliptic curve. This is in most cases simply a conjectural value (and most often just 1).

`HeegnerTorsionElement(E)`

Given a rational elliptic curve and an integer Q corresponding to an Atkin-Lehner involution (so that $\gcd(Q, N/Q) = 1$), this function computes the torsion point on the curve corresponding to the period given by the integral from $i\infty$ to $w_Q(i\infty)$.

`HeegnerPoints(E, D : parameters)`

`ReturnPoint`	BoolElt	*Default :* `false`
`Precision`	RngIntElt	*Default :* 100
Verbose	`Heegner`	*Maximum :* 1

Given an elliptic curve E over $\mathbf{Q}$, and a suitable discriminant D (of a quadratic field) this function computes the images on E under the modular parametrization of the CM points on $X_0(N)$ associated to D. It returns a tuple $\langle p_D, m\rangle$, where p_D is an irreducible polynomial whose roots are the x-coordinates of these images, and where m is the multiplicity of each of these images (the number of CM points on $X_0(N)$ mapping to a given point on E). These x-coordinates lie in the ring class field of $\mathbf{Q}(\sqrt{D})$, and the class number $h(D)$ must equal either $m\deg(p_D)$ or $2m\deg(p_D)$.

The conductor of the order $\mathbf{Q}(\sqrt{D})$ is required to be coprime to the conductor of E.

The second object returned by the function is one of the conjugate points, as an element of the point set $E(H)$ where H is the field defined by p_D, or a quadratic extension of it. This step can be time consuming; if `ReturnPoint` is set to `false`, the point is not computed.

Warning: The computation is not rigorous, as it involves computing over the complex numbers, and then recognising the coefficients of the polynomial as rationals. However, the program performs a heuristic check: it checks that the polynomial has the correct splitting at some small primes (sufficiently many to be sure that wrong answers will not occur in practice).

Example H122E20

```
> time HeegnerPoint(EllipticCurve([1,0,0,312,-3008])); //uses search
true (12 : -56 : 1)
Time: 0.240
> time HeegnerPoint(EllipticCurve([0,0,1,-22787553,-41873464535]));
true (1100363782947843220359298466100412 9/1048524607168660222036584535396 :
-1004181871409718654255966342764883958203316448270339/10736628855783147946
99393270058310986889998056 : 1)
Time: 1.380
```

Example H122E21

Here are some more complicated examples. In the first one, the curve has a 163-isogenous curve, which the algorithm uses to speed the computation. This occurs automatically, with most of the time taken being in computing the isogeny map.

```
> E := EllipticCurve([0,0,1,-57772164980,-5344733777551611]);
> b, pt:= HeegnerPoint(E);
> Height(pt);
373.47866156914237988407748041 2
```

Example H122E22

In this next example, we first use descent to get a covering on which the desired point will have smaller height.

```
> E := EllipticCurve([0,-1,0,-71582788120,-7371563751267600]);
> T := TwoDescent(E : RemoveTorsion)[1];
> T;
Hyperelliptic Curve defined by y^2 = -2896*x^4 - 57928*x^3 - 202741*x^2
 + 868870*x - 651725 over Rational Field
> S := FourDescent(T : RemoveTorsion)[1];
> b, pt := HeegnerPoint(S);
> pt;
```

```
(-34940281640330765977951793/729633174814530114300522 32 :
46087465795237503244048957/729633174814530114300522 32 :
82501230298438806677528297/729633174814530114300522 32 : 1)
```

We obtain a point on S, not on the original curve. We map it to E using the descent machinery.

```
> _, m := AssociatedEllipticCurve(S);
> PT := m(pt);
> PT;
(269356432396748248246118697936017740034773039452236777412444550587 53
924460587724041316046901475913814274843012625394997597714766923750555
669810857471061235926971161094484197799515212721830555528087646969545
65/837619099331786545303500955405701693904657590793269053332825491870
645799230089011267382251975387071464373622714525213870033427638236014
228801250428843907758749650143692004410924389857019093192586024416 4 :
410189886613094359515065087530226951251222017120181558101276037601794
678635473792334597914052557787798899390943937170051840037860568689664
871351793404398352109768736545284569745952273382778021947446766526118
621612003004627401344216069791103330332004546699363266079516476593864
985383039985673798699741432591743 95/766601839981278884919411893932635
388671474045058772153268571977045787439290021029065740244648077391646
579430077900352635000328805236464794479797855430820809227462762666048
009219642700664370632930446228302292313415544188481623273194456758440
446505454248062292834244615276350323795396202280072306278575288 : 1)
> Height(PT);
476.811182818720336949724781780
> ConjecturalRegulator(E : Precision := 5);
476.81 1
```

Example H122E23

Finally, we do some Heegner point calculation with the curve 43A and the discriminant -327. Note that the obtained trace down to the rationals is 3-divisible, but the point over the Hilbert class field is not.

```
> E := EllipticCurve([0,1,1,0,0]);
> HeegnerDiscriminants(E,-350,-300);
[ -347, -344, -340, -335, -331, -328, -327, -323, -319, -308, -303 ]
> HF := HeegnerForms(E,-327); HF;
[ <<43,19,4>, 1, (0 : 1 : 0)>, <<43,67,28>, 1, (0 : 1 : 0)>,
<<86,105,33>, 1, (0 : 1 : 0)>, <<86,153,69>, 1, (0 : 1 : 0)>,
<<129,105,22>, 1, (0 : 1 : 0)>, <<129,153,46>, 1, (0 : 1 : 0)>,
<<172,67,7>, 1, (0 : 1 : 0)>, <<258,363,128>, 1, (0 : 1 : 0)>,
<<258,411,164>, 1, (0 : 1 : 0)>, <<301,67,4>, 1, (0 : 1 : 0)>,
<<473,621,204>, 1, (0 : 1 : 0)>, <<473,841,374>, 1, (0 : 1 : 0)> ]
> H := [x[1] : x in HF]; mul := [x[2] : x in HF];
> params := ModularParametrization(E,H : Precision := 10);
> wparams := [ mul[i]*params[i] : i in [1..#H]];
> hgpt := EllipticExponential(E,&+wparams);
```

```
> hgpt;
[ 1.000000002 - 1.098466121E-9*I, 1.000000003 - 1.830776870E-9*I ]
> HeegnerPt := E![1,1];
> DivisionPoints(HeegnerPt, 3);
[ (0 : -1 : 1) ]
```

So the Heegner point is 3 times $(0, -1)$ in $E(\mathbf{Q})$. We now find algebraically one of the points (of which we took the trace to get $HeegnerPt$). This point will be defined over a subfield of the class field of $\mathbf{Q}(\sqrt{-327})$, and will have degree dividing the class number. The *poly* below is the minimal polynomial of the x-coordinate.

```
> ClassNumber(QuadraticField(-327));
12
> poly, pt := HeegnerPoints(E, -327 : ReturnPoint);
> poly;
<t^12 + 10*t^11 + 76*t^10 - 1150*t^9 + 475*t^8 - 4823*t^7 +
 997*t^6 - 5049*t^5 - 2418*t^4 - 468*t^3 - 3006*t^2 + 405*t - 675, 1>
> pt;
(u : 1/16976844562625*(58475062076*u^11 + 568781661961*u^10 +
 4238812569862*u^9 - 68932294336288*u^8 + 42534102728187*u^7 -
 238141610215111*u^6 + 134503848441911*u^5 - 122884262733563*u^4 -
 58148031982456*u^3 + 129014145075851*u^2 -
 68190988248855*u + 40320643901175) : 1)
> DivisionPoints(pt,3);
[]
```

Here is another example of Heegner points over class fields.

Example H122E24

```
> E := EllipticCurve([0,-1,0,-116,-520]);
> Conductor(E);
1460
> ConjecturalRegulator(E);
4.488874747706661735767268062 64 1
> HeegnerDiscriminants(E,-200,-100);
[ -119, -111 ]
> P := HeegnerPoints(E,-119);
> P;
<1227609449333996689*t^10 - 10626150637714398460 3*t^9 +
    24598266676939452036 84*t^8 - 12539974356047058417320*t^7 -
    29852470882365441140 8343*t^6 + 44408766844345979261611 75*t^5 +
    75735490991206189798332 41*t^4 - 39393804886040638610811 3130*t^3 -
    21531810713569162829866 8863*t^2 + 13958289016298162706904 004974*t
    + 38624559371249968900024945369, 1>
```

The function automatically performs a heuristic check that the polynomial has the right properties, using reduction mod small primes. A more expensive check is the following.

```
> G := GaloisGroup( P[1] );
```

```
> IsIsomorphic(G,DihedralGroup(10));
true
```

The extension of $\mathbf{Q}(-119)$ defined by the polynomial is the class field. We now obtain a nice representation of it, and use this to compute the height of the point (which is a bit quicker than computing the height directly).

```
> K<u> := NumberField(P[1]);
> L<v>, m := OptimizedRepresentation(K);
> _<y> := PolynomialRing(Rationals());  // use 'y' for printing
> DefiningPolynomial(L);
y^10 + 2*y^9 - y^8 - 7*y^7 - 7*y^6 + 10*y^5 + 13*y^4 - 9*y^3 - 5*y^2 +
    5*y - 1
> PT := Points(ChangeRing(E,L),m(u))[1];  // y-coord is defined over L
> Height(PT);
6.97911761109714376876370533
```

122.2.15 Analytic Information

`Periods(E: parameters)`

`Precision` RngIntElt *Default :*

Returns the sequence of periods of the Weierstrass $\wp$-function associated to the (rational) elliptic curve E, to `Precision` digits. The first element of the sequence is the real period. The function accepts a non-minimal model, and returns the periods corresponding to that model. As with many algorithms involving analytic information on elliptic curves, the implementation exploits an AGM-trick due to Mestre. There is some functionality for elliptic curves over number fields, though the exact normalisation is not always given.

`EllipticCurveFromPeriods(om: parameters)`

`Epsilon` FldReElt *Default :* 0.001

Given two complex numbers ω_1, ω_2 such that ω_2/ω_1 is in the upper half-plane that correspond to an integral model of an elliptic curve over $\mathbf{Q}$, return such a minimal model of such a curve. This uses the classical `Eisenstein` series, with the vararg `Epsilon` indicating how close to integers the computed $-27c_4$ and $-54c_6$ need to be.

`RealPeriod(E: parameters)`

`Precision` RngIntElt *Default :*

Returns the real period of the Weierstrass $\wp$-function associated to the elliptic curve E to `Precision` digits.

`EllipticExponential(E, z)`

Given a rational elliptic curve E and a complex number z, the function computes the pair $[\wp(z), \wp'(z)]$ where $\wp(s)$ is the Weierstrass $\wp$-function. The algorithm used is taken from [Coh93] for small precision, and Newton iteration on `EllipticLogarithm` is used for for high precision. The function returns a sequence rather than a point on the curve.

`EllipticExponential(E, S)`

Given a rational elliptic curve E and a sequence $S = [p, q]$, where p and q are rational numbers, this function computes the elliptic exponential of p times the `RealPeriod` and q times the imaginary period.

`EllipticLogarithm(P:` *parameters*`)`

`Precision`	RngIntElt	*Default :*

Denote by ω_1, ω_2 the periods of the Weierstrass $\wp$-function related to E. This function returns the elliptic logarithm $\phi(P)$ of the point P, such that $-\omega_1/2 \leq \mathrm{Re}(\phi(P)) < \omega_1/2$ and $-\omega_2/2 \leq \mathrm{Im}(\phi(P)) < \omega_2/2$. The value is returned to `Precision` digits. As with `Periods`, a non-minimal model can be given, and the `EllipticLogarithm` will be computed with respect to it. The algorithm is again an AGM-trick due to Mestre.

`EllipticLogarithm(E, S)`

`Precision`	RngIntElt	*Default :*
`Check`	BoolElt	*Default :* `true`

Given an elliptic curve E and a sequence $S = [z_1, z_2]$, where z_1 and z_2 are complex numbers approximating a point P on E, this function returns the elliptic logarithm $\phi(P)$. For details see the previous intrinsic. When `Check` is false, z_1 and z_2 need not have any relation to a point on the curve, in which case only the x-coordinate matters up to (essentially) a choice of sign in the resulting logarithm.

`pAdicEllipticLogarithm(P, p:` *parameters*`)`

`Precision`	RngIntElt	*Default :* 50

For a point P on an elliptic curve E which is a minimal model and a prime p, returns the p-adic elliptic logarithm of P to `Precision` digits. The order of P must not be a power of p.

Example H122E25

Verify that `EllipticExponential` and `EllipticLogarithm` are inverses.

```
> C<I>:=ComplexField(96);
> E:=EllipticCurve([0,1,1,0,0]);
> P:=EllipticExponential(E,0.571+0.221*I); P;
[ 1.6569476052101862388682981 75 + -1.7854401800676814186186959 47*I,
```

```
-2.417077284700315330385200257 + 3.894153792661208360153853530*I ]
> EllipticLogarithm(E,P);
0.571000000000000000000000000 + 0.221000000000000000000000000*I
```

Add points on a curve via Napier's method.

```
> E:=EllipticCurve([0,0,1,-7,6]);
> P1:=E![0,2]; P2:=E![2,0]; P3:=E![1,0];
> L:=EllipticLogarithm(E!P1)+EllipticLogarithm(E!P2)+EllipticLogarithm(E!P3);
> L;
0.52174821975779076985312019793 7 + 1.480548268241414993307331110 64*i
> P:=EllipticExponential(E,L);
> [Round(Real(x)) : x in P];
[ 4, -7 ]
> P1+P2+P3;
(4 : -7 : 1)
```

`RootNumber(E)`

Calculates the global root number of an elliptic curve E defined over $\mathbf{Q}$. This is equal to the sign in the functional equation of the L-series associated to the curve. The method used is to take a product of local root numbers over the primes of bad reduction.

`RootNumber(E, p)`

Calculates the local root number at a prime p of an elliptic curve E. The method used is due to Halberstadt; for $p > 3$ it is a straightforward calculation involving the valuation of the discriminant, while for $p = 2, 3$ it involves a more careful analysis of the reduction type.

`AnalyticRank(E)`

Verbose	`AnalyticRank`	*Maximum :* 1
`Precision`	RngIntElt	*Default :* 5

Determine the analytic rank of the rational elliptic curve E. The algorithm used is heuristic, computing derivatives of the L-function $L(E, s)$ at $s = 1$ until one appears to be nonzero. Local power series methods are used to compute the special functions used for higher derivatives. The function returns the first nonzero derivative $L^{(r)}(1)/r!$ as a second argument. The precision is optional, and is taken to be 17 bits if omitted; the time taken can increase dramatically if this is increased much beyond 50 digits as the time to compute the special functions then starts to dominate the running time.

`ConjecturalRegulator(E)`

`Precision` RngIntElt *Default :*

Using the `AnalyticRank` function, this function calculates an approximation, assuming that the Birch–Swinnerton-Dyer conjecture holds, to the product of the regulator of the elliptic curve E and the order of the Tate–Shafarevich group. The (assumed) analytic rank is returned as a second value.

`ConjecturalRegulator(E, v)`

Given an elliptic curve E with L-function $L(E, s)$ and the value v of the first nonzero derivative $L^{(r)}(1)/r!$, this function calculates an approximation, assuming that the Birch–Swinnerton-Dyer conjecture holds, to the product of the regulator of the elliptic curve E and the order of its Tate–Shafarevich group.

Example H122E26

```
> E:=EllipticCurve([1,1,0,-2582,48720]);
> time r, Lvalue := AnalyticRank(E : Precision:=9); r,Lvalue;
Time: 17.930
6 320.781164
> ConjecturalRegulator(E,Lvalue);
68.2713770
> time G, map := MordellWeilGroup(E : HeightBound := 8); G;
Time: 21.540
Abelian Group isomorphic to Z (6 copies)
Defined on 6 generators (free)
> Regulator([map(g): g in OrderedGenerators(G)]);
68.2713769
```

Example H122E27

```
> ConjecturalRegulator(EllipticCurve([0,-1,1,0,0]) : Precision := 8);
1.0000000 0
> ConjecturalRegulator(EllipticCurve([0,-1,1,0,0]) : Precision := 21);
1.00000000000000000000 0
> ConjecturalRegulator(EllipticCurve([0,-1,1,0,0]) : Precision := 35);
1.0000000000000000000000000000000000 0
> ConjecturalRegulator(EllipticCurve([0,-1,1,0,0]) : Precision := 50);
1.00000000000000000000000000000000000000000000000000 0
```

`ModularDegree(E)`

Verbose `ModularDegree` *Maximum :* 1

Determine the modular degree of a rational elliptic curve E. The algorithm used is described in [Wat02]. One computes the special value $L(Sym^2 E, 2)$ of the motivic symmetric-square L-function of the elliptic curve, and uses the formula

$$deg(phi) = L(Sym^2 E, 2)/(2 * Pi * \Omega) * (Nc^2) * E_p(2)$$

where Ω is the area of the fundamental parallelogram of the curve, N is the conductor, c is the Manin constant, and $E_p(2)$ is a product over primes whose square divides the conductor. The Manin constant is assumed to be 1 except in the cases described in [SW02], where it is conjectured that the optimal curves for parameterizations from $X_1(N)$ and $X_0(N)$ are different. A warning is given in these cases. The optimal curve for parameterizations from $X_1(N)$ is assumed to be the curve in the isogeny class of E that has minimal Faltings height (maximal Ω). The algorithm is based upon a sequence of real-number approximations converging to an integer — the use of verbose printing for `ModularDegree` allows the user to see the sequence of approximations.

Example H122E28

First we define a space of ModularForms.

```
> M:=ModularForms(Gamma0(389),2);
```

The first of the newforms associated to M corresponds to an elliptic curve.

```
> f := Newform(M,1); f;
q - 2*q^2 - 2*q^3 + 2*q^4 - 3*q^5 + 4*q^6 - 5*q^7 + q^9 + 6*q^10 - 4*q^11 +
O(q^12)
> E := EllipticCurve(f); E;
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

We can compute the modular degree of this using modular symbols.

```
> time ModularDegree(ModularSymbols(f));
40
Time: 0.200
```

Or via the algorithm based on elliptic curves.

```
> time ModularDegree(E);
40
Time: 0.000
```

The elliptic curve algorithm is capable of handling examples of high level, particularly when bad primes have multiplicative reduction.

```
> E := EllipticCurve([0,0,0,0,-(10^4+9)]);
> Conductor(E);
```

```
14425931664
> time ModularDegree(E);
6035544576
Time: 3.100
```

122.2.16 Integral and S-integral Points

Let E be an elliptic curve defined over the rational numbers $\mathbf{Q}$ and denote by $S = \{p_1, \ldots, p_{s-1}, \infty\}$ a finite set of primes containing the prime at infinity. There are only finitely many S-integral points on E. (That is, points where the denominators of the coordinates are only supported by primes in S.) Note that the point at infinity on E is never returned, since it is supported at every prime.

The following algorithms use the technique of linear forms in complex and p-adic elliptic logarithms.

The main routine here is `(S)IntegralPoints` for an elliptic curves. The functions listed afterwards, for determining integral points on various other kinds of genus one curves, are applications of the main routine.

`IntegralPoints(E)`

`FBasis`	[PtEll]	*Default :*
`SafetyFactor`	RngIntElt	*Default :*

Given an elliptic curve E over the $\mathbf{Q}$, this returns a sequence containing all the integral points on E, modulo negation. Secondly, a sequence is returned containing representations of the same points in terms of a fixed basis of the Mordell-Weil group (each such representation is a sequence of tuples of the form $[\langle P_i, n_i \rangle]$).

The algorithm involves first computing generators of the Mordell-Weil group, by calling `MordellWeilShaInformation` which accesses the appropriate tools available in Magma. Alternatively, the user may precompute generators and pass them to `IntegralPoints` as the optional parameter `FBasis`. This should be a sequence of points on E that are independent modulo torsion (for instance, as returned by `ReducedBasis`). `IntegralPoints` will then find all integral points on E that are in the group generated by `FBasis` and torsion.

If the optional argument `SafetyFactor` is specified, the search phase at the final step is extended as a safety check. (The height bound is increased in such a way that the search region expands by roughly the specified factor.)

`SIntegralPoints(E, S)`

`FBasis`	[PtEll]	*Default :*
`SafetyFactor`	RngIntElt	*Default :*

Given an elliptic curve E over the rationals and a set S of finite primes, returns the sequence of all S-integral points, modulo negation. The second return value, and the optional arguments `FBasis` and `SafetyFactor`, are the same as in `IntegralPoints`.

Example H122E29

We find all integral points on a certain elliptic curve:

```
> E := EllipticCurve([0, 17]);
> Q, reps := IntegralPoints(E);
> Q;
[ (-2 : -3 : 1), (8 : 23 : 1), (43 : -282 : 1), (4 : 9 : 1),
(2 : -5 : 1), (-1 : 4 : 1), (52 : -375 : 1), (5234 : 378661 : 1) ]
> reps;
[
    [ <(-2 : -3 : 1), 1> ],
    [ <(-2 : -3 : 1), 2> ],
    [ <(-2 : -3 : 1), 1>, <(4 : -9 : 1), -2> ],
    [ <(4 : -9 : 1), -1> ],
    [ <(-2 : -3 : 1), 1>, <(4 : -9 : 1), -1> ],
    [ <(-2 : -3 : 1), 1>, <(4 : -9 : 1), 1> ],
    [ <(-2 : -3 : 1), 2>, <(4 : -9 : 1), 1> ],
    [ <(-2 : -3 : 1), 1>, <(4 : -9 : 1), 3> ]
]
```

We see here that the chosen basis consists of the points $(-2 : -3 : 1)$ and $(4 : -9 : 1)$, and that coefficients which are zero are omitted.

Example H122E30

We find all S-integral points on an elliptic curve of rank 2, for the set of primes $S = \{2, 3, 5, 7\}$.

```
> E := EllipticCurve([-228, 848]);
> Q := SIntegralPoints(E, [2, 3, 5, 7]);
> for P in Q do P; end for;    // Print one per line
(4 : 0 : 1)
(-11 : 45 : 1)
(16 : 36 : 1)
(97/4 : -783/8 : 1)
(-44/9 : -1160/27 : 1)
(857/4 : -25027/8 : 1)
(6361/400 : -282141/8000 : 1)
(534256 : -390502764 : 1)
(946/49 : -20700/343 : 1)
(-194/25 : -5796/125 : 1)
(34/9 : 172/27 : 1)
(814 : 23220 : 1)
(13 : 9 : 1)
(-16 : 20 : 1)
(1/4 : -225/8 : 1)
(52 : -360 : 1)
(53 : 371 : 1)
(16/49 : 9540/343 : 1)
(-16439/1024 : -631035/32768 : 1)
```

```
(34 : 180 : 1)
(-2 : 36 : 1)
(-14 : -36 : 1)
(14 : -20 : 1)
(754 : -20700 : 1)
(94/25 : -828/125 : 1)
(2 : 20 : 1)
(94 : 900 : 1)
(-818/49 : 468/343 : 1)
(49/16 : -855/64 : 1)
(196 : -2736 : 1)
(629/25 : 13133/125 : 1)
(1534/81 : -42020/729 : 1)
(8516/117649 : -1163623840/40353607 : 1)
```

`IntegralQuarticPoints(Q)`

If Q is a sequence of five integers $[a, b, c, d, e]$ where e is a square, this function returns all integral points (modulo negation) on the curve $y^2 = ax^4 + bx^3 + cx^2 + dx + e$.

`IntegralQuarticPoints(Q, P)`

If Q is a list of five integers $[a, b, c, d, e]$ defining the hyperelliptic quartic $y^2 = ax^4 + bx^3 + cx^2 + dx + e$ and P is a sequence representing a rational point $[x, y]$, this function returns all integral points on Q.

`SIntegralQuarticPoints(Q, S)`

Given a sequence of integers $Q = [a, b, c, d, e]$ where a is a square, this function returns all S-integral points on the quartic curve $y^2 = ax^4 + bx^3 + cx^2 + dx + e$ for the set S of finite primes.

Example H122E31

We find all integral points (modulo negation) on the curve $y^2 = x^4 - 8x^2 + 8x + 1$. Since the constant term is a square we can use the first form and do not have to provide a point as well.

```
> IntegralQuarticPoints([1, 0, -8, 8, 1]);
[
    [ 2, -1 ],
    [ -6, 31 ],
    [ 0, 1 ]
]
```

`SIntegralLjunggrenPoints(Q, S)`

Given a sequence of integers $Q = [a, b, c, d]$, this function returns all S-integral points on the curve $C : ay^2 = bx^4 + cx^2 + d$ for the set S of finite primes, provided that C is nonsingular.

`SIntegralDesbovesPoints(Q, S)`

Given a sequence of integers $Q = [a, b, c, d]$, this function returns all S-integral points on $C : ay^3 + bx^3 + cxy + d = 0$ for the set S of finite primes, provided that C is nonsingular.

Example H122E32

We find the points supported by $[2, 3, 5, 7]$ on the curve $9y^3 + 2x^3 + xy + 8 = 0$.

```
> S := [2, 3, 5, 7];
> SIntegralDesbovesPoints([9, 2, 1, 8], S);
[
    [ 1, -1 ],
    [ -94/7, 172/21 ],
    [ -11/7, 2/7 ],
    [ 5, -3 ],
    [ 2, -4/3 ],
    [ 2/7, -20/21 ],
    [ -8/5, -2/15 ]
]
```

122.2.17 Elliptic Curve Database

MAGMA includes John Cremona's database of all elliptic curves over **Q** of small conductor (up to 200 000 as of December 2011). This section defines the interface to that database.

For each conductor in the range stored in the database the curves with that conductor are stored in a number of isogeny classes. Each curve in an isogeny class is isogenous (but not isomorphic) to the other curves in that class. Isogeny classes are referred to by number rather than the alphabetic form given in the Cremona tables.

All of the stored curves are global minimal models.

`EllipticCurveDatabase(: parameters)`

`CremonaDatabase(: parameters)`

`BufferSize` RNGINTELT *Default :* 10000

This function returns a database object which contains information about the elliptic curve database and is used in the functions which access it. The optional parameter `BufferSize` controls the size of an internal buffer — see the description of `SetBufferSize` for more information.

`SetBufferSize(D, n)`

The elliptic curve database D uses an internal buffer to cache disk reads; if the buffer is large enough then the entire file can be cached and will not need to be read from the disk more than once. On the other hand, if only a few curves will be accessed then a large buffer is not especially useful. `SetBufferSize` can be used to set the size n (in bytes) of this buffer. There are well defined useful minimum and maximum sizes for this buffer, and values outside this range will be treated as the nearest useful value.

`LargestConductor(D)`

Returns the largest conductor of any elliptic curve stored in the database. It is an error to attempt to refer to larger conductors in the database.

`ConductorRange(D)`

Returns the smallest and largest conductors stored in the database. It is an error to attempt to refer to conductors outside of this range.

`#D`

`NumberOfCurves(D)`

Returns the number of elliptic curves stored in the database.

`NumberOfCurves(D, N)`

Returns the number of elliptic curves stored in the database for conductor N.

`NumberOfCurves(D, N, i)`

Returns the number of elliptic curves stored in the database in the i-th isogeny class for conductor N.

`NumberOfIsogenyClasses(D, N)`

Returns the number of isogeny classes stored in the database for conductor N.

`EllipticCurve(D, N, I, J)`

`EllipticCurve(D, N, S, J)`

Returns the J-th elliptic curve of the I-th isogeny class of conductor N from the database. I may be specified either as an integer (first form) or as a label like `"A"` (second form).

`EllipticCurve(D, S)`

`EllipticCurve(S)`

Returns a representative elliptic curve with label S (e.g., `"101a"` or `"101a1"`) from the specified database (or if not specified, from the Cremona database).

`Random(D)`

Returns a random curve from the database.

`CremonaReference(D, E)`

`CremonaReference(E)`

Returns the database reference to the minimal model for E (e.g., `"101a1"`). E must be defined over $\mathbf{Q}$ and its conductor must lie within the range of the database. The second form of this function must open the database for each call, so if it is being used many times the database should be created once and the first form used instead.

Example H122E33

```
> D := CremonaDatabase();
> #D;
847550
> minC, maxC := ConductorRange(D);
> minC, maxC;
1 130000
> &+[ NumberOfCurves(D, C) : C in [ minC .. maxC ] ];
847550
```

These numbers agree (which is nice). The conductor in that range with the most curves is 100800.

```
> S := [ NumberOfCurves(D, C) : C in [ minC .. maxC ] ];
> cond := maxval + minC - 1 where _,maxval := Max(S);
> cond;
100800
> NumberOfCurves(D, cond);
924
> NumberOfIsogenyClasses(D, cond);
418
```

The unique curve of conductor 5077 has rank 3.

```
> NumberOfCurves(D, 5077);
1
> E := EllipticCurve(D, 5077, 1, 1);
> E;
Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
> CremonaReference(D, E);
5077a1
> Rank(E);
3
```

`EllipticCurves(D, N, I)`

`EllipticCurves(D, N, S)`

Returns the sequence of elliptic curves in the I-th isogeny class for conductor N from the database. I may be specified either as an integer (first form) or as a label like `"A"` (second form).

`EllipticCurves(D, N)`

The sequence of elliptic curves for conductor N from the database.

`EllipticCurves(D, S)`

The sequence of elliptic curves with label S from the database. S may specify just a conductor (like `"101"`), or both a conductor and an isogeny class (like `"101A"`).

`EllipticCurves(D)`

The sequence of elliptic curves stored in the database. Note: this function is extremely slow due to the number of curves involved. Where possible it would be much better to iterate through the database instead (see example).

Example H122E34

Here are two ways to iterate through the database:

```
> D := CremonaDatabase();
25508696848625044861003843159331265666415824
Time: 21.130
> sum := 0;
> time for E in D do sum +:= Discriminant(E); end for;
> sum;
25508696848625044861003843159331265666415824
Time: 20.060
```

Now we create a random curve and find the other curves in its isogeny class.

```
> E := Random(D);
> E;
Elliptic Curve defined by y^2 = x^3 - 225*x over Rational Field
> Conductor(E);
7200
> CremonaReference(E);
7200bg1
> EllipticCurves(D, "7200bg");
[
    Elliptic Curve defined by y^2 = x^3 - 225*x over Rational Field,
    Elliptic Curve defined by y^2 = x^3 - 2475*x - 47250 over Rational Field,
    Elliptic Curve defined by y^2 = x^3 - 2475*x + 47250 over Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 900*x over Rational Field
]
```

122.3 Curves over Number Fields

The functions in this section are for elliptic curves defined over number fields. For the most part, the functionality is a subset of that available for curves over **Q**, with functions having similar names and arguments.

The main items implemented are Tate's algorithm, 2-descent and descent by 2-isogenies, the Cassels-Tate pairing, height machinery, and analytic tools.

122.3.1 Local Invariants

The routines listed here, when nontrivial, are based on an implementation of Tate's algorithm.

`Conductor(E)`

The conductor is part of the data computed by `LocalInformation` (described below).

`BadPlaces(E)`

Given an elliptic curve E defined over a number field K, returns the places of K of bad reduction for E. i.e., the places dividing the discriminant of E.

`BadPlaces(E, L)`

Given an elliptic curve E defined over a number field K and a number field L, such that K is a subfield of L, returns the places of L of bad reduction for E.

`LocalInformation(E, P)`

`UseGeneratorAsUniformiser`

BoolElt *Default :* `false`

Implements Tate's algorithm for the elliptic curve E over a number field. This intrinsic computes local reduction data at the prime ideal P, and a local minimal model. The model is not required to be integral on input. Output is $\langle P, v_p(d), f_p, c_p, K, s\rangle$ and E_{min} where P is the prime ideal, $v_p(d)$ is the valuation of the local minimal discriminant, f_p is the valuation of the conductor, c_p is the Tamagawa number, K is the Kodaira Symbol, and s is `false` if the curve has non-split multiplicative reduction at P and `true` otherwise. E_{min} is a model of E (integral and) minimal at P.

When the optional parameter `UseGeneratorAsUniformiser` is set `true`, the computation checks whether P is principal, and if so, uses generator of P as the uniformiser. This means that at primes other that P, the returned model will still be integral or minimal if the given curve was.

`LocalInformation(E)`

Return a sequence of the tuples returned by `LocalInformation(E, P)` for all prime ideals P in the decomposition of the discriminant of E in the maximal order of the number field the elliptic curve E is over.

`Reduction(E, p)`

Given an elliptic curve E over a number field given by a model which is integral at p and has good reduction at p, returns an elliptic curve over the residue field of p which represents the reduction of E at p. The reduction map is also returned.

122.3.2 Complex Multiplication

`HasComplexMultiplication(E)`

Given an elliptic curve E over a number field, the function determines whether the curve has complex multiplication and, if so, also returns the discriminant of the CM quadratic order. The algorithm uses fairly straightforward analytic methods, which are not suited to very high degree j-invariants with CM by orders with discriminants more than a few thousand.

122.3.3 Mordell–Weil Groups

`TorsionBound(E, n)`

Given an elliptic curve E defined over a number field, returns a bound on the size of the torsion subgroup of E by looking at the first n non-inert primes of sufficiently low ramification degree and good reduction.

`pPowerTorsion(E, p)`

`Bound`	RngIntElt	*Default :* -1

Given an elliptic curve E defined over a number field or **Q**, returns the p-power torsion of E over its base field as an abelian group, together with the map from the group into the curve. One can specify a bound on the size of the p-power torsion, which may help the computation.

`TorsionSubgroup(E)`

Given an elliptic curve E defined over a number field, returns the torsion subgroup of E.

The algorithm involves using `TorsionBound` (described above).

`MordellWeilShaInformation(E: parameters)`

`DescentInformation(E: parameters)`

`RankOnly`	BoolElt	*Default :* `false`
`ShaInfo`	BoolElt	*Default :* `false`
`Silent`	BoolElt	*Default :* `false`

This is a special function which uses all relevant Magma machinery to obtain as much information as possible about the Mordell-Weil group and the Tate-Shafarevich group of the elliptic curve E. The tools used include 2-descent and

the Cassels-Tate pairing on the 2-Selmer group; analytic routines may also be used when the conductor has small norm.

The arguments and returned values are the same as for curves over $\mathbf{Q}$, and are described in Section 122.2.5.

Note: The function `PseudoMordellWeilGroup` is obsolete and should not be used; this function should be used instead.

`RankBound(E)`

`Isogeny`	MAP	*Default :*

The upper bound on the rank of $E(K)$, for an elliptic curve over a number field K (or $\mathbf{Q}$), obtained by computing the Selmer group(s) associated to some isogeny on E. The isogeny is either multiplication by 2, or a 2-isogeny if any exist, and may be specified by the optional parameter `Isogeny`.

122.3.4 Heights

`NaiveHeight(P)`

Given a point P on an elliptic curve over a number field K, the function returns the naive height of P; i.e., the absolute logarithmic height of the x-coordinate.

`Height(P : parameters)`

`Precision`	RNGINTELT	*Default :* 27
`Extra`	RNGINTELT	*Default :* 8

Given a point P on an elliptic curve defined over a number field K, returns the Néron-Tate height $\hat{h}(P)$. (This is based on the absolute logarithmic height of the x-coordinate.) The parameter `Precision` may be used to specify the desired precision of the output. The parameter `Extra` is used in the cases when one of the points $2^n P$ gets too close to the point at infinity, in which case there is a huge loss in precision in the archimedean height and increasing `Extra` remedies that problem.

`HeightPairingMatrix(P : parameters)`

Compute the height pairing matrix for an array of points. Same parameters as above.

`LocalHeight(P, Pl : parameters)`

`Precision`	RNGINTELT	*Default :* 0
`Extra`	RNGINTELT	*Default :* 8

Given a point P on an elliptic curve defined over a number field K, and a place Pl (finite or infinite) of K, returns the local height $\lambda_{Pl}(P)$ of P at Pl. The parameter `Precision` sets the precision of the output. A value of 0 takes the default precision.

When Pl is an infinite place, the parameter `Extra` can be used to remedy the huge loss in precision when one of the points $2^n P$ gets too close to the point at infinity.

122.3.5 Two Descent

The functions here have similar syntax to the corresponding ones for curves over **Q**. The current implementation from 2011 is the first complete implementation of 2-descent over number fields. Previously there have been several implementations that are effective for computing rank bounds, but not for obtaining nice models of the 2-coverings which is necessary in order to search for points. Some of the key ingredients here are the routine for solving conics over number fields (see `HasRationalPoint`), and techniques for minimisation and reduction over number fields developed by Donnelly and Fisher.

`TwoDescent(E:`*parameters*`)`

`RemoveTorsion`	BoolElt	*Default :* `false`
`RemoveGens`	{PtEll}	*Default :* {}
`WithMaps`	BoolElt	*Default :* `true`
`MinRed`	BoolElt	*Default :*
Verbose	`TwoDescent`	*Maximum :* 1

This function has the same arguments and return values as the corresponding function for curves over **Q**. It returns the 2-coverings as a sequence of hyperelliptic curves, and a corresponding list of maps.

Additionally, it returns a third object which specifies the group structure on the 2-coverings. This is a map (with inverse) from an abstract group to the sequence of 2-coverings; the abstract group is either `TwoSelmerGroup(E)`, or the appropriate quotient in the case where `RemoveTorsion` or `RemoveGens` are specified.

Currently, there is an additional optional argument `MinRed`, which controls whether the coverings are minimised and reduced. (This may be expensive for various reasons, especially when the field discriminant is not small: for one thing, a large integer may need to be factored.)

`TwoCover(e)`

`TwoCover(e)`

The purpose of this function is to obtain 2-covers arising in `TwoDescent` individually rather than all together.

The argument e is an element of a cubic extension A/F, where F is a number field, and where A may be either a number field over F or an affine algebra over F. The element e determines a 2-cover of some elliptic curve over F (by the construction given in the description of `DescentMaps` in the next section).

122.3.6 Selmer Groups

First we give a short overview of the theory of Selmer groups. This enables us to fix the notation that is used in the naming of the Magma functions. For a more complete account, see [Sil86]. The actual algorithms to compute the Selmer groups are closer to the description in [Cas66].

Let E', E be elliptic curves over a number field K and let $\phi : E' \to E$ be an isogeny. The only cases currently implemented are where ϕ is a 2-isogeny, i.e., an isogeny of degree 2, and where $E' = E$ and ϕ is multiplication by 2.

Let $E'[\phi]$ be the kernel subscheme of ϕ. By taking Galois cohomology of the short exact sequence of schemes over K,

$$0 \to E'[\phi] \to E' \to E \to 0,$$

we obtain

$$E'(K) \to E(K) \to H^1(K, E'[\phi]) \to H^1(K, E').$$

For the middle map, we write $\mu : E(K) \to H^1(K, E'[\phi])$. Thus, $E(K)/\phi(E'(K))$ injects in $H^1(K, E'[\phi])$ and the image consists exactly of the cocycles that vanish in $H^1(K, E')$.

As an approximation to this image, we define the ϕ-Selmer group of E over K to consist of those cocycles $H^1(K, E'[\phi])$ that vanish in all restrictions $H^1(K_p, E')$, where p runs though all places of K. It fits in the exact sequence

$$0 \to S^{(\phi)}(E/K) \to H^1(K, E'[\phi]) \to \prod_p H^1(K_p, E')$$

The main application of Selmer groups is that they provide the bound:

$$\#E(K)/\phi E'(K) \leq \#S^{(\phi)}(E/K).$$

If $\phi^* : E \to E'$ is the isogeny dual to ϕ, then $\phi \circ \phi^* : E \to E$ is multiplication by $d = \mathrm{Deg}(\phi)$. Thus, one can use the ϕ and ϕ^* Selmer groups to provide a bound on $\#E(K)/dE(K)$ and thus on the Mordell–Weil rank of $E(K)$.

Representation of $H^1(K, E'[\phi])$:

If ϕ is a 2-isogeny, then $H^1(K, E'[\phi]) \sim K^*/K^{*2}$. Thus, we can represent elements by $\delta \in K^*$. In MAGMA, the map μ corresponds to a representation of the map $E(K) \to K^*$. The map μ also accepts just x-coordinates.

If ϕ is multiplication-by-2 and $E : y^2 = f(x)$, we write $A = K[x]/(f(x))$ then

$$H^1(K, E[2]) \sim \{\delta \in A^*/A^{*2} | N_{A/K}(\delta) \in K^{*2}\}.$$

In fact, the full set A^*/A^{*2} corresponds to $H^1(K, E[2] \times \{\pm 1\})$.

The set $H^1(K, E'[\phi])$ also corresponds to the set of covers $\tau_\delta : T_\delta \to E$ over K, modulo isomorphy over K, that are isomorphic to $\phi : E' \to E$ over the algebraic closure of K (see [Sil86], Theorem X.2.2). In this section, we write `tor` for the map $\delta \mapsto \tau_\delta$.

`DescentMaps(phi)`

`CasselsMap(phi)`

`Fields` SETENUM *Default :* {}

Given an isogeny $\phi : E \to E_1$ of elliptic curves over a number field K (or $\mathbf{Q}$), the function returns the connecting homomorphism

$$\mu : E_1(K) \to H^1(K, E[\phi]),$$

and a map τ sending an element of $H^1(K, E[\phi])$ to the corresponding homogeneous space. Here elements of $H^1(K, E[\phi])$ are represented as elements of $A^*/(A^*)^2$ (as described above). The maps are actually given as maps to, and from, A (rather than $A^*/(A^*)^2$).

The isogeny ϕ must be either a 2-isogeny or multiplication–by–2.

When ϕ is multiplication–by–two, then the computation of μ involves a call to `AbsoluteAlgebra`. The optional parameter `Fields` is passed on to that. If the fields mentioned in this set are found to be of any use, then these will be used and whatever class group and unit data stored on the fields will be used in subsequent computations.

When ϕ is multiplication–by–two, the second map τ accepts all elements $\delta \in A^*$. An element $\delta \in A^*$ represents an element of $H^1(K, E[2])$ if and only if δ must have square norm. In general, $\delta \in A^*$ represents an element of $H^1(K, E[2] \times \{\pm 1\})$, in which case $\tau(\delta)$ is the corresponding covering $T_\delta \to \mathbf{P}^1$. (This covering is a twist of the covering $E \to E \to \mathbf{P}^1$ given by $P \mapsto 2P \mapsto x(2P)$.)

`SelmerGroup(phi)`

`Hints`	SetEnum	*Default :* {}
`Raw`	BoolElt	*Default :* `false`
`Bound`	RngIntElt	*Default :* -1
Verbose	`Selmer`	*Maximum :* 2

Given an isogeny $\phi : E \to E_1$ defined over a number field K, computes the associated Selmer group $Sel(\phi) := Sel^\phi(E/K)$.

The Selmer group is returned as a finite abelian group S, together with a map $AtoS : A \to S$, where A is as in the introduction. This is a map only in the Magma sense; it is defined only on a finite subset of A. Its "inverse" $S \to A$ provides the mathematically meaningful injection $S \hookrightarrow A^*/(A^*)^2$. The standard map

$$E_1(K) \to E_1(K)/\phi E(K) \to Sel(\phi)$$

is given by the composition of μ with $AtoS$, where $\mu : E_1(K) \to H^1(K, E[\phi])$ is the first map returned by `DescentMaps`.

If the optional parameter `Hints` is given, it is used as a list of x-coordinates to try first when determining local images. Supplying `Hints` does not change the outcome, but may speed up the computation.

The calculation of the Selmer group involves possibly expensive class group and unit group computations. If no such data has been precomputed, `SelmerGroup` will attempt to obtain this information unconditionally, unless `Bound` is positive. This

bound is then passed on to any called `ClassGroup`. However, if such data is already stored, it will be used and subsequent results will be conditional on whatever assumptions were made while computing this information. If conditional results are desired (for instance, assuming GRH), one should precompute class group information on the codomain of `CasselsMap(phi)` prior to calling `SelmerGroup(phi)`.

If `Raw` is `true`, then three technical items are also returned. The first two of these, $toVec$ and FB, enable one to represent elements of the Selmer group in terms of a "factor base" FB consisting of elements of A that generate a relevant subgroup of $A^*/(A^*)^2$. The map $toVec$ sends an element of S to an exponent vector relative to FB. The map from S to A obtained by multiplying out the results is inverse to $AtoS$.

The final returned value (when `Raw` is `true`) is a set of `Hints` (just as in the optional parameter).

`TwoSelmerGroup(E)`

`Hints`	SetEnum	*Default :* {}
`Raw`	BoolElt	*Default :* `false`
`Bound`	RngIntElt	*Default :* -1
Verbose	`EllSelmer`	*Maximum :* 2

The 2-Selmer group of an elliptic curve defined over $\mathbf{Q}$ or a number field. The function simply calls `SelmerGroup` for the multiplication–by–two isogeny. The given model for E should be integral. The options and return values are the same as for `SelmerGroup`.

Example H122E35

In this example, we determine the rank of $y^2 = x^3 + 9x^2 - 10x + 1$ by computing the 2-Selmer group.

```
> E := EllipticCurve([0,9,0,-10,1]);
> two := MultiplicationByMMap(E,2);
> mu, tor := DescentMaps(two);
```

The hard work: computing the Selmer group.

```
> S, AtoS := SelmerGroup(two);
> #S;
8
```

So the Selmer rank is 3. We deduce the following upper bound on the rank of $E(\mathbf{Q})$, taking into account any 2-torsion.

```
> RankBound(E : Isogeny := two);
3
```

In fact, there are 3 points: $(0, 1), (1, 1)$ and $(2, 5)$.

```
> g1 := E![ 0, 1 ];
```

```
> g2 := E![ 1, 1 ];
> g3 := E![ 2, 5 ];
```

We now test these points for linear independence. It will follow that the points generate $E(K)/2E(K)$, which means they are independent nontorsion points in $E(K)$ (since we know there is no 2-torsion).

```
> IsLinearlyIndependent ([g1, g2, g3]);
true
```

Next we compute the homogeneous space associated to the point $g1 + g2$. It must have a rational point mapping to $g1 + g2$. Note that @@ always denotes "preimage" in MAGMA. The algebra for `AtoS` is not the same as the original cubic, so we must translate before applying the requisite map.

```
> K := NumberField(Modulus(Domain(AtoS)));
> L := NumberField(Polynomial([1,-10,9,1]));
> b, m := IsIsomorphic (K, L); assert b;
> theta := (Rationals()! (L.1 - m(K.1))) + Domain(AtoS).1;
> H, mp := TwoCover((g1+g2)[1] - theta : E:=E); H;
Hyperelliptic Curve defined by y^2 = 9*x^4 - 4*x^3 - 18*x^2 + 4*x + 13
```

The next command finds all **Q**-rational points in the preimage of the point $g1 + g2$ on E. (In MAGMA the preimage is constructed as a scheme.)

```
> RationalPoints( (g1+g2) @@ mp);
{@ (-1 : -2 : 1) @}
```

Example H122E36

We consider the elliptic curve $E : y^2 = dx(x + 1)(x + 3)$ where d is the product of the primes less than 50. Since E has full 2-torsion, we can carry out 2-isogeny descent in three non-equivalent ways, resulting in three different rank bounds.

```
> P<x> := PolynomialRing(Integers());
> d := &*[ p : p in [1..50] | IsPrime(p) ];
> E := EllipticCurve(HyperellipticCurve(d*x*(x + 1)*(x + 3)));
```

The nontrivial 2-torsion points in $E(\mathbf{Q})$:

```
> A, mp := TorsionSubgroup(E);
> T := [ t : a in A | t ne E!0 where t := mp(a) ];
```

The corresponding 2-isogenies:

```
> phis := [ TwoIsogeny(t) : t in T ];
```

The rank bounds obtained from these isogenies:

```
> [ RankBound(E : Isogeny := phi) : phi in phis ];
[ 9, 5, 7 ]
```

We find [9, 5, 7]! Each descent gives a different rank bound. However, a full 2-descent gives:

```
> two := MultiplicationByMMap(E,2);
> RankBound(E : Isogeny := two);
```

```
1
```

Now doing full 2-descent on the three isogenous curves, we see where the obstacle for sharp rank bounds comes from.

```
> OtherTwos := [ MultiplicationByMMap(Codomain(phi), 2) : phi in phis ];
> [ RankBound(Domain(two) : Isogeny := two) : two in OtherTwos ];
[ 9, 5, 7 ]
```

We find [9, 5, 7] again.

Example H122E37

The next example is a classic one from [Kra81].

```
> E := EllipticCurve([0, 977, 0, 976, 0]);
> twoE := MultiplicationByMMap(E, 2);
> muE := CasselsMap(twoE);
> SE, toSE := SelmerGroup(twoE);
> rkEQ := RankBound(E);
> ptE := [ E | [-4, 108], [4, 140] ]; // ignore torsion
> IsLinearlyIndependent (ptE);
true
> // So E is really of rank 2
> d := 109;
> Ed := QuadraticTwist(E, d);
> Ed![ -976, -298656, 1];
(-976 : -298656 : 1)
> // So Ed is of rank at least 1
> rkEdQ := RankBound(Ed);
> _<x> := PolynomialRing(Rationals());
> K := NumberField(x^2 - d);
> EK := BaseChange(E, K);
> rkEK := RankBound(EK);
> rkEQ;
2
> rkEdQ;
3
> rkEK;
3
```

We know that the rank of $E_d(\mathbf{Q})$ must be the rank of $E(K)$ minus the rank of $E(Q)$; thus $E_d(\mathbf{Q})$ has rank 1. This is smaller than the bound of 3 we get from a 2-descent on E_d alone.

Example H122E38

Here we give some examples of using `TwoDescent`, `TwoSelmerGroup`, and `TwoCover`.

```
> E := EllipticCurve( [ 0, 0, 1, -7, 6] ); // rank 3 curve
> T := TwoDescent(E);
> T[6];
```

```
Hyperelliptic Curve defined by y^2 = 3*x^4 - 10*x^3 + 10*x + 1 over
Rational Field
> G, m := TwoSelmerGroup(E);
> G.1 @@ m;
theta^2 - 12*theta + 33
> Parent($1); // Modulus has y^2 = modulus isomorphic to E
Univariate Quotient Polynomial Algebra in theta over Rational Field
with modulus theta^3 - 112*theta + 400
> TwoCover( (G.1 + G.2) @@ m);
Hyperelliptic Curve defined by y^2 = 3*x^4 - 10*x^3 + 10*x + 1 over
Rational Field
> TwoCover( Domain(m) ! 1 );
Hyperelliptic Curve defined by y^2 = 2*x^3 - 12*x^2 - 32*x + 196 over
Rational Field
```

122.3.7 The Cassels-Tate Pairing

The pairing between elements of the 2-Selmer group is implemented, in the same way as for curves over the rationals. This is described in Section 122.2.8.

122.3.8 Elliptic Curve Chabauty

This refers to a method for finding the rational points on a curve, if the curve admits a suitable map to an elliptic curve over some extension field. The method was developed principally by Nils Bruin (see [Bru03] or [Bru04]). The first intrinsic follows a Mordell-Weil sieve based strategy, similar to that used in `Chabauty` for genus 2 curves (see [BS10]).

`Chabauty(MWmap, Ecov)`

`InertiaDegreeBound`	RngIntElt	*Default :* 20
`SmoothBound`	RngIntElt	*Default :* 50
`PrimeBound`	RngIntElt	*Default :* 30
`IndexBound`	RngIntElt	*Default :* -1
`InitialPrimes`	RngIntElt	*Default :* 50
Verbose	`EllChab`	*Maximum :* 3

Let E be an elliptic curve defined over a number field K. This function attempts to determine the subset of $E(K)$ consisting of those points that have a **Q**-rational image under a given map $E \to \mathbf{P}^1$.

The arguments are as follows:

- a map `MWmap`: $A \to E$ from an abstract abelian group into $E(K)$ (for instance, the map returned by `PseudoMordellWeilGroup`),
- A map `Ecov` : $E \to \mathbf{P}^1$ defined over K.

The returned values are a finite subset V of A and an integer R. The set V consists of all the points in the image of A in $E(K)$ that have a **Q**-rational image under

the given map `Ecov`. If the index of A in $E(K)$ is finite and coprime to R then V consists of all points in $E(K)$ with $\mathbf{Q}$-rational image. Changing the various optional parameters can change the performance of this routine considerably. In general, it is not proven that this routine will return at all.

The parameter `InertiaDegreeBound` determines the maximum inertia degree of primes at which local information is obtained.

Only places at with the group order of the rational points on the reduction of E are `SmoothBound`-smooth are used.

The Mordell-Weil sieving process only uses information from $E(K)/BE(K)$, where B is `PrimeBound`-smooth.

If `IndexBound` is set to a value different from -1, then the returned value R will only contain prime divisors from `IndexBound`. Setting this parameter may make it impossible for the routine to return.

`Chabauty(MWmap, Ecov, p)`

`Cosets`	Tup	*Default :*
`Aux`	SetEnum	*Default :*
`Precision`	RngIntElt	*Default :*
`Bound`	RngIntElt	*Default :*
Verbose	`EllChab`	*Maximum :* 3

Let E be an elliptic curve defined over a number field K. This function bounds the set of points in $E(K)$ whose images under a given map $E \to \mathbf{P}^1$ are $\mathbf{Q}$-rational.

The arguments are as follows:

- a map `MWmap` : $A \to E$ from an abstract group into $E(K)$ (for instance, the map returned by `PseudoMordellWeilGroup`),
- a map of varieties `Ecov` : $E \to \mathbf{P}^1$ defined over K, and
- a rational prime p, such that E and the map `Ecov` have good reduction at primes above p.

The returned values are N, V, R and L as follows. Let G denote the image of A in $E(K)$.

- N is an upper bound for the number of points $P \in G$ such that $\mathbf{Ecov}(P) \in \mathbf{P}^1(\mathbf{Q})$ (note that N can be ∞).
- V is a set of elements of A found by the program that have images in $\mathbf{P}^1(\mathbf{Q})$,
- R is a number with the following property: if $[E(K) : G]$ is finite and coprime to R, then N is also an upper bound for the number of points $P \in E(K)$ such that $\mathbf{Ecov}(P) \in \mathbf{P}^1(\mathbf{Q})$, and
- L is a collection of cosets in A such that $(\bigcup L) \cup V$ contains all elements of A that have images in $\mathbf{P}^1(\mathbf{Q})$.

If `Cosets` ($< C_j >$) are supplied (a coset collection of A), then the bounds and results are computed for $A \cap (\bigcup C_j)$.

If `Aux` is supplied (a set of rational primes), then the information at p is combined with the information at the other primes supplied.

If `Precision` is supplied, this is the p-adic precision used in the computations. If not, a generally safe default is used.

If `Bound` is supplied, this determines a search bound for finding V.

The algorithm used is based on [Bru03] and [Bru02].

For examples and a more elaborate discussion, see [Bru04].

Example H122E39

This example is motivated and explained in great detail in [Bru04] (Section 7). Let E be the elliptic curve

$$y^2 = x^3 + (-3\zeta^3 - \zeta + 1)dx^2 + (-\zeta^2 - \zeta - 1)d^2x, \quad d = 2\zeta^3 - 2\zeta^2 - 2$$

over $K := \mathbf{Q}(\zeta)$ where ζ is a primitive 10th root of unity.

```
> _<z> := PolynomialRing(Rationals());
> K<zeta> := NumberField(z^4-z^3+z^2-z+1);
> OK := IntegerRing(K);
> d := 2*zeta^3-2*zeta^2-2;
> E<X,Y,Z> := EllipticCurve(
>         [ 0, (-3*zeta^3-zeta+1)*d, 0, (-zeta^2-zeta-1)*d^2, 0 ]);
```

Next we determine as much as possible of $E(K)$, allowing MAGMA to choose the method.

```
> success, G, GtoEK := PseudoMordellWeilGroup(E);
> G;
Abelian Group isomorphic to Z/2 + Z/2 + Z + Z
Defined on 4 generators
Relations:
    2*G.1 = 0
    2*G.2 = 0
> success;
true
```

Here G is an abstract group and `GtoEK` injects it into $E(K)$. Since the flag is `true`, this is a subgroup of finite, odd index in $E(K)$. Next, we determine the points $(X : Y : Z)$ in this subgroup for which the function

$$u : E \to \mathbf{P}^1 : (X : Y : Z) \to d(-X + (\zeta^3 - 1)Z : X + d(-\zeta^3 - \zeta)Z)$$

takes values in $\mathbf{P}^1(\mathbf{Q})$.

```
> P1 := ProjectiveSpace(Rationals(),1);
> u := map< E->P1 | [-X + (zeta^3 - 1)*d*Z, X+(-zeta^3-zeta)*d*Z] >;
> V, R := Chabauty( GtoEK, u);
> V;
{
    0,
```

```
    G.3 - G.4,
    -G.3 + G.4,
    G.3 + G.4,
    -G.3 - G.4
}
> R;
320
```

We see that the routine assumed that the image of `GtoEK` is 2- and 5-saturated in $E(K)$. We can ask it to not assume anything outside 2.

```
> V2, R := Chabauty( GtoEK, u: IndexBound:= 2);
> V eq V2;
true
> R;
16
```

This means that we have found all points in $E(K)$ that have a rational image under u. If one wants to find the images of these points then it is necessary in this example to first extend u (by finding alternative defining equations for it) so that it is defined on all the points.

```
> u := Extend(u);
> [ u( GtoEK(P) ) : P in V ];
[ (-1 : 1), (-1/3 : 1), (-1/3 : 1), (-3 : 1), (-3 : 1) ]
```

Alternatively, we can apply Chabauty's method without Mordell-Weil sieving.

```
> N, V, R, C := Chabauty( GtoEK, u, 3);
> N;
5
> V;
{
    0,
    G.3 - G.4,
    -G.3 + G.4,
    G.3 + G.4,
    -G.3 - G.4
}
> R;
4
```

The `Chabauty` calculations prove that there are at most N elements in G whose image under u is $\mathbf{Q}$-rational. Also, V is a set of elements with this property. Since here $N = 5 = \#V$, these are the only such elements. Moreover, the calculations prove that *if* $[E(K) : G]$ is coprime to R, then N is actually an upper bound on the number of elements in $E(K)$ whose image under u is $\mathbf{Q}$-rational. We know from the `PseudoMordellWeilGroup` computation that $[E(K) : G]$ is odd. Therefore we have solved our problem for $E(K)$, not just for G.

122.3.9 Auxiliary Functions for Etale Algebras

This section contains machinery for number fields (and more generally "etale algebras", i.e. algebras of the form $\mathbf{Q}[x]/p(x)$), intended to perform the number field calculations that are involved in computing Selmer groups of geometric objects.

It has become conventional to refer to "the p-Selmer group" of a number field K (or, more generally, of an etale algebra) relative to a finite set S of K-primes. It means the finite subgroup $K(S,p)$ of $K^*/(K^*)^p$, consisting of those elements whose valuation at every prime outside S is a multiple of p.

`AbsoluteAlgebra(A)`

`Fields`	SetEnum	*Default :* {}
Verbose	`EtaleAlg`	*Maximum :* 1

Given a separable commutative algebra over **Q**, an absolute number field or a finite field, the function returns the isomorphic direct sum of absolute fields as a cartesian product and the isomorphisms to and from this product. The optional parameter `Fields` enables the user to suggest representations of the absolute fields. If this function finds it needs a field isomorphic to one occurring in the supplied list, then it will use the given field. Otherwise it will construct a field itself. The isomorphism is returned as a second argument. If called twice on the same algebra, it will recompute if the Fields argument is given. Otherwise it will return the result computed the first time.

`pSelmerGroup(A, p, S)`

Verbose	`EtaleAlg`	*Maximum :* 1

Returns the p-Selmer group of a semi-simple algebra A. The set S of ideals should be prime ideals of the underlying number field. The group returned is the direct sum of the p-Selmer groups of the irreducible summands of A and the map is the obvious embedding in the multiplicative group of A. An implied restriction is that `BaseRing(A)` be of type `FldNum`. See also pSelmerGroup on page 1006. If an algebra over the rationals is required, create a degree 1 extension by

```
RationalsAsNumberField();
```

`LocalTwoSelmerMap(P)`

Let K be the number field associated with the prime ideal P. The map returned is $K^* \to K_P^*/K_P^{*2}$, where K_P is the completion of K at P. The codomain is represented as a finite abelian group.

`LocalTwoSelmerMap(A, P)`

Let K be the base field of the commutative algebra A and let P be a prime ideal in K. Then this function returns a map $A^* \to A^*/A^{*2} \otimes K_P$, where K_P is the completion of K at P. The codomain is represented as a finite abelian group.

This map is computed by using `LocalTwoSelmerMap(Q)` for the various extensions Q of P to the fields making up `AbsoluteAlgebra(A)`. The map returned is

essentially the direct sum of all these maps. For technical purposes, one may also wish to use the components of the map coming from each number field; these are given by the second return value, which is a sequence of records (ordered in the same way as the results are concatenated in the returned map). Each record contains items `i, p, map and vmap`. Here `i` is an index indicating which number field in `AbsoluteAlgebra(A)` the record corresponds to, `p` is an extension of P to a prime in that number field, `map` is `LocalTwoSelmerMap(p)`, and `vmap` is the valuation map at p on that number field.

Example H122E40

```
> P<x> := PolynomialRing(Rationals());
> A := quo<P | x^3 - 1>;
> AA := AbsoluteAlgebra(A);
> AA;
Cartesian Product
<Number Field with defining polynomial x - 1 over the Rational Field,
 Number Field with defining polynomial x^2 + x + 1 over the Rational Field>
```

122.3.10 Analytic Information

`RootNumber(E, P)`

The local root number of the elliptic curve E (defined over a number field) at the prime ideal P. The formulae are due to Rohrlich, Halberstadt, Kobayashi and the Dokchitser brothers.

`RootNumber(E)`

Calculates the global root number of an elliptic curve E defined over a number field K. This is the product of local root numbers over all places of K (-1 from each infinite place), and is the (conjectural) sign in the functional equation relating $L(E/K, s)$ to $L(E/K, 2 - s)$.

`AnalyticRank(E)`

`Precision` RngIntElt *Default :* 6

Determine the analytic rank of the elliptic curve E, which is defined over a number field K. The algorithm used is heuristic, computing derivatives of the L-function $L(E/K, s)$ at $s = 1$ until one appears to be nonzero; it also assumes the analytic continuation and the functional equation for the L-function. The function returns the first nonzero derivative $L^{(r)}(1)/r!$ as a second argument. The precision is optional, and is taken to be 6 digits if omitted.

`ConjecturalRegulator(E)`

`Precision`	RngIntElt	*Default :* 10

Using the `AnalyticRank` function, this function calculates an approximation, assuming that the Birch–Swinnerton-Dyer conjecture holds, to the product of the regulator of the elliptic curve E and the order of the Tate–Shafarevich group. The (assumed) analytic rank is returned as a second value.

`ConjecturalSha(E, Pts)`

`Precision`	RngIntElt	*Default :* 6

For an elliptic curve E defined over a number field K and a sequence of points in $E(K)$ which purpotedly form its basis modulo torsion, computes the conjectural order of the Tate-Shafarevich group Ш(E/K). This function computes the product of the regulator and Ш from the Birch–Swinnerton-Dyer conjecture (using `ConjecturalRegulator`) and divides by the determinant of the height pairing matrix for the points supplied in `Pts`. It returns 0 if the points are linearly dependent or they generate a group of rank less than the analytic rank. If the points generated a subgroup of index $n > 1$, it returns $n^2 \cdot |Ш|$.

122.3.11 Elliptic Curves of Given Conductor

This section describes search routines for finding elliptic curves with given conductor, or with good reduction outside a given set of primes. The aim is not to provably find all such curves; in most cases, this would be a very difficult task using current algorirthms. Rather the aim is to efficiently search for these curves, using a variety of techniques, taking advantage of all available information, and taking advantage of all the tools available in Magma which can be applied to the problem.

The routine is particularly effective when some traces of Frobenius of the desired are specified. The obvious application here is to find an elliptic curve that matches a known modular form.

These functions can be used to find elliptic curves over **Q**, as well as number fields, by using `RationalsAsNumberField()`.

`EllipticCurveSearch(N, Effort)`

`EllipticCurveWithGoodReductionSearch(S, Effort)`

`Full`	BoolElt	*Default :* `false`
`Max`	RngIntElt	*Default :*
`Primes`	SeqEnum	*Default :*
`Traces`	SeqEnum	*Default :*
Verbose	`ECSearch`	*Maximum :* 2

These functions perform a search for elliptic curves with specified conductor(s). The first function finds curves with conductor equal to the given ideal N; the "good reduction" functions find curves with conductors divisible only by primes that belong

to the given set S (or that divide the given ideal N). The functions return a sequence containing all such curves that were found. This sequence will not contain curves that are isomorphic to each other. It may contain isogenous curves, however no attempt is made to find full isogeny classes.

The second argument, `Effort`, is an integer which controls how much effort is used before returning; the running time depends on this variable in a roughly linear way, except with significant step-ups at values where an additional technique begins to be used. Currently, `Effort := 400` is the smallest value which tries all available techniques (however this is subject to change).

There are two ways to specify an early stopping condition:

(i) If the optional argument `Max` is set to some positive integer, the routine will return whenever this many non-isogenous curves have been found.

(ii) The optional arguments `Primes` and `Traces`, which must correspond to eachother, may be used to specify some traces of Frobenius of the desired curve(s). Here `Primes` is a sequence of prime ideals coprime to the conductor(s), and `Traces` is a sequence of integers of the form $[a_P : P in$ Primes $]$. Alternatively, to search for several curves with different traces, `Traces` may instead by a sequence of sequences of that form.

If an early stopping condition has been specified, the routine initially tries with lower effort, in order to succeed quickly for easier input; the effort is incremented exponentially up to the specified `Effort` level. (This strategy is desirable because the algorithm involves performing a large number of independent searches in alternation.) However if the option `Full` is set, the full specified `Effort` is used from the beginning.

If `SetVerbose("ECSearch", 1)` is used, information about the search is printed during computation. In particular, curves with the desired conductor are printed when they are found, so that they can be obtained without waiting for the routine to finish.

122.4 Curves over p-adic Fields

The functions in this section are for elliptic curves defined over p-adic fields. They provide an interface to the same code for Tate's algorithm that is used for curves over number fields.

122.4.1 Local Invariants

`Conductor(E)`

The conductor of the elliptic curve E defined over a p-adic field.

`LocalInformation(E)`

Implements Tate's algorithm for the elliptic curve E over a p-adic field. This intrinsic computes local reduction data at the prime ideal P, and a local minimal model. The model is not required to be integral on input. Output is $\langle P, v_p(d), f_p, c_p, K, s\rangle$ and E_{min} where P is the uniformizer of the ground field, $v_p(d)$ is the valuation of the local minimal discriminant, f_p is the valuation of the conductor, c_p is the Tamagawa number, K is the Kodaira Symbol, and s is `false` if the curve has non-split multiplicative reduction and `true` otherwise. E_{min} is an integral minimal model of E.

`RootNumber(E)`

The local root number of the elliptic curve E (defined over a p-adic field).

122.5 Bibliography

[**BC04**] W. Bosma and J. Cannon, editors. *Discovering Mathematics with Magma*. Springer-Verlag, Heidelberg, 2004.

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**Bru02**] N. R. Bruin. *Chabauty methods and covering techniques applied to generalized Fermat equations*, volume 133 of *CWI Tract*. Stichting Mathematisch Centrum Centrum voor Wiskunde en Informatica, Amsterdam, 2002. Dissertation, University of Leiden, Leiden, 1999.

[**Bru03**] Nils Bruin. Chabauty methods using elliptic curves. *J. reine angew. Math.*, 562:27–49, 2003.

[**Bru04**] Nils Bruin. Some ternary Diophantine equations of signature $(n, n, 2)$. In Bosma and Cannon [BC04].

[**BS10**] Nils Bruin and Michael Stoll. The Mordell-Weil sieve: Proving non-existence of rational points on curves. *LMS J. Comput. Math.*, 13:272–306, 2010.

[**Cas66**] J. W. S. Cassels. Diophantine equations with special reference to elliptic curves. *J. London Math. Soc.*, 41:150–158, 1966.

[**CFO$^+$**] J.E. Cremona, T.A Fisher, C. O'Neil, D. Simon, and M Stoll. Explicit n-Descent On Elliptic Curves, III. Algorithms. ArXiv preprint. URL:http://arxiv.org/abs/1107.3516.

[**CFO$^+$08**] J. E. Cremona, T. A. Fisher, C. O'Neil, D. Simon, and M. Stoll. Explicit n-descent on elliptic curves. I. Algebra. *J. Reine Angew. Math.*, 615:121–155, 2008.

[**CFO$^+$09**] J.E. Cremona, T.A Fisher, C. O'Neil, D. Simon, and M Stoll. Explicit n-Descent On Elliptic Curves, II. Geometry. *J. reine angew. Math.*, 632:63–84, 2009.

[**CFS10**] J.E. Cremona, T.A Fisher, and M Stoll. Minimisation and reduction of 2-, 3- and 4-coverings of elliptic curves. *Algebra & Number Theory*, 4(6):763–820, 2010.

[**CM12**] B. Creutz and R.L. Miller. Second isogeny descents and the Birch and Swinnerton-Dyer conjectural formula. *J.Algebra*, 372:673–701, 2012.

[**Coh93**] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.

[**Cre99**] John Cremona. Reduction of binary cubic and quartic forms. *LMS JCM*, 2:62–92, 1999.

[**Cre01**] John Cremona. Classical invariants and 2-descent on elliptic curves. *J. Symbolic Comp.*, 31:71–87, 2001.

[**Cre10**] Brendan Creutz. *Explicit second p-descents on elliptic curves*. PhD Thesis, Jacobs University Bremen, 2010.

[**Elk00**] N. Elkies. Rational Points Near Curves and Small Nonzero $|x^3 - y^2|$ via Lattice Reduction. In Bosma [Bos00], pages 33–63.

[**Fis00**] Tom Fisher. *On 5 and 7 descents for elliptic curves*. PhD thesis, University of Cambridge, 2000.

[**Fis01**] Tom Fisher. Some examples of 5 and 7 descent for elliptic curves over Q. *J. Eur. Math. Soc.*, 3(Issue 2):169–201, 2001.

[**Fis08**] Tom Fisher. Finding rational points on elliptic curves using 6-descent and 12-descent. *J. Algebra*, 320(2):853–884, 2008.

[**FK02**] Claus Fieker and David R. Kohel, editors. *ANTS V*, volume 2369 of *LNCS*. Springer-Verlag, 2002.

[**GZ86**] Gross and Zagier. Heegner Points and Derivatives of L-series. *Invent. Math.*, 84:225–320, 1986.

[**Har08**] D. Harvey. Efficient computation of p-adic heights. *LMS J. Comput. Math.*, 11:40–59, 2008.

[**Kra81**] K. Kramer. Arithmetic of elliptic curves upon quadratic extension. *Trans. Amer. Math. Soc.*, 264(1):121–135, 1981.

[**MSS96**] J. R. Merriman, S. Siksek, and N. P. Smart. Explicit 4-descents on an elliptic curve. *Acta Arith.*, 77(4):385–404, 1996.

[**MST06**] B. Mazur, W. Stein, and J. Tate. Computation of p-adic heights and log convergence. *Documenta Mathematica*, Extra:577–614, 2006.

[**MT91**] B. Mazur and J. Tate. The p-adic sigma function. *Duke Math Journal*, 62(3):663–688, 1991.

[**Sik95**] Samir Siksek. Infinite descent on elliptic curves. *Rocky Mountain J. Math.*, 25(4):1501–1538, 1995.

[**Sil86**] J. Silverman. *The arithmetic of elliptic curves*, volume 106 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1986.

[**SS04**] Edward F. Schaefer and Michael Stoll. How to do a p-descent on an elliptic curve. *Trans. Amer. Math. Soc.*, 356(3):1209–1231 (electronic), 2004.

[**SW02**] W. A. Stein and M. Watkins. A New Database of Elliptic Curves—First Report. In Fieker and Kohel [FK02].

[**Wat02**] M. Watkins. Computing the modular degree of an elliptic curve. *Experimental Mathematics*, 11(4):487–502, 2002.

[**Wom03**] T. Womack. *Explicit descent on elliptic curves.* PhD thesis, University of Nottingham, 2003.

123 ELLIPTIC CURVES OVER FUNCTION FIELDS

Chapter 123
ELLIPTIC CURVES OVER FUNCTION FIELDS

This section involves elliptic curves with coefficients in a function field $k(C)$ where C is a regular projective curve over some field k (usually a number field or a finite field). The commands are largely parallel to those for elliptic curves over the rationals; one can compute local information (Tate's algorithm and so forth), a minimal model, the L-function, the 2-Selmer group, and the Mordell–Weil group. This goes in order of decreasing generality: Local information is available for curves over univariate function fields over any exact base field, while at the other extreme Mordell–Weil groups are available only for curves over rational function fields over finite fields for which the associated surface is a rational surface. The generality of many of the commands will be expanded in future releases.

123.1 An Overview of Relevant Theory

An elliptic curve over $K = k(C)$ may be regarded as a surface $\mathcal{E}$ over k with a map $\pi : \mathcal{E} \to C$ (in other words, an *elliptic surface*); the generic fibre of $\mathcal{E}$ is E. Under this interpretation, elements of the Mordell–Weil group $E(K)$ are in one-to-one correspondence with sections of π. (A section is a morphism $s : C \to \mathcal{E}$ such that $\pi \circ s = \mathrm{Id}_C$.) This means that one may study the Mordell–Weil group by studying the geometry of the surface.

Given E, there is a unique $\mathcal{E}$ up to isomorphism that is projective, regular, and relatively minimal. This is called the Kodaira–Néron model, and we will always assume that we are working with this model of the surface.

Let $\bar{k}$ denote the separable closure of k, and $\mathcal{E}_{\bar{k}}$ the elliptic surface considered over $\bar{k}$. The Néron–Severi group $\mathrm{NS}(\mathcal{E}_{\bar{k}})$ of $\mathcal{E}_{\bar{k}}$ is the group of divisors of $\mathcal{E}_{\bar{k}}$ modulo algebraic equivalence. It is a finitely generated group and is closely connected with the Mordell–Weil group $E(K)$.

Let N be the subgroup of $\mathrm{NS}(\mathcal{E}_{\bar{k}})$ that is generated by all components of all the fibres of π together with the section corresponding to the zero point of $E(\bar{k}(C))$; this is known as the *trivial lattice* of $\mathrm{NS}(\mathcal{E}_{\bar{k}})$. It can easily be determined since the number of components in reducible fibres can be computed by Tate's algorithm. The following divisor classes together form a basis of N

(i) the image of the section corresponding to the zero point;

(ii) one complete fibre; and

(iii) the components of all the reducible fibres, with one component from each fibre omitted.

It is known that the quotient $\mathrm{NS}(\mathcal{E}_{\bar{k}})/N$ is generated by images of sections of π, and that $\mathrm{NS}(\mathcal{E}_{\bar{k}})/N \cong E(\bar{k}(C))$ (via the identification of sections with points). In particular, this implies the Shioda–Tate formula

$$\mathrm{rank}(E(\bar{k}(C))) + 2 + \sum_{v \in C(\bar{k})} (m_v - 1) = \mathrm{rank}(\mathrm{NS}(\mathcal{E}_{\bar{k}}))$$

where m_v denotes the number of components of the fibre $\pi^{-1}(v)$.

The Galois group $G = G_{\bar{k}/k}$ acts on $\mathrm{NS}(\mathcal{E}_{\bar{k}})$, and it maps N to itself. Moreover, after extending scalars to $\mathbf{Q}$ one can split the Galois representation. That is, there exists $M \subset \mathrm{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbf{Q}$ such that $\mathrm{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbf{Q} \cong M \oplus (N \otimes \mathbf{Q})$ and $M \cong E(\bar{k}(C)) \otimes \mathbf{Q}$ as G-modules. In particular, $M^G \cong E(K) \otimes \mathbf{Q}$. In the case that k is a finite field, the Frobenius action on N can be determined with the functions `FrobeniusActionOnReducibleFiber` and `FrobeniusActionOnTrivialLattice`.

In order to study $\mathrm{NS}(\mathcal{E}_{\bar{k}})$ as a G-module, one can embed it in the ℓ-adic cohomology group $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)$. To get a G-equivariant map one must slightly change the G-action on $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)$. Let $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)(1)$ denote the (1)-Tate twist of $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)$. The main property that we need to know about this twist is that it transforms the q-eigensubspace in $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)$ of some q-Frobenius element to the 1-eigenspace of this Frobenius in $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)(1)$. The cycle class map then yields a G-equivariant embedding

$$\mathrm{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbf{Q}_\ell \hookrightarrow H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)(1).$$

It is conjectured by Tate that the image of $(\mathrm{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbf{Q}_\ell)^G$ under this map exactly equals $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)(1)^G$.

In the case that k is a finite field one can in principal determine $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)(1)^G$ via the Lefschetz trace formula. Suppose that $k \subset \mathbf{F}_q$ and let F_q denote the q-th power Frobenius map. Then

$$\#\mathcal{E}(\mathbf{F}_q) = 1 + q^2 - (1+q)\,\mathrm{Trace}(F_q | H^1(C_{\bar{k}}, \mathbf{Q}_\ell)) + \mathrm{Trace}(F_q | H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)).$$

The trace on $H^1(C_{\bar{k}}, \mathbf{Q}_\ell)$ is zero if C is a rational curve, and it can be determined by counting $\mathbf{F}_q$-rational points on C in the general case. Hence one can determine $\mathrm{Trace}(F_q | H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell))$ by counting $\mathbf{F}_q$-rational points on $\mathcal{E}$. By doing this for various powers of q one can determine the characteristic polynomial of F_q acting on $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell)$, and hence the conjectural ranks of $E(\bar{k}(C))$ and $E(K)$ by using Tate's conjectures. The conjectural ranks obtained in this way give unconditional upper bounds on the true ranks.

As the Galois action on N can be determined, the difficult part is to compute

$$\det(1 - T \cdot F_q | H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_\ell) / \mathrm{im}(N \otimes \mathbf{Q}_\ell)),$$

where $\mathrm{im}(N \otimes \mathbf{Q}_\ell)$ stands for the image of $N \otimes \mathbf{Q}_\ell$ under the cycle class map. It can be shown that this polynomial is equal to the L-function of E over K; it follows that this L-function is a polynomial. This shows that Tate's conjectures are linked to a geometric

version of the Birch and Swinnerton-Dyer conjecture. Just as in the number field case, this conjecture expresses the rank and the product of the order of the Tate–Shafarevich group and the regulator of E in terms of its L-function. The L-function can be computed with the function `LFunction` and the conjectural information on the rank, Tate–Shafarevich group, and regulator can be obtained with the function `AnalyticInformation`.

If E can be defined by a Weierstrass equation in which the coefficients a_i are polynomials of degree at most i, then $\mathcal{E}_{\bar{k}}$ is a rational surface and $\text{rank}(\text{NS}(\mathcal{E}_{\bar{k}})) = 10$. In this case $\text{rank}(E(\bar{k}(C))) = 10 - \text{rank}(N)$ can be easily determined. In the case that k is a finite field then $E(\bar{k}(C))$ and $E(K)$ can be computed using functions in this section.

123.2 Local Computations

`BadPlaces(E)`

A sequence containing the places where the given model of E has bad reduction, for an elliptic curve E defined over a function field.

`Conductor(E)`

The conductor of an elliptic curve E defined over a function field F. In general this is returned as a divisor of F. When F is a rational function field it is returned as a sequence of tuples $\langle f, e\rangle$ of places (specified by field elements f) and multiplicities e.

`LocalInformation(E, Pl)`

`LocalInformation(E, f)`

This function performs Tate's algorithm for an elliptic curve E over a function field to determine the reduction type and a minimal model at the given place Pl. When E is defined over a rational function field $F(t)$ the place is simply given as a field element f (which must either be $1/t$ or an irreducible polynomial in t.)

The model is not required to be integral on input. The output is of the form $\langle Pl, v_p(d), f_p, c_p, K, split\rangle$ and E_{min} where Pl is the place, $v_p(d)$ is the valuation of the local minimal discriminant, f_p is the valuation of the conductor, c_p is the Tamagawa number, K is the Kodaira Symbol, *split* is a boolean that is false if reduction is of nonsplit multiplicative type and true otherwise, and E_{min} is a model of E (integral and) minimal at Pl.

`LocalInformation(E)`

Returns a sequence of tuples as described above for all places of bad reduction of the elliptic curve E.

`KodairaSymbols(E)`

A sequence of tuples $\langle K, n\rangle$, corresponding to the places of bad reduction of the elliptic curve E. Here K is the Kodaira symbol and n is the degree of the corresponding place.

`NumberOfComponents(K)`

The number of components of a fibre with the Kodaira symbol K.

`MinimalModel(E)`

A model of the elliptic curve E (defined over a function field, which must have genus 0) that is minimal at all finite places, together with a map from E to this minimal model.

`MinimalDegreeModel(E)`

A model of the elliptic curve E (defined over a rational function field) which minimises the quantity Max([Degree(a_i)/i]), where a_1, a_2, a_3, a_4, a_6 are the Weierstrass coefficients.

`IsConstantCurve(E)`

For an elliptic curve E defined over a rational function field $F(t)$, the function returns `true` if and only if E is isomorphic over $F(t)$ to an elliptic curve with coefficients in F (and also returns such a curve in that case).

`TraceOfFrobenius(E, p)`

The trace of Frobenius a_p for the reduction of E at the place p, specified as an element of the base field.

123.3 Elliptic Curves of Given Conductor

`EllipticCurveSearch(N, Effort)`

`Full`	BoolElt	*Default :* `false`
`Max`	RngIntElt	*Default :*
`Primes`	SeqEnum	*Default :*
`Traces`	SeqEnum	*Default :*
Verbose	`ECSearch`	*Maximum :* 2

This routine searches for elliptic curves over a field $F_q(t)$ whose conductor is equal to the specified conductor N. Alternatively, the first argument may be a set or sequence of possible conductors. Here a conductor is specified in the same format as returned by `Conductor`.

For explanation of the optional parameters and other details, see the description of `EllipticCurveSearch` for elliptic curves over number fields.

123.4 Heights

`NaiveHeight(P)`

The naive x-coordinate height of a point P on an elliptic curve over a function field K; in other words, the degree of the point $(x(P) : 1)$ on the projective line.

`Height(P)`

The Néron–Tate height $\hat{h}(P)$ of the given point P on an elliptic curve defined over a function field.

`LocalHeight(P, Pl)`

Given a point P on an elliptic curve defined over a function field F and a place Pl of the function field F, returns the local height $\lambda_{Pl}(P)$ at Pl of P.

`HeightPairing(P, Q)`

Returns the height pairing of the points P and Q, defined as $\langle P, Q\rangle = (\hat{h}(P+Q) - \hat{h}(P) - \hat{h}(Q))/2$ (where as usual $\hat{h}$ denotes the Néron–Tate height).

`HeightPairingMatrix(S)`

Given a sequence S of points P_i on an elliptic curve defined over a function field, this function returns the matrix $(\langle P_i, P_j\rangle)$, where $\langle , \rangle$ is the height pairing.

`HeightPairingLattice(S)`

The height pairing lattice of a sequence of independent points on an elliptic curve defined over a function field.

`Basis(S)`

Given a sequence S of points on an elliptic curve, returns a sequence of points that form a basis for the free part of the subgroup generated by the points in S. The second returned value is a Gram matrix for this basis with respect to the Néron–Tate pairing.

`Basis(S, r, disc)`

Given a sequence S of points on an elliptic curve, returns a sequence of independent points in the free part of the subgroup generated by S such that these points generate a lattice of rank r and discriminant *disc*. The answer is returned as soon as such a lattice has been found, ignoring any additional points in the given sequence.

`IsLinearlyDependent(points)`

`IsLinearlyIndependent(points)`

`IndependentGenerators(points)`

These functions are available for elliptic curves over function fields, and behave the same way as for elliptic curves over the rationals.

123.5 The Torsion Subgroup

`TorsionSubgroup(E)`

Given an elliptic curve E defined over a function field F, this function returns an abelian group A isomorphic to the torsion subgroup of $E(F)$, together with a map from A to $E(F)$.

`TorsionBound(E, n)`

`TorsionBound(E, n, B)`

Given an elliptic curve over a function field F and an integer n, this function computes a bound on the size of the torsion subgroup of $E(F)$ by considering the torsion subgroups of the fibres of E at n different places of F.

When an integer B is given as a third argument then the subgroup of elements of order dividing B is bounded, rather than the whole torsion subgroup.

`GeometricTorsionBound(E)`

Given an elliptic curve E defined over a function field F, this function computes a bound for the geometric torsion subgroup of E. That is, the torsion group of $E(K)$ where K/F is the smallest extension with algebraically closed constant field. In cases where a bound cannot be computed then 0 is returned.

123.6 The Mordell–Weil Group

The machinery in this section and the next section is for curves defined over a rational function field $k(t)$ whose field of constants k is finite.

The Mordell–Weil group can be computed (and generators found) for a curve $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ where each a_i is a *polynomial* in $k[t] \subset k(t)$ of degree at most i.

When this hypothesis is not satisfied, it may still be possible to bound the Mordell–Weil rank and find Mordell–Weil generators using the 2-descent routines described in the next section.

`RankBounds(E)`

`RankBound(E)`

These functions return lower and upper bounds (or just an upper bound) on the rank of the Mordell–Weil group $E(F)$ for an elliptic curve E defined over a function field F with finite constant field. The bound is obtained by applying all the available tools (those described in this section together with the `AnalyticInformation` obtained from the L-function).

`MordellWeilGroup(E : parameters)`

`Al` MonStgElt *Default :* `"Geometric"`

This function computes the Mordell–Weil group of an elliptic curve E that satisfies the hypotheses stated in the introduction immediately above. The function returns two values: an abelian group A and a map m from A to E. The map m provides an isomorphism between the abstract group A and the Mordell–Weil group.

The algorithm used by default is the geometric method described above. However, when `Al` is set to `"Descent"` it instead uses the 2-descent tools described in the next section; if the curve admits 2-isogenies then it uses a separate implementation of descent by 2-isogenies (described in [Rob07]). These descent methods do not always determine the full Mordell–Weil group (in which case a warning is printed); their advantages are that they do not require the degrees of the coefficients to be bounded and in many cases are very efficient.

`MordellWeilLattice(E)`

This function computes the Mordell–Weil lattice of an elliptic curve E that satisfies the hypotheses stated in the introduction immediately above. This is the free part of the Mordell–Weil group with an inner product given by the Néron–Tate height pairing. The function returns two values: the lattice L and a map m from L to E.

`GeometricMordellWeilLattice(E)`

This function computes the geometric Mordell–Weil lattice of an elliptic curve E that satisfies the hypotheses stated in the introduction immediately above. This consists of the free part of the group of points on E that are defined over the function field with a possibly extended constant field, together with the Néron–Tate pairing. The function returns two values: a lattice L and a map m from L to E', where E' is a base change of E over the larger field.

`Generators(E)`

Given an elliptic curve E over a rational function field F that satisfies the hypotheses stated in the introduction immediately above, this function returns a sequence of points in $E(F)$ which are generators of the Mordell–Weil group.

Example H123E1

We find that the curve $y^2 = x^3 + (t^4 + 2t^2)x + t^3 + 2t$ over $F_3(t)$ has rank 2 and has no 2-torsion in its Tate–Shafarevich group.

```
> F<t> := FunctionField(GF(3));
> E := EllipticCurve([ t^4 + 2*t^2, t^3 + 2*t ]);
> S2E := TwoSelmerGroup(E);
> S2E;
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*S2E.1 = 0
```

```
    2*S2E.2 = 0
> MordellWeilGroup(E);
Abelian Group isomorphic to Z + Z
Defined on 2 generators (free)
Mapping from: Abelian Group isomorphic to Z + Z
Defined on 2 generators (free) to CrvEll: E given by a rule [no inverse]
```

Furthermore, we may compute the regulator of E as follows.

```
> Determinant(HeightPairingMatrix(Generators(E)));
12
```

123.7 Two Descent

In odd characteristic the 2-Selmer group can be computed and its elements can be represented as minimised 2-coverings. A search for points on these 2-coverings can be made.

In characteristic 2 descent by 2-isogeny can be performed for an ordinary curve E using the isogenies $E \to E^{frob} \to E$. The Selmer groups for both isogenies are computed; if these have ranks s_1 and s_2 then the rank of E is at most $s_1 + s_2 - 1$.

`TwoSelmerGroup(E)`

This function computes the 2-Selmer group of an elliptic curve E defined over a rational function field $\mathbf{F}_q(t)$ of odd characteristic. This is returned as an abstract group together with a map from the group to the relevant algebra.

The algorithm is standard (similar to the one used for elliptic curves over number fields).

`TwoDescent(E)`

This represents the nontrivial elements of the 2-Selmer group of E as hyperelliptic curves $C : y^2 = f(x)$ of degree 4, and returns a sequence containing the curves together with a list containing the corresponding covering maps $C \to E$. The elliptic curve should be defined over a rational function field $\mathbf{F}_q(t)$ of odd characteristic.

The curves returned have polynomial coefficients and are minimised at all finite places. The conic parametrisation step uses the algorithm by Cremona and van Hoeij.

`QuarticMinimize(f)`

This is the routine used by `TwoDescent` to minimise two-coverings. The function takes a homogeneous quartic in two variables with coefficients in a rational univariate function field and returns a minimal quartic equivalent to f together with the appropriate transformation matrix.

`Points(C : parameters)`

`Bound` RngIntElt *Default :*

This finds all rational points with height up to the given `Bound` on the hyperelliptic curve C, which must be defined over a rational function field $\mathbf{F}_q(t)$. The parameter `Bound` is *not optional*, and it refers to the x-coordinate (logarithmic) height of the points: In other words, the routine finds all projective points $(X : Y : Z)$ where X and Z are polynomials in t of degree less than or equal to `Bound`.

`PointsQI(C, H)`

`OnlyOne` BoolElt *Default :* `false`

`ExactBound` BoolElt *Default :* `false`

This is an optimised routine for finding rational points on a curve given as an intersection of two quadrics defined over a rational function field $\mathbf{F}_q(t)$. It searches for projective points whose coordinates are polynomials in t of degree up to H. To guarantee finding *all* such points the parameter `ExactBound` must be set to true.

The algorithm uses a lattice reduction method described in [Rob07].

`TwoIsogenySelmerGroups(E)`

This performs descent by 2-isogenies for a non-supersingular elliptic curve E defined over a rational function field $k(t)$ where k is finite *of characteristic 2*. The isogenies used are the Frobenius map $E \to E^{frob} : (x, y) \mapsto (x^2, y^2)$ and the dual isogeny $E^{frob} \to E$ (which is separable). The function returns four objects: the Selmer group S^{sep} of the separable isogeny (as an abstract group), followed by the Selmer group S^{Frob} of the Frobenius isogeny, followed by maps $S^{sep} \to k(t)/\Phi(k(t))$ and $S^{Frob} \to k(t)^*/(k(t)^*)^2$. Here Φ denotes the Artin–Schreier map $a \mapsto a^2 + a$.

Notes describing the algorithm will be made available (on request). The theoretical background is presented in [Kra77].

123.8 The L-function and Counting Points

To compute the L-function one can use the intrinsic `LFunction` or alternatively one can do it "by hand", using the tools in this section and the next (as illustrated in Example H123E6).

`LFunction(E)`

`LFunction(E, S)`

`Check` BoolElt *Default :* `true`

The L-function of the elliptic curve E, which should be defined over a rational function field over a finite field.

To speed up the computation, the points in S are used to reduce the unknown part of H^2; S should be a sequence of points on E or on some base change of E by an extension of the constant field. If the parameter `Check` is set to `false` then the points in S must form a basis for a subgroup of the geometric Mordell–Weil group

modulo torsion that is mapped to itself by the Galois group of the extension of the fields of constants.

`LFunction(E, e)`

The L-function for the base change of the elliptic curve E by a constant field extension of degree e.

`AnalyticRank(E)`

The Mordell–Weil rank of E as predicted by Tate's version of the Birch–Swinnerton-Dyer conjecture (or equivalently, the Artin–Tate conjecture). This is an unconditional upper bound and is proved to be the actual rank for several classes of curves, such as those arising from rational elliptic surfaces and K3 elliptic surfaces. It is required that E is defined over a function field over a finite field.

`AnalyticInformation(E)`

`AnalyticInformation(E, L)`

`AnalyticInformation(E, e)`

This function computes what is predicted by the Birch–Swinnerton-Dyer (or Artin–Tate) conjecture about E, which should be an elliptic curve defined over a function field over a finite field. The data is given as a tuple consisting of the rank, geometric rank, and the product of the order of Tate–Shafarevich group and the regulator.

When a polynomial L is given as a second argument then function will assume that L is the L-function of E.

When an integer e is given as a second argument then the function will return analytic data for the base change of E by the constant field extension of degree e. This is more efficient than simply calling `AnalyticInformation` on the base change.

Example H123E2

The curve $y^2 = x^3 + 2t^2x + (t^3 + 2t^2 + 2t)$ over $\mathbf{F}_3(t)$ has rank 0 and its Tate–Shafarevich group contains $\mathbf{Z}/3 \oplus \mathbf{Z}/3$. We may conclude this (unconditionally) from the analytic information, since the Tate conjecture is proved for rational elliptic surfaces.

```
> F<t> := FunctionField(GF(3));
> E := EllipticCurve([ 2*t^2, t^3 + 2*t^2 + 2*t ]);
> AnalyticInformation(E);
<0, 4, 9>
```

Furthermore, the rank of $E(\overline{F_3}(t))$ is 4.

Example H123E3

We consider again the curve $y^2 = x^3 + (t^4 + 2t^2)x + t^3 + 2t$ over $\mathbf{F}_3(t)$, which was found to have rank 2 and regulator 12 in Example H123E1.

```
> F<t> := FunctionField(GF(3));
> E := EllipticCurve([ t^4 + 2*t^2, t^3 + 2*t ]);
> AnalyticInformation(E);
<2, 8, 12>
```

This confirms that the rank is 2 and tells us that the regulator multiplied by the order of Sha is 12; hence the Tate–Shafarevich group is trivial. Again this is unconditional.

`NumberOfPointsOnSurface(E, e)`

Returns the number of points on the Kodaira–Néron model of the elliptic curve E over an extension of the constant field of degree e.

`NumbersOfPointsOnSurface(E, e)`

If an elliptic curve E is defined over a function field over a finite field of q elements then this intrinsic computes a sequence containing the number of points on the corresponding Kodaira–Néron model of E over extensions of $\mathbf{F}_q$ of degree up to e, together with the number of points on the base curve over these extensions.

`BettiNumber(E, i)`

Returns the i-th Betti number of the Kodaira–Néron model of the elliptic surface corresponding to the elliptic curve E.

`CharacteristicPolynomialFromTraces(traces)`

Given a sequence $traces = [t_1, t_2, \ldots, t_d]$, this function computes a polynomial of degree d with constant coefficient equal to 1 such that the j-th powersum of the reciprocal zeroes of this polynomial is equal to t_j. This is typically used when one wants to compute the characteristic polynomial of Frobenius acting on some piece of cohomology from traces of various powers of Frobenius.

`CharacteristicPolynomialFromTraces(traces, d, q, i)`

Given a sequence of field elements $[t_1, t_2, \ldots, t_k]$ and integers d, q, and i, this function computes a polynomial of degree d with constant coefficient 1 such that the reciprocals of the zeroes of this polynomial have absolute value $q^{i/2}$ and the j-th powersum of the reciprocals of these zeroes equals t_j. For this function one must have k at least equal to Floor$(d/2)$. A priori the leading coefficient of the polynomials is not determined from the list of traces; in this case two polynomials are returned, one for each choice of sign. One usually needs to provide Ceiling$(d/2)$ traces in order to determine the sign, but in certain cases more traces are required. This function is typically used when one wants to compute the characteristic polynomial of Frobenius acting on a piece of the i-th cohomology group from traces of various powers of Frobenius.

123.9 Action of Frobenius

`Frobenius(P, q)`

The q-th power Frobenius map on the point P of an elliptic curve that can be defined over a function field with constant field $\mathbf{F}_q$).

`FrobeniusActionOnPoints(S, q : parameters)`

`gram`	AlgMatElt	*Default :*

A matrix representing the q-power Frobenius map on the subgroup of the geometric Mordell–Weil group (modulo torsion) with the given basis S. (This subgroup is assumed to be invariant under the q-power Frobenius.)

The optional parameter `gram` should be the Gram matrix with respect to the height pairing of the points in S.

`FrobeniusActionOnReducibleFiber(L)`

Given reduction data L for an elliptic curve E, such as given by the command `LocalInformation`, this function returns a matrix representing the Frobenius action on the non-identity components of corresponding fibres.

`FrobeniusActionOnTrivialLattice(E)`

Given an elliptic curve E defined over a rational function field over a finite field, returns a matrix representing the Frobenius action on fibre components and the zero section of the corresponding elliptic surface.

123.10 Extended Examples

Example H123E4

In this example we construct an elliptic curve by using a pencil of cubic curves passing through 8 given points. The 8 points are defined over $GF(1831^8)$ and form a Galois orbit over $GF(1831)$.

```
> p := 1831;
> F := GF(p);
> Fe<u> := ext<F | 8>;
> K<t> := FunctionField(F);
> P2<X,Y,Z> := ProjectivePlane(K);
> // define the 8 points:
> points := [ [ u^(p^i), (u^3+u+1)^(p^i) ] : i in [0..7] ];
> M := [ [ p[1]^i*p[2]^j : i in [0..3-j], j in [0..3] ] : p in points ];
> // find the coefficients of 2 cubics that pass through the points:
> B := Basis(Kernel(Transpose(Matrix(M))));
> R<x,y> := PolynomialRing(F, 2);
> mono_aff := [ x^i*y^j : i in [0..3-j], j in [0..3] ];
> // f1 and f2 are the cubics:
> f1 := &+[ (F!B[1][i])*mono_aff[i] : i in [1..10] ];
> f2 := &+[ (F!B[2][i])*mono_aff[i] : i in [1..10] ];
```

```
> // Find the 9th intersection point, which we use as zero, to put
> // it to a nice Weierstrass model :
> P9 := Points(Scheme(Spec(R), [f1, f2]))[1];
> F1 := Homogenization(Evaluate(f1, [X, Y]), Z);
> F2 := Homogenization(Evaluate(f2, [X, Y]), Z);
> C := Curve(P2, F1 + t*F2);
> E := MinimalDegreeModel(EllipticCurve(C, C![P9[1], P9[2]]));
```

We could transfer the 8 points to this Weierstrass model, and use them to determine the Mordell–Weil group. Instead, we will see what MAGMA is able to compute just from the Weierstrass model.

```
> KodairaSymbols(E);
[ <I1, 1>, <I1, 7>, <II, 1>, <I1, 2> ]
```

All fibres are irreducible. According to the theory of rational elliptic surfaces, the geometric Mordell–Weil lattice should then be isomorphic to the root lattice E_8. We can check this. We also compute the Mordell–Weil lattice over the ground field.

```
> Lgeom := GeometricMordellWeilLattice(E);
> IsIsomorphic(Lgeom, Lattice("E", 8));
true
[-1  0 -2  1  1 -1  0 -1]
[ 0  0  0  1  0  0 -1  1]
[ 2  0  2 -2 -1  1  1  0]
[-2  0 -1  0  1  0  0 -1]
[ 1  0  0  1 -1 -1 -1  1]
[-1  0  0  0  1  0  1 -1]
[ 2 -1  2 -3 -1  1  1  1]
[ 1 -1  1 -1  0  0  0  0]
> L, f := MordellWeilLattice(E);
> L;
Standard Lattice of rank 1 and degree 1
Inner Product Matrix:
[8]
> f(L.1);
((1057*t^8 + 384*t^7 + 351*t^6 + 728*t^5 + 872*t^4 + 948*t^3 + 1473*t^2 + 257*t
    + 1333)/(t^6 + 100*t^5 + 1565*t^4 + 1145*t^3 + 927*t^2 + 1302*t + 1197) :
    (1202*t^12 + 1506*t^11 + 1718*t^10 + 1365*t^9 + 656*t^8 + 325*t^7
    + 1173*t^6    + 902*t^5 + 1555*t^4 + 978*t^3 + 616*t^2 + 779*t +
    964)/(t^9 + 150*t^8 + 1520*t^7 + 747*t^6 + 1065*t^5 + 340*t^4 +
    1618*t^3 + 1669*t^2 + 1150*t + 1768) : 1)
```

That the rank equals 1 is not surprising; this point corresponds to the $\mathbf{F}_{1831}$-rational degree 8 divisor consisting of the sum of the 8 points we started with.
To determine the L-function of E one would normally have to count points over various extension fields of $\mathbf{F}_{1831}$, in this case up to $\mathbf{F}_{1831^4}$. This would be costly using current techniques. But since MAGMA is able to determine the geometric Mordell–Weil lattice, it can compute the L-function by simply considering the Galois action on points. MAGMA automatically uses this when asked for the L-function:

```
> LFunction(E);
```

```
-126330075128099803866555841*T^8 + 1
```

Example H123E5

In this example we determine the $\mathbf{C}(t)$-rank of an elliptic curve, following [Klo07].

```
> K<t> := FunctionField(Rationals());
> E := EllipticCurve([-(2*t-1)^3*(4*t-1)^2, t*(2*t-1)^3*(4*t-1)^3]);
```

To determine where the surface has bad reduction we determine the primes at which singular fibres collapse.

```
> &*BadPlaces(E);
(t^5 - 99/32*t^4 + 337/128*t^3 - 251/256*t^2 + 3/16*t - 1/64)/t
> Discriminant(Numerator($1));
-87887055375/4503599627370496
> Factorisation(Numerator($1));
[ <3, 15>, <5, 3>, <7, 2> ]
> Factorisation(Denominator($2));
[ <2, 52> ]
```

So 11 and 13 are the smallest primes of good reduction. We remark that in [Klo07] the primes 17 and 19 were used.

```
> K11<t11> := FunctionField(GF(11));
> E11 := ChangeRing(E,K11);  // Reduce E mod 11
> LFunction(E11);
161051*T^5 - 7986*T^4 - 363*T^3 - 33*T^2 - 6*T + 1
> AnalyticInformation(E11);
<0, 1, 1>
> AnalyticInformation(E11, 2);
<1, 1, 35/2>
```

So modulo 11 the rank is 0, but over $\mathbf{F}_{11^2}(t)$ the rank equals the geometric rank of 1, and the height of a generator is congruent to $\frac{35}{2}$ modulo $\mathbf{Q}^2$. From this it can be concluded that the geometric rank in characteristic 0 is at most 1. As the L-function has odd degree 5 for any p of good reduction, it will always have a zero at $1/p$ or $-1/p$ and consequently the geometric rank modulo p will always be at least 1. To determine what the geometric rank is in characteristic zero, one can combine information at 2 different primes.

```
> K13<t13> := FunctionField(GF(13));
> E13 := ChangeRing(E, K13);  // Redude E mod 13
> AnalyticInformation(E13);
<0, 1, 1>
> AnalyticInformation(E13,2);
<1, 1, 121/2>
```

So over $\mathbf{F}_{13^2}(t)$ the rank equals the geometric rank of 1. The height of a generator is congruent to $\frac{121}{2}$ modulo $\mathbf{Q}^2$. As the quotient of the heights of generators in characteristics 11 and 13 is not a square in $\mathbf{Q}$, there cannot exist a Mordell–Weil group in characteristic zero that both modulo 11 and 13 reduces to a finite index subgroup of the Mordell–Weil group modulo p. Hence one can conclude that the geometric Mordell–Weil rank in characteristic zero is zero.

Example H123E6

In this example we calculate part of the L-function of an elliptic curve for which it is not feasible to compute the L-function completely. The simplest way to compute an L-function is to call `LFunction`, which counts points on the surface over certain constant field extensions. However, if the required extension fields are too big then `LFunction` will not terminate. One can determine the required extension degrees as follows.

```
> K<t> := FunctionField(GF(5));
> E := EllipticCurve([t^9+t^2, t^14+t^4+t^3]);
> h2 := BettiNumber(E, 2);
> N := FrobeniusActionOnTrivialLattice(E);
> [h2, h2 - NumberOfRows(N)];
[ 34, 21 ]
```

So the H^2 has dimension 34, of which a 13-dimensional piece is generated by the trivial lattice and a 21-dimensional piece is unknown. To determine the L-function one would have to at least count points over $\mathbf{F}_{5^{10}}$. As this is not feasible we will instead count points over extension degrees up to 5 and print the first few coefficients of the L-function.

```
> nop := NumbersOfPointsOnSurface(E, 5);
> traces := [ nop[i] - 1 - 25^i - 5^i*Trace(N^i) : i in [1..5] ];
> CharacteristicPolynomialFromTraces(traces);
5750*T^5 + 875*T^4 - 40*T^3 - 40*T^2 + 1
```

123.11 Bibliography

[**Klo07**] Remke Kloosterman. Elliptic K3 surfaces with geometric Mordell-Weil rank 15. *Canadian Mathematical Bulletin*, 50(2):215–226, 2007.

[**Kra77**] K. Kramer. Two-Descent for Elliptic Curves in Characteristic Two. *Trans. Amer. Math. Soc.*, 232:279–295, 1977.

[**Rob07**] David Roberts. *Explicit Descent On Elliptic Curves Over Function Fields.* PhD thesis, University of Nottingham, 2007.

124 MODELS OF GENUS ONE CURVES

Chapter 124
MODELS OF GENUS ONE CURVES

124.1 Introduction

This chapter deals with curves of genus one that are given by equations in a particular form. Most of the functionality involves invariant theory of these models and applications of this theory to arithmetic problems concerning genus one curves over number fields.

Geometrically (viewed over an algebraically closed field), a genus one model of degree n is an elliptic curve embedded in $\mathbf{P}^{n-1}$ via the linear system $|n.O|$. In general, a genus one model of degree n is a principal homogeneous space for an elliptic curve (of order n in the Weil–Châtelet group) which is embedded in $\mathbf{P}^{n-1}$ in an analogous way. Such models are sometimes called genus one normal curves. Not every element of order n in the Weil–Châtelet group admits such an embedding, although it does if it is everywhere locally soluble.

Genus one models may be defined in MAGMA over any ring. The degree n can be 2, 3, 4, or 5. Genus one models have their own type in MAGMA: `ModelG1`, which is not a subtype of any other type. In particular, these objects are not curves or even schemes. The data that defines a genus one model is one of: a multivariate polynomial (for degree 2 or 3), a pair of multivariate polynomials (degree 4), or a matrix of linear forms (degree 5). Each of these are now described in detail.

A genus one model of degree 2 in MAGMA is defined either by a binary quartic $g(x, z)$ (referred to as a model without cross terms), or more generally by an equation $y^2 + f(x, z)y - g(x, z)$ where f and g are homogeneous of degrees 2 and 4 and the variables x, z, y are assigned weights 1, 1, 2 respectively. A binary quartic $g(x, z)$ defines the same model as the equation $y^2 - g(x, z)$. (The implicit map is the projection to $\mathbf{P}^1_{x,z}$, which in this case is not an embedding but rather has degree 2.)

A genus one model of degree 3 in MAGMA is defined by a cubic form in 3 variables (in other words, the equation of a projective plane cubic curve).

A genus one model of degree 4 in MAGMA is defined by a sequence of two homogeneous polynomials of degree 2 in 4 variables. This represents an intersection of two quadric forms in $\mathbf{P}^3$, and is the standard form in which MAGMA returns curves that are obtained by doing `FourDescent` on an elliptic curve.

A genus one model of degree 5 in MAGMA is defined by a 5×5 alternating matrix whose entries are linear forms in 5 variables. The associated subscheme of $\mathbf{P}^4$ is cut out by the 4×4 Pfaffians of the matrix. It is known that every genus one normal curve of degree 5 arises in this way.

Note: Degenerate cases are allowed, which means that the scheme associated to a genus one model is not always a smooth curve of genus 1 (or even a curve).

124.2 Related Functionality

Section 122.2.11, which concerns three descent on elliptic curves, is relevant for studying rational points on plane cubics (genus one models of degree 3) over the rationals.

A very efficient point search for schemes is available; for details see Section 112.13.4. This will find all rational points up to absolute height H on the curve associated to a given genus one model in time $O(H^{2/d})$, where d is the dimension of the ambient projective space.

124.3 Creation of Genus One Models

`GenusOneModel(n, seq)`

`GenusOneModel(seq)`

`GenusOneModel(n, str)`

The genus one model of degree n (where n is 2, 3, 4, or 5) determined by the coefficients in the given sequence or string. The coefficients may belong to any ring.

A sequence $[a, b, c, d, e]$ of length 5 is interpreted as the degree 2 model $ax^4 + bx^3z + cx^2z^2 + dxz^3 + ez^4$. A sequence $[f, g, h, a, b, c, d, e]$ of length 8 is interpreted as the degree 2 model $y^2 + y(fx^2 + gxz + hz^2) - (ax^4 + bx^3z + cx^2z^2 + dxz^3 + ez^4)$.

A sequence $[a, b, c, d, e, f, g, h, i, j]$ of length 10 is interpreted as the degree 3 model $ax^3 + by^3 + cz^3 + dx^2y + ex^2z + fy^2x + gy^2z + hz^2x + iz^2y + jxyz$.

Sequences of lengths 20 or 50 are interpreted as models of degree 4 or 5 respectively; however, it is easier to create these by specifying matrices instead (see below).

The sequence of coefficients can be recovered by calling `Eltseq`.

`GenusOneModel(C)`

A genus one model that represents the given curve C.

For degree 2, C should either be a subscheme of a weighted projective space $\mathbf{P}(1, 1, 2)$, or a hyperelliptic curve. For degrees $n = 3$, 4, or 5, C should be a genus one normal curve of degree n; in other words, a plane cubic for $n = 3$, an intersection of two quadrics in $\mathbf{P}^3$ for $n = 4$, or an intersection of five quadrics in $\mathbf{P}^4$ for $n = 5$.

`GenusOneModel(f)`

`GenusOneModel(seq)`

The genus one model given by the polynomial f or the sequence of equations `seq`.

`GenusOneModel(n, E)`

A genus one model of degree n (where n is 2, 3, 4, or 5) representing the elliptic curve E embedded in $\mathbf{P}^{n-1}$ via the linear system $|n.O|$. Also returned are the image of the embedding as a curve C together with the maps of schemes $E \to C$ and $C \to E$.

`GenusOneModel(mat)`

The genus one model of degree 5 associated to the given 5×5 matrix.

`GenusOneModel(mats)`

The genus one model of degree 4 determined by the given pair of 4×4 symmetric matrices in the sequence `mats`. (The matrices can be recovered by calling `ModelToMatrices`).

`CompleteTheSquare(model)`

Given a genus one model of degree 2, returns a simplified genus one model of degree 2 without cross terms; this is computed by completing the square on the multivariate polynomial defining the original model.

`RandomGenusOneModel(n)`
`RandomModel(n)`

`Size` RngIntElt *Default :*

A random genus one model of degree n, where n is 2, 3, 4, or 5.

`GenericModel(n)`

The generic genus one model of degree n, where n is 2, 3, 4 or 5. The coefficients are indeterminates in a suitable polynomial ring.

`ChangeRing(model, B)`

The genus one model defined over the ring B obtained by coercing the coefficients of the given genus one model into B.

`CubicFromPoint(E, P)`

The 3-covering corresponding to the rational point P on an elliptic curve E. The 3-covering is returned as the equation of a projective plane cubic curve. Also returned are the covering map and a point that maps to P under the covering map.

`HesseModel(n, seq)`

A genus one model of degree n invariant under the standard representation of the Heisenberg group. The second argument should be a sequence of two ring elements.

`DiagonalModel(n, seq)`

A genus one model of degree n invariant under the diagonal action of μ_n. The second argument should be a sequence of n ring elements.

Example H124E1

We construct the genus one model of degree 5 obtained from the generic elliptic curve $E_{a,b} : y^2 = x^3 + ax + b$ over $\mathbf{Q}(a, b)$. The model is the image of $E_{a,b}$ under the embedding in $\mathbf{P}^4$ given by the linear system $|5.O|$.

```
> K<a,b> := FunctionField(Rationals(), 2);
> Eab := EllipticCurve([a, b]);
> model := GenusOneModel(5, Eab);
> model;
[           0 -b*x1 - a*x2           x5           x4           x3]
[ b*x1 + a*x2            0           x4           x3           x2]
[         -x5          -x4            0          -x2            0]
[         -x4          -x3           x2            0           x1]
[         -x3          -x2            0          -x1            0]
```

From this matrix, which is the data storing the model, the equations of the curve in $\mathbf{P}^4$ can be computed; they are quadratic forms given by the 4×4 Pfaffians of the matrix.

```
> Equations(model);
[
    -x1*x4 + x2^2,
    x1*x5 - x2*x3,
    b*x1^2 + a*x1*x2 + x2*x4 - x3^2,
    -x2*x5 + x3*x4,
    -b*x1*x2 - a*x2^2 + x3*x5 - x4^2
]
```

Note that the degree 5 model has the same invariants $c4, c6, \Delta$ as $E_{a,b}$:

```
> Invariants(model);
-48*a
-864*b
-64*a^3 - 432*b^2
> cInvariants(Eab), Discriminant(Eab);
[
    -48*a,
    -864*b
]
-64*a^3 - 432*b^2
```

124.4 Predicates on Genus One Models

`IsGenusOneModel(f)`
`IsGenusOneModel(seq)`
`IsGenusOneModel(mat)`

Returns `true` if and only if the given polynomial, sequence of polynomials, or matrix determines a "genus one model" in the sense described in the introduction to this chapter. When true, the model is also returned.

Important note: This does *not* imply that the associated scheme is a curve of genus 1, as degenerate cases are allowed!

`IsEquivalent(model1, model2)`
`IsEquivalent(cubic1, cubic2)`

Return `true` if and only if the two given cubics (or genus one models of degree 3) are equivalent as genus one models. In other words, that there exists a linear transformation of the ambient projective space $\mathbf{P}^2_{\mathbf{Q}}$ which takes one cubic to the other, up to scaling. The algorithm is given in [Fis06].

When true, the transformation is also returned as a tuple (the syntax is explained in Section 124.9, on transformations, below).

124.5 Access Functions

`Degree(model)`

The degree (2, 3, 4, or 5) of the given model.

`DefiningEquations(model)`

A sequence containing the equations by which the given genus one model (which must have degree 2, 3, or 4) is defined.

`Equations(model)`

A sequence containing equations for the scheme associated to the given genus one model (of any degree). For degree 2, 3, or 4 this is the same as `DefiningEquations`.

`Matrix(model)`

The defining matrix of a genus one model of degree 5.

`Curve(model)`

`HyperellipticCurve(model)`

`QuadricIntersection(model)`

The curve associated to the given genus one model.

In the degree 2 case the curve is hyperelliptic of the form $y^2 + f(x, z)y - g(x, z) = 0$, but is only created explicitly as a hyperelliptic curve when `HyperellipticCurve` is called; otherwise it is created as a general curve in a weighted projective space in which the variables x, z, and y have weights 1, 1, and 2).

In the degree 4 case the curve is an intersection of two quadrics in $\mathbf{P}^3$.

An error results in degenerate cases where the equations of the model do not define a curve

`Matrices(model)`

For a genus one model of degree 4 this function returns a sequence containing two 4×4 symmetric matrices representing the quadrics.

`BaseRing(model)`

The coefficient ring of the given model.

`PolynomialRing(model)`

The polynomial ring used to define the model.

`Eltseq(model)`

`ModelToString(model)`

A sequence or string containing the coefficients of the defining data of the given genus one model (which is either a polynomial, a pair of polynomials, or a matrix). The model may be recovered by using `GenusOneModel`.

124.6 Minimisation and Reduction

This section contains functions for obtaining a simpler model of a given genus one curve.

We use the following terminology, which has become fairly standard. Here a model will always mean a global integral model.

Minimisation refers to computing a model which is isomorphic to the original one (over its ground field) but possibly not integrally equivalent. A model is *minimal* if it is locally minimal at all primes; a model is locally minimal if the valuation of its discriminant is as small as possible among all integral models. For locally solvable models this minimal discriminant is equal to that of the associated elliptic curve.

Reduction refers to computing a model which is integrally equivalent to the original one (meaning there exist transformations in both directions over the ring of integers of the base field). Informally, a *reduced model* is one whose coefficients have small height. One approach to formulating this precisely is to attach to each model an *invariant* in some symmetric space: The model is then said to be reduced iff its invariant lies in some fixed

fundamental domain. This idea is also the basis for algorithms used to obtain reduced models.

The algorithms used for models of degree 2, 3, and 4 over $\mathbf{Q}$ are described in [CFS10]. For degree 5 it is necessary to use some different techniques developed by Fisher. For models of degree 2 over number fields the algorithms use additional techniques developed by Donnelly and Fisher.

`Minimise(model : parameters)`

`Transformation`	BoolElt	*Default :* `true`
`CrossTerms`	BoolElt	*Default :* `false`
`UsePrimes`	SeqEnum	*Default :* []
`ClassGroupPrimes`	SeqEnum	*Default :* []
Verbose	`Minimise`	*Maximum :* 3

Given a genus one model (of degree 2, 3, 4, or 5) with coefficients in $\mathbf{Q}$, this returns a minimal model in the sense described above.

It is also implemented for genus one models of degree 2 without cross terms, and returns a model that is minimal among models without cross terms when this is possible. When this is not possible (due to class group obstructions) it returns a model that is nearly minimal, in the sense that the extra factor appearing in the discriminant is chosen to have small norm. The primes dividing the extra factor may be specified via the optional argument `ClassGroupPrimes`.

The transformation taking the original model to the minimal model is also returned, unless the optional argument `Transformation` is set to `false`. (The syntax for transformations is explained in Section 124.9).

For degree 2, when the optional argument `CrossTerms` is set to `false` then a model without cross terms is returned that is minimal among models of this kind.

The third value returned is the set of primes where the minimal model has positive level. These are the primes where the model is not soluble over $\mathbf{Q}_p^{nr}$, the maximal non-ramified extension of $\mathbf{Q}_p$ (except for $p = 2$ for models of degree 2 when `CrossTerms` is `false`).

The degree 5 routine is not yet proven to work in all cases.

`Minimise(f)`

Given the equation f of a nonsingular projective plane cubic curve, this returns the equation of a minimal model of the curve and a tuple specifying the transformation (as explained in Section 124.9 below).

`pMinimise(f, p)`

Given the equation f of a nonsingular projective plane cubic curve, this function returns a model of the curve which is minimal at p. Also returned is a matrix M giving the transformation; up to scaling, the minimised cubic is the original cubic evaluated at $M[x, y, z]^{tr}$.

`Reduce(model)`

`Reduce(f)`

Verbose `Reduce` *Maximum :* 3

Given a genus one model of degree 2, 3, or 4 over **Q**, this function computes a reduced model in the sense described above.

The model may instead be described by the appropriate polynomial f, which must be either a homogeneous polynomial that defines a genus one model of degree 2 or 3, or a univariate polynomial that defines a genus one model of degree 2 without cross terms.

The transformation taking the original model to the reduced model is also returned. (The syntax for transformations between genus one models is explained in Section 124.9).

`ReduceQuadrics(seq)`

Verbose `Reduce` *Maximum :* 3

This function computes a reduced basis for the space spanned by the given quadrics (which should be given as a sequence of homogeneous quadratic forms in variables $x_1, \ldots, x_n$). This means making a linear change of the homogeneous coordinates, and also finding a new basis for the space of forms, with the aim of making the resulting coefficients small. The second and third objects returned are matrices S and T giving the transformation. The change of homogeneous coordinate variables is given by S, while T specifies the change of basis of the space of forms. The returned forms are therefore obtained by substituting $[x_1, \ldots, x_n].S$ for $[x_1, \ldots, x_n]$ in the original forms, and then applying T to this basis of forms (where T acts from the left).

The current implementation of this routine is not optimal; nevertheless it is useful in some situations.

124.7 Genus One Models as Coverings

The curve defined by a genus one model of degree n is a principal homogeneous space for some elliptic curve (namely the Jacobian of the curve). The data of the Jacobian, and the covering map of degree n^2, can be read from the invariants and covariants of the model.

Any two models with the same Jacobian can be added together as elements of the Weil–Châtelet group. Below are functions for adding two models of degree 3, and for "doubling" models of degree 4 or 5.

A related function for degree 3 models is `ThreeSelmerElement` (see Section 122.2.11). For degree 4 models the maps can also be computed using `AssociatedEllipticCurve` and `AssociatedHyperellipticCurve` from the package on four descent (see Section 122.2.9).

`Jacobian(model)`

`Jacobian(C)`

The Jacobian, returned as an elliptic curve, of the given genus one model, or of the curve C corresponding to a genus one model.

`nCovering(model : parameters)`

`E`	CrvEll	*Default :*

The covering map from the given genus one model to its Jacobian. Three values are returned: the curve C of degree n corresponding to the given model, its Jacobian as an elliptic curve E, and a map of schemes $C \to E$.

If an elliptic curve E is given it must be isomorphic to the Jacobian and then this curve will be the image of the map.

`AddCubics(cubic1, cubic2 : parameters)`

`AddCubics(model1, model2 : parameters)`

`model1 + model2`

`E`	CrvEll	*Default :*
`ReturnBoth`	BoolElt	*Default :* `false`

Given two ternary cubic polynomials, or two genus one models of degree 3, that both have the same invariants, returns the sum of the corresponding elements of $H^1(\mathbf{Q}, E[3])$.

An error results if the two cubics do not belong to the same elliptic curve E, See Section 122.2.11 for more information about `AddCubics`.

`DoubleGenusOneModel(model)`

Given a genus one model of degree 4 or 5, this function computes twice the associated element in the Weil–Châtelet group and returns this as a genus one model (which will have degree 2 or 5, respectively).

`FourToTwoCovering(model : parameters)`

`FourToTwoCovering(C : parameters)`

`C2`	Crv	*Default :*

Given a genus one model of degree 4 or an associated curve, this function returns three values: the curve C_4 in $\mathbf{P}^3$ corresponding to the model, a plane quartic curve C_2 representing twice the model in the Weil–Châtelet group, and the map of schemes $C_4 \to C_2$. Calling `AssociatedHyperellipticCurve(Curve(model))` provides the same information.

124.8 Families of Elliptic Curves with Prescribed n-Torsion

In [RS95], Rubin and Silverberg explicitly construct families of elliptic curve over $\mathbf{Q}$ which have the same Galois representation on the n-torsion subgroups as a given elliptic curve.

`RubinSilverbergPolynomials(n, J : parameters)`

`Parameter` RngElt *Default :*

Suppose that $n = 2$, 3, 4, or 5 and let $E : y^2 = x^3 + ax + b$ be an elliptic curve over the rationals with j-invariant $1728J$. This function returns polynomials $\alpha(t)$ and $\beta(t)$ that determine a family of elliptic curves with fixed n-torsion, in the following sense: Every nonsingular member F_t of the family $F : y^2 = x^3 + a\alpha(t)x + b\beta(t)$ has $F_t[n]$ isomorphic to $E[n]$ as $\mathbf{Z}[G]$-modules, where G is the absolute Galois group of $\mathbf{Q}$, and furthermore the isomorphisms between $F_t[n]$ and $E[n]$ preserve the Weil pairing. When n is 3, 4, or 5, all such "n-congruent" curves belong to the same family.

124.9 Transformations between Genus One Models

A transformation between two genus one models of degree $n = 2$, 3, 4, or 5 is a tuple consisting of two elements. The second element is an $n \times n$ matrix determining a linear transformation of the projective coordinates (acting on them from the right). The first element is a "rescaling", which is either a matrix or a scalar.

For degree 2 models $q(x, z)$ without cross terms, or for degree 3 models, a transformation is a tuple $\langle k, S\rangle$, where k is an element of the coefficient ring; the transformed model is obtained by making the substitution of coordinate variables determined by S and multiplying the equation by k. For degree 2 models with cross terms a transformation is a tuple $\langle k, [A, B, C], S\rangle$, where k and S are as above and in addition $y + Ax^2 + Bxz + Cz^2$ is substituted for y in the transformation. For degree 4, the first element is a 2×2 matrix; a model of degree 4 is given by two quadric equations, and the 2×2 matrix determines a change of basis of the quadrics (acting on them from the left). A transformation of degree 5 models is a tuple $\langle T, S\rangle$ where T and S are both 5×5 matrices; a model of degree 5 is given by a 5×5 matrix of linear forms M, and the transformed model is given by TM_ST^{tr} where M_S is obtained from M by making the substitution of coordinate variables specified by S.

Two genus one models are said to be equivalent if they differ by one of these transformations. Equivalent models have the same invariants up to scaling by the 4th and 6th powers of some element, given by `ScalingFactor`.

`IsTransformation(n, g)`

Return `true` if and only if the tuple g represents a transformation between genus one models of degree n.

`RandomTransformation(n : parameters)`

`Size`	RngIntElt	*Default :* 5
`Unimodular`	BoolElt	*Default :* `false`
`CrossTerms`	BoolElt	*Default :* `false`

A tuple that represents a random transformation between genus one models of degree n. When `Unimodular` is set to `true` then the returned transformation is integrally invertible.

The optional parameter `Size` is passed to `RandomSL` or `RandomGL`.

In degree 2, if `CrossTerms` is false then the returned transformation preserves the set of models with no cross terms.

`g * model`

`ApplyTransformation(g, model)`

The result of applying the transformation g to the second argument, which should be a genus one model.

`g1 * g2`

`ComposeTransformations(g1, g2)`

The composition $g1 * g2$ of two transformations of genus one models. Transformations of genus one models act on the left, so $(g1 * g2) * f = g1 * (g2 * f)$.

`ScalingFactor(g)`

The scaling factor of a transformation g between genus one models is an element λ such that if a genus one model has invariants c_4 and c_6 then the transformed model has invariants $\lambda^4 c_4$ and $\lambda^6 c_6$.

124.10 Invariants for Genus One Models

`aInvariants(model)`

The invariants $[a_1, a_2, a_3, a_4, a_6]$ of the given genus one model which must have degree 2, 3, or 4. The formulae in the degree 3 case come from [ARVT05].

`bInvariants(model)`

The invariants $[b_2, b_4, b_6, b_8]$ of the given genus one model which must have degree 2, 3, or 4. These are computed from the `aInvariants` in the standard way (as for elliptic curves).

`cInvariants(model)`

The invariants $[c_4, c_6]$ of the given genus one model. For $n = 2$, 3, or 4 these are the classical invariants, as can be found in [AKM$^+$01]. For $n = 5$ the algorithm is described in [Fis08].

`Invariants(model)`

The invariants c_4, c_6 and Δ (the discriminant) of the given genus one model.

`Discriminant(model)`

The discriminant Δ of the given genus one model.

`SL4Invariants(model)`

The SL_4-invariants of a genus one model of degree 4.

124.11 Covariants and Contravariants for Genus One Models

The functions in this section implement the invariant theory developed in [Fis].

`Hessian(model)`

We write X_n for the (affine) space of genus one models of degree n. The module of covariants $X_n \to X_n$ is a free module of rank 2 over the ring of invariants. The generators are the identity map and a second covariant which we term the Hessian. (In the cases where $n = 2$ or 3 this is the determinant of a matrix of second partial derivatives.) This function evaluates the Hessian of the given genus one model.

`CoveringCovariants(model)`

The covariants that define the covering map from the given genus one model to its Jacobian (this is the same as the defining equations of the `nCovering`).

`Contravariants(model)`

We write X_n for the (affine) space of genus one models of degree n, and X_n^* for its dual. The module of contravariants $X_n \to X_n^*$ is a free module of rank 2 over the ring of invariants. This function evaluates the generators P and Q at the given genus one model.

`HesseCovariants(model, r)`

Evaluates a pair of covariants, which depend on an integer r, at the genus one model of degree prime to r. The pencil spanned by these genus one models is a family of genus one curves invariant under the same representation of the Heisenberg group. (In other words, the universal family above a twist of $X(n)$.)

If $r \equiv 1 \pmod{n}$ then the covariants evaluated are the identity map and the Hessian. If $r \equiv -1 \pmod{n}$ then the covariants evaluated are the contravariants. If $n = 5$ there are two further possibilities. We identify $X_5 = \wedge^2 V \otimes W$ where V and W are 5-dimensional vector spaces. Then the covariants evaluated for $r \equiv 2, 3 \pmod{5}$ take values in $\wedge^2 W \otimes V^*$ and $\wedge^2 W^* \otimes V$ respectively.

`HessePolynomials(n, r, invariants : parameters)`

`Variables` [RngMPolElt] *Default :*

The Hesse polynomials $D(x, y), c_4(x, y), c_6(x, y)$. These polynomials give the invariants for the pencil of genus one models computed by `HesseCovariants`. The `RubinSilverbergPolynomials` are closely related to these formulae in the case $r \equiv 1 \pmod{n}$.

124.12 Examples

Example H124E2

We find a cubic which is a counterexample to the Hasse principle. The approach involves the idea of "visibility" of Tate–Shafarevich elements, which was introduced by Mazur (see [CM00]).
The cubic will be a nontrivial element of the Tate–Shafarevich group of the curve 4343B1 in Cremona's tables, which we call E. The cubic will be obtained from a rational point on an auxiliary elliptic curve F.
First, we compute that E has rank 0, and F has rank 1:

```
> E := EllipticCurve([ 0, 0, 1, -325259, -71398995 ]);
> F := EllipticCurve([ 1, -1, 1, -24545, 1486216 ]);
> CremonaReference(E);
4343b1
> RankBounds(E);
0 0
> RankBounds(F);
1 1
```

We take a plane cubic representing one of the nontrivial elements in the 3-Selmer group of F, which has order 3, so that its elements are all in the image of $F(\mathbf{Q})$ since $F(\mathbf{Q})$ has rank 1:

```
> SetClassGroupBounds(500);
> #ThreeSelmerGroup(F);
3
> coverings := ThreeDescent(F);
> coverings;
[
    Curve over Rational Field defined by
    -3*x^2*z - 3*x*y^2 - 27*x*y*z + 12*x*z^2 + 2*y^3 + 21*y^2*z + 3*y*z^2 - 4*z^3
]
> C := Equation(coverings[1]);
```

We now try to find a linear combination of C and its Hessian (which is also a plane cubic) that has j-invariant equal to the j-invariant of E. To find the right linear combination we may work geometrically (that is, with F instead of C since they are isomorphic over $\overline{\mathbf{Q}}$). We work with the family $tF + H$ where t is an indeterminate.

```
> B<t> := PolynomialRing(Rationals());
> F_BR := ChangeRing(Parent(Equation(F)), B);
```

```
> F_B := F_BR ! Equation(F);
> H_B := Hessian(F_B);
> c4,c6,Delta := Invariants(t*F_B + H_B);
```

Alternatively we could get these invariants as follows:

```
> D,c4,c6 := HessePolynomials(3, 1, cInvariants(F) : Variables := [t, 1] );
> Delta := Discriminant(F)*D^3;
// Solve c4(t)^3/Delta(t) = j(E) for t:
> jpoly := c4^3 - jInvariant(E)*Delta;
> Roots(jpoly);
[ <7479/7, 1> ]
```

So we take the following linear combination (and replace the equation by a nicer equation):

```
> C2raw := 7479/7*C + Hessian(C);
> C2 := Reduce(Minimise(C2raw));
> C2;
7*x^3 + 7*x^2*y + 3*x^2*z - 4*x*y^2 - 30*x*y*z + 12*x*z^2 - 13*y^3 - 2*y^2*z -
    15*y*z^2 - 17*z^3
```

The Jacobian of $C2$ is E, so $C2$ is a principal homogeneous space for E of index 3, and in fact it is everywhere locally soluble:

```
> IsIsomorphic(Jacobian(C2), E);
true
> PrimeDivisors(Integers()!Discriminant(GenusOneModel(C2)));
[ 43, 101 ]
> C2_crv := Curve(ProjectiveSpace(Parent(C2)), C2);
> IsLocallySolvable(C2_crv, 43);
true (7 + O(43) : 1 + O(43^50) : O(43))
> IsLocallySolvable(C2_crv, 101);
true (1 + O(101^50) : 32 + O(101) : 1 + O(101))
// Find the preimage of the covering map C2 -> E:
> _, _, maptoE := nCovering(GenusOneModel(C2) : E := E);
> preimage := Pullback(maptoE, E!0);
> Points(preimage);   // Q-rational points
{@ @}
> TorsionSubgroup(E);
Abelian Group of order 1
```

We conclude that $C2$ has no rational points since, as $E(\mathbf{Q})$ is trivial, any rational points on $C2$ must map to O_E.

124.13 Bibliography

[**AKM+01**] Sang Yook An, Seog Young Kim, David C. Marshall, Susan H. Marshall, William G. McCallum, and Alexander R. Perlis. Jacobians of genus one curves. *J. Number Theory*, 90(2):304–315, 2001.

[**ARVT05**] Michael Artin, Fernando Rodriguez-Villegas, and John Tate. On the Jacobians of plane cubics. *Adv. Math.*, 198(1):366–382, 2005.

[**CFS10**] J.E. Cremona, T.A Fisher, and M Stoll. Minimisation and reduction of 2-, 3- and 4-coverings of elliptic curves. *Algebra & Number Theory*, 4(6):763–820, 2010.

[**CM00**] John E. Cremona and Barry Mazur. Visualizing elements in the Shafarevich-Tate group. *Experiment. Math.*, 9(1):13–28, 2000.

[**Fis**] Tom Fisher. The Hessian of a genus one curve.

[**Fis06**] T.A. Fisher. Testing equivalence of ternary cubics. In S. Pauli F. Hess and M.Pohst, editors, *ANTS VII*, volume 4076 of *LNCS*, pages 333–345. Springer-Verlag, 2006.

[**Fis08**] T.A Fisher. The invariants of a genus one curve. *Proc. Lond. Math. Soc.*, 97(3):753–782, 2008.

[**RS95**] K. Rubin and A. Silverberg. Families of elliptic curves with constant mod p representations. In *Elliptic curves, modular forms, & Fermat's last theorem (Hong Kong, 1993)*, Ser. Number Theory, I, pages 148–161. Internat. Press, Cambridge, MA, 1995.

125 HYPERELLIPTIC CURVES

Chapter 125
HYPERELLIPTIC CURVES

125.1 Introduction

This chapter contains descriptions of functions designed to perform calculations with hyperelliptic curves and their Jacobians. A hyperelliptic curve, which is taken to include the genus one case, is given by a nonsingular generalized Weierstrass equation

$$y^2 + h(x)y = f(x),$$

where $h(x)$ and $f(x)$ are polynomials over a field K. The curve is viewed as embedded in a weighted projective space, with weights 1, $g + 1$, and 1, in which the points at infinity are nonsingular.

Functionality for hyperelliptic curves includes optimized algorithms for working on genus two curves over $\mathbf{Q}$, including heights on the Jacobian, and a datatype for the Kummer surface of the Jacobian. For Jacobians of curves over finite fields, there exist specialized algorithms for computing the group structure of the set of rational points.

The category of hyperelliptic curves is `CrvHyp` and points on curves are of type `PtHyp`. Jacobians of hyperelliptic curves are of type `JacHyp` and points `JacHypPt`. Similarly, the Kummer surface of a genus two curve is of type `SrfKum` with points of type `SrfKumPt`.

The initial development of machinery for hyperelliptic curves was undertaken by Michael Stoll, supported by members of the MAGMA group.

125.2 Creation Functions

125.2.1 Creation of a Hyperelliptic Curve

A hyperelliptic curve C given by a generalized Weierstrass equation $y^2 + h(x)y = f(x)$ is created by specifying polynomials $h(x)$ and $f(x)$ over a field K. The class of hyperelliptic curves includes curves of genus one, and a hyperelliptic curve may also be constructed by type change from an elliptic curve E. Note that the ambient space of C is a weighted projective space in which the one or two points at infinity are nonsingular.

`HyperellipticCurve(f, h)`

`HyperellipticCurve(f)`

`HyperellipticCurve([f, h])`

Given two polynomials h and $f \in R[x]$ where R is a field or integral domain, this function returns the nonsingular hyperelliptic curve $C : y^2 + h(x)y = f(x)$. If $h(x)$ is not given, then it is taken as zero. If R is an integral domain rather than a field, the base field of the curve is taken to be the field of fractions of R. An error is returned if the given curve C is singular.

`HyperellipticCurve(P, f, h)`

Create the hyperelliptic curve as described above using the projective space P as the ambient. The ambient P should have dimension 2.

`HyperellipticCurveOfGenus(g, f, h)`

`HyperellipticCurveOfGenus(g, f)`

`HyperellipticCurveOfGenus(g, [f, h])`

Given a positive integer g and two polynomials h and $f \in R[x]$ where R is a field or integral domain, this function returns the nonsingular hyperelliptic curve of genus g given by $C : y^2 + h(x)y = f(x)$. If $h(x)$ is not given, then it is taken as zero. Before attempting to create C, the function checks that its genus will be g by testing various numerical conditions on f and g. If the genus is not correct, a runtime error is raised. If R is an integral domain rather than a field, the base field of C is taken to be the field of fractions of R. An error is returned if the curve C is singular.

`HyperellipticCurve(E)`

Returns the hyperelliptic curve C corresponding to the elliptic curve E, followed by the map from E to C.

125.2.2 Creation Predicates

`IsHyperellipticCurve([f, h])`

Given a sequence containing two polynomials h, $f \in R[x]$, where R is an integral domain, return `true` if and only if $C : y^2 + h(x)y = f(x)$ is a hyperelliptic curve. In this case, the curve is returned as a second value.

`IsHyperellipticCurveOfGenus(g, [f, h])`

Given a positive integer g and a sequence containing two polynomials h, $f \in R[x]$ where R is an integral domain, return `true` if and only if $C : y^2 + h(x)y = f(x)$ is a hyperelliptic curve of genus g. In this case, the curve is returned as a second value.

Example H125E1

Create a hyperelliptic curve over the rationals:

```
> P<x> := PolynomialRing(RationalField());
> C := HyperellipticCurve(x^6+x^2+1);
> C;
Hyperelliptic Curve defined by y^2 = x^6 + x^2 + 1 over Rational Field
> C![0,1,1];
(0 : 1 : 1)
```

Now create the same curve over a finite field:

```
> P<x> := PolynomialRing(GF(7));
> C := HyperellipticCurve(x^6+x^2+1);
```

```
> C;
Hyperelliptic Curve defined by y^2 = x^6 + x^2 + 1 over GF(7)
> C![0,1,1];
(0 : 1 : 1)
```

125.2.3 Changing the Base Ring

`BaseChange(C, K)`

`BaseExtend(C, K)`

Given a hyperelliptic curve C defined over a field k, and a field K which is an extension of k, return a hyperelliptic curve C' over K using the natural inclusion of k in K to map the coefficients of C into elements of K.

`BaseChange(C, j)`

`BaseExtend(C, j)`

Given a hyperelliptic curve C defined over a field k and a ring map $j : k \to K$, return a hyperelliptic curve C' over K by applying j to the coefficients of E.

`BaseChange(C, n)`

`BaseExtend(C, n)`

If C is a hyperelliptic curve defined over a finite field k and a positive integer n, let K denote the extension of k of degree n. This function returns a hyperelliptic curve C' over K using the natural inclusion of k in K to map the coefficients of C into elements of K.

`ChangeRing(C, K)`

Given a hyperelliptic curve C defined over a field k, and a field K, return a hyperelliptic curve C' over K that is obtained from C by by mapping the coefficients of C into K using the standard coercion map from k to K. This is useful when there is no appropriate ring homomorphism between k and K (e.g., when $k = \mathbf{Q}$ and K is a finite field).

Example H125E2

We construct a curve C over the rationals and use `ChangeRing` to construct the corresponding curve $C1$ over $GF(101)$.

```
> P<x> := PolynomialRing(RationalField());
> C := HyperellipticCurve([x^9-x^2+57,x+1]);
> C1 := ChangeRing(C, GF(101));
> C1;
Hyperelliptic Curve defined by y^2 + (x + 1)*y = x^9 + 100*x^2 + 57
over GF(101)
> Q<t> := PolynomialRing(GF(101));
> HyperellipticPolynomials(C1);
t^9 + 100*t^2 + 57
t + 1
> C2, f := SimplifiedModel(C1);
> HyperellipticPolynomials(C2);
4*t^9 + 98*t^2 + 2*t + 27
0
> P1 := C1![31,30,1];
> P1;
(31 : 30 : 1)
> Q := P1@f; // evaluation
> Q;
(31 : 92 : 1)
> Q@@f; // pullback
(31 : 30 : 1)
```

An explanation of the syntax for isomorphisms of hyperelliptic curves and the functions for models is given below.

125.2.4 Models

`SimplifiedModel(C)`

Given a hyperelliptic curve C defined over a field of characteristic not equal to 2, this function returns an isomorphic hyperelliptic curve C' of the form $y^2 = f(x)$, followed by the isomorphism $C \to C'$.

`HasOddDegreeModel(C)`

Given a hyperelliptic curve C, this function returns `true` if C has a model C' of the form $y^2 = f(x)$, with f of odd degree. If so, C' is returned together with the isomorphism $C \to C'$.

`IntegralModel(C)`

`Reduce` BoolElt *Default :* `false`

Given a hyperelliptic curve C defined over the rationals, this function returns an isomorphic curve C' given by polynomials with integral coefficients, together with the isomorphism $C \to C'$. If `Reduce` is set `true`, common divisors of the coefficients are eliminated as far as possible.

`MinimalWeierstrassModel(C)`

`Bound` RngIntElt *Default :* 0

Verbose `CrvHypMinimal` *Maximum :* 3

Given a hyperelliptic curve C defined over the rationals, this function returns a globally minimal Weierstrass model C' of C. If `Bound` is set, it gives an upper bound for the bad primes that are checked. As this calculation uses trial division, `Bound` should not be set much larger than 10^7. The isomorphism $C \to C'$ is returned as a second value.

`pIntegralModel(C, p)`

Verbose `CrvHypMinimal` *Maximum :* 3

Given a hyperelliptic curve C defined over $\mathbf{Q}$ or a rational function field, this function returns a model C' of the curve that is integral at the place p given as an integer, rational, polynomial, rational function or ∞. The isomorphism $C \to C'$ is returned as a second value.

`pNormalModel(C, p)`

Verbose `CrvHypMinimal` *Maximum :* 3

Given a hyperelliptic curve C defined over $\mathbf{Q}$ or a rational function field, this function returns a model C' of the curve that is normal at the place p given as an integer, rational, polynomial, rational function or ∞. The isomorphism $C \to C'$ is returned as a second value.

`pMinimalWeierstrassModel(C, p)`

Verbose `CrvHypMinimal` *Maximum :* 3

Given a hyperelliptic curve C defined over $\mathbf{Q}$ or a rational function field, this function returns a Weierstrass model C' of the curve that is minimal at the place p given as an integer, rational, polynomial, rational function or ∞. The isomorphism $C \to C'$ is returned as a second value.

`ReducedModel(C)`

`Simple`	BoolElt	*Default :* `false`
`Al`	MonStgElt	*Default :* "*Stoll*"
Verbose	`CrvHypReduce`	*Maximum :* 3

Given a hyperelliptic curve C with integral coefficients, this computes a reduced model C'. If the Stoll algorithm is used (default) then the curve argument must have integral coefficients and reduction is performed with respect to the action of $\mathrm{SL}_2(\mathbf{Z})$ on the (x, z)-coordinates. If the Wamelen algorithm is used, (`Al := "Wamelen"`), the curve must have genus 2. The isomorphism $C \to C'$ is currently returned only for the algorithm of Stoll.

`ReducedMinimalWeierstrassModel(C)`

`Simple`	BoolElt	*Default :* `false`
Verbose	`CrvHypMinimal`	*Maximum :* 3
Verbose	`CrvHypReduce`	*Maximum :* 3

Given a hyperelliptic curve C defined over the rationals, this function returns a globally minimal integral Weierstrass model C' of C that is reduced with respect to the action of $\mathrm{SL}_2(\mathbf{Z})$, using Stoll's algorithm in `ReducedModel`. The isomorphism $C \to C'$ is returned as a second value.

`SetVerbose("CrvHypReduce", v)`

This sets the verbose printing level for the curve reduction algorithms of Stoll and Wamelen. The second argument can take integral values in the interval $[0, 3]$, or boolean values: `false` (equivalent to 0) and `true` (equivalent to 1).

125.2.5 Predicates on Models

`IsSimplifiedModel(C)`

Returns `true` if the hyperelliptic curve C is of the form $y^2 = f(x)$.

`IsIntegral(C)`

Given a hyperelliptic curve C, the function returns `true` if C has integral coefficients, and `false` otherwise.

`IspIntegral(C, p)`

Given a hyperelliptic curve C defined over $\mathbf{Q}$ or a rational function field, this function returns `true` if the given model of C is integral at the given place.

`IspNormal(C, p)`

Given a hyperelliptic curve C defined over $\mathbf{Q}$ or a rational function field, this function returns `true` if the given model of C is normal at the given place.

`IspMinimal(C, p)`

Given a hyperelliptic curve C defined over $\mathbf{Q}$ or a rational function field, this function decides whether the given model of C is minimal at the given place. The returned values as follows:

- `false`, `false` if C is not an integral minimal model at the given place.
- `true`, `false` if C is integral and minimal at the given place, but not the unique minimal model (up to transformations that are invertible over the local ring).
- `true`, `false` if C is the unique integral minimal model at the given place, (up to transformations that are invertible over the local ring).

125.2.6 Twisting Hyperelliptic Curves

There are standard functions for quadratic twists of hyperelliptic curves in characteristic not equal to 2. In addition, from the new package of Lercier and Ritzenthaler (described in more detail in the next section) there are functions to return *all* twists of a genus 2 hyperelliptic curve over a finite field of any characteristic.

`QuadraticTwist(C, d)`

Given a hyperelliptic curve C defined over a field k of characteristic not equal to 2 and an element d that is coercible into k, return the quadratic twist of C by d.

`QuadraticTwist(C)`

Given a hyperelliptic curve C defined over a finite field k, return the standard quadratic twist of C over the unique extension of k of degree 2. If the characteristic of k is odd, then this is the same as the twist of C by a primitive element of k.

`QuadraticTwists(C)`

Given a hyperelliptic curve C defined over a finite field k of odd characteristic, return a sequence containing the non-isomorphic quadratic twists of C.

`IsQuadraticTwist(C, D)`

Verbose `CrvHypIso` *Maximum :* 3

Given hyperelliptic curves C and D over a common field k having characteristic not equal to two, return `true` if and only if C is a quadratic twist of D over k. If so, the twisting factor is returned as the second value.

`Twists(C)`

For C a genus 2 or 3 hyperelliptic curve over a finite field k, returns the sequence of *all* twists of C (*ie*, a set of representatives of all isomorphism classes of curves over k isomorphic to C over $\bar{k}$) along with the abstract geometric automorphism group of C (and all of its twists) as a permutation group.

For genus 2, k can be any characteristic. For genus 3, the characteristic of k must be at least 11 and the model of C of the form $y^2 = f(x)$.

There is also a version where the argument is GI, the sequence of Cardona-Quer-Nart-Pujola invariants of a genus 2 curve over a finite field (see Section 125.3.2) which returns the full set of isomorphism classes (twists) of curves over k with the given invariants.

These functions are part of the package contributed by Lercier and Ritzenthaler which is more fully described in the Igusa invariants section.

`HyperellipticPolynomialsFromShiodaInvariants(JI)`

Computes and returns all twists of a genus 3 hyperelliptic curve and its geometric automorphism group corresponding to a sequence of Shioda invariants JI (see Section 125.3.3) over a finite field of characteristic at least 11. In fact the first return value is a sequence of polynomials $f(x)$ of degree 7 or 8 such that the twisted curves correspond to $y^2 = f(x)$. The reason for this is that JI could be a *singular* set of invariants corresponding to polynomials f with discriminant zero. In that case, these do not correspond to hyperelliptic curves, but it might be useful to get the full set of twists anyway.

This function comes from the genus 3 package contributed by Lercier and Ritzenthaler.

Example H125E3

We construct the quadratic twists of the hyperelliptic curve $y^2 = x^6 + x^2 + 1$ defined over $\mathbf{F}_7$.

```
> P<x> := PolynomialRing(GF(7));
> C := HyperellipticCurve(x^6+x^2+1);
> QuadraticTwists(C);
[
    Hyperelliptic Curve defined by y^2 = x^6 + x^2 + 1 over GF(7),
    Hyperelliptic Curve defined by y^2 = 3*x^6 + 3*x^2 + 3 over GF(7)
]
> IsIsomorphic($1[1],$1[2]);
false
```

Example H125E4

We take a hyperelliptic curve over the rationals and form a quadratic twist of it.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x);
> C7 := QuadraticTwist(C, 7);
> C7;
Hyperelliptic Curve defined by y^2 = 7*x^6 + 7*x over Rational Field
```

We now use the function `IsIsomorphic` to verify that C and C_7 are nonisomorphic. We then extend the field of definition of both curves to $\mathbf{Q}(\sqrt{7})$ and verify that the curves become isomorphic over this extension.

```
> IsIsomorphic(C, C7);
```

```
false
> K<w> := ext< Rationals() | x^2-7 >;
> CK := BaseChange(C, K);
> C7K := BaseChange(C7, K);
> IsIsomorphic(CK, C7K);
true (x : y : z) :-> (x : -1/7*w*y : z)
```

Example H125E5

We find all the twists of a supersingular genus 2 curve over $\mathbf{F}_2$.

```
> P<x> := PolynomialRing(GF(2));
> C := HyperellipticCurve(x^5,P!1);
> C;
Hyperelliptic Curve defined by y^2 + y = x^5 over GF(2)
> tws,auts := Twists(C);
> tws;
[
    Hyperelliptic Curve defined by y^2 + y = x^5 over GF(2),
    Hyperelliptic Curve defined by y^2 + y = x^5 + x^4 over GF(2),
    Hyperelliptic Curve defined by y^2 + y = x^5 + x^4 + 1 over GF(2)
]
> #auts; // auts is the geometric automorphism group of C
160
```

125.2.7 Type Change Predicates

`IsEllipticCurve(C)`

The function returns `true` if and only if C is a genus one hyperelliptic curve of odd degree, in which case it also returns an elliptic curve E isomorphic to C followed by the isomorphism $C \to E$ and the inverse isomorphism $E \to C$.

125.3 Operations on Curves

125.3.1 Elementary Invariants

`HyperellipticPolynomials(C)`

The univariate polynomials $f(x)$, $h(x)$, in that order, defining the hyperelliptic curve C by $y^2 + h(x)y = f(x)$.

`Degree(C)`

The degree of the hyperelliptic curve C or a pointset C of a hyperelliptic curve.

`Discriminant(C)`

The discriminant of the hyperelliptic curve C.

`Genus(C)`

The genus of the hyperelliptic curve C.

125.3.2 Igusa Invariants

The Clebsch, Igusa–Clebsch and Igusa invariants may be computed for curves of genus 2.

The MAGMA package implementing the functions was written by Everett W. Howe (however@alumni.caltech.edu) with some advice from Michael Stoll and is based on some `gp` routines written by Fernando Rodriguez–Villegas as part of the Computational Number Theory project funded by a TARP grant. The `gp` routines may be found at `http://www.ma.utexas.edu/users/villegas/cnt/inv.gp`.

In addition, a package of functions written by Reynald Lercier and Christophe Ritzenthaler has been added which contains, amongst other things, functionality for working with a different set of absolute (as opposed to weighted projective) invariants referred to as Cardona-Quer-Nart-Pujola invariants. These work in characteristic 2 as well as other characteristics, although the definitions are different in the two cases.

Rodriguez–Villegas's routines are based on a paper of Mestre [Mes91]. The first part of Mestre's paper summarizes work of Clebsch and Igusa, and is based on the classical theory of invariants. This package contains functions to compute three types of invariants of quintic and sextic polynomials f (or, perhaps more accurately, of binary sextic forms):

- The *Clebsch invariants* A, B, C, D of f, as defined on p. 317 of Mestre;
- The *Igusa–Clebsch invariants* A', B', C', D' of f, as defined on p. 319 of Mestre; and
- The *Igusa invariants* (or *Igusa J-invariants*, or *J-invariants*) J_2, J_4, J_6, J_8, J_{10} of f, as defined on p. 324 of Mestre.

The corresponding functions are `ClebschInvariants`, `IgusaClebschInvariants`, and `JInvariants`, respectively. For convenience, we use `IgusaInvariants` as a synonym for `JInvariants`.

Igusa invariants may be defined for a curve of genus 2 over any field and for polynomials of degree at most 6 over fields of characteristic not equal to 2. The Igusa invariants of the curve $y^2 + hy = f$ are equal to the Igusa invariants of the polynomial $h^2 + 4 * f$ except in characteristic 2, where the latter are not defined. In practice, the functions below will

not calculate the Igusa invariants of a polynomial unless 2 is a unit in the coefficient ring. However, Igusa invariants of curves are available for all coefficient rings. (But see below.)

Igusa invariants are given by a sequence $[J_2, J_4, J_6, J_8, J_{10}]$ of five elements of the coefficient ring of the polynomials defining the curve. This sequence should be thought of as living in weighted projective space, with weights 2, 4, 6, 8, and 10.

It should be noted that many of the people who work with genus 2 curves over the complex numbers prefer not to work with the real Igusa invariants, but rather work with some related numbers, $[I_2, I_4, I_6, I_{10}]$ (or $[A', B', C', D']$ in Mestre's terminology), that we call the *Igusa–Clebsch invariants*, of the curve. Once again, these live in weighted projective space. The Igusa–Clebsch invariants of a polynomial are defined in terms of certain nice symmetric polynomials in its roots, and, in characteristic zero, the J-invariants may be obtained from the I-invariants by some simple homogeneous transformations. In fact, many of the genus-2-curves-over-the-complex-numbers people refer to the elements $i1 := I_2^5/I_{10}$, $i2 := I_2^3 * I_4/I_{10}$ and $i3 := I_2^2 * I_6/I_{10}$ as the "invariants" of the curve. The problem with the Igusa–Clebsch invariants is that they do not work in characteristic 2 and it was for this reason that Igusa defined his J-invariants.

The coefficient ring of the polynomial f must be an algebra over a field of characteristic not equal to 2 or 3.

The Cardona-Quer-Nart-Pujola invariants are three absolute invariants g_1, g_2, g_3 which can be derived from the J-invariants and which provide an affine classification of all genus two curves over a basefield k up to isomorphism over $\bar{k}$. That is, there is a 1-1 correspondence between $\bar{k}$-isomorphism classes of such curves and triples (g_1, g_2, g_3) in k^3. There are also functions to construct a curve with given invariants and to find all twists of such a curve (*ie* representatives of the k-isomorphism classes in the given $\bar{k}$-isomorphism class), which will be described in later sections.

The invariants are different in the characteristic 2 and odd (or 0) characteristic cases. Details about the former case may be found in [CNP05]. See [CQ05] for the latter case. In the odd characteristic case, the formulae for $[g_1, g_2, g_3]$ in terms of the J-invariants are as follows:

$$[\frac{J_2^5}{J_{10}}, \frac{J_2^3 J_4}{J_{10}}, \frac{J_2^2 J_6}{J_{10}}] \quad J_2 \neq 0$$

$$[0, \frac{J_4^5}{J_{10}^2}, \frac{J_4 J_6}{J_{10}}] \quad J_2 = 0, J_4 \neq 0$$

$$[0, 0, \frac{J_6^5}{J_{10}^2}] \quad J_2 = J_4 = 0$$

In the characteristic 2 case, the field k must be perfect. Formulae for the invariants (labelled j_i rather than g_i) may be found on p. 191 of [CNP05].

`ClebschInvariants(C)`

Given a hyperelliptic curve C having genus 2, compute the Clebsch invariants A, B, C and D as described on p. 317 of [Mes91]. The base field of C may not have characteristic 2, 3 or 5. The invariants are found using Überschiebungen.

`ClebschInvariants(f)`

Given a polynomial f of degree at most 6, compute the Clebsch invariants A, B, C and D as described on p. 317 of [Mes91]. The coefficient ring of the polynomial f must be an algebra over a field of characteristic not equal to 2, 3 or 5. The invariants are found using Überschiebungen.

`IgusaClebschInvariants(C: parameters)`

`Quick` BoolElt *Default :* `false`

Given a curve C of genus 2 defined over a field, the Igusa–Clebsch invariants A', B', C' and D' as described on p. 319 of [Mes91] are found. These will be all be zero in characteristic 2. If `Quick` is `true`, the base field of C may not have characteristic 2, 3 or 5 and a faster method using Überschiebungen is employed; otherwise, universal formulae are used.

`IgusaClebschInvariants(f, h)`

Given a polynomial h having degree at most 3 and a polynomial f having degree at most 6, the Igusa–Clebsch invariants A', B', C' and D' of the curve $y^2 + hy - f = 0$ are found. These will be all be zero in characteristic 2.

`IgusaClebschInvariants(f: parameters)`

`Quick` BoolElt *Default :* `false`

Given a polynomial f having degree at most 6 and defined over a ring in which 2 is a unit, the Igusa–Clebsch invariants A', B', C' and D' of the polynomial f are found. These will be all be zero in characteristic 2. If `Quick` is `true`, the coefficient ring of f may not have characteristic 2, 3 or 5 and a faster method using Überschiebungen is employed; otherwise, universal formulae are used.

`IgusaInvariants(C: parameters)`

`JInvariants(C: parameters)`

`Quick` BoolElt *Default :* `false`

Given a curve C of genus 2 defined over a field, the function returns the Igusa invariants (or J-invariants) J_2, J_4, J_6, J_8, J_{10} as described on p. 324 of [Mes91]. If `Quick` is `true`, the base field of C may not have characteristic 2, 3 or 5 and a faster method using Überschiebungen is employed; otherwise, universal formulae are used.

`IgusaInvariants(f, h)`

`JInvariants(f, h)`

Given a polynomial h having degree at most 3 and a polynomial f having degree at most 6, this function returns the Igusa invariants (or J-invariants) J_2, J_4, J_6, J_8, J_{10} of the curve $y^2 + hy = f$. The coefficient ring R of the polynomials h and f must either (a) have characteristic 2, or (b) be a ring in which MAGMA can apply the operator `ExactQuotient(n,2)`. For example, R may be an arbitrary field, the ring of rational integers, a polynomial ring over a field or over the integers and so forth. However, R may not be a p-adic ring, for instance. If the desired coefficient ring does not meet either condition (a) or condition (b), then `ScaledIgusaInvariants` should be used and its invariants then scaled by the appropriate powers of $\frac{1}{2}$.

`IgusaInvariants(f: parameters)`

`JInvariants(f: parameters)`

`Quick` BOOLELT *Default :* `false`

Given a polynomial f having degree at most 6 which is defined over a ring in which 2 is a unit, return the Igusa invariants (or J-invariants) J_2, J_4, J_6, J_8, J_{10} of f. If `Quick` is `true`, the coefficient ring of f may not have characteristic 2, 3 or 5 and a faster method using Überschiebungen is employed; otherwise, universal formulae are used.

`ScaledIgusaInvariants(f, h)`

Given a polynomial h having degree at most 3 and a polynomial f having degree at most 6, return the Igusa J-invariants of the curve $y^2 + hy = f$, scaled by $[16, 16^2, 16^3, 16^4, 16^5]$.

`ScaledIgusaInvariants(f)`

Given a polynomial f having degree at most 6 which is defined over a ring not of characteristic 2, return the Igusa J-invariants of f, scaled by $[16, 16^2, 16^3, 16^4, 16^5]$.

`AbsoluteInvariants(C)`

Given a curve C of genus 2 defined over a field, the function computes the ten absolute invariants of C as described on p. 325 of [Mes91].

`ClebschToIgusaClebsch(Q)`

Convert Clebsch invariants in the sequence Q to Igusa–Clebsch invariants.

`IgusaClebschToIgusa(S)`

Convert Igusa–Clebsch invariants in the sequence S to Clebsch invariants.

`G2Invariants(C)`

Compute and return the sequence of three Cardona-Quer-Nart-Pujola invariants (see the introduction above) for C of genus 2.

`G2ToIgusaInvariants(GI)`

Convert the sequence of Cardona-Quer-Nart-Pujola invariants (see the introduction above) to a corresponding sequence of Igusa J-invariants.

`IgusaToG2Invariants(JI)`

Convert the sequence of Igusa J-invariants to Cardona-Quer-Nart-Pujola invariants (see the introduction above).

125.3.3 Shioda Invariants

The Shioda invariants may be computed for curves of genus 3 in characteristic 0 or characteristic ≥ 11. There are 9 of these, the first 6 being algebraically independent and the last 3 being algebraic over the field generated by the other 6. The discriminant is not one of these invariants. For more details, see [LR12] or [Shi67]. These invariants have weights and naturally give a point in a weighted projective space. There are also intrinsic for computing the 6 Maeda invariants of the curve ([Mae90]).

This functionality comes from a package contributed by Lercier and Ritzenthaler.

`ShiodaInvariants(C)`

`normalize` BoolElt *Default :* `false`

Returns a sequence containing the 9 Shioda invariants of a genus 3 hyperelliptic curve C along with a sequence containing the weight of the corresponding invariant (always $2, 3, 4, 5, 6, 7, 8, 9, 10$). The base field of C must be of characteristic 0 or ≥ 11. There are also versions that takes a single polynomial f of degree less than or equal to 8 as argument (corresponding to the curve $y^2 = f(x)$) or a sequence of two polynomials fh (corresponding to the curve $y^2 + h(x)y = f(x)$). In these cases if the degree of f (resp $f - h^2/4$) is less than 7, or if the discriminant is zero, the invariants will be invariants of the corresponding binary octic but not of a genus 3 hyperelliptic curve anymore. Such sequences of invariants are referred to as *singular*.

The sequence of invariants can be considered as giving a point in a dimension 8 weighted projective space with the above weights. If the parameter `normalize` is set to `true`, the sequence is scaled to give a new sequence representing the same point in a normalized fashion. This means that two isomorphic curves will not necessarily give the same sequence of invariants, but they will always give the same sequence of normalized invariants.

`ShiodaInvariantsEqual(V1,V2)`

Returns whether two sequences of Shioda invariants $V1$ and $V2$ represent the same point in the natural weighted projective space. This is equivalent to asking whether their normized versions (see previous intrinsic) are the same, which means that they represent the same isomorphism class of curves (if they are non-singular).

`DiscriminantFromShiodaInvariants(JI)`

Returns the corresponding discriminant from a sequence of Shioda invariants JI. The discriminant (of a binary octic) is just a polynomial in its 9 Shioda invariants. Note that is scaled if the invariants are scaled, so that the value is different for example, if applied to the results of calling `ShiodaInvariants` with `normalize` set to `true` or `false`. The significant thing though is whether this is non-zero or not. The value is zero if and only if JI are *singular* invariants (associated to a polynomial with multiple roots or of degree ≤ 6).

`ShiodaAlgebraicInvariants(FJI)`

`ratsolve` BoolElt *Default :* `true`

As mentioned in the introduction, the first 6 Shioda invariants are algebraically independent forms on the genus 3 hyperelliptic moduli space and the remaining 3 are algebraic over them.

This intrinsic takes a sequence FJI of 6 elements of a field k that must be of characteristic 0 or ≥ 11. If these are considered to be the first 6 Shioda invariants (of a possibly singular set), this intrinsic returns a sequence containing all possible sequences of the full 9 invariants over k that have those of FJI as the first 6, if the parameter `ratsolve` is `true` (the default). If `ratsolve` is `false`, the sequence returned instead contains the 6 polynomials in 3 variables over k, which defines a dimension 0, degree 5 system whose solutions are the possible last 3 invariants.

Example H125E6

```
> k := GF(37);
> P<t> := PolynomialRing(k);
> C := HyperellipticCurve(t^8+33*t^7+27*t^6+29*t^5+4*t^4+
>         18*t^3+20*t^2+27*t+36);
> ShiodaInvariants(C);
[ 33, 16, 30, 31, 8, 18, 0, 30, 31 ]
[ 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
> ShiodaAlgebraicInvariants([k|33,16,30,31,8,18]);
[
    [ 33, 16, 30, 31, 8, 18, 0, 30, 31 ],
    [ 33, 16, 30, 31, 8, 18, 14, 10, 25 ]
]
> ShiodaAlgebraicInvariants([k|33,16,30,31,8,18] : ratsolve := false);
[
    $.1^5 + 21*$.1^4 + 12*$.1^3 + 3*$.1^2 + 23*$.1,
    $.1^2 + 6*$.1 + 33*$.2 + 23*$.3 + 36,
    $.1*$.2 + 34*$.1 + 36*$.2 + 32*$.3,
    $.1*$.3 + 17*$.1 + 14*$.2^2 + 23*$.2 + 17*$.3 + 21,
    27*$.1 + $.2*$.3 + 36*$.2 + 27*$.3 + 2,
    8*$.1 + 27*$.2^2 + 2*$.2 + $.3^2 + 19*$.3 + 27
```

]

`MaedaInvariants(C)`

Returns the six Maeda field invariants $(I2, I3, I4, I4p, I8, I9)$ of a genus 3 hyperelliptic curve C. Again, the base field of C must be of characteristic 0 or ≥ 11. For this intrinsic, the model of C must also be in simplified $y^2 = f(x)$ form. There is also a version with argument f, a univariate polynomial of degree less than or equal to 8, which returns the Maeda invariants of the binary octic given by homogenizing f (to degree 8). This is the same as asking for the invariants of $y^2 = f(x)$ when f has degree 7 or 8.

125.3.4 Base Ring

`BaseField(C)`
`BaseRing(C)`
`CoefficientRing(C)`

The base field of the hyperelliptic curve C.

125.4 Creation from Invariants

The problem is to construct a curve of genus 2 from a given set of Igusa–Clebsch invariants defined over the same field as the field of moduli (simply the smallest field in which the invariants lie). Mestre [Mes91] shows that this is not always possible, even in theory. But over a finite field or the rationals he gives a method for deciding whether it is possible and if it is, for finding such a curve. Mestre's algorithm was implemented by P. Gaudry. Mestre's algorithm does not work when the curve has a split Jacobian. Cardona and Quer [CQ05] showed that in this case one can always find a curve defined over the field of moduli and gave equations for such a curve. This works over any field of characteristic not 2, 3 or 5.

In any characteristic, the code package of Lercier and Ritzenthaler produces a genus 2 curve from a given set of Cardona-Quer-Nart-Pujola invariants (see Subsection 125.3.2). They use the work of Cardona and Quer cited in the previous paragraph in the odd characteristic case, with some extra work for characteristics 3 and 5. In characteristic 2, they use the models from [CNP05].

Lercier and Ritzenthaler have also contributed a package for genus 3 hyperelliptic curves that produces a curve from a given set of Shioda invariants (see Subsection 125.3.3). This works in characteristic not 2, 3, 5 or 7.

Over a field of characteristic zero, the equation of the curve returned by these methods can involve huge coefficients. For curves over the rationals, this curve can be processed by the algorithm of P. Wamelen [Wam99] and [Wam01]. While, in practice, this algorithm often produces a curve with much smaller coefficients, for certain curves the algorithm may not significantly reduce their size.

`HyperellipticCurveFromIgusaClebsch(S)`

`Reduce` BoolElt *Default :* `false`

This attempts to build a curve of genus 2 with the given Igusa-Clebsch invariants (see Subsection 125.3.2) by Mestre's algorithm (and Cardona's in case of non-hyperelliptic involutions) defined over the field F in which the invariants lie. Currently this field must be the rationals, a number field, or a finite field of characteristic greater than 5. If there exists no such curve defined over F, then either a curve over some quadratic extension is returned (when F is the rationals), or an error results (when F is a number field).

When the base field is the rationals, the parameter `Reduce` invokes Wamelen's reduction algorithm, and is equivalent to a call to `ReducedModel` with the algorithm specified as `"Wamelen"`.

`HyperellipticCurveFromG2Invariants(S)`

From a given sequence of Cardona-Quer-Nart-Pujola invariants (as described in Subsection 125.3.2) over a finite field or the rationals, return a genus 2 curve with these absolute invariants. This works in all characteristics as noted in the introduction to the section. The geometric automorphism group is also returned as a finitely-presented group.

`HyperellipticCurveFromShiodaInvariants(JI)`

`HyperellipticPolynomialFromShiodaInvariants(JI)`

Given a sequence of 9 Shioda invariants JI over the rationals or a finite field k, the first intrinsic returns a genus 3 hyperelliptic curve over k with these invariants. The abstract geometric automorphism group is also returned as a permutation group. This will cause an error if JI are singular Shioda invariants.

The second intrinsic also works for singular invariants. It returns a polynomial f of degree ≤ 8 with the given invariants. If JI are non-singular, $y^2 = f(x)$ is a genus 3 curve with invariants JI.

Example H125E7

A typical run would look like:

```
> SetVerbose("Igusa",1);
> IgCl := [ Rationals() |
>   -549600, 8357701824, -1392544870972992, -3126674637319431000064 ];
> time C := HyperellipticCurveFromIgusaClebsch(IgCl);
Found conic point:
(-64822283782462146583672682837123736006679080996161747198790\
9483959707614135742187 5/136379110441009303141332726677576884 6\
0868572956675822863869630737568561534020494578840036864773 12,\
1018800933596853119179482113258441387813354952965788888004501 1\
3779781952464580535888671875/107297694738676628355433722672369\
8687686225673291912604376829353972347884806380881983953258115 6\
```

```
5610468983296)
Time: 0.990
> time C := ReducedModel(C : Al := "Wamelen");
Time: 161.680
> HyperellipticPolynomials(C);
-23*x^6 + 52*x^5 - 55*x^4 + 40*x^3 - 161*x^2 + 92*x - 409
0
```

An example in characteristic 2 from Cardona-Quer-Nart-Pujola invariants:

```
> k<t> := GF(16);
> g2_invs := [t^3,t^2,t];
> HyperellipticCurveFromG2Invariants(g2_invs);
Hyperelliptic Curve defined by y^2 + (x^2 + x)*y =
   t*x^5 + t*x^3 + t*x^2 + t*x over GF(2^4)
Finitely presented group on 3 generators
Relations
    $.2^2 = Id($)
    $.1^-3 = Id($)
    ($.1^-1 * $.2)^2 = Id($)
    $.3^2 = Id($)
    $.1 * $.3 = $.3 * $.1
    $.2 * $.3 = $.3 * $.2
> _,auts := $1;
> #auts; // auts = D_12
12
```

A genus 3 example using Shioda invariants

```
> k := GF(37);
> FJI := [k| 30, 29, 13, 13, 16, 9];
> ShiodaAlgebraicInvariants(FJI);
[
    [ 30, 29, 13, 13, 16, 9, 14, 35, 0 ],
    [ 30, 29, 13, 13, 16, 9, 36, 32, 22 ],
    [ 30, 29, 13, 13, 16, 9, 36, 32, 23 ]
]
> JI := ($1)[1];
> HyperellipticCurveFromShiodaInvariants(JI);
Hyperelliptic Curve defined by y^2 = 19*x^8 + 13*x^7 + 29*x^6 +
ΨΨ3*x^5 + 16*x^4 + 19*x^3 + x^2 + 27*x + 12
over GF(37)
Symmetric group acting on a set of cardinality 2
Order = 2
```

125.5 Function Field

125.5.1 Function Field and Polynomial Ring

```
FunctionField(C)
```

The function field of the hyperelliptic curve C.

```
DefiningPolynomial(C)
```

A weighted homogeneous polynomial for the hyperelliptic curve C.

```
EvaluatePolynomial(C, a, b, c)
EvaluatePolynomial(C, [a, b, c])
```

Evaluates the homogeneous defining polynomial of the hyperelliptic curve C at the point (a, b, c).

125.6 Points

The hyperelliptic curve is embedded in a weighted projective space, with weights 1, $g + 1$, and 1, respectively on x, y and z. Therefore point triples satisfy the equivalence relation $(x : y : z) = (\mu x : \mu^{g+1} y : \mu z)$, and the points at infinity are then normalized to take the form $(1 : y : 0)$.

125.6.1 Creation of Points

```
C ! [x, y]
C ! [x, y, z]
elt<  PS | x, y  >
elt<  PS | x, y, z  >
```

Returns the point on a hyperelliptic curve C specified by the coordinates (x, y, z). The `elt` constructor takes the pointset of a hyperelliptic curve as an argument. If z is not specified it is assumed to be 1.

```
C ! P
```

Given a point P on a hyperelliptic curve C_1, such that C is a base extension of C_1, this returns the corresponding point on the hyperelliptic curve C. The curve C can be, e.g., the reduction of C_1 to finite characteristic (i.e. base extension to a finite field) or the tautological coercion to itself.

`Points(C, x)`

`RationalPoints(C, x)`

The indexed set of all rational points on the hyperelliptic curve C that have the value x as their x-coordinate. (Rational points are those with coordinates in the coefficient ring of C). Note that points at infinity are considered to have ∞ as their x-coordinate.

`PointsAtInfinity(C)`

The points at infinity for the hyperelliptic curve C returned as an indexed set of points.

`IsPoint(C, S)`

The function returns `true` if and only if the sequence S specifies a point on the hyperelliptic curve C, and if so, returns this point as the second value.

Example H125E8

We look at the point at infinity on $y^2 = x^5 + 1$.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^5+1);
> PointsAtInfinity(C);
{@ (1 : 0 : 0) @}
```

There is only one, and to see that this really is a point on C it must be remembered that in MAGMA, all hyperelliptic curves are considered to live in weighted projective spaces:

```
> Ambient(C);
Projective Space of dimension 2
Variables : $.1, $.2, $.3
Gradings :
1       3       1
```

In fact, the point is nonsingular on C, as we now check. (It's worth remembering that all the functionality for curves, for instance `IsNonSingular`, applies to hyperelliptic curves as a special case.)

```
> pointAtInfinity := C![1,0,0];  // Entering the point by hand.
> IsNonSingular(pointAtInfinity);
true
```

125.6.2 Random Points

`Random(C)`

Given a hyperelliptic curve C defined over a finite field, this returns a point chosen at random on the curve. If the set of all points on C has already been computed, this gives a truly random point, otherwise the ramification points have a slight advantage.

125.6.3 Predicates on Points

`P eq Q`

Returns `true` if and only if the two points P and Q on the same hyperelliptic curve have the same coordinates.

`P ne Q`

Returns `false` if and only if the two points P and Q on the same hyperelliptic curve have the same coordinates.

125.6.4 Access Operations

`P[i]`

The i-th coordinate of the point P, for $1 \leq i \leq 3$.

`Eltseq(P)`

`ElementToSequence(P)`

Given a point P on a hyperelliptic curve, this returns a 3-element sequence consisting of the coordinates of the point P.

125.6.5 Arithmetic of Points

`-P`

`Involution(P)`

Given a point P on a hyperelliptic curve, this returns the image of P under the hyperelliptic involution.

125.6.6 Enumeration and Counting Points

`NumberOfPointsAtInfinity(C)`

The number of points at infinity on the hyperelliptic curve C.

`PointsAtInfinity(C)`

The points at infinity for the hyperelliptic curve C returned as an indexed set of points.

`#C`

Given a hyperelliptic curve C defined over a finite field, this returns the number of rational points on C.

If the base field is small or there is no other good alternative, a naive point counting technique is used. However, if they are applicable, the faster p-adic methods described in the `#J` section are employed (which actually yield the full zeta function of C). As for `#J`, the verbose flag `JacHypCnt` can be used to output information about the computation.

`Points(C)`

`RationalPoints(C)`

`Bound`	RngIntElt	*Default :*
`NPrimes`	RngIntElt	*Default :* 30
`DenominatorBound`	RngIntElt	*Default :* Bound

For a hyperelliptic curve C defined over a finite field, the function returns an indexed set of all rational points on C. For a curve C over $\mathbf{Q}$ of the form $y^2 = f(x)$ with integral coefficients, it returns the set of points such that the naive height of the x-coordinate is less than `Bound`.

For a curve C over a number field, looks for points using a sieve method, described in Appendix A of [Bru02]. The parameter `NPrimes` controls the number of primes which are used and `DenominatorBound` the size of the denominators used.

`PointsKnown(C)`

Returns `true` if and only if the points of the hyperelliptic curve C have been computed. This can especially be helpful when the curve is likely to have many points and when one does not wish to trigger the possibly expensive point computation.

`ZetaFunction(C)`

Given a hyperelliptic curve C defined over a finite field, this function computes the zeta function of C. The zeta function is returned as an element of the function field in one variable over the integers.

If the base field is small or there is no other good alternative, the method used is a naive point count on the curve over extensions of degree $1, ..., g$ of the base field. However, if they are applicable, the faster p-adic methods described in the `#J` section are employed. As for `#J`, the verbose flag `JacHypCnt` can be used to output information about the computation.

`ZetaFunction(C, K)`

Given a hyperelliptic curve C defined over the rationals, this function computes the zeta function of the base extension of C to K. The curve C must have good reduction at the characteristic of K.

Example H125E9

For the following classical curve of Diophantus' *Arithmetica*, it is proved by Joseph Wetherell [Wet97] that Diophantus found all positive rational points. The following MAGMA code enumerates the points on this curve.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x^2+1);
> Points(C : Bound := 1);
{@ (1 : -1 : 0), (1 : 1 : 0), (0 : -1 : 1), (0 : 1 : 1) @}
> Points(C : Bound := 2);
{@ (1 : -1 : 0), (1 : 1 : 0), (0 : -1 : 1), (0 : 1 : 1), (-1 : -9 : 2),
(-1 : 9 : 2), (1 : -9 : 2), (1 : 9 : 2) @}
> Points(C : Bound := 4);
{@ (1 : -1 : 0), (1 : 1 : 0), (0 : -1 : 1), (0 : 1 : 1), (-1 : -9 : 2),
(-1 : 9 : 2), (1 : -9 : 2), (1 : 9 : 2) @}
```

125.6.7 Frobenius

`Frobenius(P, F)`

`Check` BOOLELT *Default :* `true`

Applies the Frobenius $x -> x^{(\#F)}$ to P. If `Check` is `true`, it verifies that the curve of which P is a point is defined over the finite field F.

125.7 Isomorphisms and Transformations

A hyperelliptic curve isomorphism is defined by a linear fractional transformation $t(x : z) = (ax + bz : cx + dz)$, a scale factor e, and a polynomial $u(x)$ of degree at most $g + 1$, where g is the genus of the curve. This data defines the isomorphism of weighted projective points

$$(x : y : z) \mapsto (ax + bz : ey + \tilde{u}(x, z) : cx + dz),$$

where $\tilde{u}$ is the degree $g + 1$ homogenization of u. When not specified, the values of e and u are by default taken to be 1 and 0, respectively.

An isomorphism can be created from the parent structures by coercing a tuple $\langle [a, b, c, d], e, u\rangle$ into the structure of isomorphisms between two hyperelliptic curves, or by creating it as a transformation of a given curve, i.e. creating the codomain curve together with the isomorphism from the given data.

Note that due to the projective weighting of the ambient space of the curve, two equal isomorphisms may have different representations.

125.7.1 Creation of Isomorphisms

`Aut(C)`

Given a hyperelliptic curve C, this returns the structure of all automorphisms of the curve.

`Iso(C1, C2)`

Given hyperelliptic curves $C1$ and $C2$ of the same genus and base field, this returns the structure of all isomorphisms between them.

`Transformation(C, t)`
`Transformation(C, u)`
`Transformation(C, e)`
`Transformation(C, e, u)`
`Transformation(C, t, e, u)`

Returns the hyperelliptic curve C' which is the codomain of the isomorphism from the hyperelliptic curve C specified by the sequence of ring elements t, the ring element e and the polynomial u, followed by the the isomorphism to the curve.

Example H125E10

We create the hyperelliptic curve $y^2 = x^5 - 7$ and apply a transformation to it.

```
> P<x> := PolynomialRing(Rationals());
> H1 := HyperellipticCurve(x^5-7);
> H2, phi := Transformation(H1, [0,1,1,0], 1/2, x^2+1);
> H2;
Hyperelliptic Curve defined by y^2 + (-2*x^2 - 2)*y = -7/4*x^6 - x^4 - 2*x^2 +
    1/4*x - 1 over Rational Field
```

```
> phi;
(x : y : z) :-> (z : 1/2*y + x^2*z + z^3 : x)
> IsIsomorphic(H1, H2);
true (x : y : z) :-> (z : 1/2*y + x^3 + x*z^2 : x)
```

125.7.2 Arithmetic with Isomorphisms

Hyperelliptic curve isomorphisms can be evaluated, inverted, and composed on points as right operators. Note that the functional notation $f(P)$ is not presently available for these maps of curves. However the available map syntax `@` is more consistent with functions as operating on the right (which determines and is apparent from the way composition is defined).

`f * g`

The composition of the maps f and g as right operators.

`Inverse(f)`

The inverse of the hyperelliptic curve isomorphism f.

`f in M`

Given an isomorphism of hyperelliptic curves f, and the structure of isomorphisms M between two hyperelliptic curves, this returns `true` if and only if they share the same domains and codomains.

`P @ f`

`Evaluate(f,P)`

Returns the evaluation of f at a point P in its domain. The same functions apply equally to points in the Jacobian of the domain curve and, in the case of genus 2, to points on the associated Kummer surface.

`P @@ f`

`Pullback(f,P)`

Returns the inverse image of the isomorphism f at a point P in its codomain. The same functions apply equally to points in the Jacobian of the codomain curve and, in the case of genus 2, to points on the associated Kummer surface.

`f eq g`

Given isomorphisms f and g of hyperelliptic curves having the same domain and codomain, this function returns `true` if and only if they are equal. Note that two isomorphisms may be equal even if their defining data are distinct (see example below).

125.7.3 Invariants of Isomorphisms

`Parent(f)`

The "parent structure" of an isomorphism between two hyperelliptic curves (which contains all isomorphisms with the same domain and codomain as the given isomorphism).

`Domain(f)`

The curve which is the domain of the given isomorphism.

`Codomain(f)`

The curve which is the target of the given isomorphism.

125.7.4 Automorphism Group and Isomorphism Testing

This section details the features for computing isomorphisms between two curves and determining the group of automorphisms. The function `IsGL2Equivalent` plays a central role in the isomorphism testing, and is documented here due to its central role in these computations.

The genus 2 package of Lercier and Ritzenthaler provides a function to determine the geometric automorphism group of a genus 2 curve in any characteristic by working with Cardona-Quer-Nart-Pujola invariants (see Subsection 125.3.2). This replaces the old function that only worked in odd (or 0) characteristics. They also provide a function that returns a list of all possible geometric automorphism groups for genus 2 curves over a given finite field and the number of isomorphism classes of curves with each possible group.

Lercier and Ritzenthaler's genus 3 package provides the same for genus 3 hyperelliptic curves, working with Shioda invariants. Here, the characteristic has to be 0 or ≥ 11.

`IsGL2Equivalent(f, g, n)`

This function returns `true` if and only if f and g are in the same $\mathrm{GL}_2(k)$-orbit, where k is the coefficient field of their parent, modulo scalars. The polynomials are considered as homogeneous polynomials of degree n, where n must be at least 4. The second return value is the sequence of all matrix entries $[a, b, c, d]$ such that $g(x)$ is a constant times $f\left((ax+b)/(cx+d)\right)(cx+d)^n$.

`IsIsomorphic(C1, C2)`

Verbose `CrvHypIso` *Maximum :* 3

This function returns `true` if and only if the hyperelliptic curves $C1$ and $C2$ are isomorphic over their common base field. If the curves are isomorphic, an isomorphism is returned.

`AutomorphismGroup(C)`

Given a hyperelliptic curve C of characteristic different from 2, the function returns a permutation group followed by an isomorphism to the group of automorphisms of the curve over its base ring. The curve must be of genus at least one, and the automorphism group is defined to consist of those automorphisms which commute with the hyperelliptic involution, i.e. which induce a well-defined automorphism of its quotient projective line. A third return value gives the action $C \times G \to C$.

Example H125E11

We give an example of the computation of the automorphism group of a genus one hyperelliptic curve.

```
> P<x> := PolynomialRing(GF(3));
> C1 := HyperellipticCurve(x^3+x);
> G1, m1 := AutomorphismGroup(C1);
> #G1;
> [ m1(g) : g in G1 ];
[
    (x : y : z) :-> (x : y : z),
    (x : y : z) :-> (x : -y : z),
    (x : y : z) :-> (z : y : x),
    (x : y : z) :-> (z : -y : x)
]
```

We note that due to the weighted projective space, the same map may have a non-unique representation, however the equality function is able to identify equivalence on representations.

```
> f := m1(G1.3);
> f;
(x : y : z) :-> (z : y : x)
> g := Inverse(f);
> g;
(x : y : z) :-> (2*z : y : 2*x)
> f eq g;
true
```

We see that the geometric automorphism group is much larger. By base extending the curve to a quadratic extension, we find the remaining automorphisms of the curve.

```
> K<t> := GF(3,2);
> C2 := BaseExtend(C1, K);
> G2, m2 := AutomorphismGroup(C2);
> #G2;
48
> O := C2![1,0,0];
> auts := [ m2(g) : g in G2 ];
> [ f : f in auts | O@f eq O ];
[
    (x : y : z) :-> (x : y : z),
```

```
    (x : y : z) :-> (x : -y : z),
    (x : y : z) :-> (x : t^2*y : 2*z),
    (x : y : z) :-> (x + t^6*z : t^2*y : 2*z),
    (x : y : z) :-> (x + t^6*z : y : z),
    (x : y : z) :-> (x + t^2*z : y : z),
    (x : y : z) :-> (x + t^2*z : t^2*y : 2*z),
    (x : y : z) :-> (x : t^6*y : 2*z),
    (x : y : z) :-> (x + t^6*z : t^6*y : 2*z),
    (x : y : z) :-> (x + t^6*z : -y : z),
    (x : y : z) :-> (x + t^2*z : -y : z),
    (x : y : z) :-> (x + t^2*z : t^6*y : 2*z)
]
> #$1;
12
```

Note that this curve is an example of a supersingular elliptic curve in characteristic 3. In the final computation we restrict to the automorphisms as an elliptic curve, i.e. those which fix the point at infinity — the identity element of the group law.
In the context of hyperelliptic curves of genus one, the group of automorphisms must stabilize the ramification points of the hyperelliptic involution. These are precisely the 2-torsion elements as an elliptic curve. So we have an group extension by the 2-torsion elements, acting by translation. Converting to an elliptic curve, we find that there are two 2-torsion elements over $\mathbf{F}_3$:

```
> E1 := EllipticCurve(C1);
> A1 := AbelianGroup(E1);
> A1;
Abelian Group isomorphic to Z/4
Defined on 1 generator
Relations:
    4*$.1 = 0
```

We see that two of the 2-torsion elements are defined over $\mathbf{F}_3$, and the remaining ones appear over the quadratic extension. So in the former case the automorphism group is an extension of the elliptic curve automorphism group (of order 2) by $\mathbf{Z}/2\mathbf{Z}$, and in latter case the automorphism group is an extension (of the group of order 12) by the abelian group isomorphic to $\mathbf{Z}/2\mathbf{Z} \times \mathbf{Z}/2\mathbf{Z}$. Note that there exist other curve automorphisms given by translations by other torsion points (under the addition as an elliptic curve), but that do not commute with the hyperelliptic involution, hence do not enter into the hyperelliptic automorphism group.

`GeometricAutomorphismGroup(C)`

Given a hyperelliptic curve C of genus 2 or 3 the function returns a finitely-presented group isomorphic to the geometric automorphism group, i.e. the automorphism group of the curve over an algebraic closure of its base field.

The method used for genus 2 is to compute the Cardona-Quer-Nart-Pujola invariants (see Subsection 125.3.2) of C and use the classification of the possible automorphism groups in terms of the invariants. See [SV01] and [CQ05] for the odd (and 0) characteristic case and [CNP05] for characteristic 2.

The method used for genus 3 is to compute the Shioda invariants (see Subsection 125.3.3) of C and use the classification of the possible automorphism groups in terms of the invariants. For this case, the base field must have characteristic 0 or ≥ 11.

There is also a genus 2 version where the argument is the sequence GI of Cardona-Quer-Nart-Pujola invariants of a curve rather than the actual curve. This avoids actually constructing the curve, in case the user is starting from the invariants.

The functions are part of packages for genus 2 and 3 curves contributed by Reynald Lercier and Christophe Ritzenthaler.

`GeometricAutomorphismGroupFromShiodaInvariants(JI)`

There is a variant of the last intrinsic for genus 3 curves where the argument is the sequence JI of Shioda invariants of a curve rather than the actual curve. This avoids actually constructing the curve, in case the user is starting from the invariants. The same restrictions on the characteristic of the base field apply.

Example H125E12

We give examples of the computation of the geometric automorphism group of a genus two and a genus three hyperelliptic curve.

```
> P<x> := PolynomialRing(RationalField());
> f := x^6+x^3+13;
> C := HyperellipticCurve(f);
> time GeometricAutomorphismGroup(C);
Permutation group acting on a set of cardinality 6
Order = 12 = 2^2 * 3
    (1, 2, 3, 4, 5, 6)
    (1, 6)(2, 5)(3, 4)
Time: 0.010
> f := x^8-1;
> C1 := HyperellipticCurve(f);
> GeometricAutomorphismGroup(C1);
GrpPC of order 32 = 2^5
PC-Relations:
    $.1^2 = $.4,
    $.3^2 = $.5,
    $.2^$.1 = $.2 * $.3,
    $.3^$.1 = $.3 * $.5,
    $.3^$.2 = $.3 * $.5
```

Note that `AutomorphismGroup` can be used to retrieve the same (and more!) information but this can be much slower.

```
> aut := AutomorphismGroup(C);
> aut;
Symmetric group aut acting on a set of cardinality 2
Order = 2
```

```
  (1, 2)
  Id(aut)
```

We need to extend the field!

```
> Qbar := AlgebraicClosure();
> Cbar := BaseChange(C, Qbar);
> time autbar := AutomorphismGroup(Cbar);
Time: 332.290
> autbar;
Permutation group autbar acting on a set of cardinality 12
Order = 12 = 2^2 * 3
  (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)(11, 12)
  Id(autbar)
  (1, 3)(2, 4)(5, 7)(6, 8)(9, 11)(10, 12)
  (1, 5, 9)(2, 6, 10)(3, 11, 7)(4, 12, 8)
  (1, 7)(2, 8)(3, 9)(4, 10)(5, 11)(6, 12)
  (1, 9, 5)(2, 10, 6)(3, 7, 11)(4, 8, 12)
  (1, 11)(2, 12)(3, 5)(4, 6)(7, 9)(8, 10)
> IdentifyGroup(autbar);
<12, 4>
```

`GeometricAutomorphismGroupGenus2Classification(F)`

Given a finite field F (of any characteristic), the function returns two sequences. The first gives a list of all possible geometric automorphism groups for genus 2 curves defined over F and the second gives the corresponding number of isomorphism (over $\bar{F}$, the algebraic closure of F) classes of F-curves having the given automorphism group. The groups are represented as finitely-presented groups.

This function is part of a package for genus 2 curves contributed by Reynald Lercier and Christophe Ritzenthaler and is based on the classification analysis in [Car03] and [CNP05].

`GeometricAutomorphismGroupGenus3Classification(F)`

Given a finite field F of characteristic ≥ 11, the function returns two sequences. The first gives a list of all possible geometric automorphism groups for genus 3 curves defined over F and the second gives the corresponding number of isomorphism (over $\bar{F}$, the algebraic closure of F) classes of F-curves having the given automorphism group. The groups are represented as permutation groups.

This function is part of a package for genus 3 curves contributed by Reynald Lercier and Christophe Ritzenthaler.

Example H125E13

We determine the possible (geometric) automorphism groups for genus 2 curves over $\mathbf{F}_2$.

```
> gps,ncls :=  GeometricAutomorphismGroupGenus2Classification(GF(2));
> [#gp : gp in gps];
[ 2, 12, 32, 160 ]
> ncls;
[ 5, 1, 1, 1 ]
```

125.8 Jacobians

The Jacobian of a hyperelliptic curve is implemented as the divisor class group of the curve. In particular, no equations giving the Jacobian as a variety ever appear. The Jacobian of any hyperelliptic curve can be created, but most of the interesting functionality is over finite fields, or for genus 2 over number fields or $\mathbf{Q}$.

125.8.1 Creation of a Jacobian

`Jacobian(C)`

The Jacobian of the hyperelliptic curve C.

125.8.2 Access Operations

`Curve(J)`

The hyperelliptic curve from which the Jacobian J was constructed.

`Dimension(J)`

The dimension of the Jacobian J as an algebraic variety, equal to the genus of the curve C of which J is the Jacobian.

125.8.3 Base Ring

`BaseField(J)`

`BaseRing(J)`

`CoefficientRing(J)`

The base field of the Jacobian J.

125.8.4 Changing the Base Ring

`BaseChange(J, F)`

`BaseExtend(J, F)`

The base extension of the Jacobian J to the field F.

`BaseChange(J, j)`

`BaseExtend(J, j)`

The base extension of the Jacobian J obtained by the map j, where j is a ring homomorphism with the base field of C as its domain.

`BaseChange(J, n)`

`BaseExtend(J, n)`

The base extension of the Jacobian J over a finite field to its degree n extension.

125.9 Richelot Isogenies

Let k be a field of characteristic different from 2. We consider a curve of genus 2, given by an equation

$$C : y^2 = f(x)$$

where $f(x)$ is a square-free polynomial of degree 5 or 6. Let J be the Jacobian of C. In this section we mean by a *Richelot isogeny* a polarized isogeny $\Phi : J \to A$ between principally polarized abelian surfaces, such that the kernel of Φ over the algebraic closure has group structure $\mathbf{Z}/2\mathbf{Z} \times \mathbf{Z}/2\mathbf{Z}$. We have that $J[\Phi] \subset J[2]$ is maximal isotropic with respect to the Weil-pairing on $J[2]$.

We can represent the points of $J[\Phi]$ over the algebraic closure as divisors in the following way. We write

$$f(x) = cQ_1(x)Q_2(x)Q_3(x),$$

where the Q_i are degree 2 polynomials if $\deg(f) = 6$. If $\deg(f) = 5$ then Q_2, Q_3 are of degree 2 and Q_1 is of degree 1 and is considered to represent a degree 2 with a root at $x = \infty$. Then

$$\{0, [Q_1(x) = 0] - [Q_2(x) = 0], [Q_1(x) = 0] - [Q_3(x) = 0], [Q_2(x) = 0] - [Q_3(x) = 0]\}$$

is the kernel of some Richelot-isogeny and, conversely, any Richelot kernel can be represented in this way.

The Q_i do not have to be defined over the ground field individually. One way of specifying such a kernel is to write

$$f(x) = c\mathrm{Norm}_{L[x]/k[x]}Q(x)$$

where $L = k[t]/(h(t))$ for some square free cubic polynomial h and $Q(x) \in L[x]$. If L is totally split and A is the Jacobian of a genus 2 curve then a description the genus 2 curve

D such that $A = \mathrm{Jac}(D)$ is classically known. See [Smi05], Chapter 8 for an exposition that is relatively close to the description given here. See [BD09] for a description of D for general L.

In special cases, the codomain A can be a product of elliptic curves or the Weil-restriction of an elliptic curve with respect to a quadratic extension of k. In that case, the curve C has extra automorphisms that respects the representation $f(x) = c\mathrm{Norm}_{L[x]/k[x]}Q(x)$. and one can find the relevant elliptic curves as quotients of C.

`RichelotIsogenousSurfaces(J)`

`RichelotIsogenousSurfaces(C)`

`Kernels` BoolElt *Default :* `true`

Computes the richelot isogenies defined over the basefield of the given abelian varieties and returns a list of objects representing the codomains. If the codomain is the Jacobian of a genus 2 curve, then that Jacobian is returned or, if a curve is given instead of a Jacobian, the corresponding curve.

If the codomain is a product of elliptic curves, a Cartesian product of elliptic curves is returned. If the codomain is the Weil restriction of an elliptic curve relative to a quadratic extension, then the elliptic curve over the quadratic extension is returned.

If `Kernels` is specified then a second list is returned, consisting of quadratic polynomials over cubic algebras. Each describes the kernel of the relevant isogeny.

`RichelotIsogenousSurface(J, kernel)`

`RichelotIsogenousSurface(C, kernel)`

Given a genus 2 Jacobian and a Richelot kernel, return the codomain. The genus 2 curve must be given by a model of the form $C : y^2 = f(x)$ and the kernel must be a quadratic polynomial $Q(x)$ over a cubic algebra L such that $\mathrm{Norm}_{L[x]/k[x]}Q(x) = cf(x)$. The elements of the second list returned by `RichelotIsogenousSurfaces` when given `Kernels:=true` are valid kernel descriptions. The codomain is returned using the same conventions as for `RichelotIsogenousSurfaces`.

Example H125E14

We will determine the Richelot isogenies on the Jacobian of $y^2 = x^5 + x$. This Jacobian has the amusing property that there are 3 such isogenies and that each of the types of codomain (Jacobian, Weil restriction, product of elliptic curves) is represented.

```
> R<x>:=PolynomialRing(Rationals());
> C:=HyperellipticCurve(x^5+x);
> J:=Jacobian(C);
> RichelotIsogenousSurfaces(J);
[*
    Cartesian Product<Elliptic Curve defined by y^2 = x^3 + 5/32*x^2 -
    5/1024*x - 1/32768 over Rational Field, Elliptic Curve defined by
    y^2 = x^3 - 5/32*x^2 - 5/1024*x + 1/32768 over Rational Field>,
```

```
    Jacobian of Hyperelliptic Curve defined by y^2 = -2*x^5 - 2*x over
    Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 5/32*$.1*x^2 + 5/1024*x +
        1/32768*$.1 over Number Field with defining polynomial x^2 + 1
    over the Rational Field
*]
```

We now illustrate how the kernels are represented.

```
> codomains,kernels:=RichelotIsogenousSurfaces(J:Kernels);
> Q:=kernels[1];
> LX<X>:=Parent(Q);
> L<alpha>:=BaseRing(LX);
> Q;
(-1/2*alpha^2 + 2*alpha)*X^2 + (-1/2*alpha^2 + alpha + 1)*X -
    1/2*alpha^2 + 2*alpha
> L;
Univariate Quotient Polynomial Algebra in alpha over Rational Field
with modulus alpha^3 - 4*alpha^2 + 2*alpha
```

Let us check that the norm of `Q` gives us $x^5 + x$ again and that calling `RichelotIsogenousSurface` allows us to recreate the corresponding codomain.

```
> _,swp:=SwapExtension(LX);
> Norm(swp(Q));
x^5 + x
```

We can use Q to recreate the corresponding codomain.

```
> codomains[1] eq RichelotIsogenousSurface(J,Q);
true
```

Finally, to verify that the computed abelian surfaces are all isogenous, we verify that their L-series over $\mathbf{Q}$ are equal. For each type of return value we have to create the L-Series in a slightly different way, but once done, we can easily check that their coefficients agree.

```
> LC:=LSeries(C : ExcFactors:="Ogg");
> myL:=func< A|
>     case<Type(A) | SetCart : LSeries(A[1])*LSeries(A[2]),
>                    JacHyp  : LSeries(Curve(A) : ExcFactors:="Ogg"),
>                    CrvEll  : LSeries(A),
>                    default : false>>;
> cfs:=[c: c in LGetCoefficients(LC,1000)];
> [[c: c in LGetCoefficients(myL(A),1000)] eq cfs : A in codomains];
[ true, true, true ]
```

125.10 Points on the Jacobian

Points on $Jac(C)$ are represented as divisors on C. They can be specified simply by giving points on C, or divisors on C, or in the Mumford representation, which is the way MAGMA returns them (and which it uses to store and manipulate them). Points can be added and subtracted.

Representation of points on $Jac(C)$: Let C be a hyperelliptic curve of genus g. A triple $\langle a(x), b(x), d\rangle$ specifies the divisor D of degree d on C defined by

$$A(x, z) = 0, \quad y = B(x, z)$$

where $A(x, z)$ is the degree d homogenisation of $a(x)$, and $B(x, z)$ is the degree $(g + 1)$ homogenisation of $b(x)$. Note that the equation $y = B(x, z)$ make sense projectively because y has weight $g + 1$ (as always for hyperelliptic curves in MAGMA).

The *point on $Jac(C)$ corresponding to* $\langle a(x), b(x), d\rangle$ is then D minus a multiple of the divisor at infinity on C. So, when there is a single point P_∞ at infinity, we have $D - dP_\infty$. Otherwise, there is a $\mathbf{Q}$-rational divisor $P_{+\infty} + P_{-\infty}$ consisting of the two points at infinity; in this case d is required to be even and we have $D - (d/2)(P_{+\infty} + P_{-\infty})$.

All points on $Jac(C)$ can be expressed in this way, except when g is odd and there are no rational points at infinity (in which case the extra points can not be created in MAGMA, and arithmetic on points is not implemented). There is a uniquely determined "reduced" triple representing each point, which MAGMA uses to represent any point it encounters.

See the examples in the following section ("Creation of Points").

Technical details: In order to make sense, a triple $\langle a(x), b(x), d\rangle$ is required to satisfy:

(a) $a(x)$ is monic of degree at most g;

(b) $b(x)$ has degree at most $g + 1$, and $a(x)$ divides $b(x)^2 + h(x)b(x) - f(x)$, where $h(x)$ and $f(x)$ are the defining polynomials of C;

(c) d is a positive integer with $\deg(a(x)) \leq d \leq g + 1$, such that the degree of $b(x)^2 + h(x)b(x) - f(x)$ is less than or equal to $2g + 2 - d + \deg(a(x))$.

For uniqueness of representation, in the case of one point at infinity, (the odd degree case, in particular), we require that $d = \deg(a(x))$.

A triple $\langle a(x), b(x), d\rangle$ is reduced to a canonical representative as follows:

(a) If $d = \deg(a(x))$, we reduce b mod a, so $\deg(b(x)) < \deg(a(x))$.

(b) If $a = 1$, we assume that $\deg(B(1, z)) = d$.

(c) A unique representative for $b(x)$ is found such that the coefficient of x^k in b is zero for $\deg(a(x)) \leq k \leq \deg(a(x)) + g + 1 - d$.

For certain models of C, not all rational points can be represented in the above form (with the restrictions on d) or the representation is not unique. The bad cases are g odd and (i) 0 or (ii) 2 rational points at infinity.

In case (i) we would have to allow divisors with $\deg(a(x)) = g + 1$, which give linear pencils of equivalent divisors, and in case (ii) a Jacobian point with $d = g+1$ has precisely two canonical representatives.

In case (i), there is no obvious canonical representative for these additional Jacobian points and currently MAGMA does not deal with them. Consequently, arithmetic is not implemented in this case.

However, in case (ii), the two representatives correspond to the two distinguished elements of a linear pencil of divisors which include contributions from one or the other point at infinity. By arbitrarily selecting one of these two infinite points as the default, a unique representative can be chosen. This is now performed internally by MAGMA, the default point being chosen at the time of creation of the Jacobian. Arithmetic is also implemented. Note that in this case, extra work is generally required in point addition to keep track of points at infinity and final reduction to the unique representation. If very fast addition is crucial when C is of this type, it is generally wise to use an isomorphic model with exactly one point at infinity, if possible, by moving a rational Weierstrass point to infinity.

125.10.1 Creation of Points

```
J ! 0
```
```
Id(J)
```
```
Identity(J)
```

The identity element on the Jacobian J.

```
J ! [a, b]
```
```
elt< J | a, b >
```
```
elt< J | [a, b] >
```
```
elt< J | a, b, d >
```
```
elt< J | [a, b], d >
```

The point on the Jacobian J defined by the polynomials a and b and the positive integer d; if not specified then d is taken to be $\deg(a)$.

```
P - Q
```
```
J ! [P, Q]
```
```
elt< J | P, Q >
```

For points P and Q on a hyperelliptic curve, this constructs the image of the divisor class $[P - Q]$ as a point on its Jacobian J.

```
J ! [S, T]
```
```
elt< J | S, T >
```

Given two sequences $S = [P_i]$ and $T = [Q_i]$ of points on the hyperelliptic curve with Jacobian J, each of length n, this returns the image of the divisor class $\sum_i [P_i] - \sum_i [Q_i]$ as a point on the Jacobian J.

`JacobianPoint(J, D)`

The point on the Jacobian J (of a hyperelliptic curve C) associated to the divisor D on C. If D does not have degree 0, then a suitable multiple of the divisor at infinity is subtracted. When the divisor at infinity on C has even degree, D is required to have even degree.

The function works for any divisor such that the corresponding point is definable in MAGMA. It is not implemented for characteristic 2.

`J ! P`

Given a point P on a Jacobian J', construct the image of P on J, where J is a base extension of J'.

`Points(J, a, d)`

`RationalPoints(J, a, d)`

Find all points on the Jacobian J with first component a and degree d. Only implemented for genus 2 curves of the form $y^2 = f(x)$.

Example H125E15

Points on $y^2 = x^6 - 3x - 1$ and their images on the Jacobian.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6-3*x-1);
> C;
Hyperelliptic Curve defined by y^2 = x^6 - 3*x - 1 over Rational Field
> J := Jacobian(C);
> J;  // Magma didn't do much work to create J
Jacobian of Hyperelliptic Curve defined by y^2 = x^6 - 3*x - 1 over Rational Field
```

Find some points on C and map them to J (using the first point to define the map $C \to J$):

```
> ptsC := Points(C : Bound := 100);
> ptsC;
{@ (1 : -1 : 0), (1 : 1 : 0), (-1 : -1 : 3), (-1 : 1 : 3) @}
> ptsJ := [ ptsC[i] - ptsC[1] : i in [2,3,4] ];
> ptsJ;
[ (1, x^3, 2), (x + 1/3, x^3, 2), (x + 1/3, x^3 + 2/27, 2) ]
```

We recreate the first of these, giving it in MAGMA's notation (which is read as "the divisor of degree 2 on C determined by $z^2 = y - x^3 = 0$").

```
> pt1 := elt< J | [1,x^3], 2 >;
> pt1 eq ptsJ[1];
true
```

The degree of the divisor must be specified here, otherwise it is assumed to be $\deg(a(x)) = 0$:

```
> pt1 := J! [1,x^3];
> pt1; pt1 eq J!0;
(1, 0, 0)
true
```

Example H125E16

We define the nontrivial 2-torsion point on the Jacobian of $C : y^2 = (x^2 + 1)(x^6 + 7)$.
The divisor $(y = 0, x^2 + 1 = 0)$ should define the only nontrivial 2-torsion point in $J(\mathbf{Q})$.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve( (x^2+1)*(x^6+7) );
> J := Jacobian(C);
```

Define the point corresponding to the divisor given by $x^2 + 1 = 0, y = 0$ on C. We can use the simpler syntax, not specifying that the degree of the divisor is 2, because this equals the degree of $a(x) = x^2 + 1$.

```
> Ptors := J![x^2+1, 0];
> Ptors;
(x^2 + 1, 0, 2)
```

The alternative syntax would be as follows.

```
> Ptors1 := elt< J | [x^2+1, 0], 2 >;
> Ptors eq Ptors1;  // Are they the same?
true
```

Check that Ptors has order 2:

```
> 2*Ptors;
(1, 0, 0)
> $1 eq J!0;  // Is the previous result really the trivial point on J?
true
> Order(Ptors);  // Just to be absolutely sure ...
2
```

The other factor $x^6 + 7$ will give the same point on J:

```
> Ptors2 := J![x^6+7,0];
(x^2 + 1, 0, 2)
```

Note that MAGMA returned the unique reduced triple representing this point, which means we can easily check whether or not it is the same point as `Ptors`.
For alternative ways to obtain 2-torsion points, see the section on torsion.

Example H125E17

A Jacobian with a point not coming from the curve.
Any curve containing the point $(\sqrt{2}, \sqrt{2})$ has a $\mathbf{Q}$-rational divisor

$$D := (\sqrt{2}, \sqrt{2}) + (-\sqrt{2}, -\sqrt{2}).$$

This will give a nontrivial point in $J(\mathbf{Q})$. For instance, take $y^2 = x^6 - 6$.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6-6);
```

```
> J := Jacobian(C);
```

The divisor D has degree 2 and is given by $x^2 - 2 = y - x = 0$.

```
> pt := J![x^2-2, x];
> pt;  // What is pt?
(x^2 - 2, x, 2)
> Parent(pt);  // Where does pt live?
Jacobian of Hyperelliptic Curve defined by y^2 = x^6 - 6 over Rational Field
> pt eq J!0;  // Is pt equal to 0 on J?
false
> Order(pt);
0
```

This means that P has infinite order in $J(\mathbf{Q})$.
Alternatively we can construct the same point by first constructing the divisor, giving it as an ideal in the homogeneous coordinate ring in which C lives. (Note that Y has weight 3).

```
> P<X,Y,Z> := CoordinateRing(Ambient(C));
> D := Divisor(C, ideal<P | X^2-2*Z^2, Y-X*Z^2> );
> pt1 := J!D;
> pt eq pt1;
true
```

125.10.2 Random Points

`Random(J)`

A random point on a Jacobian J of a hyperelliptic curve defined over a finite field.

125.10.3 Booleans and Predicates for Points

For each of the following functions, P and Q are points on the Jacobian of a hyperelliptic curve.

`P eq Q`

Returns `true` if and only if the points P and Q on the same Jacobian are equal.

`P ne Q`

Returns `false` if and only if the two points P and Q on the same Jacobian are equal.

`IsZero(P)`

`IsIdentity(P)`

Returns `true` if and only if P is the zero element of the Jacobian.

125.10.4 Access Operations

`P[i]`

Given an integer $1 \leq i \leq 2$, this returns the i-th defining polynomial for the Jacobian point P. For $i = 3$, this returns the degree d of the associated reduced divisor.

`Eltseq(P)`

`ElementToSequence(P)`

Given a point P on the Jacobian J of a hyperelliptic curve, the function returns firstly, a sequence containing the two defining polynomials for the divisor associated to P and secondly, the degree of the divisor.

125.10.5 Arithmetic of Points

For each of the following functions, P and Q are points on the Jacobian of a hyperelliptic curve.

`-P`

The additive inverse of the point P on the Jacobian.

`P + Q`

The sum of the points P and Q on the same Jacobian.

`P +:= Q`

Given two points P and Q on the same Jacobian, set P equal to their sum.

`P - Q`

The difference of the points P and Q on the same Jacobian.

`P -:= Q`

Given two points P and Q on the same Jacobian, set P equal to the difference $P-Q$.

`n * P`

`P * n`

The n-th multiple of P in the group of rational points on the Jacobian.

`P *:= n`

Set P equal to the n-th multiple of itself.

125.10.6 Order of Points on the Jacobian

`Order(P)`

Returns the order of the point P on the Jacobian J of a hyperelliptic curve defined over a finite field or the rationals, or 0 if P has infinite order. This first computes $\#J$ when J is defined over a finite field.

`Order(P, l, u)`

`Alg`	MonStg	*Default :* "*Shanks*"
`UseInversion`	BoolElt	*Default :* `true`

Returns the order of the point P where u and l are bounds for the order of P or for the order of J the parent of P. This does not compute $\#J$.

If the parameter `Alg` is set to `"Shanks"` then the generic Shanks algorithm is used, otherwise, when `Alg` is `"PollardRho"`, a Pollard-Rho variant is used (see [GH00]).

If `UseInversion` is `true` the search space is halved by using point negation.

`Order(P, l, u, n, m)`

`Alg`	MonStg	*Default :* "*Shanks*"
`UseInversion`	BoolElt	*Default :* `true`

Returns the order of the point P where u and l are bounds for the order of P or for the order of J the parent of P and where n and m are such that the group order is n mod m. This does not compute $\#J$.

The two parameters `Alg` and `UseInversion` have the same use as in the previous function.

`HasOrder(P, n)`

Given a point P on the Jacobian J of a hyperelliptic curve and a positive integer n, this returns `true` if the order of the point is n.

125.10.7 Frobenius

`Frobenius(P, k)`

`Check`	BoolElt	*Default :* `true`

Given a point P that lies on the Jacobian J of a hyperelliptic curve that is defined over the finite field $k = F_q$, determine the image of P under the Frobenius map $x \to x^q$. If `Check` is `true`, Magma verifies that the Jacobian of P is defined over k.

125.10.8 Weil Pairing

`WeilPairing(P, Q, m)`

Computes the Weil pairing of P and Q, where P and Q are m-torsion points on the 2-dimensional Jacobian J defined over a finite field.

Example H125E18

The following illustrates the use of the Weil Pairing in the MOV-reduction of the discrete logarithm problem on a Jacobian.

```
> PP<x>:=PolynomialRing(GF(2));
> h := PP!1;
> f := x^5 + x^4 + x^3 + 1;
> J := Jacobian(HyperellipticCurve(f,h));  // a supersingular curve
> Jext := BaseExtend(J, 41);
> Factorization(#Jext);
[ <7, 1>, <3887047, 1>, <177722253954175633, 1> ]
> m := 177722253954175633;                 // some big subgroup order
> cofact := 3887047*7;
> P := cofact*Random(Jext);
> Q := 876213876263897634*P;               // Q in <P>
```

Say we want to recompute the logarithm of Q in base P.

```
> Jext2 := BaseExtend(Jext, 6);            // go to an ext of deg 6
> NJ := #Jext2;
>
> R := Random(Jext2);
> R *:= NJ div m^Valuation(NJ, m);
>
> eP := WeilPairing(Jext2!P, R, m);
> eQ := WeilPairing(Jext2!Q, R, m);
> assert eP^876213876263897634 eq eQ;
```

So the discrete log problem on the Jacobian has been reduced to a discrete log problem in a finite field.

125.11 Rational Points and Group Structure over Finite Fields

125.11.1 Enumeration of Points

`Points(J)`

`RationalPoints(J)`

Given a Jacobian J of a hyperelliptic curve defined over a finite field, determine all rational points on the Jacobian J.

125.11.2 Counting Points on the Jacobian

Several algorithms are used to compute the order of a Jacobian depending of its size, its genus and its type. In particular, in genus 2 it includes all the techniques described in [GH00]. The best current algorithms are the ones based on p-adic liftings. These are the defaults when they apply (see below) and the characteristic is not too large. In odd characteristic, Kedlaya's algorithm is used as described in [Ked01]. For characteristic 2, we have Mestre's canonical lift method as adapted by Lercier and Lubicz. Details can be found in [LL]. We also now have an implementation of Vercauteren's characteristic 2 version of Kedlaya (described in [Ver02]). As an added bonus, the p-adic methods actually give the Euler factor (see below). A verbose flag can be set to see which strategy is chosen and the progress of the computation.

The latest p-adic methods, faster than Kedlaya in odd characteristic, are those of Alan Lauder based on deformations over parametrised families. Magma incorporates an implementation for hyperelliptic families by Hendrik Hubrechts (see [Hub06] for details). This has a slightly different interface to the other methods (which all run through the same top-level intrinsics) as it can be used to count points on several members of the family at once.

`SetVerbose("JacHypCnt", v)`

Set the verbose printing level for the point-counting on the Jacobians. Currently the legal values for v are `true`, `false`, 0, 1, 2, 3 or 4 (`false` is the same as 0, and `true` is the same as 1).

`#J`

`Order(J)`

Given the Jacobian J of a hyperelliptic curve defined over a finite field, determine the order of the group of rational points.

There are 4 optional parameters which concern every genus.

`NaiveAlg`	BoolElt	*Default :* `false`
`ShanksLimit`	RngIntElt	*Default :* 10^{12}
`CartierManinLimit`	RngIntElt	*Default :* 5×10^5
`UseSubexpAlg`	BoolElt	*Default :* `true`

The parameter `NaiveAlg` can be set to one if one want to use the naive algorithm for point counting, consisting in counting points on the curve over the first g extensions, where g is the genus. Setting this parameter to `true` annihilates the effects of the others. By default, this strategy is used for small groups and Jacobians in characteristic 2 for which no group law is available.

In odd characteristic, Kedlaya's algorithm can be applied to curves of any genus and it is the default when the group is not too small or the characteristic too large.

In contrast, Mestre's characteristic 2 method has several conditions on its applicability. When these are satisfied and the genus is not too large or the group order too small, it is the default.

- Firstly, the Jacobian must be *ordinary*. This is equivalent to the defining polynomial $h(x)$ having degree g or $g+1$ and having distinct roots (ie non-zero discriminant).
- Next, there must be a group law available for a final elimination: the algorithm produces several possibilities for the characteristic polynomial of Frobenius. If no group law is available, we attempt to replace the curve with an isomorphic one for which one is. Note that a group law exists if there is a single point at infinity (degree of $h(x) = g$).

Warning : Even if these conditions are satisfied, the last phase of the algorithm may still fail for $g \geq 4$ in unusual cases. However, this is very rare. Finally, it should be noted that the main algorithm works over the smallest extension of the ground field K over which $h(x)$ splits. It is therefore much faster if $h(x)$ has all its roots in K.

When the genus is > 4 or the canonical lift method doesn't apply, Vercauteren's version of Kedlaya is now used in its place. It is also used for genus 4 when an extension of the base field is required for Mestre: ie, $h(x)$ doesn't split.

`UseGenus2` BoolElt *Default :* `false`

For genus 2 the methods mentioned at the start are applied if the parameter `UseGenus2` is set to `true`. Otherwise they are now only used for large characteristic.

If none of the above apply, the remaining methods are the subexponential algorithm, which uses the generic machinery of the function fields, and general group order algorithms of Shanks and Pollard. The subexponential algorithm will be the first choice if no group law is available or the genus is large and the base field small. It can be disabled by setting `UseSubexpAlg` to `false`.

The parameter `ShanksLimit` determines when we switch from a memory-consuming Shanks' algorithm to a slower Pollard's method. The value of the parameter `CartierManinLimit` is the maximal characteristic for which we may use the Cartier-Manin trick (see function `EulerFactorModChar` below).

The next 2 parameters are used only for the genus 2 methods in odd characteristic.

`UseSchoof` BoolElt *Default :* `true`

`UseHalving` BoolElt *Default :* `true`

When the base field is large enough, it is better to firstly compute the cardinality of the Jacobian modulo some odd primes and some power of two. These two parameters allow the user to disable one or both of these methods.

Example H125E19

In the following sequence of examples the different point counting methods are illustrated. The first example uses direct counting.

```
> SetVerbose("JacHypCnt",true);
> P<x> := PolynomialRing(GF(31));
> f := x^8 + 3*x^7 + 2*x^6 + 23*x^5 + 5*x^4 + 21*x^3 + 29*x^2 + 12*x + 9;
> J := Jacobian(HyperellipticCurve(f));
> time #J;
Using naive counting algorithm.
20014
Time: 0.020
```

Example H125E20

For the second example, we apply Kedlaya's algorithm to a genus curve 2 over $GF(3^{20})$.

```
> K := FiniteField(3,20);
> P<x> := PolynomialRing(K);
> f :=  x^5 + x + K.1;
> J := Jacobian(HyperellipticCurve(f));
> #J;
Using Kedlaya's algorithm.
Applying Kedlaya's algorithm
Total time: 2.310
12158175772023384771
```

Example H125E21

For the third example, we apply Vercauteren's algorithm to a non-ordinary genus curve 2 over $GF(2^{25})$..

```
> K := FiniteField(2,25);
> P<x> := PolynomialRing(K);
> f :=  x^5 + x^3 + x^2 + K.1;
> J := Jacobian(HyperellipticCurve([f,K!1]));
> time #J;
Using Kedlaya/Vercauteren's algorithm.
1125899940397057
Time: 0.680
```

Example H125E22

For the third example, we apply Mestre's algorithm to a genus 3 ordinary curve over $GF(2^{25})$.

```
> K := FiniteField(2,25);
> P<x> := PolynomialRing(K);
> h := x*(x+1)*(x+K.1);
> f :=  x^7 + x^5 + x + K.1;
> C := HyperellipticCurve([f,h]);
> J := Jacobian(C);
> #J;
Using Mestre's method.
Total time: 0.440
3778001771268503789504 0
> SetVerbose("JacHypCnt",false);
```

As the full Euler factor of J has been computed and stored the number of points on C, the Zeta function for C and the Euler factor for C are now immediate.

```
> time #C;
33555396
Time: 0.000
```

Example H125E23

We compare the Shanks and Pollard methods for large prime fields.

```
> // Comparison between Shanks and Pollard:
> P<x> := PolynomialRing(GF(1000003));
> f := x^7 + 123456*x^6 + 123*x^5 + 456*x^4 + 98*x^3 + 76*x^2 + 54*x + 32;
> J := Jacobian(HyperellipticCurve(f));
> curr_mem := GetMemoryUsage(); ResetMaximumMemoryUsage();
> time Order(J : ShanksLimit := 10^15);
1001800207033014252
Time: 19.140
> GetMaximumMemoryUsage()-curr_mem;
133583360
```

The computation took about 100 MB of central memory.

```
> delete J`Order;   // clear the result which has been stored
> curr_mem := GetMemoryUsage();
> ResetMaximumMemoryUsage();
> time Order(J : ShanksLimit := 0);
1001800207033014252
Time: 95.670
> GetMaximumMemoryUsage()-curr_mem;
0
```

Now it takes almost no memory, but it is slower (and runtime may vary a lot).

Example H125E24

In the case of genus 2 curves, the parameters `UseSchoof` and `UseHalving` do help.

```
> // Using Schoof and Halving true is generally best in genus 2
> P<x> := PolynomialRing(GF(100000007));
> f := x^5 + 456*x^4 + 98*x^3 + 76*x^2 + 54*x + 32;
> J := Jacobian(HyperellipticCurve(f));
> time Order(J);
10001648178050390
Time: 7.350
> delete J`Order;
> time Order(J: UseSchoof := false, UseHalving := false);
10001648178050390
Time: 21.080
```

But if the Jacobian is known in advance to be highly non–cyclic, it may be slightly better to switch them off. The Jacobian below is the direct product of two copies of the same supersingular elliptic curve.

```
> // ... but not always for highly non-cyclic Jacobians
> P<x> := PolynomialRing(GF(500083));
> f := x^5 + 250039*x^4 + 222262*x^3 + 416734*x^2 + 166695*x + 222259;
> J := Jacobian(HyperellipticCurve(f));
> time Order(J);
250084007056
Time: 1.920
> delete J`Order;
> time Order(J : UseSchoof:=false, UseHalving := false);
250084007056
Time: 1.870
```

`FactoredOrder(J)`

Given the Jacobian J of a hyperelliptic curve defined over a finite field, the function returns the factorization of the order of the group of rational points.

`EulerFactor(J)`

Given the Jacobian J of a hyperelliptic curve defined over a finite field, the function returns the Euler factor J, i.e. the reciprocal of the characteristic polynomial of Frobenius acting on $H^1(J)$. (see also `ZetaFunction(C)` which is essentially the same function and has the same behaviour).

`EulerFactorModChar(J)`

Given the Jacobian J of a hyperelliptic curve defined over a finite field, the function returns the Euler factor J modulo the characteristic of the base field. This function should not be used in high characteristic (say p should be $\leq 10^6$).

`EulerFactor(J, K)`

Given a Jacobian J of a hyperelliptic curve defined over the rationals and a finite field K at which J has good reduction, the function returns the Euler factor of the base extension of J to K.

125.11.3 Deformation Point Counting

`JacobianOrdersByDeformation(Q, Y)`

`EulerFactorsByDeformation(Q, Y)`

`ZetaFunctionsByDeformation(Q, Y)`

These functions compute the orders of Jacobians (*resp.* Euler factors, *resp.* Zeta functions) of one or more hyperelliptic curves in a 1-parameter family over a finite field using deformation methods.

$Q(x, z)$ should be a 2-variable polynomial in variables x and z over a finite field of odd characteristic k. Currently Q must be monic of odd degree as a polynomial in x. z is the parameter and the familiy of curves is given by $y^2 = Q(x, z)$.

Y should be a sequence of elements in a finite field extension K of k. The function then computes and returns the sequence of orders, Euler factors or Zeta functions associated to the hyperelliptic curves over K obtained by specialising the z parameter of $Q(x, z)$ to the elements of Y.

Over a field of size p^n, for n not too large, it is efficient to compute the results for several curves in the family at once, as roughly half of the computation for an individual curve is taken up in computing a Frobenius matrix that is then specialised to the parameter value. For a sequence of input parameter values, this part of the computation is only performed once at the start. However, as n becomes larger, the time taken for the specialisation at a particular value starts to dominate.

The field k over which the family is defined should be of small degree over the prime field. This is because Kedlaya's algorithm is applied over k to a particular member of the family with the parameter specialised to a k-value to initialise the deformation.

Similarly, the algorithm is much faster when $Q(x, z)$ is of small degree in the parameter z. In practice, the degree of $Q(x, z)$ as a polynomial in z, probably shouldn't exceed 3 or 4 (linear is obviously best).

There are two further conditions on Q and Y which we will try to remove in the near future. The first is that no Y-value is zero. The second is that the generic discriminant of $\hat{Q}$ (resultant of $\hat{Q}$ and $\partial\hat{Q}/\partial x$ w.r.t. x) must have leading coefficient (as a polynomial in z) a unit in $W(k)$. Here, $W(k)$ is the ring of integers of the unramified extension of $\mathbf{Q}_p$ with residue class field k and $\hat{Q}$ is a lift of Q to a 2-variable polynomial over $W(k)$. By translating and scaling the parameter, the user may be able to effect these two conditions if they aren't initially satisfied.

As with the other point-counting functions, the verbose flag `JacHypCnt` can be used to output information during processing.

Example H125E25

The following is a small example where the Euler factors for 4 random members of a linear family of elliptic curves are computed with a single call. The base field is $\mathbf{F}_9$ and the parameter values are taken in $\mathbf{F}_{3^{40}}$ which is the field over which the counting occurs.

```
> Fp := FiniteField(3^2);
> R<X,Y> := PolynomialRing(Fp,2);
> Q  := X^3+X^2-2*X+Fp.1+Y*(X^2-X+1);
> Fq := ext<Fp | 20>;
> values := [Random(Fq) : i in [1..4]];
> EFs := EulerFactorsByDeformation(Q,values);
> EFs;
[
    12157665459056928801*$.1^2 + 852459373*$.1 + 1,
    12157665459056928801*$.1^2 + 1088717005*$.1 + 1,
    12157665459056928801*$.1^2 + 6911064226*$.1 + 1,
    12157665459056928801*$.1^2 - 607949990*$.1 + 1
]
```

125.11.4 Abelian Group Structure

`Sylow(J, p)`

Given the Jacobian J of a hyperelliptic curve defined over a finite field and a prime p, this function returns the Sylow p-subgroup of the group of rational points of J, as an abstract abelian group A. The injection from A to J is also returned as well as the generators of the p-Sylow subgroup.

`AbelianGroup(J)`

`UseGenerators`	BOOL	*Default :* `false`
`Generators`	SETENUM	*Default :*

Given the Jacobian J of a hyperelliptic curve defined over a finite field K, this function returns the group of rational points of J as an abstract abelian group A. The isomorphism from A to $J(K)$ is returned as a second value.

If `UseGenerators` is set then the group structure computation is achieved by extracting relations from the user-supplied set of generators in `Generators`.

`HasAdditionAlgorithm(J)`

Returns true if and only if the Jacobian J has an addition algorithm. This is of interest when trying to construct the (abelian) group structure of J: When no user-supplied generators are given, such an algorithm must be present.

125.12 Jacobians over Number Fields or Q

Some functions in this section work for general number fields (notably `TwoSelmerGroup`), while many are only implemented over **Q**.

125.12.1 Searching For Points

`Points(J)`

`RationalPoints(J)`

Bound RngInt *Default :* 0

Given a Jacobian J of a genus 2 hyperelliptic curve defined by an integral model over the rationals, determine all rational points on the Jacobian J whose naive height on the associated Kummer surface is less than or equal to Bound.

125.12.2 Torsion

`TwoTorsionSubgroup(J)`

Given the Jacobian J of a hyperelliptic curve C which is either of genus 2 or has odd degree defined over a number field K, the function returns $J(K)[2]$ as an abstract group, together with a map sending elements of the abstract group to points on J. The curve C must be given in the simplified form $y^2 = f(x)$.

`TorsionBound(J, n)`

Given the Jacobian J of a hyperelliptic curve defined over the rationals, this function returns a bound on the size of the rational torsion subgroup of the Jacobian. The bound is obtained by examining the group $J(\mathbf{F}_p)$ for the first n good primes p.

`TorsionSubgroup(J)`

Given the Jacobian J of a genus 2 curve defined over the rationals, this function returns the rational torsion subgroup of J, and the map from the group into J. The curve must have the form $y^2 = f(x)$ with integral coefficients.

Example H125E26

For the curve

$$C : y^2 = (x+3)(x+2)(x+1)x(x-1)(x-2),$$

the only Q-rational torsion on the Jacobian is 2-torsion.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(&*[x-n : n in [-3..2]]);
> J := Jacobian(C);
> T, m := TwoTorsionSubgroup(J);
> T;
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/2
Defined on 4 generators
Relations:
```

```
    2*P[1] = 0
    2*P[2] = 0
    2*P[3] = 0
    2*P[4] = 0
> [ m(T.i) : i in [1..4] ];
[ (x^2 - 3*x + 2, 0, 2), (x^2 - 2*x, 0, 2), (x^2 - x - 2, 0, 2),
(x^2 - 4, 0, 2) ]
> #T eq #TorsionSubgroup(J);
true
```

The Jacobian for the following curve has torsion subgroup $Z/24$ over $\mathbf{Q}$.

```
> C := HyperellipticCurve((2*x^2-2*x-1)*(2*x^4-10*x^3+7*x^2+4*x-4));
> J := Jacobian(C);
> T, m := TwoTorsionSubgroup(J);
> T;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*P[1] = 0
> m(T.1);
(x^2 - x - 1/2, 0, 2)
> A,h := TorsionSubgroup(J);
> #T eq #A;
false
> A;
Abelian Group isomorphic to Z/24
Defined on 1 generator
Relations:
    24*P[1] = 0
> P := h(A.1);
> P;
(x^2 - 1/2, -1, 2)
> Order(P);
24
> 12 * P eq m(T.1);
true
```

125.12.3 Heights and Regulator

This section pertains to height functions on the Mordell–Weil group of the Jacobian of hyperelliptic curves over a number field k. However, naive heights and height constants are currently only implemented for Jacobians of genus 2 curves defined over $\mathbf{Q}$.

In the case of genus two curves defined over $\mathbf{Q}$, the canonical height is computed using the algorithm of Flynn and Smart[FS97] with improvements by Stoll[Sto99]. This algorithm computes the canonical height using local error functions on the associated Kummer surface.

In all other cases the algorithm described in chapter 5 of [Mül10a] is used. It is based on a theorem due to Faltings and Hriljac which expresses the canonical height pairing in terms of Arakelov intersection theory and works as follows:

We find divisors D_P and D_Q of degree zero on the curve representing P and Q, respectively. For this we use the canonical representative of P and Q (see Section 125.10); if $P = Q$, then the canonical representatives for P and $-P$ are used. Since the canonical representatives have common support at infinity, we subtract the divisor of a function $x-\lambda$ from one of them. These ideas are due to David Holmes [Hol06]. If there are points at infinity in the positive support of P or Q, then we might have to also subtract the divisor of a function $x - \mu$ from the other representatives.

The actual Arakelov intersection computations are performed locally using regular models of the curve (see Section 114.12.2) at the relevant primes and Groebner bases over p-adic quotient rings in the non-archimedean case. The algorithm requires factorisation of polynomials over non-archimedean local fields. For archimedean places the intersections multiplicities can be expressed using theta functions with respect to the analytic Jacobian. This relies heavily on several functions described in Section 125.18.

If the genus is less than 4, the most expensive archimedean operations are usually applications of the Abel-Jacobi map ToAnalyticJacobian. In larger genus, the computation of theta functions, whose running time grows exponentially in the genus is more expensive.

Regarding the non-archimedean computations, the main bottleneck is integer factorisation which is required to find out which primes may yield non-trivial intersection multiplicities.

Applications: Heights are a useful tool for studying rational points on varieties. The most standard applications concerning points on Jacobians are

- proving independence of points in $J(k)/J_{tors}(k)$ (in particular, the regulator is defined in terms of the canonical height), and
- proving non-divisibility: given $P \in J(k)$ and $n \in \mathbf{Z}$, proving that P is not of the form nR for some $R \in J(k)$.

The heights in this section are logarithmic, and they measure the size of the coordinates of the image of P on the Kummer variety associated to J (embedded in $\mathbf{P}^{2^g-1}$, where g is the genus of the curve). The naive height h is simply the height of the image of P in $\mathbf{P}^{2^g-1}$ using an explicit embedding, which is currently only available for $g = 2$. One can refine this, taking advantage of the group law on J, defining a *canonical height* which has nice properties with respect to the group law, for instance $\hat{h}(nP) = n^2\hat{h}(P)$. In particular, $\hat{h}(P) = 0$ if and only if P is a torsion point. The function $h - \hat{h}$ is bounded on $J(k)$.

Computationally, one generally wants an upper bound on this, because then one can find all points up to a given canonical height by doing a search for points of bounded naive height.

`NaiveHeight(P)`

Given a point P on the Jacobian of a curve of genus 2 (or on the associated Kummer surface), the function returns the logarithmic height of the image of P in $\mathbf{P}^3$ under the maps $J \to K \to \mathbf{P}^3$.

`Height(P: parameters)`

`CanonicalHeight(P: parameters)`

`lambda`	RngIntElt	*Default :* 1
`mu`	RngIntElt	*Default :* 0
`LocalPrecision`	RngIntElt	*Default :* 0
`UseArakelov`	BoolElt	*Default :* `false`
`Precision`	RngIntElt	*Default :* 0

The canonical height of a point P on the Jacobian of a hyperelliptic curve over a number field or over the rationals. If the genus is 2 and the ground field is $\mathbf{Q}$, then this computes the canonical height on the associated Kummer surface. Otherwise this function simply computes the height pairing of P with itself using Arakelov intersection theory.

`HeightConstant(J: parameters)`

`Effort`	RngIntElt	*Default :* 0
`Factor`	BoolElt	*Default :* `false`

Given the Jacobian J of a genus 2 curve over $\mathbf{Q}$ of the form $y^2 = f(x)$ with integral coefficients, this computes a real number c such that $h(P) \leq \hat{h}(P) + c$ for all P in $J(\mathbf{Q})$, where h is the naive height and $\hat{h}$ is the canonical height.

The parameter `Effort` (which can be $0, 1$ or 2) indicates how much effort should be put into finding a good bound. The second value returned is a bound for μ_∞, the contribution from the infinite place. If the parameter `Factor` is `true`, then the discriminant will be factored, and its prime divisors will be considered individually, usually resulting in an improvement of the bound.

`HeightPairing(P, Q: parameters)`

`lambda`	RngIntElt	*Default :* 1
`mu`	RngIntElt	*Default :* 0
`LocalPrecision`	RngIntElt	*Default :* 0
`UseArakelov`	BoolElt	*Default :* `false`
`Precision`	RngIntElt	*Default :* 0

The value of the canonical height pairing for rational points P and Q on the Jacobian of a hyperelliptic curve defined over a number field. The pairing can be defined as $\langle P, Q\rangle := (\hat{h}(P+Q) - \hat{h}(P) - \hat{h}(Q))/2$ and if the genus is 2 and the ground field is $\mathbf{Q}$, the pairing is computed using this definition. Otherwise the pairing is computed using Arakelov intersection theory.

Sometime these fail due to insufficient precision; if this happens, the parameter `LocalPrecision` should be changed accordingly.

Changing the parameters `lambda` and `mu` (see the introduction above) can sometimes speed up the computations because some of the required integer factorisations might be significantly easier for some values of λ and μ than for others. Note that the parameter `mu` is only used if it is nonzero.

The parameter `UseArakelov` indicates whether the algorithm based on Arakelov intersection theory should be used in genus 2.

`HeightPairingMatrix(S: Precision)`

`Precision` RngIntElt *Default :* 0

Given a sequence $[P_1, ...P_n]$ of points on the Jacobian J of a hyperelliptic curve defined over a number field, this function returns the matrix with entries $\langle P_i, P_j\rangle$, where the latter denotes the canonical height pairing between P_i and P_j.

`Regulator(S: Precision)`

`Precision` RngIntElt *Default :* 0

Given a sequence S of points on the Jacobian J of a hyperelliptic curve defined over a number field k, the function returns the determinant of the height pairing matrix of S. The regulator is equal to zero when the points are dependent in the Mordell–Weil group, and otherwise is equal to the square of the volume of the parallelotope spanned by the points in the subgroup of the free quotient of $J(k)$ generated by S.

`ReducedBasis(S: Precision)`

`Precision` RngIntElt *Default :* 0

Given a sequence of points on the Jacobian J of a hyperelliptic curve defined over a number field k, this function returns an LLL- reduced basis for the subgroup of $J(k)/J_{tors}(k)$ generated by the given points (that is,the function reduces the real lattice formed by the points, under the positive definite quadratic form given by the canonical height pairing). The height pairing matrix of the sequence is returned as a second value.

Example H125E27

This example illustrates some basic properties of heights, and proves that a certain point in $J(\mathbf{Q})$ is not a nontrivial multiple of any other point in $J(\mathbf{Q})$. Let J be the Jacobian of $y^2 = x^6 + x^2 + 2$.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x^2+2);
```

```
> J := Jacobian(C);
```

Find some small points on C and map them to J:

```
> ptsC := Points(C : Bound:=1000);
> ptsJ := [ ptsC[i] - ptsC[1] : i in [2,3,4,5,6] ];
> ptsJ;
[ (1, x^3, 2), (x + 1, x^3 - 1, 2), (x + 1, x^3 + 3, 2), (x - 1, x^3 - 3, 2),
(x - 1, x^3 + 1, 2) ]
```

The canonical heights of these five points:

```
> [ Height(P) : P in ptsJ ];
[ 0.47983979745040515202327954250 2, 0.00000000000000000000000000000000,
0.47983979745040515202327954249 1, 0.47983979745040515202327954249 1,
0.00000000000000000000000000000000 ]
```

We see that two of them are torsion ($\hat{h} = 0$), and the others are probably equal or inverse to each other modulo torsion, because they appear to have the same canonical height. If so, they would generate a subgroup of rank 1 in $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$. The next command verifies this.

```
> ReducedBasis(ptsJ);
[ (x + 1, x^3 + 3, 2) ]
[0.47983979745040515202327954249 1]
> P := ptsJ[3];
> P;
(x + 1, x^3 + 3, 2)
```

So this point (which is the third point in our list) generates the others. We proceed to check that the other two non-torsion points in our list are equal to $P + T$ or $-P + T$ for some torsion point T.

```
> Jtors, maptoJ := TorsionSubgroup(J);
> {ptsJ[1], ptsJ[4]} subset { pt + maptoJ(T) : pt in {P,-P}, T in Jtors };
true
```

Now we check the property $\hat{h}(nP) = n^2\hat{h}(P)$ for $n = 23$.

```
> Height(23*P)/Height(P);
529.00000000000000000000000000
```

Of course, the naive height does not behave so nicely, but at least $h - \hat{h}$ should be bounded by the height constant.

```
> HC := HeightConstant(J : Effort:=2, Factor);
> HC;
3.7365462328830572011347394037 6
```

In particular, all torsion points should have naive height less than this.

```
> for T in Jtors do
>     NaiveHeight(maptoJ(T));
> end for;
0.00000000000000000000000000000000
```

```
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
> // Does the inequality hold for 23*P?
> NaiveHeight(23*P) - Height(23*P) le HeightConstant(J);
true
```

Finally, we show that P is not a nontrivial multiple of another point in $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$ (implying that P generates $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$ if this has rank 1). For suppose that $P = nQ + T$ for some $Q \in J(\mathbf{Q}), T \in J_{tors}(\mathbf{Q})$. Then $\hat{h}(Q) = (1/n^2)\hat{h}(P) < \hat{h}(P)$, and the following search shows there is no Q satisfying this bound.

```
> LogarithmicBound := Height(P) + HeightConstant(J);  // Bound on the naive h(Q)
> AbsoluteBound := Ceiling(Exp(LogarithmicBound));
> PtsUpToAbsBound := Points(J : Bound:=AbsoluteBound );
> ReducedBasis( [ pt : pt in PtsUpToAbsBound ]);
[ (x^2 + 1/2*x + 1/2, 1/4*x - 5/4, 2) ]
[0.47983979745040515202327954249 1]
```

If Q exists, it would have to be in the set `PtsUpToAbsBound`. But the results of the final command indicate that the group generated by `PtsUpToAbsBound` is also generated by a single point of canonical height 0.47983979745040515202327954249 1, so there are no nontorsion points Q in `PtsUpToAbsBound` with $\hat{h}(Q) < \hat{h}(P)$.
This example is continued in Example H125E35, where we prove that $J(\mathbf{Q})$ has rank 1, and then use the fact that P generates $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$ to find all rational points on C.

Example H125E28

We compute a reduced basis for a set of points on the Jacobian J on the genus two curve $C : y^2 = x^6 + x^2 + 1$.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x^2+1);
> J := Jacobian(C);
```

We construct some points on C:

```
> Z := PointsAtInfinity(C);
> Z;
{@ (1 : -1 : 0), (1 : 1 : 0) @}
> P1 := Z[1];
> P2 := Z[2];
> P3 := C![1/2,9/8,1];
> P4 := C![-1/2,9/8,1];
```

We now map them to J (for example $Q1$ is the divisor $P1 - P2$):

```
> Q1 := J![P1, P2];
```

```
> Q2 := J![P1, P3];
> Q3 := J![P3, P4];
> B, M := ReducedBasis([Q1, Q2, Q3] : Precision := 12);
> B;  // This will be a basis for <Q1, Q2, Q3>
[ (1, -x^3, 2), (x^2, 1, 2) ]
> M;  // The height pairing matrix for the new basis B
[0.27973179336882782876670568 39      -5.005347463630776922E-17]
[     -5.005347463630776922E-17 0.952446212809730769780007795 9]
> Determinant(M);
0.26642948719661422476551649 84
```

Since Det(M) is nonzero, the two generators in B are independent.

125.12.4 The 2-Selmer Group

The principal functions in this section provide information about the Mordell-Weil group $J(K)$ of a hyperelliptic Jacobian defined over the rationals or a number field. Two descent provides an upper bound (since $J(K)/2J(K)$ embeds in the 2-Selmer group of J). Finer information is provided by `HasSquareSha`, which determines the parity of the 2-rank of the Shafarevich–Tate group, and `RankBounds` collects together all the information that is computable in MAGMA.

Starting with MAGMA 2.13, a simpler user interface for computing 2-Selmer groups was introduced. The two implementations that were previously available still exist internally. In MAGMA 2.18-4 a new implementation has been included. When `TwoSelmerGroup` is called, it chooses one of them (or the user may choose).

`BadPrimes(C)`

`BadPrimes(J)`

`Badness` RNGINTELT *Default :* 1

Given a hyperelliptic curve C with integral coefficients defined over a number field (or the Jacobian of such a curve), the function returns a sequence containing the primes where the given model has bad reduction.

`HasSquareSha(J)`

`IsEven(J)`

Given the Jacobian of a hyperelliptic curve over $\mathbf{Q}$ or a number field, and assuming that the Shafarevich-Tate group of J is finite, this returns `true` if and only if the order of the Shafarevich-Tate group is a square. (Otherwise, the order is twice a square, as shown by Poonen and Stoll in [PS99]). The order is square if and only if the number of "deficient" primes for C is even (see below).

`IsDeficient(C, p)`

For a genus 2 curve defined over $\mathbf{Q}$ or a number field, this returns `true` if C is "deficient" at p. By definition, the curve is deficient if there are no points on C defined over any extension of odd degree over $\mathbf{Q}_p$ (when p is a prime) or over $\mathbf{R}$ (when p is 0). Equivalently, C is deficient if there is no $\mathbf{Q}_p$-rational divisor of odd degree on C.

`HasIndexOne(C,p)`

`HasIndexOne(C,p)`

Given a hyperelliptic curve C over $\mathbf{Q}$ or a number field, this returns true if and only if there is a divisor of odd degree on C which is defined over the completion at the prime p. An even genus curve is deficient at p if and only if it does not have index one at p.

`HasIndexOneEverywhereLocally(C)`

Returns true if and only if C has index one over all completions of its base field (including real completions).

`TwoSelmerGroup(J)`

`Al`	MonStgElt	*Default :*
`Fields`	SetEnum	*Default :*
`ReturnFakeSelmerData`	BoolElt	*Default :* `false`
`ReturnRawData`	BoolElt	*Default :* `false`
Verbose	`Selmer`	*Maximum :* 3

The 2-Selmer group of the Jacobian of a hyperelliptic curve defined over $\mathbf{Q}$ or a number field. If the curve is not defined over $\mathbf{Q}$ then it must have a single rational point at infinity (for curves in the simplified form $y^2 = f(x)$, this is equivalent to $f(x)$ having odd degree), The algorithm may work better when an integral model of the curve is given (or better yet, a minimal model).

The Selmer group is returned as a finite abelian group S, together with a map from S to some affine algebra A, which represents the standard map from the Selmer group to $A^*/(A^*)^2$ or to $A^*/(A^*)^2\mathbf{Q}^*$ (depending whether the degree of f is odd or even).

Three separate implementations exist internally in Magma. (Prior to V2.13 two of these were available as the intrinsics `TwoSelmerGroup` and `TwoSelmerGroupData`). The user can specify which implementation is used by setting the optional parameter `Al` to `"TwoSelmerGroupOld"`, `"TwoSelmerGroupNew"` or to `"TwoSelmerGroupData"`; otherwise, an appropriate choice is made automatically.

Much of the computation time is usually spent on class group and unit calculations. These computations can be speeded up by using non-rigorous bounds, and there are two ways to control which bounds are used. The recommended way is to preset them using one of the intrinsics `SetClassGroupBounds` or

SetClassGroupBoundMaps (see 37.6.1). The other way is to precompute the class groups of the fields involved (with the desired optional parameters in ClassGroup), and then pass these fields to TwoSelmerGroup via the optional parameter Fields. The class group data is stored, and if TwoSelmerGroup requires a field that is isomorphic to one of the given Fields, the stored data will automatically be used. The relevant fields are those given by the roots of f, where $y^2 = f(x)$ is the SimplifiedModel of the curve.

When called with ReturnFakeSelmerData, the program returns an additional item, which specifies the "fake Selmer group" in the terminology of [Sto01]. (This is only relevant to the even degree case.) The returned object is a tuple $\langle B_1, B_2, B_3 \rangle$, where B_1 is a sequence of elements in A that span a subgroup S_1 of $A^*/(A^*)^2$, and B_2 and B_3 are sequences of integers that span subgroups S_2 and S_3 of $\mathbf{Q}^*/(\mathbf{Q}^*)^2$. The fake Selmer group S is then determined by the exact sequence $0 \to S_3 \to S_2 \to S_1 \to S \to 0$. See [Sto01] for a full explanation of this.

When called with ReturnRawData, the program additionally returns a third item expvecs and a fourth item factorbase. These specify the images in $A^*/(A^*)^2$ of the Selmer group generators (in unexpanded form). The factorbase is a sequence of elements f_j of A, and expvecs is a sequence of vectors in $\mathbf{Z}^F$, where F is the number of elements in the factorbase. The image in $A^*/(A^*)^2$ of the ith Selmer group generator is then the sum over j of $f_j^{e_j}$ where (e_j) is the ith exponent vector.

RankBound(J)

RankBounds(J)

An upper, or a lower and an upper bound, on the rank of the Mordell-Weil group of J, which should be the Jacobian of a hyperelliptic curve over the rationals or a number field. RankBound can be computed provided both TwoSelmerGroup and TwoTorsionSubgroup are implemented for J, while RankBounds is only implemented for Jacobians defined over $\mathbf{Q}$.

For curves of even degree and odd genus it is not always possible to compute an upper bound for the rank. One can compute instead an upper bound for the rank of the subgroup of the Mordell-Weil group consisting of points which can be represented by rational divisors. When the curve has index one everywhere locally, every rational divisor class can be represented by a rational divisor. RankBound and RankBounds check if this condition is satisfied and print a warning if it is not.

An initial upper bound is furnished by computing the 2-Selmer group. For even degree hyperelliptic curves this can potentially be sharpened by determining if the torsor T parameterizing divisor classes of degree 1 on the curve represents a nontrivial element of Sha[2]. This can be achieved in two ways. If Sha does not have square order (see HasSquareSha) then T is nontrivial and the bound can be decreased by 1. If there are rational divisor classes of degree 1 everywhere locally, then T lies in Sha[2], and RankBounds will attempt to determine whether T is divisible by 2 in Sha using the algorithm described in [Cre12]. When T is not divisible by 2, the bound may be lowered by 2. Examples of both phenomena are given below.

The lower bound is computed by searching for points on the Jacobian, then determining their independence using either the height pairing machinery or by considering their images in $J(\mathbf{Q})/2J(\mathbf{Q})$.

Example H125E29

In this example, we find out as much as we can about the Mordell-Weil group of the Jacobian of

$$y^2 = x(x + 1344^2)(x + 10815^2)(x + 5406^2)(x + 2700^2).$$

First define the curve and its Jacobian:

```
> _<x> := PolynomialRing(Rationals());
> pol := x*(x+1344^2)*(x+10815^2)*(x+5406^2)*(x+2700^2);
> C := HyperellipticCurve( pol );
> J := Jacobian(C);
> J;
Jacobian of Hyperelliptic Curve defined by
y^2 = x^5 + 155285397*x^4 + 4761213301312596*x^3 + 330186894143664707856000*x^2
+ 450122990999339719434240000000*x over Rational Field
```

We can search for points on C (with x-coordinate up to Bound):

```
> ptsC := Points(C : Bound := 10^6);
> ptsC;
{@ (1 : 0 : 0), (0 : 0 : 1), (43264 : -44828581639628800 : 1),
(43264 : 44828581639628800 : 1) @}
> pointAtInfinity := ptsC[1];
```

This `pointAtInfinity` $(1 : 0 : 0)$ might not seem to lie on C, but recall that in MAGMA, a hyperelliptic curve lives in a weighted projective plane. We can also search for points on the Jacobian, which takes longer:

```
> time ptsJ := Points(J : Bound := 2000);
Time: 0.670
> ptsJ;
{@ (1, 0, 0), (x, 0, 1) @}
```

The notation for points on J is explained in the section "Points on the Jacobian" in this chapter. The points appearing above are the trivial point on J, and the point corresponding to the divisor $(0 : 0 : 1) - (1 : 0 : 0)$ on C.

```
> ptsJ[1] eq J!0;  // Is the first point equal to 0 on J?
true
> Order( ptsJ[2] );
2
```

So far, we have only found some trivial points on J. In fact, we can see from the equation that J has full 2-torsion, which we now confirm.

```
> Jtors, map := TorsionSubgroup(J);
> Jtors;
```

```
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4
Defined on 4 generators
Relations:
2*P[1] = 0
2*P[2] = 0
2*P[3] = 0
4*P[4] = 0
```

`Jtors` is an abstract group, and `map` converts elements of `Jtors` to actual points on J. Here are the generator of the $\mathbf{Z}/4$ and its inverse.

```
> map(Jtors.4);
> map(3*Jtors.4);
(x^2 + 32963094*x - 212161021632000, 94792247622*x - 2005558137487296000, 2)
(x^2 + 32963094*x - 212161021632000, -94792247622*x + 2005558137487296000, 2)
```

Looking at the points on C we found above, the third of them looks nontrivial, so we find its order on J:

```
> P := ptsC[3];
> PJ := J![ P, pointAtInfinity ];
> PJ;
(x - 43264, -44828581639628800, 1)
> Order(PJ);
0
```

This means PJ has infinite order on J. Now we do two-descent on J:

```
> #TwoSelmerGroup(J);
64
```

We already knew that $J(\mathbf{Q})[2]$ has order 16, and that the rank of $J(\mathbf{Q})$ is at least 1, so we now know that the rank is either 1 or 2. We now ask whether the order of $Sha(J)$ is a square or twice a square (assuming it is finite):

```
> HasSquareSha(J);
true
```

It follows (assuming $Sha(J)$ is finite) that $Sha(J)$ has square order, and that $J(\mathbf{Q})$ has rank 2, even though the other generator of $J(\mathbf{Q})$ is probably not easy to find, especially if does not come from a point on C.
For more examples of Selmer group computations, see the next section.

Example H125E30

We produce some Jacobians for which the order of the Tate-Shafarevich group is twice a square (assuming it is finite). We then observe how this affects the rank bounds.
A good source of examples are curves $y^2 = 3(x^6 - x^2 + a)$ for $a = 1 \bmod 3$, since these curves are "deficient" at 3 (for the definition, see above); this can be proved by elementary arguments. We now list those (for a up to 50) which have nonsquare Sha.

```
> _<x> := PolynomialRing(Rationals());
> for a := 1 to 50 do
```

```
>    if a mod 3 eq 1 then
>       Ca :=  HyperellipticCurve( 3*(x^6-x^2+a) );
>       assert IsDeficient(Ca,3);
>          // (This causes a failure if our assertion is wrong.)
>       if not HasSquareSha(Jacobian(Ca)) then print Ca; end if;
>    end if;
> end for;
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 12 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 21 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 30 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 48 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 57 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 66 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 84 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 93 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 102 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 129 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 138 over Rational Field
```

Aside: we could have tried simply $y^2 = 3(x^6 + a)$ for $a = 1 \bmod 3$, but this won't produce any examples: it's a cute exercise to show that for these curves, the number of deficient primes is always even.
Now consider the Mordell-Weil rank of the second curve listed (where $a = 7$),

```
> C7 :=  HyperellipticCurve( 3*(x^6-x^2+7) );
> J := Jacobian(C7);
> #TwoTorsionSubgroup(J);
1
> #TwoSelmerGroup(J);
2
> RankBound(J);
0
```

There is no nontrivial 2-torsion in $J(\mathbf{Q})$, and the Selmer group has order 2. The program has checked that the order of the Tate-Shafarevich group is twice a square (hence the 2-rank of Sha is exactly 1), and therefore it has returned a `RankBound` indicating that $J(\mathbf{Q})$ has rank 0. This rank bound is unconditional. In fact, the computation proves that there is a subgroup $\mathbf{Z}/2$ in Sha modulo the subgroup of infinitely divisible elements; therefore one may also deduce unconditionally that the 2-power part of Sha is finite cyclic (since if there was an infinitely 2-divisible part, it would still contribute to the 2-Selmer group).

Example H125E31

We demonstrate a non-trivial Tate-Shafarevich group on the Jacobian of a genus 2 curve. We consider the following Jacobian.

```
> P<x>:=PolynomialRing(Rationals());
> C:=HyperellipticCurve(-x^6 + 2*x^5 + 3*x^4 + 2*x^3 - x - 3);
```

```
> JC:=Jacobian(C);
```

We determine an upper bound on its Mordell-Weil rank

```
> RankBound(JC);
3
```

But we can only find one independent rational point on this Jacobian.

```
> V:=RationalPoints(JC,x^2 + 83/149*x + 313/596,2);
> B:=ReducedBasis(V);
> #B;
1
```

We test if there is a quadratic twist of our curve that does have high Mordell-Weil rank.

```
> d:=-1;
> Cd:=QuadraticTwist(C,d);
> JCd:=Jacobian(Cd);
> Vd:=RationalPoints(JCd:Bound:=100);
> Bd:=ReducedBasis(Vd);
> #Bd;
4
```

This shows that the Mordell-Weil rank of $\mathrm{Jac}(C_d)(\mathbf{Q})$ is at least four. We note that $\mathrm{rankJac}(C)(\mathbf{Q}(\sqrt{d})) = \mathrm{rankJac}(C)(\mathbf{Q}) + \mathrm{rankJac}(C_d)(\mathbf{Q})$, so if we can find an upper bound on the Mordell-Weil rank of $\mathrm{Jac}(C)$ over $\mathbf{Q}(\sqrt{d})$ then we also find an upper bound on $\mathrm{rankJac}(C_d)(\mathbf{Q})$. We first try this with unproven class group data.

```
> SetClassGroupBounds(1000);
> K:=QuadraticField(d);
> CK:=BaseChange(C,K);
> JCK:=Jacobian(CK);
> NumberOfGenerators(TwoSelmerGroup(JCK));
5
```

We find that $\mathrm{rankJac}(C)(\mathbf{Q}(\sqrt{d})) \leq 5$, which means that $\mathrm{Jac}(C)$ must indeed have Mordell-Weil rank 1 over $\mathbf{Q}$. To make the class group computation unconditional, we must check up to the Minkowski bound for each of the number fields utilized in the computation

```
> Aa:=AbsoluteAlgebra(JCK`Algebra);
> L:=Aa[1];
> MinkowskiBound(L);
3763009
```

The command `FactorBasisVerify(IntegerRing(L),1000,MinkowskiBound(L))` would perform the required check, but will take considerable time to complete.

Example H125E32

In this example we compute the Mordell-Weil rank of a genus 3 hyperelliptic Jacobian J with nontrivial 2-torsion in Sha using a different method. Namely we perform a 2-descent on the torsor parameterizing divisor classes of degree one on the curve as described in [Cre12].

```
> f := Polynomial([Rationals()| 30, 10, 30, 20, 10, 10, 10, 30, 10 ]);
> f;
10*x^8 + 30*x^7 + 10*x^6 + 10*x^5 + 10*x^4 + 20*x^3 + 30*x^2 + 10*x + 30
> X := HyperellipticCurve(f);
> J := Jacobian(X);
> SetClassGroupBounds("GRH");
> S := TwoSelmerGroup(J);
> #S;
4
> TorsionBound(J,5);
1
```

Here the 2-Selmer rank is 2 and there is no nontrivial 2-torsion. So the 2-descent on J gives an upper bound of 2 for the Mordell-Weil rank. However, one can do better.

```
>J`TwoSelmerSet;
{}
```

Calling `TwoSelmerGroup` also computes the 2-Selmer set of the torsor T parameterizing rational divisor classes of degree 1 (because C is everywhere locally solvable, the parameter `Al := "TwoSelmerGroupNew"` was used). This extra information allows us to deduce a better bound for the rank.

```
>RankBounds(J);
0 0
```

In this example the 2-Selmer set is empty, which implies that T is not divisible by 2 in Sha. Consequently Sha[2] has rank at least 2 (because of well known properties of the Cassels-Tate pairing). The theory behind these computations is described in [Cre12].

Example H125E33

For hyperelliptic curves of even degree and odd genus it is not always possible to obtain an upper bound for the Mordell-Weil rank. When the curve does not have index one everywhere locally, it may not be the case that every rational divisor class can be represented by a rational divisor. Here we provide an example of an even degree odd genus curve which does not have index one everywhere locally, but for which we can still compute an upper bound for the Mordell-Weil rank. The use of `SetClassGroupBounds` speeds up the computation, but is not necessary.

```
> f := Polynomial([Rationals()|-9, 8, 8, 1, -8, -8, -7, -2, -7 ]);
> C := HyperellipticCurve(f);
> Genus(C);
3
> Degree(C);
8
> J := Jacobian(C);
```

```
> SetClassGroupBounds("GRH");
> RankBound(J);
WARNING: upper bound is only an upper bound for the rank of Pic^0(X).
2
```

The warning appears because C does not have index one everywhere locally. In fact it is easy to see C has no real points. So over the reals it has no rational divisors of odd degree. However C has points locally at all nonarchimedean primes. This means C fails to have index 1 at exactly one prime. The obstruction to a rational divisor class being represented by a rational divisor is an element of the Brauer group. It follows from reciprocity in the Brauer group that every **Q**-rational divisor class is represented by a **Q**-rational divisor. In particular $Pic^0(C) = J(\mathbf{Q})$. So we can disregard the warning.

```
> HasIndexOneEverywhereLocally(C);
false 0
> Roots(f,RealField(50));
[]
> Evaluate(f,0) lt 0;
true
> &and[ HasIndexOne(C,p) : p in BadPrimes(C) ];
true
```

125.13 Two-Selmer Set of a Curve

Let $C : y^2 = f(x)$ be a hyperelliptic curve of genus g over a number field k. We say $\pi_\delta : D_\delta \to C$ is a *two-cover* of C if, over an algebraic closure, π_δ is isomorphic as a cover to the pull-back of an embedding of C into its Jacobian along multiplication by 2. The two-Selmer set classifies the k-isomorphism classes of two-covers of C that have points everywhere locally.

The hyperelliptic involution acts on such covers by pull-back: $\iota^*(\pi_\delta) = \pi_\delta \circ \iota$ (apply ι after π_δ). The *fake* two-Selmer set is the set

$$\{\delta : D_\delta(k_v) \neq \emptyset \text{ for all places } v \text{ of } k\}/\iota^*$$

If this set is empty, then C has no k-rational points. This can happen even if C itself does have points everywhere locally. If the set is non-empty, it gives information about rational points on C. See [BS09] for the underlying theory as well as a description of an algorithm to compute the set.

`TwoCoverDescent(C)`

`Bound`	RngIntElt	*Default :* -1
`Fields`	SetEnum	*Default :*
`Raw`	BoolElt	*Default :* `false`
`PrimeBound`	RngIntElt	*Default :* 0

`PrimeCutoff` RngIntElt *Default :* 0

Computes the fake 2-Selmer set as an abstract set. The map returned as second value can be used to obtain a representation of these abstract elements as ellements in an algebra, which allows explicit construction of the corresponding cover.

The optional parameters `Bound`, `Fields` and `Raw` perform the same function as for `TwoSelmerGroup` and we refer to it for their description. The remaining optional parameters are specific to this routine and we describe them here.

`PrimeBound`: Two covers are of very high genus. Hence, according to the Weil bounds, they can have local obstructions at very large good primes. For instance, for genus 2 one should check all primes up to norm 1153 and for genus 3 one should check all primes up to norm 66553. This is very time consuming. One can use `PrimeBound` to restrict the good primes to be considered to only those whose norm do not exceed the given bound. In principle, this can result in a larger set being returned than the proper two-selmer set.

`PrimeCutoff`: If the curve has bad reduction at some large prime, it can be prohibitively expensive to check the local conditions at this prime. This bound allows a restriction on the norm of bad primes where local conditions are considered. Setting this bound can result in a larger set being returned than the proper two-selmer set.

Example H125E34

As an illustration of `TwoCoverDescent`, we give the Magma-code to perform the computations related to the examples in [BS09]. First we give some examples of genus 2 curves that have points everywhere locally, but have an empty 2-Selmer set and hence have no rational points.

```
> Q:=Rationals();
> Qx<x>:=PolynomialRing( Q );
> C:=HyperellipticCurve(2*x^6+x+2);
> Hk,AtoHk:=TwoCoverDescent(C);
> #Hk;
0
> C:=HyperellipticCurve(-x^6+2*x^5+3*x^4-x^3+x^2+x-3);
> Hk,AtoHk:=TwoCoverDescent(C);
> #Hk;
0
```

In the following we consider a curve of genus 2 that does have rational points. In fact, its Jacobian has Mordell-Weil rank 2. We compute its two-covers with points everywhere locally. There are two. They both cover an elliptic curve over a quadratic extension. We use `Chabauty` following [Bru02] to determine the rational points on C. First we check that we can represent the fake two-Selmer set of the curve with some nice elements.

```
> f:=2*x^6+x^4+3*x^2-2;
> C:=HyperellipticCurve(f);
> Hk,AtoHk:=TwoCoverDescent(C:PrimeBound:=30);
> A<theta>:=Domain(AtoHk);
> deltas:={-1-theta,1-theta};
```

```
> {AtoHk(d): d in deltas} eq Hk;
true
```

Next, we determine a factorisation of f into a quartic and a quadratic polynomial.

```
> L<alpha>:=NumberField(x^2+x+2);
> LX<X>:=PolynomialRing(L);
> g:=(X^2-1/2)*(X^2-alpha);
> h:=2*(X^2+alpha+1);
> g*h eq Evaluate(f,X);
true
```

For some $\gamma = \gamma(\delta)$, we have that a 2-cover D_δ covers the elliptic curve

$$E : y^2 = \gamma\, g(x).$$

This allows us to translate the question about rational points on D_δ into a question about L-rational points on E with the additional property that x is rational. We verify that the two values of δ we found above, correspond to $\gamma = (1 - \alpha)/2$.

```
> LTHETA<THETA>:=quo<LX|g>;
> j:=hom<A->LTHETA|THETA>;
> gamma:=1/2*(-alpha + 1);
> {Norm(j(delta)):delta in deltas} eq {gamma};
true
> E:=HyperellipticCurve( gamma * g );
> P1:=ProjectiveSpace(Rationals(),1);
> EtoP1:=map<E->P1|[E.1,E.3]>;
```

We present E explicitly as an elliptic curve to Magma

```
> P0:=E![1,(1-alpha)/2];
> Eprime,EtoEprime:=EllipticCurve(E,P0);
> Etilde:=EllipticCurve(X^3+(1-alpha)*X^2+(2-9*alpha)*X+(16-2*alpha));
> EprimeToEtilde:=Isomorphism(Eprime,Etilde);
> EtoEtilde:=EtoEprime*EprimeToEtilde;
> EtildeToP1:=Expand(Inverse(EtoEtilde)*EtoP1);
```

We determine a group of finite odd index in $E(L)$.

```
> success,MWgrp,MWmap:=PseudoMordellWeilGroup(Etilde);
> success;
true
> MWgrp;
Abelian Group isomorphic to Z/2 + Z
Defined on 2 generators
Relations:
    2*MWgrp.1 = 0
```

We determine the set of L-rational points on E that have a $\mathbf{Q}$-rational image under `EtildeToP1`.

```
> V,R:=Chabauty(MWmap,EtildeToP1:IndexBound:=2);
> V;
```

```
{
    0,
    MWgrp.1 - MWgrp.2,
    MWgrp.1,
    -MWgrp.2
}
> R;
4
```

Since we found earlier that in this case, there is only one value of γ to consider, we know that any rational point on C must correspond to one of these points on E. The correspondence is given by the fact that E and C both cover the x-line. We determine these points explicitly.

```
> CtoP1 := map< C -> P1 | [C.1,C.3] >;
> pi := Extend( EtildeToP1 );
> { pi( MWmap( v ) ) : v in V };
{ (1 : 1), (-1 : 1) }
> [ RationalPoints( p@@CtoP1 ): p in { P1(Q)| pi( MWmap( v ) ) : v in V } ];
[
    {@ (1 : -2 : 1), (1 : 2 : 1) @},
    {@ (-1 : -2 : 1), (-1 : 2 : 1) @}
]
```

125.14 Chabauty's Method

This is a method for finding the rational points on a curve C of genus at least 2, that applies when the Mordell-Weil group of $Jac(C)$ has rank less than the genus of C. It involves doing local calculations at some prime where C has good reduction.

For (hyperelliptic) curves of genus 2 over $\mathbf{Q}$, two separate routines are available in MAGMA. In the older one, the user specifies the prime, and the results usually do not determine the set of rational points precisely. In the new implementation, which is far more powerful, Chabauty calculations are combined with a "Mordell-Weil sieve" which puts together information obtained using many different primes, to determine precisely the set of rational points on the curve. However, this routine does not apply to curves with no rational points, in fact one rational point must be known, as this plays a role in the algorithm. Both routines require a generator of the Mordell-Weil group of the Jacobian (which must have rank 1) to be provided.

In addition a routine `Chabauty0` is provided to handle the trivial case where the Jacobian has rank 0.

How Chabauty's method works: Under the assumption that the rank of $J = Jac(C)$ is strictly less than the genus of C, the closure V of $J(\mathbf{Q})$ in $J(\mathbf{Q}_p)$ (in the p-adic topology) can be described as the locus where certain power series vanish. The dimension of V is at most the rank of $J(\mathbf{Q})$. The image of C in J (under a natural embedding) will intersect V in a finite set, and this set must contain $C(\mathbf{Q})$. By examining the power series that define this finite intersection, one obtains upper bounds for the number of points on $C(\mathbf{Q})$ in each congruence class modulo p (or any power of p).

`Chabauty0(J)`

Given the Jacobian of a hyperelliptic curve C over $\mathbf{Q}$, the function finds all rational points on C, under the assumption that $J(\mathbf{Q})$ has rank zero.

The assumption is not checked. When it does not hold, the function computes a subset of $C(\mathbf{Q})$ corresponding (in some sense) to the torsion in $J(\mathbf{Q})$.

`Chabauty(P : ptC)`

`ptC` PT *Default :*

For a curve C of genus 2 over $\mathbf{Q}$, this returns the full set of rational points. The algorithm involves Chabauty's method combined with a Mordell-Weil sieve. The argument P should be a rational point on the Jacobian of C, and P is assumed to generate the Mordell-Weil group (which must have rank 1 in order for Chabauty's method to be applicable).

The algorithm requires knowledge of one rational point on the curve. (In particular, it cannot be used to show that a curve has no rational points!) Such a point may be supplied as the optional argument `ptC`; otherwise, one is found by searching.

`Chabauty(P, p: Precision)`

`Precision` RNGINTELT *Default :* 5

Given a point on the Jacobian of a hyperelliptic curve C over $\mathbf{Q}$, the function uses Chabauty's method (at the prime p) to bound the number of rational points on the curve C, The curve must have good reduction at the prime p, and the point P should generate a subgroup of index coprime to p in $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$. The algorithm assumes that C is in the simplified form $y^2 = f(x)$, where f has integral coefficients.

The function returns an indexed set of tuples $\langle x, z, v, k\rangle$ such that there are at most k pairs of rational points on C whose image in P^1 under the x-coordinate map are congruent to $(x : z)$ modulo p^v, and such that the only rational points on C outside these congruences classes are Weierstrass points.

If the optional parameter `Precision` is supplied, then in the returned data each of the v's will be greater than or equal to `Precision`. (This might take some time because the algorithm is not optimised for this.)

Example H125E35

In this example, which continues Example H125E27, we use both implementations of Chabauty to prove that the curve $y^2 = x^6 + x^2 + 2$ has only the obvious six rational points.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x^2+2);
> ptsC := Points(C : Bound:=1000); ptsC;
{@ (1 : -1 : 0), (1 : 1 : 0), (-1 : -2 : 1), (-1 : 2 : 1),
   (1 : -2 : 1), (1 : 2 : 1) @}
> J := Jacobian(C);
> RankBound(J);
```

```
1
```

So the Jacobian has rank at most 1. Next we ask whether any of the points on C give us points of infinite order on J.

```
> PJ1 := J! [ ptsC[3], ptsC[1] ];
> Order(PJ1);
8
> PJ2 := J! [ ptsC[5], ptsC[1] ];
> Order(PJ2);
0
```

This means $PJ2$ has infinite order in $J(\mathbf{Q})$. In Example H125E27 we proved that $PJ2$ is not divisible in $J(\mathbf{Q})$. Since we now know $J(\mathbf{Q})$ has rank 1, we can conclude that $PJ2$ is a generator of $J(\mathbf{Q})/J(\mathbf{Q})_{tors}$. Therefore we can use it for Chabauty's method.

```
> all_pts := Chabauty(PJ2); all_pts;
{ (1 : -2 : 1), (-1 : -2 : 1), (1 : 2 : 1),
  (1 : -1 : 0), (1 : 1 : 0), (-1 : 2 : 1) }
```

So we find that we already had the full set of rational points. We could have reached the same conclusion using the other implementation of Chabauty, where we specify that the method be applied at the prime 3 (the most natural choice, since it is the smallest prime of good reduction).

```
> BadPrimes(C);
[ 2, 7 ]
> Chabauty(PJ2, 3);
{@ <1, 1, 4, 1>, <80, 1, 4, 1>, <1, 0, 4, 1> @}
```

This tells us that there are at most 3 pairs of rational points on C, apart from Weierstrass points. In fact, one can easily check that there are no rational Weierstrass points. Therefore, the 6 known points are the only rational points on C.

Example H125E36

In this example we show that there are precisely 6 rational points on the curve $y^2 = x^6 + 8$.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+8);
> ptsC := Points(C : Bound:= 1000); ptsC;
{@ (1 : -1 : 0), (1 : 1 : 0), (-1 : -3 : 1), (-1 : 3 : 1),
   (1 : -3 : 1), (1 : 3 : 1) @}
> J := Jacobian(C);
> PJ := J! [ ptsC[5], ptsC[1]];
> Order(PJ);
0
> RankBound(J);
1
```

This shows that $J(\mathbf{Q})$ has rank 1, and PJ has infinite order. Now we check that PJ generates $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$. (For more explanation, see the similar calculation in Example H125E27).

```
> heightconst := HeightConstant(J : Effort:=2, Factor);
```

```
> LogarithmicBound := Height(PJ) + heightconst;  // Bound on h(Q)
> AbsoluteBound := Ceiling(Exp(LogarithmicBound));
> PtsUpToAbsBound := Points(J : Bound:=AbsoluteBound );
> ReducedBasis([ pt : pt in PtsUpToAbsBound ]);
[ (x^2 - x + 1, 3, 2) ]
[0.32661733877148842826007676859 0]
> Height(PJ);
0.32661733877148842826007676859 0
```

This shows there are no points in $J(\mathbf{Q})$ with canonical height smaller than that of PJ, so PJ is a generator modulo torsion and we may use it for Chabauty's method.

```
> all_pts := Chabauty(PJ : ptC:=ptsC[1]); all_pts;
{@ (1 : -1 : 0), (-1 : -3 : 1), (1 : 3 : 1),
   (-1 : 3 : 1), (1 : -3 : 1), (1 : 1 : 0) @}
```

Example H125E37

In Example H125E29 (in the section on 2-Selmer groups) we found a curve of rank 2. Although we cannot apply Chabauty's method to get information about the full set of rational points on C, we can still apply the routines to find all points on C whose images on J lie in a given subgroup of rank 1.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve( x*(x+1344^2)*(x+10815^2)*(x+5406^2)*(x+2700^2) );
> J := Jacobian(C);
> ptsC := Points(C : Bound := 10^6); ptsC;
{@ (1 : 0 : 0), (0 : 0 : 1), (43264 : -44828581639628800 : 1),
   (43264 : 44828581639628800 : 1) @}
> PJ := J! [ ptsC[3], ptsC[1] ]; Order(PJ);
0
```

Now we find all the points $P \in C(\mathbf{Q})$ for which the image $P - P_0 \in J(\mathbf{Q})$ lies in the subgroup generated by PJ, where P_0 denotes `ptsC[1]`.

```
> pts := Chabauty(PJ : ptC:=ptsC[1] ); pts;
{ (-1806336 : 0 : 1), (0 : 0 : 1), (43264 : 44828581639628800 : 1),
  (1 : 0 : 0), (-29224836 : 0 : 1), (-7290000 : 0 : 1),
  (-116964225 : 0 : 1), (43264 : -44828581639628800 : 1) }
```

Example H125E38

We find all rational points on $y^2 = x^6 + 1$, whose Jacobian J has rank 0, ie the group $J(\mathbf{Q})$ is finite.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+1);
> J := Jacobian(C);
> RankBound(J);
0
```

```
// So we may use Chabauty0.
> Chabauty0(J);
{@ (1 : -1 : 0), (1 : 1 : 0), (0 : 1 : 1), (0 : -1 : 1) @}
```

This Selmer computation required class/unit computations in the fields given by roots of $x^6 + 1$. In this example, there's clearly an approach requiring less work: to use the self-evident map from C to the elliptic curve $y^2 = x^3 + 1$, which has rank 0.
To define the map, we need to refer to the space where C lives, which is a weighted projective space in which Y has weight 3.

```
> E := EllipticCurve([0,1]);
> Pr2<X,Y,Z> := Ambient(C);
> CtoE := map< C -> E | [X^2*Z,Y,Z^3] >;
```

The map is well-defined (otherwise MAGMA would have already complained).

```
> CtoE;
Mapping from: CrvHyp: C to CrvEll: E
with equations :
X^2*Z
Y
Z^3
```

Now find the rational points on E.

```
> Rank(E);
0
> #TorsionSubgroup(E);
6
> ptsE := Points(E : Bound:=100 ); ptsE;
{@ (0 : 1 : 0), (-1 : 0 : 1), (0 : 1 : 1), (0 : -1 : 1),
(2 : 3 : 1), (2 : -3 : 1) @}
```

So, these are all the rational points on E. We can see with the naked eye that only those with $X = 0$ pull back to rational points on C, giving us the same six points on C that we found before. Of course, we could instead ask MAGMA to pull them back:

```
> for P in ptsE do
>     preimageofP := P @@ CtoE;
>     Points(preimageofP);
> end for;
{@ (1 : -1 : 0), (1 : 1 : 0) @}
{@ @}
{@ (0 : 1 : 1) @}
{@ (0 : -1 : 1) @}
{@ @}
{@ @}
```

125.15 Cyclic Covers of $\mathbf{P}^1$

A q-cyclic cover of the projective line is a curve C which admits a degree q morphism to $\mathbf{P}^1$ such that the associated extension of function fields is (geometrically) cyclic. By Kummer theory, such a curve has an affine model of the form $y^q = f(x)$ (at least when the base field contains the q-th roots of unity). This class of curve includes hyperelliptic curves. Current functionality for cyclic covers includes testing local solubility, searching for points and performing descents on curves and their Jacobians when they are defined over a number field.

Most functionality is implemented in the cases where $f(x)$ is a separable polynomial of arbitrary degree and q is prime. One notable exception is that the method of descent on the curve is available for any $f(x)$ that is q-th power free.

125.15.1 Points

A point on a cyclic cover $C : y^q = f_d x^d + f_{d-1}x^{d-1} + ... + f_0$ is specified by a triple representing a point in the appropriate weighted projective plane.

When $d = \text{Degree}(f(x))$ is divisible by q, we consider points the weighted projective plane $\mathbf{P}^2(1 : d/q : 1)$. In this case a point is represented by a triple $[X, Y, Z]$ satisfying

$$Y^q = f_d X^d + f_{d-1}X^{d-1}Z + ... + f_0 Z^d.$$

When $d = \text{Degree}(f(x))$ is not divisible by q, we consider points in the weighted projective plane $\mathbf{P}^2(q : d : 1)$. In this case a point is represented by a triple $[X, Y, Z]$ satisfying

$$Y^q = f_d X^d + f_{d-1}X^{d-1}Z^q + ... + f_0 Z^{dq}.$$

`RationalPoints(f,q)`

`Bound`	RngIntElt	*Default :* 0
`DenominatorBound`	RngIntElt	*Default :* 0
`NPrimes`	RngIntElt	*Default :* 0

For a polynomial f defined over a number field (or $\mathbf{Q}$) this Searches for rational points on the curve $y^q = f(x)$ in a box bounded by `Bound` (which must be specified). Returns a set of triples (x, y, z) satisfying $y^q = F(x, z)$ where $F(x, z)$ is the homogenisation of $f(x)$ taking x to have weight LCM(q,Degree($f(x)$))/Degree($f(x)$).

The search is by a sieve method described in Appendix A of [Bru02]. The parameter `NPrimes` controls the number of primes which are used and `DenominatorBound` the size of the denominators used.

`HasPoint(f,q,v)`

`HasPoint(f,q,v)`

For a polynomial f defined over a number field k, this checks if the curve defined by $y^q = f(x)$ has any points over the completion k_v. The second return value is a triple of elements in the completion of k at v giving a point on the curve. The triple

(x, y, z) returned satisfies $y^q = F(x, z)$ where $F(x, z)$ is the homogenisation of $f(x)$ taking x to have weight LCM(q,Degree($f(x)$))/Degree($f(x)$).

The current implementation for testing local solubility of cyclic covers is polynomial time in the cardinality of the residue field, making it somewhat impractical for larger primes.

`HasPointsEverywhereLocally(f,q)`

For a polynomial f defined over a number field k, this returns true if and only if the curve $y^q = f(x)$ has points over all nonarchimedean completions of k.

125.15.2 Descent

Let C be a curve with Jacobian J and suppose $\psi : A \to J$ is an isogeny of abelian varieties. We say that a morphism of curves $\pi : D \to C$ is a ψ-covering of C if, over an algebraic closure, π is isomorphic as a cover to the pull-back of an embedding of C into J along ψ.

Now suppose that $C : y^q = f(x)$ is a q-cyclic cover of the projective line defined over a number field k. Any primitive q-th root of unity ζ gives rise to an automorphism: ι sending a point (x, y) to $(x, \zeta y)$. This induces an automorphism of the Jacobian. Hence the endomorphism ring of the Jacobian contains the cyclotomic ring $\mathbf{Z}[\zeta]$, and in particular the endomorphism $\phi = 1 - \zeta$. The ϕ-Selmer set of C classifies isomorphism classes of ϕ-coverings of C which are everywhere locally soluble. There is an action of ι on these coverings. The quotient by this action is known as the fake ϕ-Selmer set of C. If this set is empty, then C has no k-rational points. This can happen even if C itself has points everywhere locally. Even when this set is non-empty, it gives information about rational points on C. When $q = 2$, ϕ is multiplication by 2, in which case the ϕ-Selmer set is the same as the 2-Selmer set described in `TwoCoverDescent`.

`qCoverDescent(f,q)`

`PrimeBound`	RngIntElt	*Default :* 0
`PrimeCutOff`	RngIntElt	*Default :* 0
`KnownPoints`	Set	*Default :*
Verbose	`CycCov`	*Maximum :* 3

Computes the fake ϕ-Selmer set of the cyclic cover $C : y^q = f(x)$ as an abstract set. The first return value is the fake ϕ-Selmer set given as a subset of the cartesian product of number fields having defining polynomials the irreducible factors of f. The map returned is from this cartesian product into the abstract group $A(S, q)$ involved in the computation (the unramified outside S subgroup of the quotient of the etale algebra associated to the ramification points of C by the subgroup generated by scalars and q-th powers).

The input `f` must be a q-th power free polynomial defined over a number field k with class number 1.

The optional parameters `PrimeBound` and `PrimeCutoff` have the same meaning as those used the `TwoCoverDescent` for hyperelliptic curves. Only good primes up to `PrimeBound` and bad primes up to `PrimeCutoff` will be considered in the local

computations. If either of these parameters is not set to default, the set returned might be strictly larger than the fake ϕ-Selmer set.

One can obtain a lower bound for the size of the ϕ-Selmer set by considering the images of known global points on C. This often speeds up the computation considerably as the first few primes might be sufficient to show that the ϕ-Selmer set is equal to the set of images of the known rational points. A set of known points may be passed in using the parameter `KnownPoints`. These should be given as triples of the type returned by `RationalPoints`.

Example H125E39

The following computation shows that the genus 15 curve defined by the equation $y^7 = 2x^7 + 6$ has no rational points even though it is everywhere locally solvable.

```
> _<x> := PolynomialRing(Rationals());
> RationalPoints(3*x^7+6,7 :  Bound := 1000);
{@ @}
> HasPointsEverywhereLocally(3*x^7+6,7);
true
> time Sel := qCoverDescent(3*x^7+6,7 : PrimeBound := 1000);
Time: 0.450
> Sel;
{}
```

125.15.3 Descent on the Jacobian

Suppose J is the Jacobian of the cyclic cover C defined by $y^q = f(x)$ over a field k containing the q-th roots of unity. If k is a number field, then the ϕ-Selmer group of J is a finite group which contains $J(k)/\phi(J(k))$. Computing the ϕ-Selmer group can be used to give an upper bound for the rank of $J(k)$. In the case $q = 2$, this reduces to the 2-Selmer group computed with `TwoSelmerGroup`.

For q prime the ϕ-Selmer group may be computed using an algorithm of Poonen-Schaefer [PS97]. Their algorithm actually computes a fake Selmer group, which is a quotient of the true Selmer group by a subgroup of order dividing q.

When the base field does not contain the q-th roots of unity, one still obtains information by computing the ϕ-Selmer group over the cyclotomic extension. For details see [PS97], or consider the example below.

`PhiSelmerGroup(f,q)`

`ReturnRawData`	BoolElt	*Default :* `false`
Verbose	`Selmer`	*Maximum :* 3

For a polynomial f over a number field k, ths computes the (fake) ϕ-Selmer group of the Jacobian of the curve $y^q = f(x)$ over the field $k(\mu_q)$ obtained by adjoining the q-th roots of unity to k. The polynomial $f(x)$ must be separable and q must be prime.

There are two steps in the algorithm which may require a significant amount of time: the first is the computation of class groups of the extensions of $k(\mu_q)$ given by the irreducible factors of $f(x)$; the second is the computation of the local images of the descent map at the prime(s) above q.

The (fake) Selmer group is returned as a finite abelian group S, together with a map from S to some affine algebra A, which represents the standard map from the Selmer group to $A^*/k^*(A^*)^q$ or to $A^*/(A^*)^q$ correspondingly as the degree of $f(x)$ is or is not divisible by q.

When called with `ReturnRawData` the program will yield three additional return values, `expvecs`, `factorbase` and `selpic1`. The first two are used to specify the images in $A^*/k^*(A^*)^q$ of the (fake) Selmer group generators (in unexpanded form). This is done exactly as for `TwoSelmerGroup`. `selpic1` is an element of the supergroup containing the (fake) Selmer group, or it is the empty set. This specifies the coset of the (fake) Selmer group corresponding to the (fake) Selmer set of the Pic^1 torsor as described in [Cre12]. When the curve has points everywhere locally this determines whether or not the torsor is divisible by ϕ in Sha, which in turn yields more information on the rank.

`RankBound(f,q)`

`RankBounds(f,q)`

`ReturnGenerators`	BoolElt	*Default :* `false`
Verbose	`Selmer`	*Maximum :* 3

An upper (or a lower and upper) bound for the rank of the Jacobian of the cyclic cover defined by $y^q = f(x)$. The upper bound is obtained by performing a descent as described above and incorporates the information on the Pic^1 torsor thus obtained. The lower bound is obtained by a naive (and rather inefficient) search for divisors on the curve.

If the optional parameter `ReturnGenerators` is set to `true`, then a third value will be returned. This will be a list of univariate polynomials defining divisors on $\mathbf{P}^1$ that lift to divisors on the curve. The images of these in the Mordell-Weil group generate a subgroup of rank at least as large as the lower bound provided. Independence of these points is determined by considering their images in the (fake) Selmer group.

Example H125E40

In this example we use descents to determine both the rank of the Mordell-Weil group and the 3-primary part of Sha for the Jacobian J of a genus 4 curve C over the rationals.

```
> _<x> := PolynomialRing(Rationals());
```

```
> f := 3*(x^6 + x^4 + 4*x^3 + 2*x^2 + 4*x + 3);
> k := CyclotomicField(3);
> IsIrreducible(PolynomialRing(k)!f);
true
> SelJ,m,expvecs,fb,SelPic1 := PhiSelmerGroup(f,3 : ReturnRawData);
> Ilog(3,#SelJ);
2
```

In this example the map from the genuine Selmer group to the fake Selmer group has a kernel of order 3. This is because f has degree divisible by $q = 3$ and is irreducible. Hence the ϕ-Selmer group has 3-rank 3. The fact that f is irreducible also implies that there is no nontrivial ϕ-torsion on J defined over k. Thus the ϕ-Selmer group gives an upper bound of $2 * 3 = 6$ for the rank of J over k and an upper bound of 3 for the rank over $\mathbf{Q}$ (see [PS97]). We can use the information about the Pic^1 torsor to get a better bound

```
> HasPointsEverywhereLocally(f,3);
true
> SelPic1;
{}
```

This means that the Pic^1 torsor represents a ϕ-torsion class in the Shafarevich-Tate group of J that is not divisible by ϕ. In particular it is nontrivial, and because the Cassels-Tate pairing is alternating this imples the ϕ-torsion subgroup of Sha contains a nontrivial subgroup with even 3-rank. Hence, we deduce that $J(k)/\phi(J(k))$ has 3-rank at most $3 - 2 = 1$, and consequently that $J(\mathbf{Q})$ has rank ≤ 1. Calling `RankBounds` will deduce these bounds, and search for generators to obtain a lower bound.

```
> r_l, r_u, gens := RankBounds(f,3 : ReturnGenerators);
> [r_l,r_u];
[1,1]
> gens;
[*
    T^3 - T^2 + 4*T + 4
*]
```

This polynomial defines a divisor on $\mathbf{P}^1$ that lifts to a degree 3 divisor D on $C : y^3 = f(x)$. One way to verify this is the following:

```
> K := NumberField(gens[1]);
> IsPower(Evaluate(f,K.1),3);
true 1/4*(-3*K.1^2 + 9*K.1 + 6)
```

Let D_0 denote the pullback of some point $P \in \mathbf{P}^1$ under the degree 3 map $C \to \mathbf{P}^1$. Then $D - D_0$ is a degree 0 defined over the rationals. According to `RankBounds` its class has infinite order in $J(\mathbf{Q})$. This was determined by considering its image under the composition, $J(\mathbf{Q}) \to J(k) \to Sel^{\phi}(J/k)$. Since there is no nontrivial ϕ-torsion, this amounts to saying that the image is nontrivial. We can verify this with the following computations (carried out internally by `RankBounds`).

```
> A<theta> := Domain(m); // the algebra representing Sel
> D := Evaluate(gens[1],theta);
> imD := m(D);
> assert imD ne SelJ!0;
```

```
> assert imD in SelJ;
```

Since the rank over **Q** has been computed unconditionally it follows that the ϕ-torsion subgroup of Sha is isomorphic to $(\mathbf{Z}/3)^2$, and since it contains a element not divisible by ϕ in Sha one can deduce that the 3-primary part of Sha is also isomorphic to $(\mathbf{Z}/3)^2$.

125.15.4 Partial Descent

The (fake) ϕ-Selmer set of the curve $C : y^q = f(x)$ defined over a number field k is computed using a map which sends a point (x, y) of C to $x - \theta$, where θ is a generic root of f. This map takes values in the product of the extensions of k defined by the irreducible factors of $f(x)$. More generally one can specify a partial descent map as follows. For each irreducible factor $f_i(x)$ of $f(x)$, fix an extension K_i/k and an irreducible factor $h_i(x) \in K_i[x]$ of $f_i(x)$. The map sending (x, y) to $(h_1(x), ..., h_n(x))$ induces a well defined map from $C(k)$ to an appropriate quotient of the units in the product of the K_i. A partial descent computes the set of elements which restrict into the image of $C(k_v)$ for every prime v of k. This set gives information on the set of rational points of C. If it is empty, then the curve has no rational points.

Geometrically this set can be understood as follows. The coverings parameterised by the ϕ-Selmer set are Galois covers of C with group isomorphic (as a Galois module) to the ϕ-torsion subgroup of the Jacobian. By Galois theory, any Galois submodule gives rise to intermediate coverings. A partial descent computes a set of everywhere locally solvable intermediate coverings corresponding to some Galois submodule of $J[\phi]$.

`qCoverPartialDescent(f,factors,q)`

`PrimeBound`	RngIntElt	*Default :* 0
`PrimeCutoff`	RngIntElt	*Default :* 0
`KnownPoints`	Set	*Default :*
Verbose	`CycCov`	*Maximum :* 3

Performs a partial descent on the curve $y^q = f(x)$ with respect to the descent map given by `factors`. The map returned is from this cartesian product into the abstract group $A(S, q)$ involved in the computation (the unramified outside S subgroup of the quotient of the etale algebra associated to the orbits of ramification points of C determined by `factors` by the subgroup generated by scalars and q-th powers).

The input `f` must be a separable polynomial defined over a number field k with class number 1. For each irreducible factor $h_i \in k[x]$ of f, the list `factors` must contain an irreducible factor of h_i defined over some (possibly trivial) extension of k. The optional parameters are the same as for `qCoverDescent`.

Example H125E41

Consider the cyclic cover $D : y^3 = 3(x^2+1)(x^2+17)(x^2-17)$.

```
> _<x>:=PolynomialRing(Rationals());
> f := 3*(x^2+1)*(x^2+17)*(x^2-17);
> pts := RationalPoints(f,3 : Bound:=100);
> pts;
{@
    [ -1, -12, 1 ],
    [ 1, -12, 1 ]
@}
```

So this appears to have only two rational points. We now apply partial descent by only factorising the third factor as follows:

```
> K<th> := NumberField(x^2-17);
> factor := Factorisation(x^2-17,K)[1,1];
> time selmerset, algebra:=qCoverPartialDescent(
>  3*(x^2+1)*(x^2+17)*(x^2-17),[*x^2+1,x^2+17,factor*],3
>  : KnownPoints := pts, PrimeBound := 1000 );
Time: 0.530
> selmerset;
{
    <1, 9, $.1 + 9>,
    <1, 9, 2*$.1 + 2>
}
> delta := <1,9,2+2*th>;
```

Each element of `selmerset` corresponds to a covering of D and every rational point on D lifts to one of these two coverings. For example δ corresponds to a covering Y_δ of D with defining equations

$$\lambda^2 y_1^3 = x^2 + 1,$$

$$9\lambda^2 y_2^3 = x^2 + 17,$$

$$(2\theta + 2)\lambda y_3^3 = x - \theta,$$

where θ is a square root of 17. This covering of D also evidently covers the genus one curve defined over $\mathbf{Q}(\theta)$,

$$C : (2\theta + 2)v^3 = (x^2 + 1)(x - \theta)\,.$$

One can now use the method of Elliptic Curve Chabauty to find all points of $Y_\delta(K)$ which map to points of $D(\mathbf{Q})$. Namely any such point gives rise to a point of $C(K)$ with x-coordinate in $\mathbf{Q}$. Provided we can obtain generators for $C(K)$, such points can be determined using `Chabauty`.

```
> P2<x,v,w>:=ProjectivePlane(K);
> C:=Curve(P2, -(delta[3])*v^3 + (x^2+w^2)*(x-th*w) );
> E,CtoE:=EllipticCurve(C);
> twotors := TorsionSubgroup(E);
> #twotors;
1
```

```
> covers,coverstoE:=TwoDescent(E);
> #covers;
1
> cover:=covers[1];
> covertoE:=coverstoE[1];
> pts:={@pt@covertoE : pt in RationalPoints(cover : Bound:=100)@};
> pts;
{@ (1/128*(-153*th + 1377) : 1/1024*(-8109*th + 28917) : 1),
(1/128*(-153*th + 1377) : 1/1024*(6885*th - 17901) : 1),
(1/194688*(-171346689*th + 708942857) : 1/60742656*(-4566609746937*th
    + 18826158509345) : 1), (1/194688*(-171346689*th + 708942857) :
1/60742656*(4566537140481*th - 18825505051241) : 1) @}
> gen := pts[1];
```

This shows that $E(K) \simeq \mathbf{Z}$, so `gen` generates a finite index subgroup.

```
> A:=FreeAbelianGroup(1);
> AtoE:=map<A -> E | elt :-> Eltseq(elt)[1]*gen >;
```

To apply Chabauty, we want to use a map $E \to \mathbf{P}^1$ such that the composition $C \to E \to \mathbf{P}^1$ sends (x, v, w) to x.

```
> P1:=ProjectiveSpace(Rationals(),1);
> CtoE;
Mapping from: CrvPln: C to CrvEll: E
with equations :
1/16*(9*th - 153)*v
1/128*(-729*th + 1377)*x
-1/17*th*x + w
and inverse
th*$.2
1/16*(-9*th + 153)*$.1
$.2 + 1/128*(1377*th - 12393)*$.3
> EtoP1:=map<E -> P1 | [th*E.2,E.2 + 1/128*(1377*th - 12393)*E.3]>;
> Chabauty(AtoE,EtoP1);
{
    -A.1
}
126
> Chabauty(AtoE,EtoP1 : IndexBound:=126);
{
    -A.1
}
126
> pointC:= (-A.1)@AtoE@@CtoE;
> pointC;
(-1 : -1 : 1)
```

Any preimage of this point on Y_δ has $x = -1$, and as such must lie above $(-1, -12) \in D(\mathbf{Q})$. A similar argument proves that the only rational point coming from the other element of the partial Selmer set is $(1, -12)$, proving $D(\mathbf{Q}) = \{(1, 12), (-1, -12)\}$.

125.16 Kummer Surfaces

The Kummer surface K associated to the Jacobian J of a genus 2 curve is the quotient of J by the inverse map. It can be embedded as a quartic hypersurface in projective 3-space whose only singularities are 16 ordinary double points. The Jacobian is a double cover of K ramified at these double points; they are the images of the two-torsion points on J. Resolving the singularities on K yields a $K3$-surface.

Currently, Kummer surfaces in MAGMA are not schemes, but are of type SrfKum. They can be used to perform arithmetic on the Jacobian without the need for reduction of divisors. The other nontrivial functionality that uses them is point searching.

The Kummer surface and arithmetic on it are implemented for Jacobians in arbitrary characteristic following [Mül10b] which extends earlier work by Flynn, described in chapter 3 of [CF96].

125.16.1 Creation of a Kummer Surface

`KummerSurface(J)`

The Kummer surface of the Jacobian J of a genus 2 curve.

125.16.2 Structure Operations

`DefiningPolynomial(K)`

The defining polynomial of the Kummer surface K.

125.16.3 Base Ring

`BaseField(K)`

`BaseRing(K)`

`CoefficientRing(K)`

The base field of the Kummer surface K.

125.16.4 Changing the Base Ring

`BaseChange(K, F)`

`BaseExtend(K, F)`

Extends the base field of the Kummer surface K to the field F.

`BaseChange(K, j)`

`BaseExtend(K, j)`

Extends the base field of the Kummer surface K by the map j, where j is a ring homomorphism with the base field of C as its domain.

`BaseChange(K, n)`

`BaseExtend(K, n)`

Extends the finite base field of the Kummer surface K over a finite field to the degree n extension.

125.17 Points on the Kummer Surface

Points are given by their projective coordinates, normalized depending on the base field.

125.17.1 Creation of Points

`K ! 0`

Returns the image of the identity element on the Kummer surface K, which is normalized to be the origin $(0 : 0 : 0 : 1)$.

`K ! [x1, x2, x3, x4]`

Returns the point on the Kummer surface K defined by the projective coordinates x_1, x_2, x_3, and x_4.

`K ! P`

Given a point P on the Jacobian of K, or on a Kummer surface for which K is a base extension, this returns the point on K.

`IsPoint(K, S)`

Given a sequence $S = [x_1, x_2, x_3, x_4]$ of elements of the base field of K, the function returns `true` if the point specified by the sequence defines the homogeneous coordinates of a point on the Kummer surface K. If so, the corresponding point on K is returned as the second value.

`Points(K,[x1, x2, x3])`

Returns the indexed set of points on the Kummer surface K with first three coordinates given by the sequence $[x_1, x_2, x_3]$.

125.17.2 Access Operations

`P[i]`

Returns the i-th coordinate of the point P, for $1 \leq i \leq 4$.

`Eltseq(P)`

`ElementToSequence(P)`

Given a point P on a Kummer surface, the function returns the coordinates of P as a sequence.

125.17.3 Predicates on Points

`P eq Q`

Given two points on the same Kummer surface, this returns `true` if and only if the points P and Q are equal.

`P ne Q`

Given two points on the same Kummer surface, this returns `false` if and only if the points P and Q are equal.

125.17.4 Arithmetic of Points

`-P`

Returns the negation of the point P on the Kummer surface, equal to P itself.

`n * P`

`P * n`

Returns the n-th multiple of the point P on the Kummer surface K.

`Double(P)`

Returns the double $2 * P$ of the point P.

`PseudoAdd(P1, P2, P3)`

Let P and Q be points on the Jacobian J of a genus 2 curve. Given the images P_1, P_2, and P_3 on the Kummer surface of points P, Q, and $P - Q$ on J, the function returns the image of $P + Q$.

`PseudoAddMultiple(P1, P2, P3, n)`

Let P and Q be points on the Jacobian J of a genus 2 curve. Given the images P_1, P_2, and P_3 on the Kummer surface of points P, Q, $P - Q$ on J, the function returns the image of $P + n * Q$.

125.17.5 Rational Points on the Kummer Surface

`RationalPoints(K, Q)`

Given the Kummer surface of the Jacobian of a genus 2 hyperelliptic curve defined over a ring R and sequence Q of three elements of R, the function returns an indexed set containing those points on K whose first three coordinates correspond to the three terms of Q.

Example H125E42

We search for some points on the Kummer surface of the hyperelliptic curve $y^2 = x^5 - 7$ defined over the rational field.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^5-7);
> Genus(C);
2
> J := Jacobian(C);
> K := KummerSurface(J);
> K;
Kummer surface of Jacobian of Hyperelliptic Curve defined by
    y^2 = x^5 - 7 over Rational Field
> Points(K, [0,1,2]);
{@ (0 : 1 : 2 : 4) @}
> Points(K, [1,3,2]);
{@ @}
> Points(K, [0,1,3]);
{@ (0 : 1 : 3 : 9) @}
```

125.17.6 Pullback to the Jacobian

`Points(J, P)`

`RationalPoints(J, P)`

Given a point P on the Kummer surface associated to the Jacobian J (of a genus 2 curve), the function returns the indexed set of points on J mapping to P.

125.18 Analytic Jacobians of Hyperelliptic Curves

This section contains descriptions of functions pertaining to the creation and use of analytic Jacobians for hyperelliptic curves.

Suppose C is a curve of genus g defined over the complex numbers. The analytic Jacobian of the curve is an abelian torus and is constructed as follows: We view the complex points on the curve, $C(\mathbf{C})$, as a compact Riemann surface of genus g. It is known that the dimension of the vector space of holomorphic differentials is equal to the genus of the curve. So let ϕ_i, $i = 1, 2, \ldots, g$ be a basis for this vector space and set $\overline{\phi} = {}^t(\phi_1, \ldots, \phi_g)$, a vector of holomorphic 1-forms. For a hyperelliptic curve there is a natural choice of holomorphic differentials, namely $\phi_i = x^{i-1}dx/y$. We can now define a mapping $\mathrm{Int} : C \to \mathbf{C}^g$ by

$$P \mapsto \int_P^\infty \overline{\phi}.$$

To remove the dependency of this mapping on the path of integration, we define Λ to be the image of the map from the first homology group of C, $H_1(C, \mathbf{Z})$, to $\mathbf{C}^g$ which sends a closed path, $\sigma \in H_1(C, \mathbf{Z})$, on the Riemann surface to $\int_\sigma \overline{\phi}$. Then Λ turns out to be a lattice in $\mathbf{C}^g$ and so we get the complex torus $\mathbf{C}^g/\Lambda$. We then see that Int is a well-defined mapping from C into $\mathbf{C}^g/\Lambda$. We extend this mapping additively to the divisors of degree zero on C. The Abel part of the Abel-Jacobi theorem states that two divisors map to the same image under Int if and only if they are linearly equivalent. The Jacobi part of the theorem asserts that the map is onto. So there is a bijection between the points of $\mathbf{C}^g/\Lambda$ and the points of the (algebraic) Jacobian.

So far the analytic Jacobian $\mathbf{C}^g/\Lambda$ is just a torus, but it can be made into an abelian manifold by choosing a polarization. Let $A_1, \ldots, A_g, B_1, \ldots, B_g$ be a symplectic homology basis. That is, the A_i and B_i are closed paths on the Riemann surface such that all intersection numbers are 0 except that A_i intersects B_i with intersection number 1 for each i. Let ω_1 and ω_2 be the $g \times g$-matrices with entries

$$\omega_{1ij} = \int_{B_j} \phi_i,$$

and

$$\omega_{2ij} = \int_{A_j} \phi_i.$$

Then the columns of $P = (\omega_1, \omega_2)$ form a $\mathbf{Z}$-basis for the lattice $\Lambda = P\mathbf{Z}^{2g}$. The matrix P is called the (big) period matrix. The space of complex, symmetric $g \times g$ matrices with positive definite imaginary part is called the Siegel upper-half space of degree g. Because we chose a symplectic basis for the homology it follows that $\tau = \omega_2^{-1}\omega_1$ is an element of Siegel upper half-space. We will refer to τ as the small period matrix.

Set

$$J = \begin{pmatrix} 0 & \mathbf{1}_g \\ -\mathbf{1}_g & 0 \end{pmatrix}.$$

If we define

$$E(Px, Py) = {}^t x J y,$$

for any $x, y \in \mathbf{R}^{2g}$, then E is a Riemann form for the torus $\mathbf{C}^g/\Lambda$ and so the torus acquires a polarization. The existence of a Riemann form on a torus is a necessary and sufficient condition for the torus to be embeddable into projective space. The image is isomorphic to the (algebraic) Jacobian.

In order for the theory describing the map from the analytic Jacobian to the algebraic Jacobian (see Mumford [Mum84]) to work we actually need a special symplectic basis linked to a particular ordering of the roots of the hyperelliptic polynomial. For an example of such a basis see Mumford [Mum84, Chap III.5]. The basis used can be obtained using the function `HomologyBasis` and the roots in the corresponding order are stored as the attribute `A'Roots` (where `A` is an analytic Jacobian).

125.18.1 Creation and Access Functions

`AnalyticJacobian(f)`

Given $f \in \mathbf{C}[x]$ where $\mathbf{C}$ is a complex field (see Chapter 25), this function returns the analytic Jacobian of the hyperelliptic curve defined by $y^2 = f(x)$. The polynomial must have degree at least 3 and the complex field precision at least 20 and, for the moment, less than 2000.

`HyperellipticPolynomial(A)`

Returns the polynomial defining the hyperelliptic curve whose Jacobian is A.

`SmallPeriodMatrix(A)`

The small period matrix of the analytic Jacobian A. If A has dimension g or equivalently the hyperelliptic curve has genus g, then this is a symmetric $g \times g$ matrix with positive definite imaginary part. If $P = (\omega_1, \omega_2)$ is the full period matrix (see `BigPeriodMatrix`) then the small period matrix is defined by $\omega_2^{-1}\omega_1$.

`BigPeriodMatrix(A)`

The full period matrix of the hyperelliptic curve of the Jacobian A. If A has dimension g or equivalently the hyperelliptic curve has genus g, then this is a $g \times 2g$ matrix. The analytic Jacobian as a torus equals $\mathbf{C}^g/\Lambda$, where Λ is the $\mathbf{Z}$-lattice spanned by the columns of the period matrix. The period matrix is computed with respect to the holomorphic differentials $\phi_i = x^{i-1}dx/y$ and a symplectic basis for the homology that can be retrieved using `HomologyBasis`.

`HomologyBasis(A)`

The symplectic homology basis used for the computation of the period matrix of the Jacobian A. First assume that `HyperellipticPolynomial(A)` has odd degree. The function `HomologyBasis(A)` returns three values which we label `basepoints`, `loops` and `S`. Then `basepoints` is a list of points in the complex plane. The return value`loops` is a list of $2g$ lists of indices into `basepoints`. So the list `[i1, i2, i3, ...]` corresponds to the loop consisting of joining the straight lines from `basepoints[in]` to `basepoints[in+1]` for `i` $= 1, 2, \ldots$. These closed loops form a

homology basis for the curve but probably not a symplectic basis. The third return value, S, is a matrix giving the linear combinations of the loops that were used to form a symplectic homology basis and to compute the period matrix.

If HyperellipticPolynomial(A) has even degree then the returned homology basis is not for the Riemann surface for this curve, but for the curve of odd degree obtained by sending $a =$ A`InfiniteRoot to infinity through the linear fractional transformation given by $x \mapsto 1/(x-a)$. So one needs to apply the inverse transformation to the returned basis in order to get the basis that was used to compute the period matrix.

Dimension(A)

Genus(A)

The dimension of the Jacobian A as a complex abelian variety. This is equal to the genus of the curve C for which A is the Jacobian.

BaseField(A)

BaseRing(A)

CoefficientRing(A)

The base field of the Jacobian A.

125.18.2 Maps between Jacobians

ToAnalyticJacobian(x, y, A)

Let A be the analytic Jacobian of $y^2 = f(x)$. This function maps the point (x, y) on the curve to the analytic Jacobian. More precisely, let $a = \infty$ when HyperellipticPolynomial has odd degree and $a =$A`InfiniteRoot otherwise. Then it maps the divisor $(x, y) - (a, 0)$ to the analytic Jacobian. As any point on the algebraic Jacobian is simply a sum of such divisors, we can get its image by linearity of the map. The function returns a $g \times 1$ matrix. This should be thought of as an element of $\mathbf{C}^g/\Lambda$ where Λ is the $\mathbf{Z}$-lattice generated by the columns of the BigPeriodMatrix.

FromAnalyticJacobian(z, A)

Let A be an analytic Jacobian of $y^2 = f(x)$, with small period matrix τ. Let z be a $g \times 1$ complex matrix (thought of as an element of $\mathbf{C}^g/\Lambda$ where Λ is the $\mathbf{Z}$-lattice generated by the columns of the BigPeriodMatrix). This function returns a list of g (the dimension of A), or fewer, pairs $P_i = \langle x_i, y_i \rangle$ satisfying $y^2 = f(x)$. This is an element of the algebraic Jacobian when interpreted as the divisor $\sum_{i=1}^{g} P_i - g\infty$.

Example H125E43

We give an example of moving between the algebraic and analytic Jacobians. We take two points on an algebraic Jacobian, map them to the analytic Jacobian, add them there and map the answer back to the algebraic Jacobian. We check that the answer corresponds to adding the points on the algebraic Jacobian.

```
> P<x> := PolynomialRing(RationalField());
> f := -8*x^5 - 14*x^4 + 28*x^3 - 30*x^2 + 6*x;
> C := HyperellipticCurve(f);
> J := Jacobian(C);
> pts := ReducedBasis(RationalPoints(J:Bound:=500)); pts;
[ (x^2 + 1, -3*x + 5, 2), (x^2 - 4/9*x + 1/9, -55/27*x + 43/27, 2) ]
```

This Jacobian has rank 2 and these are two generators. We will use them as our two points.

```
> P1 := pts[1]; P2 := pts[2];
> C<I> := ComplexField(50);
> K<x> := PolynomialRing(C);
> f2 := Evaluate(f,x);
> div1 := [r[1] : r in Roots(Evaluate(ElementToSequence(P1)[1],x))];
> div1 := [<d1,Evaluate(ElementToSequence(P1)[2],d1)> : d1 in div1];
> div2 := [r[1] : r in Roots(Evaluate(ElementToSequence(P2)[1],x))];
> div2 := [<d1,Evaluate(ElementToSequence(P2)[2],d1)> : d1 in div2];
> A := AnalyticJacobian(f2);
```

We are working with 50 decimal places. The sequences `div1` and `div2` are the two points on the algebraic Jacobian represented as pairs of $\langle x, y\rangle$ coordinates.

```
> pt1 := &+[ToAnalyticJacobian(d[1],d[2],A) : d in div1];
> pt2 := &+[ToAnalyticJacobian(d[1],d[2],A) : d in div2];
```

Now `pt1` and `pt2` are the images of the two points `P1` and `P2` on the analytic Jacobian (two 2×1 matrices).

```
> sum := FromAnalyticJacobian(pt1+pt2,A);
> sum;
[ <0.62500000000000000000000000000000000000000000000024 +
-1.49999999999999999999999999999999999999999999992*I,
1.73437500000000000000000000000000000000000000000053 +
-11.71874999999999999999999999999999999999999999985*I>,
<0.62500000000000000000000000000000000000000000000036 +
1.49999999999999999999999999999999999999999999992*I,
1.73437500000000000000000000000000000000000000000075 +
11.71874999999999999999999999999999999999999999986*I> ]
```

Now we have mapped sum back to the algebraic Jacobian where it is given as a sequence of two $\langle x, y\rangle$ coordinates. We compare the polynomial having these two x-coordinates as roots, with the algebraic sum of `P1` and `P2`.

```
> clst := Coefficients((x-sum[1][1])*(x-sum[2][1]));
> xpol := P![BestApproximation(Re(c),1000) : c in clst];
```

```
> xpol;
x^2 - 5/4*x + 169/64
> P1+P2;
(x^2 - 5/4*x + 169/64, 125/16*x - 403/128, 2)
```

125.18.2.1 Isomorphisms, Isogenies and Endomorphism Rings of Analytic Jacobians

In this section we discuss the functionality provided for finding isomorphisms and isogenies between different analytic Jacobians and also for computing the endomorphism ring of an analytic Jacobian.

Suppose we have two abelian varieties $A_1 = \mathbf{C}^g/\Lambda_1$ and $A_2 = C^g/\Lambda_2$ with a morphism $\phi : A_1 \mapsto A_2$. This map lifts to a map $\mathbf{C}^g \mapsto \mathbf{C}^g$, given by some complex $g \times g$ matrix, α. This is called the complex representation of ϕ. Suppose Λ_i is spanned by the columns of the $g \times 2g$ matrix P_i. As $\phi(\Lambda_1) \subset \Lambda_2$, we see that there must exist an integral $2g \times 2g$ matrix M such that $\alpha P_1 = P_2 M$. M is called the rational representation of ϕ. If A_1 and A_2 are isomorphic and P_1 and P_2 are Frobenius bases, that is, they provide a basis with respect to which the polarization is given by the matrix $J = \begin{pmatrix} 0 & \mathbf{1}_g \\ -\mathbf{1}_g & 0 \end{pmatrix}$, then it follows that M must be a symplectic matrix. That is, M must be such that $MJ^tM = J$. The symplectic matrices act on Siegel upper half-space. If $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is a symplectic matrix and τ an element of Siegel upper half-space, then the action is given by $\tau \mapsto (a\tau + b)(c\tau + d)^{-1}$. Note that if there is an isomorphism from A_1 to A_2 with rational representation M then τ_1 equals the result of letting ${}^tM$ (not M) act on τ_2.

In the case of finding isomorphisms between abelian surfaces, the code makes use of a fundamental domain for 2-dimensional upper half-space (see [Got59]). This often works well even with relatively low precision. The other functions all rely on the function `LinearRelation` and work with greater reliability if a high precision is chosen. This also means that it can happen that these functions miss finding a map. Even if a map is reported it might be an artifact of insufficient precision causing `LinearRelation` to identify a relation that disappears at higher precision. Of course, every effort was made to make these functions work as well as possible and no cases are known where sufficient precision fails to give correct results.

`To2DUpperHalfSpaceFundamentalDomian(z)`

Given a complex matrix in 2-dimensional Siegel upper half-space (that is a symmetric matrix with positive definite imaginary part), this function returns two matrices. The first is an element of the fundamental domain for 2-dimensional Siegel upper half-space described by Gottschling in [Got59] and the second is a symplectic matrix that takes the input to the first return value. Note that if the input is equivalent to an element on the border of the fundamental domain the returned value might depend heavily on the least significant digits of the input.

`AnalyticHomomorphisms(t1, t2)`

Given two small period matrices t_1 and t_2, this function returns a basis for the **Z**-module of $2g \times 2g$ integer matrices M such that there exists a complex $g \times g$ matrix α such that $\alpha(t_1, 1) = (t_2, 1)M$.

`IsIsomorphicSmallPeriodMatrices(t1,t2)`

For two small period matrices t_1 and t_2, this function finds a $2g \times 2g$ symplectic integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha(t_1, 1) = (t_2, 1)M$. Such matrices define isomorphisms between the corresponding analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second return value is the matrix, if found. Zero matrix otherwise.

`IsIsomorphicBigPeriodMatrices(P1, P2)`

For two big period matrices P_1 and P_2, this function finds a $2g \times 2g$ symplectic integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha P_1 = P_2 M$. Such matrices define isomorphisms between the corresponding analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second and third return values are M and α, if found.

`IsIsomorphic(A1, A2)`

For two analytic Jacobians A_1 and A_2 with big period matrices P_1 and P_2, this function finds a $2g \times 2g$ symplectic integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha P_1 = P_2 M$. Such matrices define isomorphisms between the analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second and third return values are M and α, if found.

`IsIsogenousPeriodMatrices(P1, P2)`

For two period matrices (small or big) P_1 and P_2, this function finds a nonsingular $2g \times 2g$ integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha(P_1, 1) = (P_2, 1)M$, in case of small period matrices, or $\alpha P_1 = P_2 M$ for big period matrices. Such a matrix defines an isogeny between the corresponding analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second return value is M, if found. In the case of big period matrices α is the third return value.

`IsIsogenous(A1, A2)`

For two analytic Jacobians A_1 and A_2 with big period matrices P_1 and P_2, this function finds a nonsingular $2g \times 2g$ integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha P_1 = P_2 M$. Such a matrix defines an isogeny between the analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second and third return values are M and α, if found.

`EndomorphismRing(P)`

This function returns the endomorphism ring, as a matrix algebra, of the analytic Jacobian associated to the given period matrix P. If a big period matrix, P, is given then it also returns a list of α-matrices such that $\alpha P = PM$, for each generator, M, of the matrix algebra.

`EndomorphismRing(A)`

This function returns the endomorphism ring, as a matrix algebra, of the given analytic Jacobian A. Suppose the analytic Jacobian has big period matrix P. The second return value is a list of α-matrices such that $\alpha P = PM$, for each generator, M, of the matrix algebra.

Example H125E44

We give an example of how MAGMA can be used to find rational isogenies between the Jacobians of genus 2 curves. Let us consider the two curves

$$y^2 = x^5 - 4x^4 + 8x^2 - 4x,$$

and

$$y^2 = x^5 + 4x^4 + 10x^3 + 12x^2 + x.$$

These are curves 1 and 3 in the twenty second isogeny class of Smart [Sma97]. We compute their analytic Jacobians to 100 decimal places.

```
> K<x> := PolynomialRing(RationalField());
> C<i> := ComplexField(100);
> KC<xc> := PolynomialRing(C);
> f1 := x^5 - 4*x^4 + 8*x^2 - 4*x;
> f1C := Evaluate(f1,xc);
> A1 := AnalyticJacobian(f1C);
> f2 := x^5 + 4*x^4 + 10*x^3 + 12*x^2 + x;
> f2C := Evaluate(f2,xc);
> A2 := AnalyticJacobian(f2C);
```

We now get a basis for the $\mathbf{Z}$-module of isogenies from `A1` to `A2`. Note that `IsIsogenous` returns `true` for these two Jacobians, but it does not return an isogeny defined over $\mathbf{Q}$.

```
> Mlst := AnalyticHomomorphisms(SmallPeriodMatrix(A1),SmallPeriodMatrix(A2));
> Mlst;
[
  [ 1  0 -1  0]
  [ 0  1  0  0]
  [ 1  1  1  0]
  [ 1  1 -1  2],

  [ 1  0  1  0]
  [ 0  0  0 -1]
  [-1  0  1 -1]
```

```
  [ 1  2  1 -1],

  [ 0  1  0 -1]
  [ 1  0  0  0]
  [ 1  1  0  1]
  [ 1  1  2 -1],

  [ 0  1  0  1]
  [ 0  0 -1  0]
  [ 0 -1 -1  1]
  [ 2  1 -1  1]
]
>
```

For each of these four "M" matrices let us find the corresponding α matrices.

```
> P1 := BigPeriodMatrix(A1);
> P2 := BigPeriodMatrix(A2);
> alst := [Submatrix(P2*Matrix(C,M),1,1,2,2)
>          *Submatrix(P1,1,1,2,2)^-1 : M in Mlst];
> alst[1][1,1];
9.498572673182793269164480489911434797891377549198242765578603486437979481059498
481543257426864218501E-101 + -1.414213562373095048801688724209698078569671875376
769480731766797379907324784621070388503875343276415 73*i
>
```

So at least `alst[1]` does not correspond to a rational isogeny. In general none of the four α-matrices might correspond to a rational isogeny. But it is possible that some $\mathbf{Z}$-linear combination of them is defined over the rationals and we want to know whether this actually happens. We can find out by recognizing the entries of the α-matrices as algebraic numbers. This can be done using the `PowerRelation` function.

```
> SetDefaultRealFieldPrecision(100);
> pol := PowerRelation(C!alst[4][1,1],8:Al:="LLL");
> Evaluate(pol,x);
x^4 + 2*x^2 - 1
```

It turns out that every entry of each of the `alst` matrices is in the number field defined by the polynomial $x^8 + 12x^6 + 34x^4 + 52x^2 + 1$. We can use `LinearRelation` to write each entry as a linear combination of the elements of a power basis for this number field. For example:

```
> aroot := C!Roots(Evaluate(x^8 + 12*x^6 + 34*x^4 + 52*x^2 + 1,xc))[1][1];
> basis := [aroot^i : i in [0..7]];
> LinearRelation(Append(basis,alst[1][1,1]));
[ 0, 411, 0, 293, 0, 107, 0, 9, -40 ]
```

So we can write each α-matrix as an algebraic number in the number field defined by $x^8 + 12x^6 + 34x^4 + 52x^2 + 1$. It is then a simple matter to find a linear combination of the α-matrices that is defined over $\mathbf{Q}$. In this particular case we get lucky and `alst[3]` itself is already rational:

```
> alpha := alst[3];
```

```
> alpha;
[1.1429873912822749822157835483053409594519099948227986612151258432276326359067
  381956754480218601720 30E-100 + 1.2392853976334069864667648932075686036279851 2
  520125066621751762100135189353096335779088349296 2362769E-100*i
  -0.99999999999999999999999999999999999999999999999999999999999999999999999999
  99999999999999999999999999 + -1.186746029522145753827915120679934647863639129
  87256639374529369743809277710674524919691817563 2802507E-100*i]
[-2.000000000000000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000 + 8.334710799700090834903499884806428065464872993741
  2275687347263052454397784385678876631050632 71942125E-101*i
  1.000000000000000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000 + -1.29064186778300376477121365149925086148800581946
  53319435140080394516627397309641436258161370 36994307E-100*i]
> Matrix(2,2,[Round(Real(x)) : x in Eltseq(alpha)]);
[ 0  1]
[ 2 -1]
```

So the two Jacobians are **Q**-isogenous. We can use `alpha` to map points from the one Jacobian to the other. We check that it maps rational points to rational points.

```
> C1 := HyperellipticCurve(f1);
> J1 := Jacobian(C1);
> pts1 := RationalPoints(J1:Bound:=500);
> pts1;
{@ (1, 0, 0), (x^2 - 2*x + 1, 1/2*x + 1/2, 2), (x^2 - 2*x + 1, -1/2*x - 1/2, 2),
(x, 0, 1), (x - 1, 1, 1), (x - 1, -1, 1), (x^2 - 2, 0, 2), (x^2 + 2, 4*x, 2),
(x^2 + 2, -4*x, 2), (x^2 - x, x, 2), (x^2 - x, -x, 2), (x^2 - 4*x - 2, 4*x, 2),
(x^2 - 4*x - 2, -4*x, 2), (x^2 - 4*x + 2, 0, 2), (x^2 - 9/4*x - 4, 3/8*x - 2,
2), (x^2 - 9/4*x - 4, -3/8*x + 2, 2), (x^2 - 52/9*x + 50/9, 152/27*x - 160/27,
2), (x^2 - 52/9*x + 50/9, -152/27*x + 160/27, 2), (x^2 + 16/25*x - 18/25,
  152/125*x - 96/125, 2), (x^2 + 16/25*x - 18/25, -152/125*x + 96/125, 2) @}
> P1 := pts1[8]; P1;
(x^2 + 2, 4*x, 2)
> div1 := [r[1] : r in Roots(Evaluate(ElementToSequence(P1)[1],xc))];
> div1 := [<d1,Evaluate(ElementToSequence(P1)[2],d1)> : d1 in div1];
> pt1 := &+[ToAnalyticJacobian(d[1],d[2],A1) : d in div1];
> P2 := FromAnalyticJacobian(alpha*pt1,A2);
> clst := Coefficients((xc-P2[1][1])*(xc-P2[2][1]));
> xpol := K![BestApproximation(Re(c),10^50) : c in clst];
> xpol;
x^2 + x + 1
> C2 := HyperellipticCurve(f2);
> J2 := Jacobian(C2);
> pts2 := RationalPoints(J2:Bound:=500);
> pts2;
{@ (1, 0, 0), (x^2 + 2*x + 1, x - 1, 2), (x^2 + 2*x + 1, -x + 1, 2), (x + 1, 2,
1), (x + 1, -2, 1), (x, 0, 1), (x^2 + x, 2*x, 2), (x^2 + x, -2*x, 2), (x^2 + x +
  1, 2*x - 1, 2), (x^2 + x + 1, -2*x + 1, 2), (x^2 - 23*x - 16, 122*x + 84, 2),
(x^2 - 23*x - 16, -122*x - 84, 2), (x^2 + 23/4*x + 9, 59/8*x + 57/2, 2), (x^2 +
```

```
  23/4*x + 9, -59/8*x - 57/2, 2), (x^2 - 106/9*x + 1/9, 2599/54*x - 19/54, 2),
(x^2 - 106/9*x + 1/9, -2599/54*x + 19/54, 2), (x^2 - 9/16*x - 1/16, 269/64*x +
  21/64, 2), (x^2 - 9/16*x - 1/16, -269/64*x - 21/64, 2) @}
```

125.18.3 From Period Matrix to Curve

`RosenhainInvariants(t)`

Given a small period matrix t corresponding to an analytic Jacobian A of genus g, this function returns a set S of $2g-1$ complex numbers such that the hyperelliptic curve $y^2 = x(x-1)\prod_{s\in S}(x-s)$ has Jacobian isomorphic to A. The name of the function comes from the fact that a hyperelliptic curve in the form $y^2 = x(x-1)(x-\lambda_1)...(x-\lambda_{2g-1})$ is said to be in Rosenhain normal form.

Example H125E45

We give an example of how MAGMA can be used to find the equation of a genus 2 curve whose Jacobian has Complex Multiplication by a given field. We use the field $\mathbf{Q}(\sqrt{-2+\sqrt{2}})$.

```
> SetSeed(1);
> Q := RationalField();
> P<x> := PolynomialRing(Q);
> R<s> := NumberField(x^2-2);
> PP<x> := PolynomialRing(R);
> RF := NumberField(x^2-(-2+s));
> CMF<t> := AbsoluteField(RF);
> O := MaximalOrder(CMF);
```

Now, for a CM type Φ, and $\mathcal{O}$ the ring of integers, $\mathbf{C}^2/\Phi(\mathcal{O})$ is a torus with Complex Multiplication by the maximal order of our field $\mathbf{Q}(\sqrt{-2+\sqrt{2}})$. In order for this to be an abelian variety we have to find a principal polarization. This can be done using Algorithm 1 in [Wam99]. In this case it turns out that a principal polarization is essentially given by a generator of the inverse different.

```
> D := Different(O);
> IsPrincipal(D);
true
> xi := -1/8*O.2; // a chosen generator of D^-1
> xi*O eq D^-1;
true
> auts := Automorphisms(CMF : Abelian := true);
> cc := auts[3];
> cc(cc(t)) eq t;
true
> cc(xi^2) eq xi^2;
```

```
true
```

So `cc` is complex multiplication and `xi` squared is in the real subfield. By Theorem 3 of [Wam99] we see that this gives a principal polarization. We now find a Frobenius basis for our lattice with respect to the non-degenerate Riemann form given by `xi`.

```
> Z := IntegerRing();
> E := Matrix(Z,4,4,[Trace(xi*cc(a)*b) : b in Basis(O), a in Basis(O)]);
> D, C := FrobeniusFormAlternating(E); D;
[ 0  0  1  0]
[ 0  0  0  1]
[-1  0  0  0]
[ 0 -1  0  0]
> newb := ElementToSequence(Matrix(O,C)*Matrix(O,4,1,Basis(O)));
> SetKantPrecision(O,100);
> Abs(Re(Conjugate(xi,2))) lt 10^-10 and Im(Conjugate(xi,2)) gt 0;
true
> Abs(Re(Conjugate(xi,4))) lt 10^-10 and Im(Conjugate(xi,4)) gt 0;
true
```

The CM type Φ is given by the second and fourth complex embeddings. We can finally write down a big period matrix and find the element in the Siegel upper half-space corresponding to our CM Jacobian:

```
> C := ComplexField(100);
> BigPM := Matrix(C,2,4,[Conjugate(b,2) : b in newb] cat
>                       [Conjugate(b,4) : b in newb]);
> tau := Submatrix(BigPM,1,3,2,2)^-1*Submatrix(BigPM,1,1,2,2);
```

We can use `EndomorphismRing` to check that the analytic Jacobian corresponding to `tau` does have CM by the correct field:

```
> MA := EndomorphismRing(tau); Dimension(MA);
4
> MAGens := SetToSequence(Generators(MA)); MAGens;
[
    [-1  0  0  0]
    [ 0 -1  0  0]
    [ 0  0 -1  0]
    [ 0  0  0 -1],
    [  0   0 -17  -7]
    [  0   0  -7  -3]
    [  5 -12   0   0]
    [-12  29   0   0]
]
> MP := [MinimalPolynomial(g): g in MAGens];
> IsIsomorphic(NumberField(rep{f: f in MP | Degree(f) gt 1}), CMF);
true
```

Now let us find a curve with this Jacobian.

```
> S := RosenhainInvariants(tau);
```

```
> P<x> := PolynomialRing(C);
> f := x*(x-1)*&*{x-a : a in S};
> IC := IgusaClebschInvariants(f);
> ICp := [BestApproximation(Re(r),10^50) : r in
>          [IC[1]/IC[1], IC[2]/IC[1]^2, IC[3]/IC[1]^3, IC[4]/IC[1]^5]];
> ICp;
[ 1, 5/324, 31/5832, 1/1836660096 ]
> C1 := HyperellipticCurveFromIgusaClebsch(ICp);
> C2 := ReducedWamelenModel(C1);
> C2;
Hyperelliptic Curve defined by y^2 = -x^5 - 3*x^4 + 2*x^3 + 6*x^2 - 3*x - 1
over Rational Field
```

Let's check that this curve's analytic Jacobian is isomorphic to the one corresponding to `tau`.

```
> P<x> := PolynomialRing(C);
> f := -x^5 - 3*x^4 + 2*x^3 + 6*x^2 - 3*x - 1;
> A := AnalyticJacobian(f);
> IsIsomorphicSmallPeriodMatrices(tau,SmallPeriodMatrix(A));
true
[ 1 -2  0  0]
[ 0  0  2  1]
[ 1 -2  5  2]
[ 2 -5  4  2]
```

125.18.4 Voronoi Cells

This section describes two functions that are concerned with the computation of Voronoi cells.

`Delaunay(sites)`

This function computes the Delaunay triangulation for the sites given by *sites*. The argument *sites* should be a sequence of pairs (of type `Tup`) of real numbers. The real numbers should all belong to the same real field. For n sites, a sequence of n sequences is returned. The ith sequence contains the list of indices of the sites to which site i should be joined to form the triangulation.

`Voronoi(sites)`

This function computes the Voronoi cells for the sites given by *sites*. A Voronoi cell of a site consists of all those points in the plane closer to that site than to any other. The argument *sites* should be a sequence of pairs (of type `Tup`) of real numbers. The real numbers can be either fixed precision or free reals, but they should all be in the same real field.

Three sequences, *siteedges*, *dualsites* and *cells* are returned. The sequence *siteedges* is what the intrinsic `Delaunay` would have returned (so it defines the Delaunay triangulation). The sequence *dualsites* is a sequence of triples $\langle x, y, m \rangle$,

interpreted as follows: If m is zero the triple represents the point x, y. If m is non-zero the triple represents a point "at infinity" in the direction of the vector x, y. For n sites, *cells* is a sequence of n sequences. The ith sequence is a list $[i1, i2, i3, ..., im]$ such that the cell around site i is formed by connecting *dualsites[i1]* to *dualsites[i2]*, ... If a cell is infinite the first and last indices in the sequence point to infinite points. That is, the two infinite sides are given by the lines $(x_0, y_0) + t(x_1, y_1)$, for $t \geq 0$ where (x_0, y_0) equals *dualsites[i2][1], dualsites[i2][2]* and (x_1, y_1) equals *dualsites[i1][1], dualsites[i1][2]* or (x_0, y_0) equals *dualsites[im-1][1], dualsites[im-1][2]* and (x_1, y_1) equals *dualsites[im][1], dualsites[im][2]*.

125.19 Bibliography

[**BD09**] Nils Bruin and Kevin Doerksen. The arithmetic of genus two curves with (4,4)-split Jacobians. ArXiv preprint. URL:http://arxiv.org/abs/0902.3480, 2009.

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**Bru02**] N. R. Bruin. *Chabauty methods and covering techniques applied to generalized Fermat equations*, volume 133 of *CWI Tract*. Stichting Mathematisch Centrum Centrum voor Wiskunde en Informatica, Amsterdam, 2002. Dissertation, University of Leiden, Leiden, 1999.

[**BS09**] Nils Bruin and Michael Stoll. Two-cover descent on hyperelliptic curves. *Math. Comp.*, 78:2347–2370, 2009.

[**Car03**] G. Cardona. On the number of curves of genus 2 over a finite field. *Finite Fields and Their Applications*, 9(4):505–526, 2003.

[**CF96**] J.W.S. Cassels and E.V. Flynn. *Prolegomena to a Middlebrow Arithmetic of Curves of Genus* 2. Cambridge University Press, Cambridge, 1996.

[**CNP05**] G. Cardona, E. Nart, and J. Pujolas. Curves of genus two over fields of even characteristic. *Mathematische Zeitschrift*, 250:177–201, 2005.

[**CQ05**] Gabriel Cardona and Jordi Quer. Field of moduli and field of definition for curves of genus 2. *Lecture Notes Ser. Comput.*, 13:71–83, 2005.

[**Cre12**] Brendan Creutz. Explicit descent in the picard group of a cyclic cover of the projective line. In Everett Howe and Kiran Kedlaya, editors, *ANTS X: Proceedings of the Tenth Algorithmic Number Theory Symposium*, volume 1 of *OBS*. Mathematics Sciences Publishers, 2012.

[**FS97**] E.V. Flynn and N.P. Smart. Canonical heights on the Jacobians of curves of genus 2 and the infinite descent. *Acta Arith.*, 79:333 – 352, 1997.

[**GH00**] P. Gaudry and R. Harley. Counting Points on Hyperelliptic Curves over Finite Fields. In Bosma [Bos00], pages 313–332.

[**Got59**] E. Gottschling. Explizite Bestimmung der Randflächen des Fundamentalbereiches der Modulgruppe zweiten Grades. *Math. Annalen*, 138:103–124, 1959.

[**Hol06**] David Holmes. Canonical heights on hyperelliptic curves. Preprint URL:http://arxiv.org/abs/1004.4503, 2006.

[**Hub06**] Hendrik Hubrechts. Point counting in families of hyperelliptic curves. Preprint URL:http://arxiv.org/abs/math.NT/0601438, 2006.

[**Ked01**] Kiran S. Kedlaya. Counting Points on Hyperelliptic Curves using Monsky-Washnitzer Cohomology. *J. Ramanujan Math. Soc.*, 16:323 – 338, 2001.

[**LL**] R. Lercier and D. Lubicz. A Quasi-Quadratic Time Algorithm for Hyperelliptic Curve Point Counting. *to appear*.

[**LR12**] R. Lercier and C. Ritzenthaler. Hyperelliptic curves and their invariants: geometric, arithmetic and algorithmic aspects. *Journal of Algebra*, 2012. URL:http://dx.doi.org/10.1016/j.jalgebra.2012.07.054.

[**Mae90**] T. Maeda. On the invariant fields of binary octavics. *Hiroshima Math. J.*, 20:619–632, 1990.

[**Mes91**] J.-F. Mestre. Construction de courbes de genre 2 à partir de leurs modules. In T. Mora and C. Traverso, editors, *Effective methods in algebraic geometry*, volume 94 of *Progr. Math.*, pages 313–334. Birkhäuser, 1991. Proc. Congress in Livorno, Italy, April 17–21, 1990.

[**Mül10a**] Jan Steffen Müller. *Computing canonical heights on Jacobians*. PhD Thesis, Universität Bayreuth, 2010. available at URL:http://www.math.uni-hamburg.de/home/js.mueller/.

[**Mül10b**] Jan Steffen Müller. Explicit Kummer surface formulas for arbitrary characteristic. *LMS J. Comput. Math.*, 4:47–64, 2010.

[**Mum84**] David Mumford. *Tata Lectures on Theta II*, volume 43 of *Progress in Mathematics*. Birkhäuser, 1984.

[**PS97**] Bjorn Poonen and Ed Schaefer. Explicit descent for Jacobians of cyclic covers of the projective line. *J. Reine Angew. Math.*, 488:141–188, 1997.

[**PS99**] Bjorn Poonen and Michael Stoll. The Cassels-Tate pairing on polarized abelian varieties. *Ann. of Math. (2)*, 150(3):1109–1149, 1999.

[**Shi67**] T. Shioda. On the Graded Ring of Invariants of Binary Octavics. *Am. Jour. Math.*, 89(4):1022–1046, 1967.

[**Sma97**] N. P. Smart. S-unit equations, binary forms and curves of genus 2. *Proc. London Math. Soc. (3)*, 75(2):271–307, 1997.

[**Smi05**] Benjamin A. Smith. *Explicit endomorphisms and correspondences*. PhD thesis, University of Sydney, 2005. URL:http://hdl.handle.net/2123/1066.

[**Sto99**] M. Stoll. On the height constant for curves of genus two. *Acta Arith.*, 90(2):183 – 201, 1999.

[**Sto01**] Michael Stoll. Implementing 2-descent for Jacobians of hyperelliptic curves. *Acta Arith.*, 98(3):245–277, 2001.

[**SV01**] Tony Shaska and Völklein. *Elliptic subfields and automorphisms of genus* 2 *function fields*. Springer, 2001. URL:http://au.arxiv.org/abs/math.AG/0107142.

[**Ver02**] F. Vercauteren. Computing zeta functions of hyperelliptic curves over finite fields of characteristic 2. In *Advances in cryptology—CRYPTO 2002*, volume 2442 of *LNCS*, pages 369–384. Springer, Berlin, 2002.

[**Wam99**] P. Van Wamelen. Examples of Genus Two CM Curves Defined over the Rationals. *Mathematics of Computation*, 68:307–320, 1999.

[**Wam01**] P. Van Wamelen. Computing with the Jacobian of a Genus 2 Curve. URL:http://www.math.lsu.edu/~wamelen/genus2.html, 2001.

[**Wet97**] J. L. Wetherell. *Bounding the number of rational points on certain curves of high rank*. PhD thesis, U.C. Berkeley, 1997.

126 HYPERGEOMETRIC MOTIVES

Chapter 126
HYPERGEOMETRIC MOTIVES

126.1 Introduction

Let $\vec{\alpha}, \vec{\beta} \in \mathbf{C}^n$ be n-tuples (or multisets) of complex numbers. For arithmetic applications we will eventually take them to be rationals, and for purposes of monodromy will largely need only to consider them modulo 1.

Consider the generalised hypergeometric differential equation

$$z(\theta+\alpha_1)\cdots(\theta+\alpha_n)F(z) = (\theta+\beta_1-1)\cdots(\theta+\beta_n-1)F(z), \qquad \theta = z\frac{d}{dz},$$

whose only singularities are regular at 0, 1, and ∞. For simplicity of exposition, we assume that the β's are distinct modulo 1, when a basis of solutions around $z=0$ is given by

$$z^{1-\beta_i}{}_nF_{n-1}\left(\frac{\alpha_1-\beta_i+1,\ldots,\alpha_n-\beta_i+1}{\beta_1-\beta_i+1,..\vee..,\beta_n-\beta_i+1}\;\middle|\; z\right)$$

for $i = 1\ldots n$, and the ith term $\beta_i-\beta_i+1$ is suppressed. The generalised hypergeometric function ${}_nF_{n-1}$ is given by

$${}_nF_{n-1}\left(\frac{a_1,\ldots,a_n}{b_1,\ldots,b_{n-1}}\;\middle|\; z\right) = \sum_{k=0}^{\infty}\frac{(a_1)_k\cdots(a_n)_k}{(b_1)_k\cdots(b_{n-1})_k}\frac{z^k}{k!},$$

where the Pochhammer symbol is given by $(x)_k = (x)(x+1)\cdots(x+k-1)$; the $k!$ in the denominator of the above display can thus be thought of as $(1)_k$, which was the suppressed term. Note that shifting all the α and β by some fixed amount keeps the ${}_nF_{n-1}$ expression the same, while only modifying the $z^{1-\beta_i}$ term. Also, switching α and β can be envisaged in terms of the map $z \to 1/z$ that swaps 0 and ∞.

A theorem of Pochhammer says that the above differential equation has $(n-1)$ independent *holomorphic* solutions around $z=1$. Let G denote the fundamental group of the thrice punctured Riemann sphere, and $V_{\vec{\alpha},\vec{\beta}}$ the solution space around a base point. We have a monodromy representation $M : G \to GL_n(V_{\vec{\alpha},\vec{\beta}})$. Writing $g_0, g_1, g_\infty \in G$ for loops about 0,1,∞, we find that $M(g_0)$ has eigenvalues $e^{-2\pi i\beta_j}$ and $M(g_\infty)$ has eigenvalues $e^{2\pi i\alpha_j}$, implying that we are mainly concerned with $\vec{\alpha}$ and $\vec{\beta}$ only modulo 1. Indeed, one can note that if we take $F = {}_nF_{n-1}(\vec{a},\vec{b}|z)$ and $\vec{x} \in \mathbf{Z}^n, \vec{y} \in \mathbf{Z}^{n-1}$, then generically ${}_nF_{n-1}(\vec{a}+\vec{x},\vec{b}+\vec{y}|z)$ is a linear combination of rational functions times derivatives of F (this is a contiguity relation). Meanwhile, the above fact from Pochhammer implies that $M(g_1)$ must have $(n-1)$ eigenvalues equal to 1 (all with independent eigenvectors), and so this element is a pseudo-reflection.

It turns out that if $H \subseteq GL_n(\mathbf{C})$ is generated by A and B with AB^{-1} a pseudo-reflection, the H-action on $\mathbf{C}^n$ is irreducible if and only if A and B have disjoint sets of eigenvalues. This is equivalent to all the $\alpha_i - \beta_j$ being nonintegral. Moreover, in his 1961 Amsterdam thesis, Levelt showed that, given any eigenvalues, there are (up to conjugacy) unique A and B realising these eigenvalues with AB^{-1} a pseudo-reflection. (Much of the above comes from notes of Beukers.)

For arithmetic purposes, one usually also desires that the eigenvalues be roots of unity and the sets of them be Galois-invariant. Thus we can specify hypergeometric data H by (say) two products of cyclotomic polynomials, these products being coprime and of equal degree. Given such an H, Rodriguez-Villegas conjectures the existence of a family of pure motives (defined over $\mathbf{Q}$), for which the trace of Frobenius at good primes is given by a hypergeometric sum defined by Katz [Kat90] (see also [Kat96]). For each rational $t \neq 0, 1$, there should be a motive H_t whose L-function satisfies a functional equation of a prescribed type, with the Euler factors at good primes given in terms of Gauss sums (the bad Euler factors are less understood, and depend on deformation theory).

One can also relate such motives to more traditional objects in many cases. For instance, there is one hypergeometric datum in degree 1, which can be specified by $\alpha = [\frac{1}{2}]$ and $\beta = [0]$, these being rationals corresponding to the second and first cyclotomic polynomials respectively (by convention, we take β to contain the smallest such rational). The L-function here corresponds to the quadratic field $\mathbf{Q}(\sqrt{t(t-1)})$. In degree 2 there are 13 such data, of which 3 are of weight 0 (see below) and give Artin representations of number fields, while the other 10 are of weight 1, and yield elliptic curves (explicitly calculated by Cohen). An example in higher degree is $\alpha = [\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}]$ and $\beta = [0, 0, 0, 0]$, corresponding respectively to the 5th cyclotomic polynomial and the 4th power of the first cyclotomic polynomial, and this is associated to the Calabi-Yau quintic 3-fold given by

$$x_1^5 + x_2^5 + x_3^5 + x_4^5 + x_5^5 = 5t x_1 x_2 x_3 x_4 x_5.$$

The weight w of a hypergeometric motive can be defined in terms of how much the α and β interlace (considered as roots of unity). In particular, if they are completely interlacing, then the weight is 0, and the resulting motive corresponds to an Artin representation. Write $D(x) = \#\{\alpha : \alpha \leq x\} - \#\{\beta : \beta \leq x\}$. Then $w + 1 = \max_x D(x) - \min_x D(x)$, so that the above 3-fold has weight 3 (from the four β's at 0). This weight controls how large the coefficients of the Euler factors will be.

The trace at q of a hypergeometric motive (for the parameter t) is given in terms of Gauss sums g_q over $\mathbf{F}_q$. Associated to hypergeometric data is a `GammaArray`, and one defines $G_q(r) = \prod_v g_q(-rv)^{\gamma_v}$, and also the `MValue` by $M = \prod_v v^{v\gamma_v}$. For primes p with $v_p(Mt) = 0$ the hypergeometric trace is then given by

$$U_q(t) = \frac{1}{1-q}\left(\sum_{r=0}^{q-2} \omega_p(Mt)^r Q_q(r)\right),$$

where ω_p is the Teichmüller character and $Q_q(r) = (-1)^{m_0} q^{D+m_0-m_r} G_q(r)$ where m_r is the multiplicity of $\frac{r}{q-1}$ in the β and D is a scaling parameter that involves the Hodge

structure (one expression is $m_0 = w + 1 - 2D$). One uses p-adic Γ-functions to expedite the computation of the above Gauss sums. The Euler factor is given by the standard recipe $E_p(T) = \exp\bigl(-\sum_n U_{p^n}(t)T^n/n\bigr)$, and this is a polynomial that satisfies a local functional equation.

126.2 Functionality

126.2.1 Creation Functions

`HypergeometricData(A, B)`

`HypergeometricData(F, G)`

These are two of the principal ways of specifying hypergeometric data. The first takes two sequences A and B (of the same length) of rationals, which must be disjoint upon reduction modulo 1, and each of which must be Galois-invariant when taking the corresponding roots of unity (for instance, if $\frac{1}{6}$ is specified, $\frac{5}{6}$ is also given). The second takes two products F and G of cyclotomic polynomials, these products being coprime and of the same degree. *NOTE*: the routine will always switch A and B (or f and g) so that the latter has the smallest rational in $[0, 1)$ in it.

`HypergeometricData(G)`

This is a third way to specify hypergeometric data, by giving a sequence of integers G such that $\sum_v vG[v] = 0$. Here we have $P_\alpha(T)/P_\beta(T) = \prod_v (T^v - 1)^{G[v]}$, and the polynomials can be determined via Möbius inversion. Again α and β will be reversed if necessary (which corresponds to negating all the entries of G).

`HypergeometricData(G)`

This is a fourth way to specify hypergeometric data, by giving a *list* G of nonzero integers (with repetition possible) corresponding to the sequence G of the previous intrinsic, with negative integers for those where $G[v]$ is negative. The sum of the members of the list must be 0.

`HypergeometricData(F, G)`

This is a fifth way to specify hypergeometric data, by giving two arrays F and G of integers, corresponding to the cyclotomic polynomials to be used.

`HypergeometricData(E)`

This is a utility intrinsic that take a sequence E of two sequences and then passes these two sequences to the intrinsics above.

`Twist(H)`

This intrinsic takes hypergeometric data H, and adds 1/2 to every element in α and β, returning new hypergeometric data. Note that the new α and β may be switched due to this twisting.

`PrimitiveData(H)`

Given hypergeometric data H, return its primitive associated data. This is most easily described in terms of `GammaList`, where one divides all the elements by the gcd.

`PossibleHypergeometricData(d)`

`Weight`	RngIntElt	*Default :* `false`
`TwistMinimal`	BoolElt	*Default :* `false`
`CyclotomicData`	BoolElt	*Default :* `false`
`Primitive`	RngIntElt	*Default :* 0

Given a degree d, generate all possible examples of hypergeometric data of that degree, returned as a sequence of pairs of sequences, each sequence therein having d rationals. If `Weight` is specified, restrict to data of this weight. If `TwistMinimal` is specified, only give twist-minimal data. If `CyclotomicData` is specified, return the sequences of cyclotomic data rather than rationals. If `Primitive` is `true`, only return data that are primitive; if `Primitive` is a positive integer, return the data that have this imprimitivity.

126.2.2 Access Functions

`Weight(H)`

The weight of the given hypergeometric data H.

`Degree(H)`

The degree of the given hypergeometric data H.

`DefiningPolynomials(H)`

The (products of cyclotomic) polynomials corresponding to α and β corresponding to hypergeometric data H.

`CyclotomicData(H)`

This returns two arrays of integers, specifying which cyclotomic polynomials occur for α and β corresponding to hypergeometric data H. Thus, for example, $\Phi_3\Phi_4^2\Phi_6$ would be represented by [3,4,4,6]).

`AlphaBetaData(H)`

This returns two arrays of rationals, giving the α and β of the hypergeometric data H.

`MValue(H)`

This returns the scaling parameter M of the given hypergeometric data H. This is defined by taking $M_n = \prod_{d|n} d^{d\mu(n/d)}$ for the nth cyclotomic polynomial, and combining these into the products for α and β, and then dividing these. Another definition is $M = \prod_v v^{v\gamma_v}$.

`GammaArray(H)`

This returns an array of integers corresponding to γ_v, where these are defined by $P_\alpha(T)/P_\beta(T) = \prod_v (T^v - 1)^{\gamma_v}$. We also have $\sum_v v\gamma_v = 0$.

`GammaList(H)`

This returns a *list* of integers corresponding to γ_v, where $\mathrm{sign}(\gamma_v) \cdot v$ appears in the list $|\gamma_v|$ times.

`H1 eq H2`

`H1 ne H2`

Two instances of hypergeometric data $H1$ and $H2$ are equal if they have the same α and β.

`IsPrimitive(H)`

Returns `true` if the given hypergeometric data H is primitive, and the index of imprimitivity. The latter is the gcd of the elements in the `GammaList`.

126.2.3 Functionality with L-series and Euler Factors

`EulerFactor(H, t, p)`

`Precision`	RngIntElt	*Default :* 0
`Check`	BoolElt	*Default :* `false`
`Fake`	BoolElt	*Default :* `false`

This intrinsic is the heart of the hypergeometric motive package. It takes hypergeometric data H, a rational $t \neq 0, 1$, and a prime p, and computes the pth Euler factor of the hypergeometric motive at t. This uses p-adic Γ-functions, as indicated by Cohen. The `Precision` vararg specifies how many terms in the Euler factor should be computed – if this is 0, then the whole polynomial is computed. The `Check` vararg allows one to turn off the use of the (local) functional equation that is used to expedite the computation process.

The `Fake` vararg allows one to compute the hypergeometric trace(s) for t with $v_p(Mt) = 0$ (including wild primes). Whether or not this is the actual Euler factor depends on how inertia acts. The use of this vararg inhibits the use of the local functional equation, but one can curtail via `Precision`, and apply it manually (if known).

In general, the given prime must not be wild, that is, it must not divide the denominator of any of the α or β.

At other bad primes, the Euler factor will depend upon the relevant monodromy. The primes for which $v_p(t-1) > 0$ can perhaps be called multiplicative, in that p should divide the conductor only once (this is related to the pseudoreflection). Since $v_p(Mt) = 0$ here, the Euler factor (of degree $d-1$) can be recovered by the p-adic Γ-function methods (even when $p = 2$). Also it is often possible to relate the (presumed) hypergeometric motive to objects from a deformation. When $v_p(t-1)$ is even and the weight is also even, the prime p is actually good and has a degree d Euler factor, even though the hypergeometric trace only gives one of degree $(d-1)$. In such a case, the `EulerFactor` intrinsic with the `Fake` vararg will return the part from the hypergeometric trace.

The p with $v_p(1/t) > 0$ correspond to monodromy at ∞. The associated inertia is given by the roots of unity with the β, with maximal Jordan blocks when eigenvalues are repeated. The same is true for p with $v_p(t) > 0$, where the monodromy is around 0 (so that the α are used). In Example H126E7), an instance is given where the inertia is trivialised due to having $\zeta^{v_p(t)} = 1$.

We can compute the Euler factors at such tame primes as follows. Suppose that $t = t_0 p^{vm}$ with $v > 0$, where m occurs as a denominator of the α (similarly with $v < 0$ and β). Then one takes the smallest $q = p^f$ that is 1 mod m, and from the hypergeometric trace formula extracts the terms

$$\omega_p(Mt_0)^{j(q-1)/m} Q_q\left(\frac{j(q-1)}{m}\right)$$

for $0 \le j < m$ with $\gcd(j,m) = 1$. Denoting these by η, we then have that $\prod_\eta(1-\eta T^f)$ is an fth power (due to repetitions in the above extraction), and the fth root of this is the desired Euler factor of degree $\phi(m)$.

When m does not divide $v_p(t)$, the Euler factor from it is trivial. One then multiplies together all such Euler factors corresponding to the m from the α and β. Each m is only considered once, even if it appears multiple times in the `CyclotomicData`, as the Jordan blocks of the eigenvalues are maximal. Note that the local functional equation is not used for tame primes, though the computation should not be too onerous unless $q = p^f$ is large.

`LSeries(H, t)`

`BadPrimes`	SeqEnum	*Default :* []
`GAMMA`	SeqEnum	*Default :* []
`Identify`	BoolElt	*Default :* `true`

Given hypergeometric data H and a rational $t \neq 0, 1$, try to construct the L-series of the associated motive. This will usually need the Euler factors at wild primes to be specified. Everything else, including tame/multiplicative Euler factors and γ-factors, can be computed automatically by Magma (these can also be given via `BadPrimes` and `GAMMA`). The `Identify` vararg indicates whether an attempt should be made to identify motives of weight 0 as Artin representations, and similarly with (hyper)elliptic curves for weight 1.

126.2.3.1 Identification of Hypergeometric Data as Other Objects

`ArtinRepresentation(H, t)`

`Check` BoolElt *Default :* `true`

Given hypergeometric data H of weight 0 and a rational $t \neq 0, 1$, try to determine the associated Artin representation. This is implemented for all such H of degree 3 or less, and for some of degree 4 and higher. The condition needed is that `GammaList(H)` have cardinality 3. When `Check` is true, good primes up to 100 have their Euler factors checked for correctness.

`EllipticCurve(H)`

`EllipticCurve(H, t)`

Given hypergeometric data H of degree 2 and weight 1 (there are 10 such families) and a rational $t \neq 0, 1$, return the associated elliptic curve, as catalogued by Cohen. When t is not given, return the result over a function field.

For each of the 10 families, the same function can be called for the corresponding imprimitive data of index r, and the result will generically be an elliptic curve over an extension of degree r. However, when the $x^r - 1/Mt$ splits, the intrinsic will return an array of elliptic curves corresponding to this splitting.

`HyperellipticCurve(H)`

`HyperellipticCurve(H, t)`

Given hypergeometric data H of degree 4 and weight 1 and a rational $t \neq 0, 1$, return the associated hyperelliptic curve, if this data is known to correspond to such. When t is not given, return the result over a function field. There are 18 cases where one gets a genus 2 curve from the `CanonicalCurve` (making 36 cases when twisting is considered), and a few others where `CanonicalCurve` gives a higher genus curve and there is a genus 2 quotient. In general, one can try to call `IsHyperelliptic` on the `CanonicalCurve`.

`Identify(H, t)`

Given hypergeometric data H and a rational $t \neq 0, 1$, return any known associated object (else returns `false`). The return value can (currently) be: an Artin representation (weight 0); an elliptic curve over $\mathbf{Q}$ (weight 1 in degree 2); an elliptic curve over a number field (weight 1 in degree $2r$ with imprimitivity r), or possibly multiple such curves; or a hyperelliptic curve over $\mathbf{Q}$ (weight 1 in degree 4).

126.2.4 Associated Schemes and Curves

`CanonicalScheme(H)`

`CanonicalScheme(H, t)`

Given hypergeometric data H, this constructs a canonical associated scheme. When the parameter t is given, the specialization is returned, otherwise the result returned will be a scheme over a function field.

The scheme is determined from the `GammaList`, with a variable (X_i or Y_j) for every element in the list. The scheme is the intersection of $\sum_i X_i = \sum_j Y_j = 1$ with

$$\prod_i X_i^{g_i^+} \prod_j Y_j^{g_j^-} = \frac{1}{Mt},$$

where the g_i^+ are the positive elements in the `GammaList` and the g_j^- are the negative ones (one usually moves the latter to the other side of the equation, to make the exponents positive).

`CanonicalCurve(H)`

`CanonicalCurve(H, t)`

Given suitable hypergeometric data H, this tries to construct an associated plane curve. When the parameter t is given, the specialization at t is returned, otherwise the return value will be a plane curve over a function field. The curve is constructed using the `GammaList` (which indicates the Jacobi sums that need to be taken). When this list has four elements, it is always possible to get a curve. When the list has six elements, it is sometimes possible, depending on whether the largest element (in absolute value) is the negation of the sum of two of the other elements. If it is not possible to construct such a curve, the intrinsic returns `false`.

126.2.5 Utility Functions

`HypergeometricMotiveSaveLimit(n)`

`HypergeometricMotiveClearTable()`

These are utility intrinsics that will cache the pre-computation of p-adic Γ-functions. The first indicates to save all computed values when the prime power is less than n, and the second clears the table. The qth table entry will have $(q-1)$ elements in it.

126.3 Examples

Example H126E1

Our first example constructs some hypergeometric motives, and recognises them as being related to elliptic curves or Artin representations.

```
> H := HypergeometricData([1/2],[0]); // weight 0
> t := 3/5;
> A := ArtinRepresentation(H,t);
> D := Discriminant(Integers(Field(A))); // -24
> assert IsSquare(D/(t*(t-1))); // Q(sqrt(t(t-1)))
> R := ArtinRepresentationQuadratic(-24);
> assert A eq R;
> //
> H := HypergeometricData([1/4,3/4],[0,0]);
> Weight(H);
1
> DefiningPolynomials(H);
y^2 + 1, y^2 - 2*y + 1
> t := 3/2;
> E := EllipticCurve(H,t); E;
Elliptic Curve defined by y^2 + x*y = x^3 + 1/96*x over Q
> P := PrimesInInterval(5,100);
> &and[EulerFactor(E,p) eq EulerFactor(H,t,p) : p in P];
true
> //
> f := CyclotomicPolynomial(6)*CyclotomicPolynomial(2);
> g := CyclotomicPolynomial(1)^3;
> H := HypergeometricData(f,g); H;
Hypergeometric data given by [ 1/6, 1/2, 5/6 ] and [ 0, 0, 0 ]
> Weight(H);
2
> GammaList(H);
[* -1, -1, -1, -3, 6 *]
> GammaArray(H);
[ -3, 0, -1, 0, 0, 1 ]
> [EulerFactor(H,4,p) : p in [5,7,11,13,17,19]];
[ 125*y^3 + 20*y^2 + 4*y + 1, 343*y^3 - 42*y^2 - 6*y + 1,
  -1331*y^3 - 22*y^2 + 2*y + 1, -2197*y^3 - 156*y^2 + 12*y + 1,
  4913*y^3 + 323*y^2 + 19*y + 1, 6859*y^3 - 57*y^2 - 3*y + 1 ]
> //
> _<u> := FunctionField(Rationals());
> H := HypergeometricData([-2,0,0,-1,0,1]); H; // weight 1
Hypergeometric data given by [1/6,1/3,2/3,5/6] and [0,0,1/4,3/4]
> HyperellipticCurve(H); // defined over Q(u)
Hyperelliptic Curve defined by y^2 = 4*x^6 - 8*x^5 + 4*x^4 - 64/729/u
> t := 4;
> C := Specialization($1,t); // only works over Q(u)
```

```
> &and[EulerFactor(C,p) eq EulerFactor(H,t,p) : p in P];
true
> //
> H := HypergeometricData([0,-1,0,1,0,1,0,-1]); H; // weight 1
Hypergeometric data given by [1/6,1/3,2/3,5/6] and [1/8,3/8,5/8,7/8]
> MValue(H);
729/4096
> t := 3; // could alternatively specialize later
> E := EllipticCurve(H,t); aInvariants(E);
[ 0, 0, -s, -s, 0] where s^2 is 4096/2187
> &and[EulerFactor(E,p) eq EulerFactor(H,t,p) : p in P];
true
```

Example H126E2

This is a simple example of twisting hypergeometric data, showing that a related Artin motive is obtained for the given weight 0 data.

```
> f := CyclotomicPolynomial(6);
> g := CyclotomicPolynomial(1)*CyclotomicPolynomial(2);
> H := HypergeometricData(f,g); H; assert(Weight(H)) eq 0;
Hypergeometric data given by [ 1/6, 5/6 ] and [ 0, 1/2 ]
> A := ArtinRepresentation(H,-4/5);
> K := OptimisedRepresentation(Field(A));
> DefiningPolynomial(K);
y^6 - 3*y^5 + 3*y^4 - y^3 + 3*y^2 - 3*y + 1
> T := Twist(H); T;
Hypergeometric data given by [ 1/3, 2/3 ] and [ 0, 1/2 ]
> A := ArtinRepresentation(T,-4/5); // can be 1/t sometimes
> L := OptimisedRepresentation(Field(A));
> IsSubfield(L,K), DefiningPolynomial(L);
true Mapping from: L to K,  y^3 + 3*y - 1
```

The same can be said for twisting for (hyper)elliptic curves.

```
> H := HypergeometricData([2,2],[3]); // Phi_2^2 and Phi_3
> E := EllipticCurve(H,3);
> T := EllipticCurve(Twist(H),1/3); // 1/t here
> IsQuadraticTwist(E,T);
true -4/9
> //
> H := HypergeometricData([5],[8]); // Phi_5 and Phi_8
> C := HyperellipticCurve(H);
> t := 7;
> S := Specialization(C,t);
> T := HyperellipticCurve(Twist(H),1/t);
> Q := QuadraticTwist(T,5*t); // get right parameter
> assert IsIsomorphic(Q,S);
```

Example H126E3

This example exercises the primitivity functionality.

```
> H := HypergeometricData([3],[4]); // Phi_3 and Phi_4
> GammaList(H);
[* -1, 2, 3, -4 *]
> H2 := HypergeometricData([* -2, 4, 6, -8 *]);
> IsPrimitive(H2);
false 2
> PrimitiveData(H2) eq H;
true
> H3 := HypergeometricData([* -3, 6, 9, -12 *]);
> IsPrimitive(H3);
false 3
> PrimitiveData(H3) eq H;
true
> aInvariants(EllipticCurve(H));
[ 0, 0, -64/27/u, -64/27/u, 0 ] where u is FunctionField(Q).1
> aInvariants(EllipticCurve(H2));
[ 0, 0, -s, -s, 0 ] where s^2=(-64/27)^2/u
> aInvariants(EllipticCurve(H3));
[ 0, 0, -s, -s, 0 ] where s^3=(-64/27)^3/u
```

Example H126E4

Here is an example with the canonical schemes and curves associated to various hypergeometric data.

```
> _<u> := FunctionField(Rationals());
> H := HypergeometricData([* -2, 3, 4, -5 *]); // degree 4
> C := CanonicalScheme(H);
> _<[X]> := Ambient(C); C;
Scheme over Univariate rational function field over Q defined by
X[1] + X[2] - 1, X[3] + X[4] - 1,
X[1]^2*X[2]^5 - 3125/1728/u*X[3]^3*X[4]^4
> Dimension(C), Genus(Curve(C)); // genus 2 curve
1 2
> assert IsHyperelliptic(Curve(C));
> CC := CanonicalCurve(H);
> _<x,y> := Ambient(CC); CC;
Curve over Univariate rational function field over Q defined by
x^7 - 2*x^6 + x^5 + 3125/1728/u*y^7 - 3125/576/u*y^6 +
 3125/576/u*y^5 - 3125/1728/u*y^4
> b, C2 := IsHyperelliptic(CC); assert b;
> HyperellipticCurve(H); // in the degree 4 catalogue
Hyperelliptic Curve defined over Univariate function field
 over Q by y^2 = 4*x^5 - 3125/432/u*x^3 + 9765625/2985984/u^2
> assert IsIsomorphic(HyperellipticCurve(H),C2);
```

```
> // and an example where the curve is reducible
> H := HypergeometricData([* 6,6,-8,-4 *]); // weight 1
> C := CanonicalCurve(H);
> A := AlgorithmicFunctionField(FunctionField(C));
> E<s> := ExactConstantField(A);
> CE := BaseChange(C,E);
> I := IrreducibleComponents(CE); assert #I eq 2;
> _<x,y> := Ambient(I[1]); I[1];
Scheme over E defined by [ where s^2 = 1048576/531441/u ]
 x^6 - 2*x^5 + x^4 - s*y^6 + 3*s*y^5 - 3*s*y^4 + s*y^3
> b, C2 := IsHyperelliptic(Curve(I[1])); assert b;
```

Example H126E5

Here is an example in degree 4 and weight 3. It turns out that the motive from $t = -1$ has complex multiplication, and the L-series appears to be the same as that of a Siegel modular form given by [vGvS93, S8.7]. (This was found by Cohen and Rodriguez-Villegas). This L-series also appears in Example H127E28.

```
> H := HypergeometricData([1/2,1/2,1/2,1/2],[0,0,0,0]);
> L := LSeries(H,-1 : BadPrimes:=[<2,9,1>]); // guessed
> CheckFunctionalEquation(L);
-5.916456789157588540587964239 62E-31
> LGetCoefficients(L,100);
[* 1, 0, 0, 0, -4, 0, 0, 0, -6, 0, 0, 0, -84, 0, 0, 0, 36, 0, 0, 0, 0,
   0, 0, 0, 146, 0, 0, 0, 140, 0, 0, 0, 0, 0, 0, 0, 60, 0, 0, 0, -140,
   0, 0, 0, 24, 0, 0, 0, -238, 0, 0, 0, 924, 0, 0, 0, 0, 0, 0, 0, -820,
   0, 0, 0, 336, 0, 0, 0, 0, 0, 0, 0, -396, 0, 0, 0, 0, 0, 0, 0, -693,
   0, 0, 0, -144, 0, 0, 0, -300, 0, 0, 0, 0, 0, 0, 0, -252, 0, 0, 0 *]
> // compare to the Tensor product way of getting this example
> E := EllipticCurve("32a");
> NF := Newforms(ModularForms(DirichletGroup(32).1,3)); // wt 3 w/char
> L1 := LSeries(E); L2 := LSeries(ComplexEmbeddings(NF[1][1])[1][1]);
> TP := TensorProduct(L1, L2, [ <2, 9> ]); // conductor 2^9 (guessed)
> [Round(Real(x)) : x in LGetCoefficients(TP,100)];
[ 1, 0, 0, 0, -4, 0, 0, 0, -6, 0, 0, 0, -84, 0, 0, 0, 36, 0, 0, 0, 0,
  0, 0, 0, 146, 0, 0, 0, 140, 0, 0, 0, 0, 0, 0, 0, 60, 0, 0, 0, -140,
  0, 0, 0, 24, 0, 0, 0, -238, 0, 0, 0, 924, 0, 0, 0, 0, 0, 0, 0, -820,
  0, 0, 0, 336, 0, 0, 0, 0, 0, 0, 0, -396, 0, 0, 0, 0, 0, 0, 0, -693,
  0, 0, 0, -144, 0, 0, 0, -300, 0, 0, 0, 0, 0, 0, 0, -252, 0, 0, 0 ]
```

Example H126E6

Here is an example showing how to handle bad primes in some cases. The Euler factors at $\{3, 5, 17\}$ [where $p|(t-1)$] were determined via a recipe from deformation theory by Rodriguez-Villegas, while at $p = 2$, Roberts suggested a t-value that would trivialise the conductor (from a number field analogy), and Tornaría then computed the full degree 4 factor (at $p = 2$) for $t = 2^8$.

```
> H := HypergeometricData([1/2,1/2,1/2,1/2],[0,0,0,0]);
> Lf := LSeries(Newforms(ModularForms(8,4))[1][1]);
> T := PolynomialRing(Integers()).1; // dummy variable
> f3 := EulerFactor(Lf,3 : Integral)*(1-3*T); // make it a poly
> f5 := EulerFactor(Lf,5 : Integral)*(1-5*T); // via Integral
> f17 := EulerFactor(Lf,17 : Integral)*(1-17*T);
> f2 := 1+T+6*T^2+8*T^3+64*T^4; // determined by Tornaria
> BP := [<2,0,f2>,<3,1,f3>,<5,1,f5>,<17,1,f17>];
> L := LSeries(H,256 : BadPrimes:=BP);
> Conductor(L);
255
> assert Abs(CheckFunctionalEquation(L)) lt 10^(-28);
```

One need not specify all the bad prime information as in the above example. Here is a variation on it, with $t = 1/2^8$ (note that this actually gives the same L-series, as the data is a self-twist, with the character induced by twisting being trivial for this choice of t). Note that only the information at 2 is given to `LSeries`.

```
> H := HypergeometricData([1/2,1/2,1/2,1/2],[0,0,0,0]);
> MValue(H);
256
> t := 1/2^8; // makes v_2(Mt)=0
> f2 := EulerFactor(H,t,2 : Fake);
> f2;
64*T^4 + 8*T^3 + 6*T^2 + T + 1
> L := LSeries(H,t : BadPrimes:=[<2,0,f2>]);
> Conductor(L);
255
> assert Abs(CheckFunctionalEquation(L)) lt 10^(-28);
```

Example H126E7

Here is an example with the quintic 3-fold. The deformation theory at $p = 11$ here is related to the Grössencharacter example over $\mathbf{Q}(\zeta_5)$ given in Example H34E24. The action on inertia at $p = 11$ involves ζ_5 when $11|t$, and here it is raised to the 5th power, thus trivialising it. As with previous example, the deformation theory also involves a weight 4 modular form, here of level 25.

```
> f := CyclotomicPolynomial(5); g := CyclotomicPolynomial(1)^4;
> H := HypergeometricData(f,g); H, Weight(H); // weight 3
Hypergeometric data given by [1/5,2/5,3/5,4/5] and [0,0,0,0]
3
> t := 11^5; // 11 is now good, as is raised to 5th power
> T := PolynomialRing(Rationals()).1;
> f2 := (1-T+8*T^2)*(1+2*T); // could have Magma compute these
> f3221 := (1-76362*T+3221^3*T^2)*(1-3221*T); // wt 4 lev 25
> // degree 4 factor at 11 comes from Grossencharacter
> // in fact, this is the t=0 deformation: sum_i x_i^5 = 0
> K<z5> := CyclotomicField(5);
> p5 := Factorization(5*Integers(K))[1][1]; // ramified
```

```
> G := HeckeCharacterGroup(p5^2);
> psi := Grossencharacter(G.0,[[3,0],[1,2]]);
> f11 := EulerFactor(LSeries(psi),11 : Integral); f11;
1771561*x^4 - 118459*x^3 + 3861*x^2 - 89*x + 1
> BP := [<2,1,f2>,<5,4,1>,<11,0,f11>,<3221,1,f3221>];
> L := LSeries(H,t : BadPrimes:=BP);
> Conductor(L); // 2*5^4*3221, 5^4 is somewhat guessed
4026250
> LSetPrecision(L,5); LCfRequired(L);
12775
> CheckFunctionalEquation(L); // takes about 40s
-3.8147E-6
```

Again one need not specify all the bad prime information, as Magma can automatically compute it at multiplicative and tame primes (however, the local conductor at 5 must be specified).

```
> EulerFactor(H,t,11); // tame
1771561*T^4 - 118459*T^3 + 3861*T^2 - 89*T + 1
> EulerFactor(H,t,2); // multiplicative
16*T^3 + 6*T^2 + T + 1
> EulerFactor(H,t,3221); // multiplicative
-107637325775281*T^3 + 33663324863*T^2 - 79583*T + 1
```

One can also choose t so as to trivialise the wild prime 5.

```
> MValue(H); // 5^5;
3125
> t := 11^5/5^5;
> f5 := EulerFactor(H,t,5 : Fake); // v_5(Mt)=0
> f5;
15625*T^4 - 125*T^3 - 45*T^2 - T + 1
> L := LSeries(H,t : BadPrimes:=[<5,0,f5>]);
> Conductor(L); // 2*3*26321, Magma computes Euler factors
157926
> LSetPrecision(L,9); // about 4000 terms
> CheckFunctionalEquation(L);
-2.32830644E-10
> t := -11^5/5^5; // another choice with v_5(Mt)=0
> f5 := EulerFactor(H,t,5 : Fake); // v_5(Mt)=0
> f5; // four possible Euler factors, one for each Mt mod 5
15625*T^4 + 1750*T^3 + 230*T^2 + 14*T + 1
> L := LSeries(H,t : BadPrimes:=[<5,0,f5>]);
> Conductor(L); // 2*31*331, Magma computes Euler factors
20552
> LSetPrecision(L,9); // about 1300 terms
> CheckFunctionalEquation(L);
4.65661287E-10
```

Example H126E8

Here is an example with tame primes. This derives from comments of Rodriguez-Villegas. The idea is to take hypergeometric data that has weight 0 or 1, and compare it to Artin representations or hyperelliptic curves.

```
> T := PolynomialRing(Rationals()).1; // dummy variable
> H := HypergeometricData([3,4,6,12],[1,1,5,5]); // degree 10
> b, HC := IsHyperelliptic(CanonicalCurve(H)); // genus 5
> assert b; Genus(HC);
5
> EulerFactor(Specialization(HC,13^12),13); // 13 becomes good
371293*T^10 - 285610*T^9 + 125229*T^8 - 31096*T^7 + 4810*T^6
 - 540*T^5 + 370*T^4 - 184*T^3 + 57*T^2 - 10*T + 1
> EulerFactor(H,13^12,13); // use hypergeometric methods
371293*T^10 - 285610*T^9 + 125229*T^8 - 31096*T^7 + 4810*T^6
 - 540*T^5 + 370*T^4 - 184*T^3 + 57*T^2 - 10*T + 1
> assert &and[EulerFactor(Specialization(HC,p^12),p)
>             eq EulerFactor(H,p^12,p) : p in [11,13,17,19]];
> assert &and[EulerFactor(Specialization(HC,t0*13^12),13)
>             eq EulerFactor(H,t0*13^12,13) : t0 in [1..12]];
```

One can take a smaller power than the 12th, but then the curve will not become completely good at the prime. However, the hypergeometric calculations will still be possible.

```
> EulerFactor(H,17^4,17);
17*T^2 + 2*T + 1
> EulerFactor(H,19^9,19); // takes the Phi_3 term
19*T^2 + 7*T + 1
> EulerFactor(H,19^6,19);
361*T^4 + 114*T^3 + 31*T^2 + 6*T + 1
> EulerFactor(H,1/11^5,11); // degree is phi(1)+phi(5)
-T^5 + 1
> EulerFactor(H,4/11^5,11); // degree is phi(1)+phi(5)
-T^5 + 5*T^4 - 10*T^3 + 10*T^2 - 5*T + 1
```

A similar exploration is possible with a weight 0 example. Here the Artin representation machinery is better able to cope with partially good primes.

```
> H := HypergeometricData([2,3,6],[1,5]); // degree 5
> Q := Rationals();
> EulerFactor(ArtinRepresentation(H,7^6),7 : R:=Q);
-T^5 - T^4 - T^3 + T^2 + T + 1
> EulerFactor(ArtinRepresentation(H,7^3),7 : R:=Q);
T^2 + T + 1
> EulerFactor(ArtinRepresentation(H,7^2),7 : R:=Q);
-T + 1
> EulerFactor(ArtinRepresentation(H,2/11^5),11 : R:=Q);
-T^5 + 5*T^4 - 10*T^3 + 10*T^2 - 5*T + 1
> EulerFactor(H,7^6,7); // compute it directly from H
-T^5 - T^4 - T^3 + T^2 + T + 1
```

```
> EulerFactor(H,7^3,7);
T^2 + T + 1
> EulerFactor(H,7^2,7);
-T + 1
> EulerFactor(H,2/11^5,11);
-T^5 + 5*T^4 - 10*T^3 + 10*T^2 - 5*T + 1
```

Example H126E9

Here is an example of the use of `PossibleHypergeometricData`, enumerating the number of possibilities in small degree. A speed test is also done for the save-limit code.

```
> for d in [1..8] do
>    [#PossibleHypergeometricData(d : Weight:=w) : w in [0..d-1]];
>    end for;
[ 1 ]
[ 3, 10 ]
[ 3, 0, 10 ]
[ 11, 74, 30, 47 ]
[ 7, 0, 93, 0, 47 ]
[ 23, 287, 234, 487, 84, 142 ]
[ 21, 0, 426, 0, 414, 0, 142 ]
[ 51, 1001, 1234, 3247, 894, 1450, 204, 363 ]
> D4w1 := PossibleHypergeometricData(4 : Weight:=1);
> D := [HypergeometricData(x) : x in D4w1];
> #[x : x in D | Twist(x) eq x]; // 12 are self-twists
12
> #PossibleHypergeometricData(4 : Weight:=1,TwistMinimal);
43
> #PossibleHypergeometricData(4 : Weight:=1,Primitive);
64
> // speed test for SaveLimit
> H := HypergeometricData([1/2,1/2,1/2,1/2],[0,0,0,0]);
> HypergeometricMotiveSaveLimit(2000);
> time _:=LGetCoefficients(LSeries(H,-1),2000);
Time: 1.040
> time _:=LGetCoefficients(LSeries(H,-1),2000);
Time: 0.540
> HypergeometricMotiveClearTable();
> time _:=LGetCoefficients(LSeries(H,-1),2000);
Time: 1.030
```

126.4 Bibliography

[**Kat90**] N. M. Katz. *Exponential Sums and Differential Equations*, volume 124. Annals of Math. Studies., 1990.

[**Kat96**] N. M. Katz. *Rigid Local Systems*, volume 139. Annals of Math. Studies., 1996.

[**vGvS93**] B. van Geemen and D. van Straten. The cusp forms of weight 3 on $\Gamma_2(2,4,8)$. *Math. Comp.*, 61(204):849–872, 1993.

127 L-FUNCTIONS

Chapter 127
L-FUNCTIONS

127.1 Overview

A large variety of ζ-functions and L-functions occur in number theory and algebraic geometry. Some well-known L-functions include the Riemann ζ-function, the Dedekind ζ-function of a number field, Dirichlet series associated to characters, and L-series of curves (e.g., elliptic curves) over the rationals. MAGMA provides functionality for constructing such L-functions and computing their values in the complex plane. A typical calculation might go as follows:

```
> L := LSeries(EllipticCurve([0, -1, 1, 0, 0]));
> Evaluate(L,2);
0.546048036215013518334126660433
```

The first line defines an L-series $L(E, s)$ of the elliptic curve

$$E : y^2 + xy = x^3 - x^2$$

while the second line computes its value at $s = 2$. An impatient reader may wish simply to type `LSeries;` at the prompt and look at the various `LSeries` signatures and mimic the code above, thus getting access to much of the functionality.

Topics covered in this chapter include:

- The built-in L-series which include the Riemann ζ-function, the Dedekind ζ-function of a number field, Dirichlet series associated to characters, Artin representations, modular forms, and L-series of elliptic curves;
- The calculation of values, derivatives and Taylor expansions of L-series at a complex point s_0 to desired accuracy;
- A technical description of the L-series object in MAGMA, together with a description of how to construct user-defined L-series with any number of gamma factors, provided that the L-series satisfies a functional equation of the standard type;
- Operations such as division, multiplication and the tensor product of two L-series.

The reader is referred to Manin-Panchishkin [Sha95] Chapter 4, Serre [Ser65] and articles in [JKS94] for a background on L-functions. The algorithms mostly follow Dokchitser [Dok04] and the Pari implementation ComputeL [Dok02]. See also Lavrik [Lav67], Tollis [Tol97] and the exposition in Cohen [Coh00], 10.3.

127.2 Built-in L-series

An *L-series* or an *L-function* is an infinite sum $L(s) = \sum_{n=1}^{\infty} a_n/n^s$ in the complex variable s with complex coefficients a_n. Such functions arise in many places in mathematics and they are usually naturally associated with some kind of mathematical object, for instance a character, a number field, a curve, a modular form or a cohomology group of an algebraic variety. The coefficients a_n are certain invariants associated with that object. For example, in the case of a character $\chi : (\mathbf{Z}/m\mathbf{Z})^* \to \mathbf{C}^*$ they are simply its values $a_n = \chi(n)$ when $\gcd(n, m) = 1$ and 0 otherwise.

MAGMA is able to associate an L-series to various types of object. The intrinsic which provides access to such pre-defined L-series (apart from the Riemann zeta function that is not quite associated with anything) is

`LSeries(`*object: optional parameters*`)`

Every such function returns a variable of type `LSer`. A range of functions may now be applied to this L-series object as described in the following sections, and these are independent of the object to which the L-series was originally associated. In fact, an object of type `LSer` only "remembers" its origin for printing purposes.

`RiemannZeta()`

`Precision`	RNGINTELT	*Default :*

The Riemann zeta function $\zeta(s)$ is returned.

The number of digits of precision to which the values $\zeta(s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted, the precision of the default real field will be used.

Example H127E1

Check that $\zeta(2)$ agrees numerically with $\pi^2/6$.

```
> L := RiemannZeta( : Precision:=40);
> Evaluate(L,2);
1.6449340668482264364724151666460251892 19
> Pi(RealField(40))^2/6;
1.6449340668482264364724151666460251892 19
```

`LSeries(K)`

`Method`	MONSTGELT	*Default :* "*Default*"
`ClassNumberFormula`	BOOLELT	*Default :* `false`
`Precision`	RNGINTELT	*Default :*

Create the Dedekind zeta function $\zeta(K, s)$ of a number field K. The series is defined by $\sum_I \mathrm{Norm}_{K/\mathbf{Q}}(I)^{-s}$, where the sum is taken over the non-zero ideals I of the maximal order of K. For $K = \mathbf{Q}$, the series coincides with the Riemann zeta function.

The optional parameter `Method` may be `"Artin"`, `"Direct"` or `"Default"` and specifies whether the zeta function should be computed as a product of L-series of Artin representations or directly, by counting prime ideals. (The default behaviour depends upon the field.)

For the `"Direct"` method, the Dedekind zeta function has a simple pole at $s = 1$ whose residue must be known in order to compute the L-values. The class number formula gives an expression for this residue in terms of the number of real/complex embeddings of K, the regulator, the class number and the number of roots of unity in K. If the optional parameter `ClassNumberFormula` is set to `true`, then these quantities are computed on initialization (using MAGMA's functions `Signature(K), Regulator(K), #ClassGroup(MaximalOrder(K))` and `#TorsionSubgroup(UnitGroup(K))`) and it might take some time if the discriminant of K is large. If `ClassNumberFormula` is `false` (default) then the residue is computed numerically from the functional equation. This is generally faster, unless the discriminant of K is small and the precision is set to be very high.

The number of digits of precision to which the values $\zeta(K, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

Example H127E2

This code computes the value of $\zeta(\mathbf{Q}(i), s)$ at $s = 2$.

```
> P<x> := PolynomialRing(Integers());
> K := NumberField(x^2+1);
> L := LSeries(K);
> Evaluate(L, 2);
1.5067030099229850308865650 4818
```

This particular example could have been expressed somewhat more succinctly by replacing the first two lines by either `K:=CyclotomicField(4)` or `K:=QuadraticField(-1)`.

Example H127E3

The code computes $\zeta(F, 2)$ for $F = \mathbf{Q}(\sqrt[12]{3})$.

```
> R<x> := PolynomialRing(Rationals());
> F := NumberField(x^12-3);
> L := LSeries(F: Method:="Direct");
Time: 0.078
> Conductor(L), LCfRequired(L);
1579460446107205632 92968955438
```

The set-up time for the direct method is negligible, but the L-value computation will take days for this number of coefficients. On the other hand, the normal closure of F is not too large and has only representations of small dimension:

```
> G := GaloisGroup(F);
> #G, [Degree(ch): ch in CharacterTable(G)];
```

```
24 [ 1, 1, 1, 1, 2, 2, 2, 2, 2 ]
> time L := LSeries(F : Method:="Artin");
Time: 5.234
```

It took longer to define the L-series, but the advantage is that it is a product of L-series with very small conductors, and the L-value calculations are almost instant:

```
> [Conductor(f[1]) : f in Factorisation(L)];
[ 1, 12, 3888, 576, 243, 15552, 15552 ]
> time Evaluate(L, 2);
1.63925427193646882835990708818
Time: 3.250
```

Example H127E4

This code follows an example of Serre and Armitage (see [Ser71, Arm71, Fri76]) where the ζ-function of a field vanishes at the central point.

```
> _<x> := PolynomialRing(Rationals());
> K<s5> := NumberField( x^2-5 );
> L<s205> := NumberField( x^2-205 );
> C := Compositum(K,L);
> e1 := C!(5+s5);
> e2 := C!(41+s205);
> E:=ext<C | Polynomial( [ -e1*e2, 0, 1] )>;
> A:=AbsoluteField(E);
> DefiningPolynomial(A);
x^8 - 820*x^6 + 223040*x^4 - 24206400*x^2 + 871430400
> Signature(A); // totally real
8 0
> L := LSeries(A);
> LCfRequired(L);
2739
> CheckFunctionalEquation(L);
1.57772181044202361082345713057E-30
> Evaluate(L, 1/2); // zero as expected
-9.98707556173617338749102627597E-62
```

So the evaluation of L at $1/2$ is zero as expected. In fact, L is a product, and one factor has odd sign:

```
> L`prod;
[ <L-series of Riemann zeta function, 1>,
  <L-series of Artin representation of Number Field A with
            character ( 1, 1, 1, -1, -1 ) and conductor 41, 1>,
  <L-series of Artin representation of Number Field A with
            character ( 1, 1, -1, -1, 1 ) and conductor 5, 1>,
  <L-series of Artin representation of Number Field A with
            character ( 1, 1, -1, 1, -1 ) and conductor 205, 1>,
  <L-series of Artin representation of Number Field A with
```

```
            character ( 2, -2, 0, 0, 0 ) and conductor 42025, 2> ]
> [ ComplexField(9)!Sign(x[1]) : x in $1 ];
[ 1.00000000, 1.00000000, 1.00000000, 1.00000000, -1.00000000 ]
> Sign(L`prod[5][1]);
-1.0000000000000000000000000000
```

`LSeries(A)`

`Precision` RngIntElt *Default :*

Creates the L-series of an Artin representation A. (For information about Artin representations see Chapter 44.)

Example H127E5

L-series of the two characters of $\mathrm{Gal}(\mathbf{Q}(i)/\mathbf{Q}) \cong C_2$.

```
> K := QuadraticField(-1);
> triv,sign := Explode(ArtinRepresentations(K));
> Evaluate(LSeries(triv), 2);    // zeta(2)=pi^2/6
1.6449340668482264364724151666 5
> Evaluate(LSeries(sign), 2);
0.9159655941772190150546035149 33
```

Example H127E6

We take a polynomial over $\mathbf{Q}$ with Galois group A_7 and the unique 6-dimensional irreducible Artin representation a of this group

```
> load galpols;
> f:=PolynomialWithGaloisGroup(7,6);
> K:=NumberField(f);
> A:=ArtinRepresentations(K);
> a:=A[2];a;
Artin representation of Number Field with defining polynomial
x^7 - 2*x^6 - 7*x^5 + 11*x^4 + 16*x^3 - 14*x^2 - 11*x + 2 over
the Rational Field with character ( 6, 2, 3, 0, 0, 1, -1, -1, -1 )
and conductor 988410721
```

Its L-series with the default precision of 30 digits needs a lot of coefficients to compute with

```
> L:=LSeries(a);
> LCfRequired(L);
1830809
```

We decrease the precision to 8 digits

```
> L:=LSeries(a: Precision:=8);
> LCfRequired(L);
```

```
52584
```

Now it takes under 30 seconds to verify the functional equation of $L(a, s)$ and to compute its value at $s = 2$.

```
> CheckFunctionalEquation(L);
-2.9802322E-7
> Evaluate(L,2);
1.1105970
```

`LSeries(E)`

`Precision` RngIntElt *Default :*

Create the L-series $L(E, s)$ of an elliptic curve E defined over $\mathbf{Q}$ or over a number field.

The number of digits of precision to which the values $L(E, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

Note that the computation time for evaluating an L-series grows roughly like the square root of the conductor (or its norm to $\mathbf{Q}$ if the base field is a number field). Therefore an evaluation might take an unreasonable amount of time if the conductor of E is much larger than, say, 10^{10} or so. If only the leading term at $s = 1$ is required, over $\mathbf{Q}$ it is faster to use `AnalyticRank` or `ConjecturalRegulator`.

Note also for general number fields it is only conjectured that $L(E/K, s)$ has a meromorphic continuation to $\mathbf{C}$ and also possesses a functional equation. This conjecture is implicitly used in the computations.

Example H127E7

Consider the curve $E : y^2 + y = x^3 + x^2$ over $\mathbf{Q}$ of conductor 43. It has Mordell-Weil rank equal to 1, so we expect $L(E/\mathbf{Q}, 1) = 0$ and $L'(E/\mathbf{Q}, 1) \neq 0$ by the Birch-Swinnerton-Dyer conjecture.

```
> E := EllipticCurve([0,1,1,0,0]);
> Conductor(E);
43
> L:=LSeries(E);
> Evaluate(L, 1);
0.00000000000000000000000000000000
> Evaluate(L, 1 : Derivative:=1);
0.34352397461847823061807116392 2
```

Now base change E to $K = \mathbf{Q}(i)$. The Mordell-Weil rank of E over K is 2:

```
> Rank(E) + Rank(QuadraticTwist(E,-1));
2
```

So we expect $L(E/K, s)$ to have a zero of order 2:

```
> K := QuadraticField(-1);
```

```
> EK := BaseChange(E, K);
> L := LSeries(EK);
> Evaluate(L, 1);
0.00000000000000000000000000000000
> Evaluate(L, 1 : Derivative:=1) lt 10^-20;
true
> Evaluate(L, 1 : Derivative:=2);
1.623995450256000307225469103 42
```

`LSeries(E, K)`

`Method`	MonStgElt	*Default :* "*Default*"
`Precision`	RngIntElt	*Default :*

Given an elliptic curve E defined over the rationals and a number field K, create the L-series $L(E/K, s)$ associated with E/K. Note that in general it is only conjectured that $L(E/K, s)$ has an analytic continuation to $\mathbf{C}$ and possesses a functional equation. This conjecture is implicitly used in the computations.

Technically, the resulting L-series is the tensor product of two l-adic representations, the one associated to $E/\mathbf{Q}$ and the one associated to $K/\mathbf{Q}$. `Method` specifies how `LSeries(K)` should be defined. It is the same parameter as for `LSeries(FldNum)`, except that when E and K both have wild ramification at either 2 or 3, `"Direct"` is always used. Note that the conductor of the L-series $L(E/K, s)$ usually increases very rapidly with the discriminant of K. Consequently, if the used method is `"Direct"` or the irreducible constituents of `PermutationCharacter(K)` have large dimension, the computation time may be quite substantial.

The number of digits of precision to which the values $L(E/K, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

Example H127E8

We take the curve $E : y^2 = x^3 + x$ over the rationals and apply base change to obtain a curve over $\mathbf{Q}(\sqrt{5})$. The resulting L-series $L(E, K, s)$ is in fact the product of $L(E/\mathbf{Q}, s)$ and $L(F/\mathbf{Q}, s)$ where F is E twisted by 5.

```
> E := EllipticCurve([0, 0, 0, 1, 0]);
> F := QuadraticTwist(E, 5);
> L := LSeries(E, QuadraticField(5));
> Evaluate(L, 1);
1.5373382847036052245896606919 5
> Evaluate(LSeries(E),1) * Evaluate(LSeries(F),1);
1.5373382847036052245896606919 5
```

Here is another example over a cyclotomic field. The L-value computations are fast, as here they only involve one-dimensional twists:

```
> E := EllipticCurve([0, 0, 0, 1, 0]);
```

```
> L := LSeries(E, CyclotomicField(11));
> time Evaluate(L, 1);
-1.03578452039312258255988860081E-28
Time: 4.797
```

`LSeries(E, A)`

Precision RngIntElt *Default :*

Twisted L-series of an elliptic curve $E/\mathbf{Q}$ by an Artin representation A. Currently does not allow E and A to be simultaneously wildly ramified at either 2 or 3.

Example H127E9

We take the elliptic curve 11A3 and twist it by the characters of $\mathbf{Q}(\zeta_5)/\mathbf{Q}$:

```
> E := EllipticCurve(CremonaDatabase(),"11A3");
> K := CyclotomicField(5);
> art := ArtinRepresentations(K);
> for A in art do Evaluate(LSeries(E,A),1); end for;
0.25384186085591068433775892335 1
0.68597671458851643816988951422 3 + 1.10993363969520543571381847366*$.1
2.83803828204429619496466743334 2
0.68597671458851643816988951422 3 - 1.10993363969520543571381847366*$.1
```

All the L-values are non-zero, so according to the Birch-Swinnerton-Dyer conjecture E has rank 0 over $\mathbf{Q}(\zeta_5)$. Indeed:

```
> #TwoSelmerGroup(BaseChange(E,K));
1
```

Example H127E10

As a higher-dimensional example, we twist $E = X_1(11)/\mathbf{Q}$ by a 2-dimensional Artin representation that factors through a quaternion Galois group.

```
> load galpols;
> E:=EllipticCurve("11a3");           // X_1(11)
> f:=PolynomialWithGaloisGroup(8,5);  // Quaternion Galois group
> K:=NumberField(f);
> A:=ArtinRepresentations(K);
> assert exists(a){a: a in A | Degree(a) eq 2};a;
Artin representation of Number Field with defining polynomial
x^8 - 12*x^6 + 36*x^4 - 36*x^2 + 9 over the Rational Field with
character ( 2, -2, 0, 0, 0 )
> L:=LSeries(E,a: Precision:=10);
> LCfRequired(L);
208818
> time Evaluate(L,1);
1.678012769
```

```
Time: 23.688
> Sign(L);
1.000000002
```

LSeries(C)

Precision	RngIntElt	*Default :*
ExcFactors	SeqEnum	*Default :* []

L-series of a hyperelliptic curve C defined over the rationals.

The number of digits of precision to which the values $L(C, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field. If the conductor exponents and the local factors at (some of) the bad primes are known in advance, they can be passed as a sequence of tuples `<prime,conductor exponent,local factor>`, e.g. ExcFactors:=$[\langle 2, 11, 1 - x\rangle]$.

Example H127E11

We take the hyperelliptic curve $y^2 = x^5 + 1$

```
> R<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^5+1);
> L := LSeries(C: Precision:=18);
> LCfRequired(L);   // need this number of coefficients
1809
> Evaluate(L,1);  // L(C,1)
1.03140710417331775
> Sign(L); // sign in the functional equation
1.00000000000000000
```

The L-value is non-zero, indicating that the Jacobian should have rank 0. In fact, it does:

```
> RankBound(Jacobian(C));
0
```

LSeries(Chi)

Precision	RngIntElt	*Default :*

Given a primitive dirichlet character $\chi : (\mathbf{Z}/m\mathbf{Z})^* \to \mathbf{C}^*$, create the associated Dirichlet L-series $L(\chi, s) = \sum_{n=1}^{\infty} \chi(n)/n^s$. The character χ must be defined so that its values fall in either the ring of integers, the rational field or a cyclotomic field.

The number of digits of precision to which the values $L(\chi, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

For information on Dirichlet characters, see Section 19.8.

Example H127E12

We define a primitive character $\chi : (\mathbf{Z}/37\mathbf{Z})^* \to \mathbf{C}^*$ and construct the associated Dirichlet L-function.

```
> G<Chi> := DirichletGroup(37, CyclotomicField(36));
> L := LSeries(Chi);
> Evaluate(L,1); // depends on the chosen generator of G
1.65325576836885655776002342451 - 0.55160789892291080587553771593 4*$.1
```

`LSeries(hmf)`

Given a cuspidal newform in a space of Hilbert modular forms, this creates the associated L-series.

Example H127E13

```
> K := QuadraticField(5);
> H := HilbertCuspForms(K,7*Integers(K),[2,2]);
> f := NewformDecomposition(NewSubspace(H))[1];
> L := LSeries(Eigenform(f));
> LSetPrecision(L,9);
> LCfRequired(L);
198
> time CheckFunctionalEquation(L);
-2.32830644E-10
Time: 16.590
```

`LSeries(psi)`

`LSeries(psi)`

`Precision` RngIntElt *Default :*

Given a primitive Hecke (Grössen)character on ideals, construct the associated L-series.

For more information on these see Section 34.9.

`LSeries(f)`

`Embedding`	MAP/USERPROGRAM	*Default :*
`Precision`	RNGINTELT	*Default :*

Given a modular form f, construct the L-series $L(f,s) = \sum_{n=1}^{\infty} a_n/n^s$, where f has the q-expansion $\sum_{n=0}^{\infty} a_n q^n$. It is assumed that $L(f,s)$ satisfies a functional equation of the standard kind (see Section 127.5.1 for the precise form of the functional equation).

The optional parameter `embedding` specifies a map which embeds the coefficients of f into the complex field. By default this is the identity map, so that the coefficients of f must be coercible into $\mathbf{C}$. Otherwise, the value of the parameter must either be an object of type `Map` or a user-defined function $e(x)$ each having domain the base ring of f and codomain the complex field (or values than can be coerced into the complex field).

The number of digits of precision to which the values $L(f,s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

Example H127E14

We define a newform of weight 2 and conductor 16. It is not defined over the integers but rather over $B = \mathbf{Z}[i]$.

```
> f := Newforms("G1N16k2A")[1]; f;
q + (-a - 1)*q^2 + (a - 1)*q^3 + 2*a*q^4 + (-a - 1)*q^5 + 2*q^6 -
2*a*q^7 + O(q^8)
> B:=BaseRing(f); B;
Equation Order with defining polynomial x^2 + 1 over Z
```

The two distinct embeddings of B into the complex numbers give rise to two modular forms, which can be accessed using the `ComplexEmbeddings` function.

```
> f1, f2 := Explode(ComplexEmbeddings(f)[1]);
> Coefficient(f,2), Coefficient(f1,2), Coefficient(f2,2);
-a - 1
-1.0000000000000000000000000000 + 1.0000000000000000000000000000*$.1
-1.0000000000000000000000000000 - 1.0000000000000000000000000000*$.1
```

Thus, f_1 and f_2 have genuine complex coefficients and we can construct the associated L-series and compute their L-values, for instance at $s = 1$.

```
> L1 := LSeries(f1);
> L2 := LSeries(f2);
> CheckFunctionalEquation(L1);
-2.3665827156630354162351856958 5E-30
> CheckFunctionalEquation(L2);
-2.3665827156630354162351856958 5E-30
> v1 := Evaluate(L1,1); v2 := Evaluate(L2,1); v1,v2;
0.35930643700350568406632720 7778 + 0.07147049399911726865884580 66910*$.1
```

```
0.359306437003505684066327207778 - 0.071470493999117268658845806691O*$.1
```

If instead we invoke `LSeries(f)`, MAGMA will note that f is defined over a number field and complain that the coefficients of f are not well-defined complex numbers.

```
> L := LSeries(f);
For f over a number field, you have to specify a complex embedding
```

Instead of using `ComplexEmbeddings`, one can instead explicitly specify an embedding of the coefficients of B into the complex numbers using the parameter `Embedding` with the function `LSeries` The following statements define the same L-function as L_2 above.

```
> C<i> := ComplexField();
> L2A := LSeries(f: Embedding:=hom< B -> C | i > );
> L2B := LSeries(f: Embedding:=func< x | Conjugates(x)[1] > );
> L2C := LSeries(f1: Embedding:=func< x | ComplexConjugate(x) > );
```

Finally, we illustrate the very important fact that MAGMA expects, but does *not* check that the L-function associated to a modular form satisfies a functional equation.

```
> L := LSeries(f1+f2);  // or L:=LSeries(f: Embedding:=func<x|Trace(B!x)>);
```

Although MAGMA is happy with this definition, it is in fact illegal. The modular form f has a character whose values lie in the field of the 4-th roots of unity.

```
> Order(DirichletCharacter(f));
4
```

The two embeddings f_1 and f_2 of f have *different* (complex conjugate) characters and $f_1 + f_2$ does not satisfy a functional equation of the standard kind. MAGMA will suspect this when it tries to determine the sign in the functional equation and thereby print a warning:

```
> Evaluate(L,1);
  |Sign| is far from 1, wrong functional equation?
0.736718188651826073550560964422
> CheckFunctionalEquation(L);
0.008143380371344820260617212212 44
```

The function `CheckFunctionalEquation` should return 0 (to current precision), so the functional equation is not satisfied, and the result of evaluating L will be a random number. So it is the *user's* responsibility to ensure that the modular form does satisfy a functional equation as described in Section 127.5.1. Here are some examples of modular forms that do.

```
> CheckFunctionalEquation(LSeries(f1^2*f2));
1.5777218104420236108234571305 7E-30
> f3 := ModularForm(EllipticCurve([0, -1, 1, 0, 0]));
> CheckFunctionalEquation(LSeries(f3));
0.00000000000000000000000000000
> M := Newforms("37k2");
> f4 := M[1,1]; f5 := M[2,1]; f6 := M[3,1];
> CheckFunctionalEquation(LSeries((f5+2*f6)*f4));
5.916456789157588540587964239 62E-31
```

127.3 Computing L-values

Once an L-series $L(s)$ has been constructed using either a standard zeta- or L-function (Section 127.2), a user defined L-function (Section 127.5.2) or constructed from other L-functions (Section 127.4), MAGMA can compute values $L(s_0)$ for complex s_0, values for the derivatives $L^{(k)}(s_0)$ and Taylor expansions.

`Evaluate(L, s0)`

`Derivative` RNGINTELT *Default :* 0

`Leading` BOOLELT *Default :* `false`

Given the L-series L and a complex number s_0, the intrinsic computes either $L(s_0)$, or if $D > 0$, the value of the derivative $L^{(D)}(s_0)$. If $D > 0$ and it is known that all the lower derivatives vanish,

$$L(s_0) = L'(s_0) = ... = L^{(D-1)}(s_0) = 0\,,$$

the computation time can be substantially reduced by setting `Leading:=true`. This is useful if it is desired to determine experimentally the order of vanishing of $L(s)$ at s_0 by successively computing the first few derivatives.

`CentralValue(L)`

Given an L-function of even weight $2k$ (in the MAGMA sense), the value of L is computed at $s = k$.

`LStar(L, s0)`

`Derivative` RNGINTELT *Default :* 0

Given the L-series L and a complex number s_0, the intrinsic computes either the value $L^*(s_0)$ or, if $D > 0$, the value of the derivative $L^{*(D)}(s_0)$. Here $L^*(s) = \gamma(s)L(s)$ is the modified L-function that satisfies the functional equation (cf. Section 127.5.1)

$$L^*(s) = \text{sign} \cdot \bar{L}^*(\text{weight} - s)$$

(cf. Section 127.5.1).

`LTaylor(L,s0,n)`

`ZeroBelow` RNGINTELT *Default :* 0

Compute the first $n + 1$ terms of the Taylor expansion of the L-function about the point $s = s_0$, where s_0 is a complex number:

$$L(s_0) + L'(s_0)x + L''(s_0)x^2/2! + \ldots + L^{(n)}(s_0)x^n/n! + O(x^{n+1})\ .$$

If the first few terms $L(s_0), ..., L^{(k)}(s_0)$ of this expansion are known to be zero, the computation time can be reduced by setting `ZeroBelow:=`$k + 1$.

Example H127E15

We define an elliptic curve E of conductor 5077 and compute derivatives at $s = 1$ until a non-zero value is reached:

```
> E := EllipticCurve([0, 0, 1, -7, 6]);
> L := LSeries(E : Precision:=15);
> Evaluate(L, 1);
0.000000000000000
> Evaluate(L, 1 : Derivative:=1, Leading:=true);
-1.69522909186553E-17
> Evaluate(L, 1 : Derivative:=2, Leading:=true);
6.27623031179552E-17
> Evaluate(L, 1 : Derivative:=3, Leading:=true);
10.3910994007158
```

This suggests that $L(E, s)$ has a zero of order 3 at $s = 1$. In fact, E is the rational elliptic curve of smallest conductor with Mordell-Weil rank 3:

```
> Rank(E);
3
```

Consequently, a zero of order 3 is predicted by the Birch–Swinnerton-Dyer conjecture. We can also compute a few terms of the Taylor expansion about $s = 1$, with or without specifying that the first three terms vanish.

```
> time LTaylor(L, 1, 5 : ZeroBelow:=3);
1.73184990011930*$.1^3 - 3.20590558844390*$.1^4 + 2.93970849657696*$.1^5
Time: 11.070
> time LTaylor(L, 1, 5);
-1.69522909186553E-17*$.1 + 3.13811515589776E-17*$.1^2 +
1.73184990011930*$.1^3 - 3.20590558844390*$.1^4 + 2.93970849657696*$.1^5
Time: 20.320
```

And this is the leading derivative, with the same value as `Evaluate(L,1:D:=3)`.

```
> c := Coefficient($1,3)*Factorial(3);c;
10.3910994007158
```

Finally, we compute the 3rd derivative of the modified L-function $L^*(s) = \gamma(s)L(s)$ at $s = 1$. For an elliptic curve over the rationals, $\gamma(s) = (N/\pi^2)^{s/2}\Gamma(s/2)\Gamma((s+1)/2)$, where N is the conductor. So, by the chain rule, $L^{*\prime\prime\prime}(1) = \gamma(1)L'''(1) = \sqrt{N/\pi}L'''(1)$.

```
> LStar(L, 1 : Derivative:=3);
417.724689268266
> c*Sqrt(Conductor(E)/Pi(RealField(15)));
417.724689268267
```

127.4 Arithmetic with L-series

With the exception of some modular forms, all the built-in L-series have weakly multiplicative coefficients, so that $L(s) = \sum a_n/n^s$ with $a_{mn} = a_m a_n$ for m, n coprime. For two such L-series, MAGMA allows the user to construct their product and, provided that it makes sense, their quotient.

`L1 * L2`

`Poles`	SEQENUM	*Default :* []
`Residues`	SEQENUM	*Default :* []
`Precision`	RNGINTELT	*Default :*

Let $L_1(s)$ and $L_2(s)$ be two L-series of the same weight whose coefficients are weakly multiplicative, that is, they satisfy $a_{mn} = a_m a_n$ for m, n coprime. This function constructs their product $L(s) = L_1(s)L_2(s)$.

If one of the L-series has zeros that cancel the poles of the other L-series, the user should specify the list of poles for $L_1^*(s)L_2^*(s)$ using the `Poles` parameter and the corresponding residues using the `Residues` parameter. See Section 127.5.1 for the terminology and Section 127.5.2 for the format of the poles and residues parameters.

The number of digits of precision to which the values $L(s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted, the precision is taken to be that of the default real field.

`L1 / L2`

`Poles`	SEQENUM	*Default :* []
`Residues`	SEQENUM	*Default :* []
`Precision`	RNGINTELT	*Default :*

Let $L_1(s)$ and $L_2(s)$ be two L-series whose coefficients a_{mn} are weakly multiplicative, that is, they satisfy $a_{mn} = a_m a_n$ for m, n coprime. This function constructs their quotient $L(s) = L_1(s)/L_2(s)$.

This function assumes (but does not check!) that this quotient exists and is a genuine L-function with finitely many poles.

If $L_2(s)$ happens to have zeros that give poles in the quotient, the user must specify the list of poles of $L_1^*(s)/L_2^*(s)$ using the `Poles` parameter and the corresponding residues using the `Residues` parameter. See Section 127.5.1 for the terminology and Section 127.5.2 for the format of `Poles` and `Residues`.

The number of digits of precision to which the values $L(s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted, the precision is taken to be that of the default real field.

`TensorProduct(L1, L2, ExcFactors)`

`TensorProduct(L1, L2)`

`TensorProduct(L1, L2, ExcFactors, K)`

`TensorProduct(L1, L2, K)`

`Precision`	RngIntElt	*Default :*
`Sign`	FldComElt	*Default :*

Let L_1 and L_2 be L-functions such that $L_1(s) = L(V_1, s)$ and $L_2(s) = L(V_2, s)$ are associated to systems of l-adic representations V_1 and V_2 (à la Serre). This function computes their tensor product $L(s) = L(V_1 \otimes V_2, s)$. This can be used, for example, to twist an L-function by characters or higher-dimensional Artin representations (see Examples H127E24, H127E25).

Note that, in particular, both $L_1(s)$ and $L_2(s)$ must have integer conductor, weakly multiplicative coefficients and an underlying Hodge structure (which is computed from the γ-shifts). The argument `ExcFactors` is a list of tuples of the form $\langle p, v\rangle$ or $\langle p, v, F_p(x)\rangle$ that give, for each of the primes p where V_1 and V_2 both have bad reduction, the valuation v of the conductor of $V_1 \otimes V_2$ at p and the inverse local factor at p. If the data is not provided for such a prime p, Magma will attempt to compute the local factors by assuming that the inertia invariants behave well at p,

$$(V_1 \otimes V_2)^{I_p} = V_1^{I_p} \otimes V_2^{I_p}.$$

It will also compute the conductor exponents by predicting the tame and wild degrees from the degrees of the local factors, but this does not work if both V_1 and V_2 are wildly ramified at p.

The sign in the functional equation of $L(V_1 \otimes V_2, s)$ cannot be determined from the signs of the factors, so it will be calculated numerically from the functional equation. If the sign is known, the user may specify it by means of the `Sign` parameter.

The number of digits of precision to which the values $L(s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted, the precision is taken to be that of the default real field.

If L_1 and L_2 have Euler products over the same field K, this field can be given as an additional argument, and the tensor product will be taken with respect to that field.

See Examples H127E18 and H127E24, H127E25, and Section 127.9.7.

127.5 General L-series

In addition to the built-in L-series for the standard objects, Magma provides machinery that allows the user to define L-series for arbitrary objects, provided that they admit a meromorphic continuation to the whole complex plane and satisfy a functional equation of the standard kind. To describe how this may be done, we need to introduce some terminology.

127.5.1 Terminology

Recall that an *L-series* is a sum

$$L(s) = \sum_{n=1}^{\infty} \frac{a_n}{n^s},$$

where the coefficients a_n are complex numbers, known as the Dirichlet coefficients of the L-series. For example, the Dirichlet coefficients of the classical Riemann zeta function are $a_n = 1$ for all n. We assume that, as in the case of Riemann zeta function, every $L(s)$ has a meromorphic continuation to the complex plane and has a functional equation. Technically, our assumptions are as follows:

Assumption 1. The defining series for $L(s)$ converges for $\mathrm{Re}(s)$ sufficiently large. Equivalently, the *coefficients* a_n grow at worst as a polynomial function of n.

Assumption 2. $L(s)$ admits a meromorphic continuation to the entire complex plane.

Assumption 3. The following exist: real positive *weight*, complex *sign* of absolute value 1, real positive *conductor* and the Γ-*factor*

$$\gamma(s) = \Gamma\Big(\frac{s+\lambda_1}{2}\Big) \cdots \Gamma\Big(\frac{s+\lambda_d}{2}\Big)$$

of *dimension* $d \geq 1$ and rational γ-*shifts* $\lambda_1, \ldots \lambda_d$, such that

$$L^*(s) = \Big(\frac{\text{conductor}}{\pi^d}\Big)^{s/2} \gamma(s)\, L(s)$$

satisfies the *functional equation*

$$L^*(s) = \text{sign} \cdot \bar{L}^*(\text{weight} - s).$$

Here $\bar{L}$ is the *dual* L-series with complex conjugate coefficients $L(s) = \sum_{n=1}^{\infty} \overline{a_n}/n^s$. Note that the "weight" here is one more than the usual motivic notion of weight.

Assumption 4. The series $L^*(s)$ has finitely many simple *poles* and no other singularities.

Assumption 1 will almost certainly be true for any naturally arising L-function and Assumptions 2–4 are expected (but not proven) to be satisfied for most of the L-functions arising in geometry and number theory. In fact, there is a large class of so-called *motivic L-functions* for which all of the above assumptions are conjectured to be true. Essentially this class consists of L-functions associated to cohomology groups of varieties over number fields. This is an extremely large class of L-functions that includes all of the standard examples.

127.5.2 Constructing a General L-Series

When computing the values $L(s)$ for a complex number s, MAGMA relies heavily on the functional equation. This means that the *emphasized* parameters in Assumptions 1–4 (coefficients, weight, conductor, poles, etc.) must be known before the computations can be carried out.

The generic `LSeries` function allows the user to construct an L-series for a new object by specifying values for these parameters. In fact, this function is used to construct all of the built-in L-series in MAGMA and some of these will be used as examples to illustrate its use (see the advanced examples section).

`LSeries(weight, gamma, conductor, cffun)`

`Sign`	FLDCOMELT	*Default :* 0
`Poles`	SEQENUM	*Default :* []
`Residues`	SEQENUM	*Default :* []
`Parent`	ANY	*Default :*
`CoefficientGrowth`	USERPROGRAM	*Default :*
`Precision`	RNGINTELT	*Default :*
`ImS`	FLDREELT	*Default :* 0
`Asymptotics`	BOOLELT	*Default :* `true`

The function takes four arguments: real positive *weight*, sequence of rational numbers *gamma*, real positive *conductor* and a coefficient function $cffun$ together with a number of optional parameters. It constructs an L-series with specified coefficients and the form of the functional equation.

Compulsory arguments (see Section 127.5.1 for terminology):

weight — weight of the L-series.

Specifies the functional equation; for instance, it is `1` for the Riemann zeta function, `2` for curves over the rationals and `Weight(f)` for modular forms f.

Note that the "weight" in this sense is the w such that $s \to w - s$ satisfies a functional equation. The usual motivic notion of weight is thus one less than this – for instance, a weight k modular form has weight k in Magma, but weight $k-1$ for motives, as its coefficients are typically of size $(\sqrt{p})^{k-1}$.

gamma — gamma shifts/parameters.

List (of type `SeqEnum`) of rational numbers that specify the gamma factor. For instance, this is `[0]` for the Riemann zeta function, `[0,1]` for an elliptic curve over $\mathbf{Q}$ and `[0,...,0,1,...,1]` with $r_1 + r_2$ zeros and r_2 ones for the Dedekind zeta function of a number field K with r_1 real and r_2 pairs of complex embeddings.

conductor — conductor of the L-series.

Part of the exponential factor $(\frac{\text{conductor}}{\pi^d})^{s/2}$ in the functional equation. It is usually an integer and is 1 for the Riemann zeta function, the level for a modular form,

|Discriminant(K)| for the Dedekind zeta of a number field K and the conductor of the Jacobian for an algebraic curve over the rationals.

cffun — specifies the coefficients a_n. The following possibilities are allowed:

- A finite sequence $[a_1, ..., a_n]$;
- A function $f(n)$ that returns a_n;
- A function $f(p, d)$ that computes the inverse of the local factor at p up to degree d;
- The value `0`, signifying that the coefficients will be provided later with a call to `LSetCoefficients`.

See Section 127.5.3 for a detailed explanation and examples.

Optional parameters for `Lseries:`:

`Sign` — the sign appearing in the functional equation.

This is the sign that enters the functional equation and is usually ± 1. It must be a complex number of absolute value 1. Alternatively, this can be set to 0 (default), in which case MAGMA will determine it numerically from the functional equation.

Instead of providing the actual sign, the user may also use an option `Sign:=s` where s is the name of a function $s(p)$ or $s(L, p)$ that computes the sign to precision p.

`Poles` — the poles z of $L^*(s)$ with Re $z \geq$weight/2 and

`Residues` — the residues at these poles

If $L^*(s)$ happens to have poles, there are only finitely many of them and each must be simple by Assumption 4 of the previous subsection. The poles have to be known and specified here when the L-function is created. The poles are symmetric about the point $s =$weight/2 by the functional equation and only the "right half" of the set of them has to be supplied in the form of a sequence as the value of the parameter `Poles`. The default setting is that $L^*(s)$ has no poles.

The residues at the poles of $L^*(s)$ are also required and they can be specified using the `Residues` parameter, which also takes a sequence as its value. It can be either a sequence of the same length as `Poles` or left to its default setting `[]`. In that case, MAGMA will attempt to determine the residues numerically using the functional equation. However, this is only possible if `Sign` is known and will not work to full precision if there is more than one pair of poles.

As an example, the Riemann zeta function has weight 1 and the modified function $\zeta^*(s)$ has poles at $s = 0$ and $s = 1$ with residues 1 and -1 respectively. So the parameter assignments `Poles:=[1], Residues:=[-1]` would be used in its definition.

As in the case of `Sign`, instead of providing the actual residues, the user may give the name of a function $r(p)$ or $r(L, p)$ that computes the residues up to precision p and returns them as a sequence.

`Parent` — any MAGMA object.

This is only used for printing purposes. When a variable type `LSer` is printed, MAGMA prints "L-series of" and then prints the parent object.

`Precision` — the precision to which L-values are to be computed (default is 0, use current precision).

`CoefficientGrowth` — name f of a function $f(x)$ or $f(L, x)$ such that $|a_n| < f(n)$.

`ImS` — the largest imaginary part of s for which $L(s)$ will be evaluated.

`Asymptotics` — whether to use asymptotic expansions (default is yes).

These four settings are related to the precision to which values are calculated in L-series computations. They are the same as for the `SetPrecision` function and are described in detail in Section 127.7.

As an illustration, the following code creates the Riemann zeta function from its invariants. This is essentially what `RiemannZeta()` does:

```
> Z := LSeries(1, [0], 1, func<n|1>
>                         : Sign:=1, Poles:=[1], Residues:=[-1]);
> CheckFunctionalEquation(Z);
0.00000000000000000000000000000000
```

For more examples, see the "Advanced Examples" Section 127.9.

It is strongly advised that the user apply the function `CheckFunctionalEquation` whenever a generic L-series is defined. If one of the specified parameters (conductor, sign, etc.) happens to be incorrect, the resulting L-series will not have a functional equation and the values returned by the evaluation functions will be nonsense. Only by checking the functional equation will the user have an indication that something is wrong.

`CheckFunctionalEquation(L)`

`t` FldReElt *Default :* 1.2

Given an L-series L, this function tests the functional equation numerically and should ideally return 0 (to the current precision), meaning that the test was passed. If the value returned is a significant distance from 0, either the L-function does not have a functional equation or some of the defining parameters (conductor, poles etc.) have been incorrectly specified or not enough coefficients have been given. In the latter case, `CheckFunctionalEquation` is likely to return a number reasonably close to 0 that measures the accuracy of computations.

As already mentioned, Magma can only work with L-functions that satisfy the functional equation as in Section 127.5.1. Whenever an L-function is constructed, its functional equation is implicitly used in all the L-series evaluations, even when L is evaluated in the region where the original Dirichlet series is absolutely convergent.

If either the sign in the functional equation or the residues of $L^*(s)$ have not yet been computed, this function will first compute them. If the sign was undefined, it returns $|\texttt{Sign}| - 1$ which, again, must be 0.

The optional technical parameter `t` is a point on the real line where Magma evaluates the two Theta functions associated to the L-series and subtracts one from the other to get the value returned by `CheckFunctionalEquation`. The parameter

`t` must be a real number satisfying $1.05 < t < 1.2$ and for every such number the function should return 0, provided that the functional equation is indeed correct.

Example H127E16

We attempt to define a truncated L-series of a quadratic character mod 3 (so $\chi(n) = 0, 1, -1$ if n is 0,1 or 2 mod 3 respectively.)

```
> L := LSeries(1, [0], 3, [1,-1,0,1,-1,0] : Sign:=-1);
> CheckFunctionalEquation(L);
1.15245561556037369749660548 1547
```

This does not look right. In fact, the γ-shifts are 0 for even quadratic characters, but 1 for odd ones,

```
> L := LSeries(1, [1], 3, [1,-1,0,1,-1,0] : Sign:=-1);
> CheckFunctionalEquation(L);
1.06372623587922089871296822 6243
```

This still does not look right. We gave the wrong sign in the functional equation.

```
> L := LSeries(1, [1], 3, [1,-1,0,1,-1,0] : Sign:=1);
> CheckFunctionalEquation(L);
3.32817107758805778728258386 3548E-15
```

This certainly looks better but it indicates that we did not give enough coefficients to our L-series. We determine how many coefficients are needed to do computations with default precision (30 digits),

```
> LCfRequired(L);
11
```

So 11 coefficients are needed and we provide them in the code below.

```
> L := LSeries(1, [1], 3, [1,-1,0,1,-1,0,1,-1,0,1] : Sign:=1);
> CheckFunctionalEquation(L);
9.86076131526264756764660706 6035E-32
```

This a correct way to define our L-series. Even better is the following variant:

```
> L := LSeries(1, [1], 3, func<n|((n+1) mod 3)-1> : Sign:=1);
> CheckFunctionalEquation(L);
9.86076131526264756764660706 6035E-32
```

This allows MAGMA to calculate as many a_n as it deems necessary using the provided function.

127.5.3 Setting the Coefficients

`LSetCoefficients(L,cffun)`

This function defines the coefficients a_n of the L-series L. The argument $cffun$ can be one of the following:

- A sequence $[a_1, ..., a_n]$;
- A function $f(n)$ that returns a_n;
- A function $f(p, d)$ that computes the inverse of the local factor at p up to degree d.

When a user-defined L-series is constructed by invoking the function

```
> L := LSeries(weight,gamma,conductor,cffun: <optional parameters>);
```

this is actually equivalent to invoking the pair of functions

```
> L := LSeries(weight,gamma,conductor,0: <optional parameters>);
> LSetCoefficients(L,cffun);
```

where the first line indicates that the coefficients will be supplied later and the second line actually specifies the coefficients. So the following description of the `cffun` parameter for the function `LSetCoefficients(L,cffun)` applies to the main `LSeries` signature as well.

The first two ways to specify the a_n are either to give a pre-computed sequence of coefficients up to a certain bound or to give the name of a function $f(n)$ that computes the a_n. (The other two ways are described in the two subsections that follow.) For instance, the following both define the Riemann zeta function with weight 1, one γ-shift 0, conductor 1, sign 1, pole at $s = 1$ with residue -1, all $a_n = 1$:

```
> V := [ 1 : k in [1..100] ];
> L := LSeries(1, [0], 1, V : Sign:=1, Poles:=[1], Residues:=[-1]);
```

or

```
> f := func<n|1>;
> L := LSeries(1, [0], 1, f : Sign:=1, Poles:=[1], Residues:=[-1]);
```

Of the two possibilities, the second one is safer to use in a sense that it lets MAGMA decide how many coefficients it needs in order to perform the calculations to the required precision. However, if the computation of a_n is costly and it involves previous coefficients, then it is probably better to write a function that computes a_n recursively, storing them in a sequence and then passing it to `LSeries`.

127.5.4 Specifying the Coefficients Later

When specifying a finite list of coefficients it is necessary to know in advance how many coefficients have to be computed. For this, MAGMA provides a function `LCfRequired(L)` (see Section 127.6).

Example H127E17

We define L to be our own version of the Riemann zeta function (weight 1, one γ-shift 0, conductor 1, sign 1, pole at $s = 1$ with residue -1), but tell MAGMA that we will specify the coefficients later with the 4th parameter set to 0. Then we ask how many coefficients it needs to perform computations:

```
> L := LSeries(1, [0], 1, 0: Sign:=1, Poles:=[1], Residues:=[-1]);
> N := LCfRequired(L); N;
6
```

Now we compute the coefficient vector $[a_1, ..., a_N]$.

```
> vec := [1, 1, 1, 1, 1, 1];
> LSetCoefficients(L,vec);
```

Now we can evaluate our ζ-function

```
> Evaluate(L,2);
1.64493406684822643647241516665
> Pi(RealField())^2/6;
1.64493406684822643647241516665
```

If we provide fewer coefficients, the computations will not have full precision but we can use `CheckFunctionalEquation` to get an indication of the resulting accuracy

```
> LSetCoefficients(L, [1,1]);
> CheckFunctionalEquation(L);
4.94876281053441137266582049650 8E-9
> Evaluate(L, 2);
1.64493406684811250127872978182
> $1 - Pi(RealField(28))^2/6;
-1.13935193685384864674677434 0E-13
```

Note that such good accuracy with just two coefficients is a singular phenomenon that only happens for L-functions with very small conductors. Normally, at least hundreds of coefficients are needed to compute L-values to this precision.
Note also that the last value `Evaluate(L,2)` in the example is much more precise than what one would get with the truncated version of the original defining series $\zeta_{trunc}(s) = 1 + 1/2^s$ at $s = 2$. The reason for this is that even in the region where the original Dirichlet series converges, the use of the functional equation usually speeds up the convergence.

127.5.5 Generating the Coefficients from Local Factors

The final and, in many cases, mathematically the most natural way to supply the coefficients is by specifying the so-called *local factors*. Recall that the coefficients for most L-series are *weakly multiplicative*, meaning that $a_{mn} = a_m a_n$ for m coprime to n. For such L-series, $L(s)$ admits a *product formula*

$$L(s) = \prod_{p \text{ prime}} \frac{1}{F_p(p^{-s})},$$

where $F_p(x)$ is a formal power series in x with complex coefficients. For example, if $L(s)$ arises from a variety over a number field (and this, as we already mentioned, essentially covers everything), then such a product formula holds. In this case the $F_p(x)$ are polynomials of degree d for those primes p not dividing the conductor and are of smaller degree otherwise. Moreover, their coefficients lie in the ring of integers of the field of definition of the variety.

For L-functions with weakly multiplicative coefficients, MAGMA allows the coefficients to be defined by specifying the local factors, using the function `LSetCoefficients(L,f)` where f is a user-defined function with two arguments p and d that computes $F_p(x)$, either as a full polynomial in x or as a power series in x of precision $O(x^{d+1})$. For example, the Riemann zeta function has $F_p(x) = 1 - x$ for all p, so we can define it as follows,

```
> Z := LSeries(1, [0], 1, 0 : Poles:=[1], Residues:=[-1], Sign:=1);
> P<x> := PolynomialRing(Integers());
> LSetCoefficients(Z, func<p,d | 1-x> );
```

or as

```
> P<x> := PowerSeriesRing(Integers());
> Z := LSeries(1, [0], 1, func<p,d|1-x+O(x^(d+1))> :
               Poles:=[1], Residues:=[-1], Sign:=1);
```

127.6 Accessing the Invariants

`LCfRequired(L)`

The number of Dirichlet coefficients a_n that have to be calculated in order to compute the values $L(s)$. This function can be also used with a user-defined L-series before its coefficients are set, see Section 127.5.4.

`LGetCoefficients(L, N)`

Compute the vector of first N coefficients `[*` $a_1, ..., a_N$ `*]` of the L-series given by L.

`EulerFactor(L, p)`

Given an L-series and a prime p, this computes the pth Euler factor, either as a polynomial or a power series. The optional parameter `Degree` will truncate the series to that length, and the optional parameter `Precision` is of use when the series is defined over the complex numbers.

`Conductor(L)`

Conductor of the L-series (real number, usually an integer). This invariant enters the functional equation and measures the 'size' of the object to which the L-series is associated. Evaluating an L-series takes time roughly proportional to the square root of the conductor.

`Sign(L)`

Sign in the functional equation of the L-series. This is a complex number of absolute value 1, or 0 if the sign has not been computed yet. (Calling `CheckFunctionalEquation(L)` or any evaluation function sets the sign.)

`GammaFactors(L)`

A sequence of Gamma factors $\lambda_1, \ldots, \lambda_d$ for $L(s)$. Each one represents a factor $\Gamma((s+\lambda_i)/2)$ entering the functional equation of the L-function.

`LSeriesData(L)`

Given an L-series L, this function returns the weight, the conductor, the list of γ-shifts, the coefficient function, the sign, the poles of $L^*(s)$ and the residues of $L^*(s)$ as a tuple of length 7. If Sign $= 0$, this means it has not been computed yet. Residues= [] means they have not yet been computed. From this data, L can be re-created with a general `LSeries` call (see Section 127.5.2).

Example H127E18

For a modular form, the q-expansion coefficients are the same as the Dirichlet coefficients of the associated L-series:

```
> f := Newforms("30k2")[1,1];
> qExpansion(f,10);
q - q^2 + q^3 + q^4 - q^5 - q^6 - 4*q^7 - q^8 + q^9 + O(q^10)
> Lf := LSeries(f);
> LGetCoefficients(Lf,20);
[* 1, -1, 1, 1, -1, -1, -4, -1, 1, 1, 0, 1, 2, 4, -1, 1, 6, -1, -4, -1*]
```

The elliptic curve of conductor 30 that corresponds to f has, of course, the same L-series.

```
> E := EllipticCurve(f); E;
Elliptic Curve defined by y^2 + x*y + y = x^3 + x + 2 over Rational Field
> LE := LSeries(E);
> LGetCoefficients(LE,20);
```

```
[* 1, -1, 1, 1, -1, -1, -4, -1, 1, 1, 0, 1, 2, 4, -1, 1, 6, -1, -4, -1*]
```

Now we change the base field of E to a number field K and evaluate the L-series of E/K at $s = 2$.

```
> P<x> := PolynomialRing(Integers());
> K := NumberField(x^3-2);
> LEK := LSeries(E,K);
> i := LSeriesData(LEK); i;
<2, [ 0, 0, 0, 1, 1, 1 ], 8748000, ... >
```

The conductor of this L-series (second entry) is very large and this is an indication that the calculations of $L(E/K, 2)$ to the required precision (30 digits) will take some time. We can also ask how many coefficients will be used in this calculation.

```
> LCfRequired(LEK); // a lot!
280632
```

Decreasing the precision will help somewhat.

```
> LSetPrecision(LEK,9);
> LCfRequired(LEK);
17508
```

And realizing that our L-series has a piece that can be factored out will certainly help (see the Arithmetic section).

```
> Q := LEK/LE;
> LSetPrecision(Q,9);
> LCfRequired(Q);
3780
> time Evaluate(Q,2) * Evaluate(LE,2);  // = L(E/K,2)
0.892165947
Time: 8.020
```

`Factorization(L)`

If an L-series is represented internally as a product of other L-series, say $L(s) = \prod_i L_i(s)^{n_i}$, return the sequence `[...<`L_i`,`n_i`>...]`.

Example H127E19

```
> L := RiemannZeta();
> Factorization(L);
[
    <L-series of Riemann zeta function, 1>
]
> R<x> := PolynomialRing(Rationals());
> K := SplittingField(x^3-2);
> L := LSeries(K);
> Factorization(L);
[
```

```
    <L-series of Riemann zeta function, 1>,
    <L-series of Artin representation of Number Field with
      defining polynomial x^6 + 108 over the Rational Field
      with character ( 1, -1, 1 ) and conductor 3, 1>,
    <L-series of Artin representation of Number Field with
      defining polynomial x^6 + 108 over the Rational Field
      with character ( 2, 0, -1 ) and  conductor 108, 2>
]
```

127.7 Precision

The values of an L-series are computed numerically and the precision required in these computations can be specified using the `Precision` parameter when an L-function is created. For example,

```
> K := CyclotomicField(3);
> L := LSeries(K: Precision:=60);
> Evaluate(L,2);
1.285190955484149402917511798699574603969178397028921243951 33
```

computes $\zeta(K, 2)$ to 60 digits precision. The default value `Precision:=0` is equivalent to the precision of the default real field at the time of the `LSeries` initialization call. If the user only wants to check numerically whether an L-function vanishes at a given point s and the value itself is not needed, it might make sense to decrease the precision (to 9 digits, say) to speed up the computations.

```
> E := EllipticCurve([0,0,1,-7,6]);
> RootNumber(E);
-1
> L := LSeries(E: Precision:=9);
> Evaluate(L, 1: Derivative:=1);
1.50193628E-11
> Rank(E);
3
```

This example checks numerically that $L'(E, 1) = 0$ for the elliptic curve E of conductor 5077 from Example H127E15.

The precision may be changed at a later time using `LSetPrecision`.

`LSetPrecision(L,precision)`

`CoefficientGrowth`	UserProgram	*Default :*
`ImS`	FldReElt	*Default :* 0
`Asymptotics`	BoolElt	*Default :* `true`

Change the number of digits to which the L-values are going to be computed to *precision*. The parameter `CoefficientGrowth` is described below in Section 127.7.1.

127.7.1 L-series with Unusual Coefficient Growth

The parameter `CoefficientGrowth` is the name f of a function $f(x)$ (or $f(L, x)$, where L is an L-series) which is an increasing function of a real positive variable x such that $|a_n| \leq f(n)$. This is used to truncate various infinite series in computations. It is set by default to the function $f(x) = 1.5 \cdot x^{\rho-1}$ where ρ is the largest real part of a pole of $L^*(s)$ if $L^*(s)$ has poles and $f(x) = 2x^{(weight-1)/2}$ if $L^*(s)$ has no poles. The user will most likely leave this setting untouched.

127.7.2 Computing $L(s)$ when $\mathrm{Im}(s)$ is Large (`ImS` Parameter)

If s is a complex number having a large imaginary part, a great deal of cancellation occurs while computing $L(s)$, resulting in a loss of precision. (The time when all the precision-related parameters are pre-computed is when the function `LSeries` is invoked, and at that time MAGMA has no way of knowing whether $L(s)$ is to be evaluated for complex numbers s having large imaginary part.) If this happens, a message is printed, warning of a precision loss. To avoid this, the user may specify the largest $\mathrm{Im}(s)$ for which the L-values are to be calculated as the value of the `ImS` parameter at the time of the L-series initialization or, later, with a call to `LSetPrecision`:

```
> C<i> := ComplexField();
> L := RiemannZeta();
> Evaluate(L, 1/2+40*i);            // wrong
Warning: Loss of 13 digits due to cancellation
0.79304495256192867198212885 1045 - 1.041274614651065020074521950 86*i
> LSetPrecision(L, 30: ImS:=40);
> Evaluate(L, 1/2+40*i);            // right
0.79304495256192867196489258 8898 - 1.041274614651065020051890595 39*i
```

127.7.3 Implementation of L-series Computations (`Asymptotics` Parameter)

The optional parameter `Asymptotics` in `LSetPrecision` and in the general `LSeries` function specifies the method by which the special functions needed for the L-series evaluation are computed.

If set to `false`, MAGMA will use only Taylor expansions at the origin and these are always known to be convergent. With the default behaviour (`Asymptotics:=true`) MAGMA will use both the Taylor expansions and the continued fractions of the asymptotic expansions at infinity. This is much faster, but these continued fractions are not proved to be convergent in all cases.

127.8 Verbose Printing

There is one verbose printing variable, called "LSeries" that controls verbose printing for all of the functions in this chapter.

```
> SetVerbose("LSeries",n);
```

It accepts verbosity levels n such that $0 \leq n \leq 3$. The values are as follows:

$n = 0$ is the (default) quiet mode. The functions do not print anything apart from loss-of-precision warnings.

$n = 1$ is possibly the most useful setting. The sign and the residues of $L^*(s)$ are printed whenever they are determined from the functional equation.

$n = 2$ generates messages that allow the user to keep track of what is currently happening. So messages are printed when a new L-function is constructed, when coefficients are generated or when expansions of special functions are computed.

$n = 3$ is a "keep-alive" setting that basically informs the user that the computer has not crashed yet and, yes, something is still happening. For L-functions of large conductor or very high precision that require a large number of coefficients, MAGMA will print a "progress indicator" every time 1000 new coefficients are generated and every 5000 terms in the series computations that test the functional equation or compute the L-values. (The frequencies 1000 and 5000 are stored in the attributes L`vprint_coeffs and L`vprint_series of a variable L of type LSeries and may be changed at the user's desire.)

127.9 Advanced Examples

127.9.1 Handmade L-series of an Elliptic Curve

Example H127E20

This is an example of how the general LSeries function can be used to define the L-series for elliptic curves. This is essentially what LSeries(E) does:

```
> E := EllipticCurve([1,2,3,4,5]);
> N := Conductor(E);N;
10351
> P<x> := PolynomialRing(Integers());
```

The easiest way to define the coefficients is to provide the local factors.

```
> cf := func< p,d|1 - TraceOfFrobenius(E,GF(p,1))*x
>                   +(N mod p ne 0 select p else 0)*x^2 >;
> L := LSeries(2,[0,1],N,cf : Parent:=E, Sign:=RootNumber(E));
```

Compare this with the built-in function LSeries(E)

```
> Evaluate(L,2);
0.97743186689450050867903912764 7
> Evaluate(LSeries(E),2);
0.97743186689450050867903912764 7
```

127.9.2 Self-made Dedekind Zeta Function

Example H127E21

This is an example of how the general `LSeries` function may be used to define the Dedekind zeta function of a number field K. This is essentially what the function `LSeries(K)` does:

```
> function DedekindZeta(K)
>   M := MaximalOrder(K);
>   r1,r2 := Signature(K);
>   gamma := [0: k in [1..r1+r2]] cat [1: k in [1..r2]];
>   disc := Abs(Discriminant(M));
>   P<x> := PolynomialRing(Integers());
```

The coefficients are defined by means of local factors; for a prime p we take the product of $1 - x^{f_k}$ where the f_i are the residue degrees of primes above p. Note that this local factor has (maximal) degree $[K : \mathbf{Q}]$ if and only if p is unramified or, equivalently, if and only if p does not divide the discriminant of K which is (up to sign) the conductor of our zeta-function.

```
>   cf := func<p,d|&*[1-x^Degree(k[1]): k in Decomposition(M,p)]>;
```

Finally, the Dedekind zeta function has a pole at $s = 1$ and we need its residue (or, rather, the residue of $\zeta^*(s)$) which we compute using the class number formula.

```
>   h := #ClassGroup(M);
>   reg := Regulator(K);
>   mu := #TorsionSubgroup(UnitGroup(M));
>   return LSeries(1, gamma, disc, cf: Parent:=K, Sign:=1, Poles:=[1],
>     Residues := [-2^(r1+r2)*Pi(RealField())^(r2/2)*reg*h/mu]);
> end function;
> Z := DedekindZeta(CyclotomicField(5)); Z;
L-series of Cyclotomic Field of order 5 and degree 4
> Evaluate(Z,1);
L*(s) has a pole at s = 1
> L := Z/RiemannZeta();
> Evaluate(L,1);
0.33983727824052353546427878 1159
```

127.9.3 L-series of a Genus 2 Hyperelliptic Curve

Example H127E22

We compute an L-series of a hyperelliptic curve of genus 2.

```
> P<x> := PolynomialRing(Integers());
> C := HyperellipticCurve([x^5+x^4,x^3+x+1]); C;
Hyperelliptic Curve defined by y^2 + (x^3 + x + 1)*y = x^5 + x^4
```

```
over Rational Field
```

There is an L-series attached to $\mathrm{H}^1(C)$ or, equivalently, the H^1 of the Jacobian J of C. To define this L-series, we need its local factors which, for primes p of good reduction of C, are given by `EulerFactor(J,GF(p,1))`. Let us look at the primes of bad reduction:

```
> Abs(Discriminant(C));
169
> Factorization(Integers()!$1);
[ <13, 2> ]
```

There is one prime $p = 13$ where the curve has bad reduction. There the fibre is a singular curve whose normalization is elliptic.

```
> A<X,Y> := AffineSpace(GF(13,1),2);
> C13 := Curve(A,Equation(AffinePatch(C,1)));
> GeometricGenus(C), GeometricGenus(C13);
2 1
> SingularPoints(C13);
{@ (9, 1) @}
> p := $1[1]; IsNode(p), IsCusp(p);
false true
> C13A := Translation(C13,p)(C13);
> C13B := Blowup(C13A); C13B;
Curve over GF(13) defined by
X^3 + 12*X^2*Y + 7*X^2 + 12*X*Y + 12*Y^2 + 3*Y + 1
> E := EllipticCurve(ProjectiveClosure(C13B));
```

The local factor of C at $p = 13$ is given by the Euler factor of this elliptic curve

```
> EulerFactor(E);
13*x^2 + 5*x + 1
```

The conductor of $L(C, s)$ is 13^2, the same as the discriminant in this case. So, we define $L(C, s)$:

```
> J := Jacobian(C);
> loc := func<p,d|p eq 13 select EulerFactor(E) else EulerFactor(J, GF(p,1))>;
> L := LSeries(2,[0,0,1,1],13^2,loc);
```

Now we check the functional equation and compute a few L-values. (Most of the execution time will be spent generating coefficients, so the first call takes some time. After that, the computations are reasonably fast.)

```
> CheckFunctionalEquation(L);
-2.95822839457879427029398211981 0E-31
> Evaluate(L,1);
0.090490390832429629 1135897572580
> Evaluate(L,2);
0.364286342944364068154291450139
```

127.9.4 Experimental Mathematics for Small Conductor

Example H127E23

This example shows how one may construct the first 20 coefficients of the L-series of an elliptic curve of conductor 11 without knowing anything about either the curve or modular form theory. The point is that for an L-series $L(s) = \sum a_n/n^s$, the associated theta function (used in `CheckFunctionalEquation`) is a series with a_n as coefficients and terms that decrease very rapidly with n. This means that if we truncate the series to, for instance, just $a_1 + a_2/2^s$ and call `LSeries` with the same parameters (weight, sign, etc.) as those for $L(s)$ but just $[a_1, a_2, 0, 0, 0, 0, ...]$ as the coefficient vector, then `CheckFunctionalEquation` will return a number reasonably close to 0, because the coefficients a_n for $n > 2$ only make a small contribution.
With this in mind, we create an L-series that looks like that of an elliptic curve with conductor 11 and sign 1, that is, we take weight = 2, conductor = 1, gamma = [0,1], sign = 1 and no poles:

```
> L := LSeries(2, [0,1], 11, 0 : Sign:=1);
```

Then taking $a_1 = 1$, and recalling the requirement that all the coefficients must be weakly multiplicative, we try various a_2 to see which of the truncated L-series $1 + a_2/2^s$ is closest to satisfying a functional equation. Using the knowledge that a_2 (and every other a_n) is an integer together with the Hasse-Weil bound, $|a_p| < 2\sqrt{p}$, we find just 5 choices:

```
> for a_2 := -2 to 2 do
>   LSetCoefficients(L,[1,a_2]);
>   print a_2, CheckFunctionalEquation(L);
> end for;
-2 0.00844809870747809018757958838063 5
-1 0.07557498328782515608604001309880
0 0.14270186786817222198450043781 69
1 0.20982875244851928788296086253 50
2 0.27695563702886635378142128725 32
```

It seems that $a_2 = -2$ is the best choice, so we set it. Note that this also determines a_4, a_8, etc. We next proceed to find a_3 in the same way. It might come as a surprise, but in this way we can find the first 30 or so coefficients correctly. (Actually, by using careful bounds, it is even possible to prove that these are indeed uniquely determined.) Here is code that finds $a_1, ..., a_{20}$:

```
> N := LCfRequired(L); N;
48
> V := [0 : k in [1..N] ];   // keep a_p in here
> P<x> := PolynomialRing(Integers());
> function Try(V,p,a_p)  // set V[p]=a_p and try functional equation
>   V[p] := a_p;
>   LSetCoefficients(L, func<p,d | 1-V[p]*x+(p eq 11 select 0 else p)*x^2 >);
>   return Abs(CheckFunctionalEquation(L));
> end function;
> for p := 2 to 20 do     // try a_p in Hasse-Weil range and find best one
>   if IsPrime(p) then
>     hasse := Floor(2*Sqrt(p));
>     _,V[p] := Min([Try(V,p, a_p): a_p in [-hasse..hasse]]);
```

```
>     V[p] -:= hasse+1;
>   end if;
> end for;
> LSetCoefficients(L, func<p,d | 1-V[p]*x+(p eq 11 select 0 else p)*x^2 >);
```

We list the coefficients that we found and apply CheckFunctionalEquation to them:

```
> LGetCoefficients(L,20);
[* 1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2*]
> CheckFunctionalEquation(L);
4.1865797906741233744189856688l6E-16
```

Compare this with the actual truth:

```
> qExpansion(Newforms("11A")[1],21);
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 - 2*q^9 - 2*q^10 + q^11
    - 2*q^12 + 4*q^13 + 4*q^14 - q^15 - 4*q^16 - 2*q^17 + 4*q^18 +
    2*q^20 + O(q^21)
```

Such methods have been used by Stark in his experiments with L-functions of number fields, and by Mestre to find restrictions on possible conductors of elliptic curves. In fact, by modifying our example (changing 11 to $1 \ldots 10$ and sign $= 1$ to sign $= \pm 1$) one can show that for $N < 11$ there are no modular elliptic curves of conductor N.

127.9.5 Tensor Product of L-series Coming from l-adic Representations

Example H127E24

This is an example of using the tensor product for L-functions coming from l-adic representations. This is what `LSeries(E,K)` uses to construct L-series of elliptic curves over number fields.

```
> E := EllipticCurve([ 0, 0, 0, 0, 1]); // Mordell curve
> P<x> := PolynomialRing(Integers());
> K := NumberField(x^3-2);
> LE := LSeries(E);
> LK := LSeries(K);
```

We have now two L-functions, one associated to an elliptic curve and one to a number field. They both come from l-adic representations, and we can try to construct their tensor product. Actually, the function `TensorProduct` requires us to specify exponents of the conductor and the bad local factors of the tensor product representation, but we can try and let MAGMA do it, hoping that nothing unusual happens. We take the tensor product of the two L-series and divide it by $L(E, s)$ to speed up the computations. Remember that $L(K, s)$ has a copy of the Riemann zeta function as a factor.

```
> L := TensorProduct(LE, LK, []) / LE;
> CheckFunctionalEquation(L);
  |Sign| is far from 1, wrong functional equation?
```

```
0.023406257100607511430153563640 1
```

The resulting L-function does not satisfy the required functional equation, so something unusual does happen at a prime where both E and K have bad reduction, which in this case must be either $p = 2$ or $p = 3$.
Let us check $p = 2$. We change the base field of E to K and look at its reduction at the unique prime above 2:

```
> EK := BaseChange(E,K);
> p := Decomposition(MaximalOrder(K),2)[1,1];
> LocalInformation(E,2);
<2, 4, 2, 3, IV>
> loc, model:=LocalInformation(EK,p);loc,model;
<Principal Prime Ideal
Generator:
    [0, -1, 0], 0, 0, 1, I0>
Elliptic Curve defined by y^2 - y = x^3 over K
```

We see that E has acquired good reduction at $p|2$. This means that the inertia invariants at 2 of the l-adic representation associated to L, namely $\rho_E \otimes (\rho_K - \rho_{\mathbf{Q}})$ are *larger* than the tensor product of the corresponding inertia invariants, which is zero. Thus the local factor of L at 2 is not $F_p(x) = 1$ (which is what `TensorProduct` assumed), but a polynomial of higher degree. In fact, it is the characteristic polynomial of Frobenius of the reduced curve E/K mod p, so $F_p(x) = 1 - a_2x + 2x^2$ with a_2 given by

```
> TraceOfFrobenius(Reduction(model, p));
0
```

This is now the correct L-function (at $p = 3$ the default procedure works) and the functional equation is satisfied

```
> L := TensorProduct(LE,LK,[<2,4,1+2*x^2>])/LE;
> CheckFunctionalEquation(L);
3.155443620884047221646914261 13E-30
```

In fact, our L-series is the same as that constructed by `LSeries(E,K)/LE`.

127.9.6 Non-abelian Twist of an Elliptic Curve

Example H127E25

In this example we illustrate how to use `LSeries(E,K)` to construct non-abelian twists of elliptic curves.

```
> E := EllipticCurve([0, 0, 0, 0, 1]);
> P<x> := PolynomialRing(Integers());
> K := NumberField(x^3-2);
> L := LSeries(E,K) / LSeries(E);
> lval := Evaluate(L, 1); lval;
```

```
0.65537155216497443571237884249 1
```

The series L is the L-function of non-abelian twist of E and it appears that $L(1)$ is non-zero. In other words, the analytic rank of $L(E/K, s)$ at $s = 1$ is the same as that of $L(E/\mathbf{Q}, s)$. Tate's version of the Birch-Swinnerton-Dyer conjecture then predicts that the Mordell-Weil rank of E/K is the same as the rank over $\mathbf{Q}$, so the elliptic curve has not acquired new independent points. This is something that can be confirmed by a Selmer group computation:

```
> EK := BaseChange(E,K);
> twoE := MultiplicationByMMap(E, 2);
> #SelmerGroup(twoE);
2
> twoEK := MultiplicationByMMap(EK, 2);
> #SelmerGroup(twoEK);
2
```

Indeed, the 2-Selmer rank is 1 in both cases, as expected. Also, the second part of the Birch-Swinnerton-Dyer conjecture predicts that the following quotient should be rational:

```
> p1, p2 := Explode(Periods(E));
> lval*Sqrt(Abs(Discriminant(K))) / (p1*Im(p2));
1.33333333333333333333333333334
```

Finally, the same L-series can be defined via the Artin representation machinery:

```
> triv,sign,rho:=Explode(ArtinRepresentations(K));
> L:=LSeries(E,rho);
> CentralValue(L);
0.65537155216497443571237884249 2
```

127.9.7 Other Tensor Products

We take two L-functions with Euler products over a field K:

$$L_1(M,s) = \prod_{\mathfrak{p}} \prod_{i=1}^{m} \big(1 - \alpha_i(\mathfrak{p})/\mathrm{N}\mathfrak{p}^s\big)^{-1} \quad \text{and} \quad L_2(N,s) = \prod_{\mathfrak{p}} \prod_{j=1}^{n} \big(1 - \beta_j(\mathfrak{p})/\mathrm{N}\mathfrak{p}^s\big)^{-1},$$

where M and N are of degrees m and n respectively, and we ignore bad Euler factors in the above. Their tensor product is an L-function of degree mn given by

$$L(A \otimes B, s) = \prod_{\mathfrak{p}} \prod_{i=1}^{m} \prod_{j=1}^{n} \big(1 - \alpha_i(\mathfrak{p})\beta_j(\mathfrak{p})/\mathrm{N}\mathfrak{p}^s\big)^{-1}.$$

It is not clear in general how to compute bad Euler factors, and even the gamma factors can be tricky. Internally, the functionality tries to compute a "Hodge structure" corresponding to the gamma factors of each object, with the "Hodge structure" for the tensor product then following via combinatorics, before passing back to the gamma factors.

Some tensor products can induce poles, particularly when tensoring an L-series with itself. Also in some cases, the tensor product will be taken over $\mathbf{Q}$ unless the user specifies otherwise.

Example H127E26

A tensor product of two elliptic curves over $\mathbf{Q}$.

```
> E1 := EllipticCurve("11a");
> E2 := EllipticCurve("17a");
> L1 := LSeries(E1);
> L2 := LSeries(E2);
> L := TensorProduct(L1, L2, []);
> LSeriesData(L); // level is 11^2 * 17^2
<3, [ 0, 1, -1, 0 ], 34969, ... >
> CheckFunctionalEquation(L);
-2.83989925879564249948222283502E-29
```

Example H127E27

A tensor product of two modular forms of level 1.

```
> f1 := ModularForms(1,12).2;
> f2 := ModularForms(1,26).2;
> L1 := LSeries(f1);
> L2 := LSeries(f2);
> L := TensorProduct(L1, L2, []);
> LSeriesData(L); // weight is (12-1)+(26-1)+1 -- motivic weight is 11+25
<37, [ 0, 1, -11, -10 ], 1, ... >
> CheckFunctionalEquation(L);
-5.75014913311889483177106362349E-25
> Pi(RealField(30))^2 * Evaluate(L,24) / Evaluate(L,25);
9.87142857142857142857142857136
> 691/70.; // Ramanujan congruence
9.87142857142857142857142857142
```

Example H127E28

An example related to Siegel modular forms (see [vGvS93, S8.7]).

```
> E := EllipticCurve("32a"); // congruent number curve
> chi := DirichletGroup(32).1; // character of conductor 4 lifted
> MF := ModularForms(chi, 3); // weight 3 modular forms on Gamma1(32,chi)
> NF := Newforms(MF);
> NF[1][1]; // q-expansion of the desired form
q + a*q^3 + 2*q^5 - 2*a*q^7 - 7*q^9 - a*q^11 + O(q^12)
> Parent(Coefficient(NF[1][1], 3)); // defined over Q(i)
Number Field with defining polynomial x^2 + 16 over the Rational Field
> f1, f2 := Explode(ComplexEmbeddings(NF[1][1])[1]);
> L1 := LSeries(E);
> L2 := LSeries(f1); // first complex embedding
> L := TensorProduct(L1, L2, [ <2, 9> ]); // conductor 2^9 (guessed)
> time CheckFunctionalEquation(L);
```

```
-3.1554436208840472216469142611 3E-30
Time: 2.090
```

Example H127E29

An example of a tensor product over a field larger than the rationals.

```
> K<s> := QuadraticField(-3);
> I := Factorization(3 * IntegerRing(K))[1][1];
> H := HeckeCharacterGroup(I^2);
> G := Grossencharacter(H.0, [[1, 0]]); // canonical character
> E := EllipticCurve([1, (3+s)/2, 0, (1+s)/2, 0]);
> Norm(Conductor(E));
73
> LG := LSeries(G);
> LE := LSeries(E);
> TP := TensorProduct(LE, LG, [<I, 5>], K); // ensure 3-part correct
> LSetPrecision(TP, 9); // there is another factor of 3 from K
> LCfRequired(TP);
1642
> CheckFunctionalEquation(TP);
4.65661287E-10
```

127.9.8 Symmetric Powers

Symmetric power L-functions form a natural analogue to tensor products. Here we essentially tensor an L-function with itself repeatedly, but remove redundant factors.

In the case of $GL(1)$, the kth symmetric power is simply the L-function associated to the kth power of the underlying character, though we must be careful to ensure primitivity (this disregards the bad primes). Explicitly, we have

$$L(\mathrm{Sym}^k \psi, s) = \prod_{\mathfrak{p}} \Big(1 - \psi(\mathfrak{p})^k / \mathrm{N}\mathfrak{p}^s\Big)^{-1},$$

(again ignoring bad primes) and so the eigenvalues are just the kth powers of the original. It is relatively easy to compute the bad Euler factors, given those for $L(\psi, s)$.

In the case of $GL(2)$, the kth symmetric power is an L-function of degree $(k+1)$ over the field of definition, given by

$$L(\mathrm{Sym}^k A, s) = \prod_{\mathfrak{p}} \prod_{i=0}^{k} \Big(1 - \alpha_1(\mathfrak{p})^{k-i} \alpha_2(\mathfrak{p})^i / \mathrm{N}\mathfrak{p}^s\Big)^{-1},$$

where $\alpha_1(\mathfrak{p})$ and $\alpha_2(\mathfrak{p})$ are the eigenvalues at the prime $\mathfrak{p}$. In the case of elliptic curves, one can use the fact that $\alpha_1(p)\alpha_2(p) = p$ to rewrite these via symmetric functions.

A similar definition can be made for higher degree L-functions, yielding that the kth symmetric power of an L-function of degree d will have degree $\binom{k+d-1}{d-1}$.

If the object itself has a natural powering operation (as with, say, Hecke characters), Magma will simply take the primitivization therein. The bad Euler factors have been explicitly calculated for elliptic curves over the rationals in [MW06] and [DMW09]. In other cases, the user will likely have to provide them. Furthermore, the full ability to take symmetric powers over fields other than the rationals is not yet fully implemented.

`SymmetricPower(L, m)`

`BadEulerFactors` SeqEnum *Default :* []

Return the L-series corresponding to the mth symmetric power of L. The `BadEulerFactors` consists of $\langle p, f, E\rangle$ triples, where p is a prime, f the conductor exponent, and E a polynomial that gives the Euler factor at that prime.

Example H127E30

Some basic code, showing the special cases where Magma will just take the power of the underlying object.

```
> G := FullDirichletGroup(3*5*7);
> chi := G.1*G.2*G.3; // Random now gives an imprimitive character
> L := LSeries(chi);
> LS3 := SymmetricPower(L, 3);
> Lc3 := LSeries(chi^3);
> Evaluate(LS3, 1);
0.84396449805350779437298478447 0 + 0.19923299213711678303364580375 3*i
> Evaluate(Lc3, 1);
0.84396449805350779437298478447 0 + 0.19923299213711678303364580375 3*i
```

Example H127E31

```
> K := QuadraticField(-23);
> I := Factorization(23 * IntegerRing(K))[1][1];
> G := HeckeCharacterGroup(I);
> psi := G.1^14; // Random now gives an imprimitive character
> L := LSeries(psi);
> LS5 := SymmetricPower(L, 5);
> Lp5 := LSeries(psi^5);
> Evaluate(LS5, 1);
0.87080188482438164758363114981 4 + 0.62258929156395483122972653081 3*i
> Evaluate(Lp5, 1);
0.87080188482438164758363114981 4 + 0.62258929156395483122972653081 3*i
```

Example H127E32

In this example we take symmetric powers of the L-function of a Grossencharacter.

```
> GR := Grossencharacter(psi, [[1,0]]);
> L := LSeries(GR);
> LS4 := SymmetricPower(L, 4);
> Lp4 := LSeries(GR^4);
> Evaluate(LS4, 3);
0.35465127571691531343026809804 2 - 0.14751987727745317365661305654 8*i
> Evaluate(Lp4, 3);
0.35465127571691531343026809804 2 - 0.14751987727745317365661305654 8*i
```

Example H127E33

An example with symmetric powers of elliptic curves. In the case of the symmetric square of 389A, the `ModularDegree` code will do the same calculation, but more efficiently.

```
> E := EllipticCurve("389a");
> L := LSeries(E);
> L2 := SymmetricPower(L, 2);
> LSeriesData(L2);
<3, [ 0, 1, 0 ], 389^2, function(p, d) ... end function, 1, [], []>
> LSetPrecision(L2, 9);
> Evaluate(L2, 2);
3.17231145
> ($1 * Conductor(E)) / (2 * Pi(RealField()) * FundamentalVolume(E));
40.0000000
> ModularDegree(E);
40
```

Example H127E34

This in an example where a higher symmetric power has a vanishing central value, here a zero of (presumably) order 4. It comes from work of Buhler, Schoen, and Top [BST97].

```
> E := EllipticCurve("73a");
> L := LSeries(E);
> L3 := SymmetricPower(L, 3);
> LSeriesData(L3); // Magma knows the Sign is +1
<4, [ 0, -1, 1, 0 ], 73^3, function(p, d) ... end function, 1, [], []>
> LSetPrecision(L3, 9);
> CentralValue(L3);
-2.16367048E-12
```

127.10 Weil Polynomials

The characteristic polynomial of the Frobenius on the etale cohomology is called a Weil polynomial. This section contains some auxiliary routines for simpler handling of these polynomials. All routines make frequently use of `PowerSumToCoefficients`, `CoefficientsToElementarySymmetric` and `ElementarySymmetricToPowerSums`.

`SetVerbose("WeilPolynomials", v)`

Set the verbose printing level for the analysis of weil polynomials. Maximal value 2.

`HasAllRootsOnUnitCircle(f)`

Given a polynomial f with rational coefficients this routine checks that the complex roots of the polynomial are all of absolute value 1. The algorithm does not use floating point approximations.

`FrobeniusTracesToWeilPolynomials(tr, q, i, deg)`

`KnownFactor`	RngUPolElt	*Default :* 1
Verbose	`WeilPolynomials`	*Maximum :* 2

Given the sequence of Frobenius traces on the i-th etale cohomology this function returns a list of possible characteristic polynomials. Here q is the order of the basefield and *deg* is the degree of the characteristic polynomial. In the case of even i the sign in the functional equation has to be determined. This is done by checking the absolute values of the roots of the hypothetical polynomials. In the case that this does not work both candidates are returned. In the case of contradictory data an empty sequence is returned.

If a factor of the characteristic polynomial is known it can be given as optional argument.

`WeilPolynomialToRankBound(f, q)`

Given a polynomial f this routine counts the zeros (with multiplicity) of f that are q times a root of unity. This is an upper bound for the Picard rank of the corresponding algebraic surface; according to the Tate conjecture, it is precisely the Picard rank.

`ArtinTateFormula(f, q, h20)`

Given a Weil polynomial corresponding to H^2 of an algebraic surface with Hodge number $h^{2,0}$ this routine evaluates the Artin-Tate formula. If the Tate conjecture holds, the two values returned are the arithmetic Picard rank, and the absolute value of the discriminant of the Picard group times the order of the Brauer group of the surface.

`WeilPolynomialOverFieldExtension(f, deg)`

Given the characteristic polynomial f of the Frobenius on an etale cohomology group. This routine returns the characteristic polynomial of the *deg* times iterated Frobenius.

`CheckWeilPolynomial(f, q, h20)`

`SurfDeg` RngIntElt *Default :* -1

Given a polynomial f this routine checks several conditions that must be satisfied by the characteristic polynomial of the Frobenius on H^2 of an algebraic surface. Here q is the size of the basefield, $h20$ is the Hodge number $h^{2,0}$ and `SurfDeg` is the degree of the surface (-1 means degree unknown). For example, this routine can be used to determine the sign in the functional equation.

The routine checks the valuation of the roots at all places (including p and ∞). It also checks the functional equation and the Artin-Tate conditions (see [EJ10] for details).

Example H127E35

```
> q1<t> := PolynomialRing(RationalField());
> z4<x,y,z,w> := PolynomialRing(IntegerRing(),4);
> f := x^4 + x^3*z + x^2*y^2 + x^2*y*w + x^2*z^2 + x^2*z*w +
>      x*y^3 + x*y*z*w + x*y*w^2 + x*z^2*w + y^3*w + y^2*z^2 +
>      y^2*z*w + y^2*w^2 + y*z^2*w + z^4 + z*w^3;
> Tr := [];
> for i := 1 to 10 do
>    P3 := ProjectiveSpace(GF(2^i),3);
>    S := Scheme(P3,f);
>    time
>    Tr[i] := #Points(S) - 1 - 2^(2*i);
> end for;
> cpl := FrobeniusTracesToWeilPolynomials(Tr, 2, 2, 22: KnownFactor := t-2);
> cpl;
[
    t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
         + 16384*t^7 - 32768*t^6 - 65536*t^5 + 262144*t^3
         + 524288*t^2 + 1048576*t - 4194304,
    t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
         - 16384*t^7 + 32768*t^6 + 65536*t^5 - 262144*t^3
         - 524288*t^2 - 1048576*t + 4194304
]
```

Here the sign in the functional equation can not be determined by the absolute values of roots test. Further point counting would be very slow, as the coefficients of t^{11}, t^{10}, t^9, t^8 are zero, thus point counting over $\mathbf{F}_{2^{15}}$ would be necessary to determine the sign in this way. Instead, we can try to use more invariants of the surface.

```
> cpl_2 := [wp : wp in cpl | CheckWeilPolynomial(wp,2,1: SurfDeg := 4)];
> cpl_2;
[
    t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
         + 16384*t^7 - 32768*t^6 - 65536*t^5 + 262144*t^3
         + 524288*t^2 + 1048576*t - 4194304
```

```
]
> WeilPolynomialToRankBound(cpl_2[1],2);
2
> ArtinTateFormula(cpl_2[1],2,1);
1 4
> ArtinTateFormula(WeilPolynomialOverFieldExtension(cpl_2[1],2),2^2,1);
2 68
```

Thus the minus sign in the functional equation is correct. The surface has arithmetic Picard rank 1 and relative to the Tate conjecture the geometric Picard group has rank 2 and discriminant either -17 or -68 (depending on the order of the Brauer group).
Advanced example: Bounding Picard rank with less point counting:
The computation is based on the same equation as the example above. Now we do not use the most costly Frobenius trace, but we assume additional cyclotomic factors.

```
> Tr := [ 1, 5, 19, 33, 11, -55, 435, -31, -197]; /* First 9 Frobenius traces */
> cycl := [ Evaluate(CyclotomicPolynomial(i),t/2)*2^EulerPhi(i)
>         : i in [1..100] | EulerPhi(i) lt 23];
> cycl[1] := cycl[1]^2;
> cycl[2] := cycl[2]^2; /* linear factors need multiplicity two */
> hyp := &cat [FrobeniusTracesToWeilPolynomials(Tr, 2, 2, 22 :
>                       KnownFactor := (t-2)*add) : add in cycl];
> hyp := SetToSequence(Set(hyp));
> hyp;
[
    t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
         - 16384*t^7 + 32768*t^6 + 65536*t^5 - 262144*t^3
         - 524288*t^2 - 1048576*t + 4194304
]
```

Using only the number of point over $\mathbf{F}_2, \ldots, \mathbf{F}_{2^9}$ we can prove that either the Picard rank is bounded by 2 or the correct Weil polynomial is in the list.

```
> CheckWeilPolynomial(hyp[1],2,1:SurfDeg := 4);
false
```

The invariants of the surface and the hypothetical Weil polynomial are contradictory. Thus the Picard rank is at most 2.

127.11 Bibliography

[**Arm71**] J. V. Armitage. Zeta functions with a zero at $s = 1/2$. *Invent. Math.*, 15(3):199–205, 1971.

[**BST97**] J. Buhler, C. Schoen, and J. Top. Cycles, L-functions and triple products of elliptic curves. *J. Reine. Angew. Math.*, 492:93–133, 1997.

[**Coh00**] Henri Cohen. *Advanced Topics in Computational Number Theory*. Springer, Berlin–Heidelberg–New York, 2000.

[**DMW09**] N. Dummigan, P. Martin, and M. Watkins. Euler factors and local root numbers for symmetric powers of elliptic curves. *Pure and Appl. Math. Qu.*, 5(4): 1311–1341, 2009.

[**Dok02**] Tim Dokchitser. ComputeL, pari package to compute motivic L-functions. URL:http://www.maths.dur.ac.uk/∼dma0td/computel/, 2002.

[**Dok04**] Tim Dokchitser. Computing special values of motivic L-functions. *Experiment. Math.*, 13(2):137–149, 2004.

[**EJ10**] Andreas-Stephan Elsenhans and Jörg Jahnel. Weil polynomials of K3 surfaces. In *Algorithmic number theory*, volume 6197 of *Lecture Notes in Computer Science*, pages 126–141, Berlin, 2010. Springer.

[**Fri76**] J. B. Friedlander. On the class numbers of certain quadratic extensions. *Acta Arith.*, 28(4):391–393, 1975/76.

[**HPP06**] F. Hess, S. Pauli, and M. Pohst, editors. *ANTS VII*, volume 4076 of *LNCS*. Springer-Verlag, 2006.

[**JKS94**] Uwe Jannsen, Steven Kleiman, and Jean-Pierre Serre, editors. *Motives*, volume 55 of *Proceedings of Symposia in Pure Mathematics*, Providence, RI, 1994. American Mathematical Society.

[**Lav67**] A. F. Lavrik. An approximate functional equation for the Hecke zeta-function of an imaginary quadratic field. *Mat. Zametki*, 2:475–482, 1967.

[**MW06**] P. Martin and M. Watkins. Symmetric powers of elliptic curve L-functions. In Hess et al. [HPP06], pages 377–392.

[**Ser65**] Jean-Pierre Serre. Zeta and L functions. In *Arithmetical Algebraic Geometry (Proc. Conf. Purdue Univ., 1963)*, pages 82–92. Harper & Row, New York, 1965.

[**Ser71**] J.-P. Serre. Conducteurs d'Artin des caractères réels. *Invent. Math.*, 14(3): 173–183, 1971.

[**Sha95**] I. R. Shafarevich, editor. *Number theory I*, volume 49 of *Encyclopaedia of Mathematical Sciences*. Springer-Verlag, Berlin, 1995. Fundamental problems, ideas and theories, A translation of *Number theory 1* (Russian), Akad. Nauk SSSR, Vsesoyuz. Inst. Nauchn. i Tekhn. Inform., Moscow, 1990, Translation edited by A. N. Parshin and I. R. Shafarevich.

[**Tol97**] Emmanuel Tollis. Zeros of Dedekind zeta functions in the critical strip. *Math. Comp.*, 66(219):1295–1321, 1997.

[**vGvS93**] B. van Geemen and D. van Straten. The cusp forms of weight 3 on $\Gamma_2(2, 4, 8)$. *Math. Comp.*, 61(204):849–872, 1993.

PART XVII
MODULAR ARITHMETIC GEOMETRY

128 MODULAR CURVES

Chapter 128
MODULAR CURVES

128.1 Introduction

Modular curves in MAGMA are a special type `CrvMod` of plane curve. A modular curve X is defined in terms of standard affine modular polynomials which are stored in precomputed databases. Those modular curves presently available are defined by bivariate polynomials relating the j-invariant and one of several standard functions on $X_0(N)$. These give singular models for $X_0(N)$ designed for computing isogenies of elliptic curves.

128.2 Creation Functions

Several different models for modular curves are available. The possible model types are "Atkin", "Canonical", and "Classical", each giving affine models defined by the modular polynomial databases of the same names. For more details on these polynomials see Section 128.4 on modular polynomials and databases.

128.2.1 Creation of a Modular Curve

`ModularCurve(X,t,N)`

Returns a model of the modular curve $X_0(N)$, in an affine plane specified by X. The string t must be one of "Atkin", "Canonical", or "Classical", with N a level in the corresponding modular curve database.

`ModularCurve(D, N)`

Returns an affine model of the modular curve $X_0(N)$ of level N from a database D of modular curves.

128.2.2 Creation of Points

Points on modular curves can be created in the same way as points on curves or schemes in general. In addition, there exist several specific constructors, defined in terms of the moduli structure, which take a parameterized elliptic curve as an argument.

`ModuliPoints(X,E)`

Given a modular curve $X = X_0(N)$ and an elliptic curve E, with compatible base rings, returns the sequence of points over the base field of E, corresponding to E with additional level structure.

Example H128E1

Below we give an example of the use of the moduli interpretation of modular curves in order to construct the corresponding subgroup scheme structures defined over a finite field.

```
> FF := FiniteField(NextPrime(10^6));
> A2 := AffineSpace(FF,2);
> X0 := ModularCurve(A2,"Canonical",17);
> E := EllipticCurve([FF|1,23]);
> mp := ModuliPoints(X0,E); mp;
[ (259805, 350390), (380571, 350390) ]
```

We will see later that it is possible to construct the structure of an elliptic curve parameterized by the corresponding moduli points.

```
> P, Q := Explode(mp);
> SubgroupScheme(E,P);
Subgroup of E defined by x^8 + 377217*x^7 + 190510*x^6 + 872850*x^5
  + 816054*x^4 + 457629*x^3 + 64955*x^2 + 361795*x + 460146
> SubgroupScheme(E,Q);
Subgroup of E defined by x^8 + 796070*x^7 + 308587*x^6 + 62023*x^5
  + 976430*x^4 + 380273*x^3 + 200328*x^2 + 892738*x + 536749
```

In this example we see that the prime 17 splits in the endomorphism ring of E, so we have exactly two parameterized isogenies defined over the base field of E.

128.3 Invariants

The defining data and invariants of the curves are accessed through standard functions.

`Level(X)`

Returns the level of the modular curve X as an integer.

`Genus(X)`

Returns the genus of the modular curve X .

`ModelType(X)`

Returns the type of the model for the modular curve X, presently limited to "Atkin", "Canonical", or "Classical". This is used to determine the algorithm by which parameterized isogenies are determined.

`Indices(X)`

Returns a sequence of integers, of the form $[N, M, P]$, classifying the class of the modular curve X. The integer sequence defines a congruence subgroup of $\mathrm{PSL}_2(\mathbf{Z})$ defined by

$$\Gamma(N, M, P) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \;\middle|\; c \equiv 0 \bmod N,\ b \equiv 0 \bmod P,\ a \equiv 1 \bmod M \right\},$$

where M divides $\mathrm{LCM}(N, P)$. For the models presently available, this will be the sequence $[N, 1, 1]$, where is the level of $X = X_0(N)$.

128.4 Modular Polynomial Databases

MAGMA contains several databases of standard defining polynomials for modular curves which are used throughout the system for the construction of isogenies of elliptic curves, and which are made available to the user. These define singular models for modular curves $X_0(N)$, in terms for standard functions on the curves.

The classical model for $X_0(N)$ is in terms of the polynomial $\Phi_N(X, Y)$ such that $\Phi_N(j(\tau), j(N\tau)) = 0$, where $j(\tau)$ is the j-function. The bivariate polynomial $\Phi_N(X, Y)$ is defined to be the *classical modular polynomial*. The classical modular polynomial has the property of being symmetric in X and Y, and moreover, the canonical involution is defined by $(X, Y) \mapsto (Y, X)$.

Suppose that N is a prime and set $s = 12/\text{GCD}(N-1, 12)$. Using the property that the Dedekind η-function is holomorphic with no zeros on the upper half plane $\mathbf{H}$, the function

$$f(\tau) = N^s \left(\frac{\eta(N\tau)}{\eta(\tau)} \right)^{2s},$$

is a holomorphic function whose reciprocal is also holomorphic on $\mathbf{H}$. The transformation properties of the η-function imply that $f(\tau)$ is invariant under $\Gamma_0(N)$. Together $j(\tau)$ and $f(\tau)$ generate the function field for $X_0(N)$. The polynomial $\Psi_N(X, Y)$ such that $\Psi_p(f(\tau), j(\tau)) = 0$ is called the *canonical modular polynomial*. The Atkin–Lehner involution sends $f(\tau)$ to $N^s/f(\tau)$, so that the relation $\Psi_N(N^s/f(\tau), j(N\tau)) = 0$ also holds.

The third distinguished class of modular polynomials are the *Atkin modular polynomials* $\Xi_N(X, Y)$, satisfying the property that $\Xi_N(f(\tau), j(\tau)) = 0$ for a modular function $f(\tau)$ on $X_0(N)$ invariant under the Atkin-Lehner involution. Since the function $f(\tau)$ is well-defined as a function on the Atkin-Lehner quotient curve $X_0^+(N)$, sometimes denoted $X_0^*(N)$, these polynomials are also referred to as *star* modular polynomials in the literature. The construction of $f(\tau)$ and the modular polynomial $\Xi_N(X, Y)$ is described in articles of Elkies [Elk98] and Morain [Mor95]. The database of Atkin modular polynomials were provided by A.O.L. Atkin.

`AtkinModularPolynomial(N)`

Given a prime number N, represented in the database of Atkin modular curves, this function returns the Atkin modular polynomial $\Xi_N(X, Y)$.

`CanonicalModularPolynomial(N)`

Given a prime number N represented in the database of canonical modular curves, this function returns the canonical modular polynomial $\Psi_N(X, Y)$.

`ClassicalModularPolynomial(N)`

Given an integer N represented in the database of classical modular curves, this function returns the defining classical modular polynomial $\Phi_N(X, Y)$.

`ModularCurveDatabase(t)`

`ModularCurveDatabase(t,i)`

Given an identifier string t, which must be one of "Atkin", "Canonical", or "Classical", this function returns the corresponding database object of modular curves $X_0(N)$.

Because of its size, the Atkin database is split into database objects of levels $200(i-1)+1 \leq N < 200i$ for i in $1, 2, 3, 4, 5$, of which only the first two are provided by default. Additional datafiles are available from the MAGMA website. The canonical database contains a subset of the curves for prime levels below 200, and only curves for levels below 60 are present in the classical modular curve database.

`N in D`

Returns `true` if and only if N is a level represented in the given modular curve database D.

`ExistsModularCurveDatabase(t)`

`ExistsModularCurveDatabase(t,i)`

Returns `true` if and only if the data file given by the string t and integer i exists

Example H128E2

Here we compare the defining polynomials for the Atkin, Canonical, and Classical models for the modular curves $X_0(N)$. For the modular curve $X_0(3)$ we list the corresponding polynomial.

```
> P2<x,j> := PolynomialRing(Integers(),2);
> P2!AtkinModularPolynomial(3);
x^4 - x^3*j + 744*x^3 + 193752*x^2 + 2348*x*j + 19712160*x + j^2
  + 24528*j + 538141968
> P2!CanonicalModularPolynomial(3);
x^4 + 36*x^3 + 270*x^2 - x*j + 756*x + 729
> P2!ClassicalModularPolynomial(3);
x^4 - x^3*j^3 + 2232*x^3*j^2 - 1069956*x^3*j + 36864000*x^3 +
    2232*x^2*j^3 + 2587918086*x^2*j^2 + 8900222976000*x^2*j +
    452984832000000*x^2 - 1069956*x*j^3 + 8900222976000*x*j^2 -
    770845966336000000*x*j + 1855425871872000000000*x + j^4 +
    36864000*j^3 + 452984832000000*j^2 + 1855425871872000000000*j
```

For larger values of N the Atkin modular polynomials tend to have smaller coefficients.

```
> P2!CanonicalModularPolynomial(11);
x^12 - 5940*x^11 + 14701434*x^10 - 139755*x^9*j - 19264518900*x^9 +
    723797800*x^8*j + 13849401061815*x^8 + 67496*x^7*j^2 -
    1327909897380*x^7*j - 4875351166521000*x^7 + 2291468355*x^6*j^2 +
    1036871615940600*x^6*j + 400050977713074380*x^6 - 5346*x^5*j^3 +
    4231762569540*x^5*j^2 - 310557763459301490*x^5*j +
    1224711544564336158000*x^5 + 161201040*x^4*j^3 +
    755793774757450*x^4*j^2 + 1730954664564250620 0*x^4*j +
    651339173406982403 1615*x^4 + 132*x^3*j^4 - 49836805205*x^3*j^3 +
```

```
    6941543075967060*x^3*j^2 - 6481517942976139866O*x^3*j +
    1042648844831301800367OO*x^3 + 468754*x^2*j^4 + 51801406800*x^2*j^3
    + 214437541826475*x^2*j^2 + 77380735840203400*x^2*j +
    804140494949359194*x^2 - x*j^5 + 3732*x*j^4 - 4586706*x*j^3 +
    2059075976*x*j^2 - 253478654715*x*j + 2067305393340*x + 1771561
> P2!CanonicalModularPolynomial(13);
x^14 + 26*x^13 + 325*x^12 + 2548*x^11 + 13832*x^10 + 54340*x^9 +
    157118*x^8 + 333580*x^7 + 509366*x^6 + 534820*x^5 + 354536*x^4 +
    124852*x^3 + 15145*x^2 - x*j + 746*x + 13
> P2!AtkinModularPolynomial(11);
x^12 - x^11*j + 744*x^11 + 196680*x^10 + 187*x^9*j + 21354080*x^9 +
    506*x^8*j + 830467440*x^8 - 11440*x^7*j + 16875327744*x^7 -
    57442*x^6*j + 208564958976*x^6 + 184184*x^5*j + 1678582287360*x^5 +
    1675784*x^4*j + 9031525113600*x^4 + 1867712*x^3*j +
    32349979904000*x^3 - 8252640*x^2*j + 74246810880000*x^2 -
    19849600*x*j + 98997734400000*x + j^2 - 8720000*j + 58411072000000
> P2!AtkinModularPolynomial(13);
x^14 - x^13*j + 744*x^13 + 196716*x^12 + 156*x^11*j + 21377304*x^11 +
    364*x^10*j + 835688022*x^10 - 8502*x^9*j + 17348897592*x^9 -
    37596*x^8*j + 224269358092*x^8 + 149786*x^7*j + 1949972557416*x^7 +
    1161420*x^6*j + 11858099339697*x^6 + 700323*x^5*j +
    51262531538400*x^5 - 9614956*x^4*j + 157275877324800*x^4 -
    23669490*x^3*j + 335383326720000*x^3 - 5859360*x^2*j +
    473336381440000*x^2 + 32384000*x*j + 397934592000000*x + j^2 +
    24576000*j + 150994944000000
```

In general the Atkin modular polynomials have coefficients of a smaller size when $N \equiv 11 \bmod 12$ and are largest when $N \equiv 1 \bmod 13$, while the opposite is true for the canonical modular polynomials.

128.5 Parametrized Structures

The modular curves $X_0(N)$ parametrize elliptic curves together with the structure of a cyclic isogeny, or equivalently, a cyclic subgroup scheme of the N-torsion of the elliptic curve. over the modular curve, singularities of the chosen surface may obstruct the construction of the corresponding isogeny.

`Isogeny(E,P)`

Given an elliptic curve E and a point P on some $X_0(N)$ corresponding to a cyclic level structure on E, the function returns an isogeny $f : E \to F$ corresponding to P. The isogeny is defined by the formulae of Velu in such a way that the pull-back of the invariant differential of F is the invariant differential on E.

`SubgroupScheme(E,P)`

Given an elliptic curve E and a point P on some $X_0(N)$ corresponding to a cyclic level structure on E, the function returns the subgroup scheme of E parameterized by $X_0(N)$.

Example H128E3

In this example we create the function field of the modular curve and compute the corresponding parameterized subgroup scheme for the canonical model of the modular curve $X_0(7)$.

```
> A2 := AffineSpace(RationalField(),2);
> X0 := ModularCurve(A2,"Canonical",7);
> K0<u,j> := FunctionField(X0);
> j;
(u^8 + 28*u^7 + 322*u^6 + 1904*u^5 + 5915*u^4 + 8624*u^3 + 4018*u^2 +
   748*u + 49)/u
```

We can now create an elliptic curve with this j-invariant and compute the corresponding subgroup scheme.

```
> E := EllipticCurveFromjInvariant(j);
> E;
Elliptic Curve defined by y^2 + x*y = x^3 - 36*u/(u^8 + 28*u^7 + 322*u^6 +
 1904*u^5 + 5915*u^4 + 8624*u^3 + 4018*u^2 - 980*u + 49)*x - u/(u^8 +
 28*u^7 + 322*u^6 + 1904*u^5 + 5915*u^4 + 8624*u^3 + 4018*u^2 - 980*u + 49)
 over Function Field of Modular Curve over Rational Field defined by
 $.1^8 + 28*$.1^7*$.3 + 322*$.1^6*$.3^2 + 1904*$.1^5*$.3^3 +
 5915*$.1^4*$.3^4 + 8624*$.1^3*$.3^5 + 4018*$.1^2*$.3^6 - $.1*$.2*$.3^6 +
 748*$.1*$.3^7 + 49*$.3^8
> ModuliPoints(X0,E);
[ (u, (u^8 + 28*u^7 + 322*u^6 + 1904*u^5 + 5915*u^4 + 8624*u^3 +
  4018*u^2 + 748*u + 49)/u) ]
> P := $1[1];
> SubgroupScheme(E,P);
Subgroup of E defined by x^3 + (-u^3 - 13*u^2 - 47*u - 14)/(u^4 + 14*u^3 +
 63*u^2 + 70*u - 7)*x^2 + (u^5 + 19*u^4 + 133*u^3 + 373*u^2 + 271*u + 49)/
 (u^8 + 28*u^7 + 322*u^6 + 1904*u^5 + 5915*u^4 + 8624*u^3 + 4018*u^2 -
 980*u + 49)*x + (-u^6 - 21*u^5 - 166*u^4 - 569*u^3 - 750*u^2 - 349*u -
 49)/(u^12 + 42*u^11 + 777*u^10 + 8246*u^9 + 54810*u^8 + 233730*u^7 +
 628425*u^6 + 999306*u^5 + 801738*u^4 + 159838*u^3 - 93639*u^2 +
 10290*u - 343)
```

We now replay the same computation for the Atkin model for $X_0(7)$.

```
> X0 := ModularCurve(A2,"Atkin",7);
> K0<u,j> := FunctionField(X0);
> j;
j
> E := EllipticCurveFromjInvariant(j);
> E;
Elliptic Curve defined by y^2 + x*y = x^3 + (36/(u^8 - 984*u^7 + 196476*u^6 +
 21843416*u^5 + 805505190*u^4 + 14493138072*u^3 + 138563855164*u^2 +
 677923505640*u + 1338887352609)*j + (-36*u^7 + 12852*u^5 + 51408*u^4 -
 1139292*u^3 - 7372512*u^2 + 6730380*u + 76688208)/(u^8 - 984*u^7 +
 196476*u^6 + 21843416*u^5 + 805505190*u^4 + 14493138072*u^3 +
 138563855164*u^2 + 677923505640*u + 1338887352609))*x + (1/(u^8 - 984*u^7 +
```

```
 196476*u^6 + 21843416*u^5 + 805505190*u^4 + 14493138072*u^3 +
 138563855164*u^2 + 677923505640*u + 1338887352609)*j + (-u^7 + 357*u^5 +
 1428*u^4 - 31647*u^3 - 204792*u^2 + 186955*u + 2130228)/(u^8 - 984*u^7 +
 196476*u^6 + 21843416*u^5 + 805505190*u^4 + 14493138072*u^3 +
 138563855164*u^2 + 677923505640*u + 1338887352609)) over
 Function Field of Modular Curve over Rational Field defined by
 $.1^8 - $.1^7*$.2 + 744*$.1^7*$.3 + 196476*$.1^6*$.3^2 +
 357*$.1^5*$.2*$.3^2 + 21226520*$.1^5*$.3^3 + 1428*$.1^4*$.2*$.3^3 +
 803037606*$.1^4*$.3^4 - 31647*$.1^3*$.2*$.3^4 + 14547824088*$.1^3*$.3^5 -
 204792*$.1^2*$.2*$.3^5 + 138917735740*$.1^2*$.3^6 + 186955*$.1*$.2*$.3^6 +
 $.2^2*$.3^6 + 677600447400*$.1*$.3^7 + 2128500*$.2*$.3^7 + 1335206318625*$.3^8
> P := X0![u,j];
> SubgroupScheme(E,P);
Subgroup of E defined by x^3 + ((-2*u^2 - 316*u - 2906)/(u^9 - 497*u^8 -
 20532*u^7 - 81388*u^6 + 4746950*u^5 + 56167290*u^4 - 1279028*u^3 -
 2650748588*u^2 - 11224313439*u - 8626202865)*j + (2*u^9 + 315*u^8 +
 2686*u^7 - 93700*u^6 - 1255594*u^5 + 3135966*u^4 + 104728146*u^3 +
 352853636*u^2 - 681445096*u - 3227101785)/(u^9 - 497*u^8 - 20532*u^7 -
 81388*u^6 + 4746950*u^5 + 56167290*u^4 - 1279028*u^3 - 2650748588*u^2 -
 11224313439*u - 8626202865))*x^2 + ((-u^6 - 262*u^5 - 16695*u^4 - 248404*u^3 -
 457567*u^2 + 11521914*u + 54297783)/(u^13 - 989*u^12 + 201198*u^11 +
 21056034*u^10 + 657230331*u^9 + 6165550233*u^8 - 88238124492*u^7 -
 2578695909108*u^6 - 20624257862361*u^5 + 1318238025445*u^4 +
 1038081350842750*u^3 + 6551865190346034*u^2 + 15514646620480317*u +
 9981405213700095)*j + (u^13 + 262*u^12 + 16338*u^11 + 153443*u^10 -
 5846023*u^9 - 115347903*u^8 + 30945748*u^7 + 15440847094*u^6 +
 109278156555*u^5 - 206898429120*u^4 - 5031013591446*u^3 -
 17024110451577*u^2 - 2556327655701*u + 47383402701465)/(u^13 - 989*u^12 +
 201198*u^11 + 21056034*u^10 + 657230331*u^9 + 6165550233*u^8 -
 88238124492*u^7 - 2578695909108*u^6 - 20624257862361*u^5 + 1318238025445*u^4 +
 1038081350842750*u^3 + 6551865190346034*u^2 + 15514646620480317*u +
 9981405213700095))*x + (-u^10 - 338*u^9 - 33893*u^8 - 1121560*u^7 -
 8257018*u^6 + 187293764*u^5 + 3504845638*u^4 + 15046974856*u^3 -
 62184511493*u^2 - 623227561058*u - 1183475711457)/(7*u^17 - 10367*u^16 +
 4654944*u^15 - 389781392*u^14 - 97999195364*u^13 - 5984563052076*u^12 -
 171524908893072*u^11 - 2216173007598816*u^10 + 4849421263003170*u^9 +
 61349830231624030*u^8 + 929601834094601 2480*u^7 + 5943878355618241 6176*u^6 -
 237426233800123901 96*u^5 - 323811129579449249 9900*u^4 -
 243531549841757418195 36*u^3 - 864749171915268570 48384*u^2 -
 146131978942592594496657*u - 808465974189161471 73495)*j +
 (u^17 + 338*u^16 + 33536*u^15 + 999466*u^14 - 4293800*u^13 - 625188410*u^12 -
 6912544240*u^11 + 82395591738*u^10 + 2064805256370*u^9 + 6482044820554*u^8 -
 152652278669056*u^7 - 1482689798210194*u^6 - 1214725952426000*u^5 +
 43848981757984690*u^4 + 215958476310275824*u^3 + 192371928062911406*u^2 -
 828594020518663419*u - 1409818017397793940)/(7*u^17 - 10367*u^16 + 4654944*u^15 -
 389781392*u^14 - 97999195364*u^13 - 5984563052076*u^12 - 171524908893072*u^11 -
 2216173007598816*u^10 + 4849421263003170*u^9 + 61349830231624030*u^8 +
 929601834094601 2480*u^7 + 5943878355618241 6176*u^6 - 237426233800123901 96*u^5 -
```

```
32381112957944924999000*u^4 - 24353154984175741819536*u^3 -
8647491719152685704838 4*u^2 - 14613197894259259449 6657*u -
8084659741891614717 3495)
```

128.6 Associated Structures

`FunctionField(X)`

Returns the function field of the modular curve X.

`jFunction(X)`

Given a modular curve X over a field, returns the j-invariant as a function on the curve.

`BaseCurve(X)`

Given one of the standard models X for the modular curve $X_0(N)$, returns the base model curve $X(1)$ and the morphism $\pi : X_0(N) \to X(1)$.

Example H128E4

In this example we demonstrate the relation of a modular curve with its base curve $X(1)$.

```
> D := ModularCurveDatabase("Atkin");
> X0 := ModularCurve(D,17);
> X1, pi := BaseCurve(X0);
```

The discriminants -4, -8, -16, -19, -43, and -67 are the class number 1 discriminants in which 17 is a split prime. We use this to construct the corresponding moduli points on the curve $X_0(17)$ and map these back down to the curve $X(1)$. Refer to Section 128.8 for a description of the function `HilbertClassPolynomial`.

```
> discs := [ -4, -8, -16, -19, -43, -67 ];
> jinvs := [ Roots(HilbertClassPolynomial(D))[1][1] : D in discs ];
> jinvs;
[ 1728, 8000, 287496, -884736, -884736000, -147197952000 ]
> pnts := &cat[ ModuliPoints(X0, EllipticCurveFromjInvariant(j))
>               : j in jinvs ];
> pnts;
[ (-2, 1728), (-3, 8000), (-4, 287496), (-1, -884736), (2, -884736000),
(7, -147197952000) ]
> [ pi(P) : P in pnts ];
[ (1728, 1728), (8000, 8000), (287496, 287496), (-884736, -884736),
(-884736000, -884736000), (-147197952000, -147197952000) ]
```

We note that $X(1)$ is defined to be the classical modular curve, defined by the diagonal image of the j-line in $\mathbf{P}^2$.

128.7 Automorphisms

`CanonicalInvolution(X)`

`AtkinLehnerInvolution(X,N)`

Given a projective modular curve $X = X_0(N)$, the function returns the Atkin-Lehner involution of the modular curve as a map of schemes. Currently, the only Atkin-Lehner involution returned is that for N equal to the level of X.

128.8 Class Polynomials

Class polynomials are invariants of elliptic curves with complex multiplication by an imaginary quadratic order of discriminant D. As such the Hilbert class polynomials can be interpreted as defining a subscheme or divisor on the modular curve $X(1) \cong \mathbf{P}^1$, while the Weber variants define a subscheme of a modular curve of higher level.

`HilbertClassPolynomial(D)`

Given a negative discriminant D, returns the Hilbert class polynomial, defined as the minimal polynomial of $j(\tau)$, where $\mathbf{Z}[\tau]$ is an imaginary quadratic order of discriminant D.

`WeberClassPolynomial(D)`

Given a negative discriminant D not congruent to 5 modulo 8, returns the Weber class polynomial, defined as the minimal polynomial of $f(\tau)$, where $\mathbf{Z}[\tau]$ is an imaginary quadratic order of discriminant D and f is a particular normalized Weber function generating the same class field as $j(\tau)$. The particular $f(\tau)$ used depends on whether D is odd ($\equiv 1 \bmod 8$) or even, in which case it also depends on $D/4 \bmod 8$, as well as whether 3 divides D or not. A root $f(\tau)$ of the Weber class polynomial is an algebraic integer (a unit in most cases and always a unit outside of 2) generating the ring class field related to the corresponding root $j(\tau)$ of the Hilbert class polynomial by an expression $j(\tau) = F(f(\tau))$. Here F is a rational function of the form $A(Bx^r + C)^3/x^r$ with $r|24$ and A,B,C rational integers which are positive or negative powers of 2. The function F is also returned. For example, if $D \equiv 1 \bmod 8$ then

$$j(\tau) = \frac{(f(\tau)^{24} - 16)^3}{f(\tau)^{24}},$$

where $\mathrm{GCD}(D, 3) = 1$, and

$$j(\tau) = \frac{(f(\tau)^{8} - 16)^3}{f(\tau)^{8}},$$

if 3 divides D and $f(\tau)$ is a unit.

In fact, $f(\tau)$ can only be a non-unit when $4|D$ and $D/4 \equiv 0, 4$ or $5 \bmod 8$. For further details, consult Yui and Zagier [YZ97] for the case of odd D and Schertz [Sch76] for the case of even D.

`WeberToHilbertClassPolynomial(f,D)`

`Al`	MonStgElt	*Default :* "*Roots*"

Given a negative discriminant D, and the corresponding Weber class polynomial f, returns the Hilbert class polynomial for D. The default algorithm, as specified by the `Al` parameter, is to compute complex approximations to the roots of the latter polynomial from approximations to the roots of the Weber polynomial and the rational function F linking the two. The other method is algebraic and uses resultants and the function F which was discussed in the description of the intrinsic `WeberClassPolynomial`.

Example H128E5

Class polynomials are typically used for constructing elliptic curves with a known endomorphism ring or known number of points over some finite field. The Weber (and other) variants of the class polynomials were introduced as a means of obtaining class invariants – defining the j-invariant of curves with given CM discriminant – with much smaller coefficients. In this example we give the classical example of $D = -71$, where the

```
> HilbertClassPolynomial(-71);
x^7 + 313645809715*x^6 - 3091990138604570*x^5 + 98394038810047812049302*x^4
- 823534263439730779968091389*x^3 + 513880036645397678032372632944 6*x^2 -
425319473946139603274605151187659*x + 737707086760731113357714241006081263
> WeberClassPolynomial(-71);
x^7 + x^6 - x^5 - x^4 - x^3 + x^2 + 2*x - 1
```

As indicated by the constant term -1, the roots of the WeberClassPolynomial are units in a particular ring class order.

128.9 Modular Curves and Quotients (Canonical Embeddings)

`ModularCurveQuotient(N,A)`

`Raw`	BoolElt	*Default :* `false`
`Reduce`	BoolElt	*Default :* `false`

Given a level N and a (possibly empty) set of Atkin-Lehner involutions represented as a sequence of integers A, this function computes a model for a quotient of $X_0(N)$ by A. When `Raw` is not set to `true`, an initial "semi-reduction" is not performed. If the `Reduce` option is `true`, then a complete `LLL`-reduction is performed on the resulting equations (this is impractical for genus greater than 50 or so). The returned curve can be: P^1 for a genus 0 curve, an elliptic or hyperelliptic curve, or the canonical embedding of the curve in P^{g-1} where g is the genus of the quotient curve (in this general case, the coordinates correspond to cusp forms invariant under the specified Atkin-Lehner involutions).

Example H128E6

We compute a model for $X_0(13 \cdot 29)$ quotiented by the Atkin-Lehner involutions w_{13} and w_{29}.

```
> C := X0NQuotient(13*29,[13,29]); C; // defined by cubics in P^4
Curve over Rational Field defined by
-x[1]*x[2]^2 + x[1]*x[2]*x[3] - x[2]^2*x[3] + x[1]*x[2]*x[4] +
    x[2]^2*x[4] + x[2]*x[4]^2 + x[1]*x[2]*x[5],
x[1]*x[2]^2 - x[2]^3 - x[2]^2*x[3] + x[1]*x[2]*x[5] + x[2]^2*x[5] +
    x[2]*x[4]*x[5],
x[1]*x[2]*x[3] - x[2]^2*x[3] - x[2]*x[3]^2 + x[1]*x[3]*x[5] +
    x[2]*x[3]*x[5] + x[3]*x[4]*x[5],
x[1]*x[2]*x[5] - x[2]^2*x[5] - x[2]*x[3]*x[5] + x[1]*x[5]^2 +
    x[2]*x[5]^2 + x[4]*x[5]^2,
-x[1]^2*x[2] + x[1]*x[2]^2 + x[1]*x[2]*x[3] - x[1]^2*x[5] -
    x[1]*x[2]*x[5] - x[1]*x[4]*x[5],
x[1]*x[2]*x[3] + x[2]^2*x[4] + x[2]*x[3]*x[4] + x[2]*x[4]*x[5] +
    x[3]*x[4]*x[5] + x[2]*x[5]^2 + x[4]*x[5]^2,
x[1]*x[2]*x[3] + x[1]*x[2]*x[4] + x[1]*x[2]*x[5] - x[1]*x[3]*x[5] -
    x[3]^2*x[5] + x[1]*x[4]*x[5] - x[2]*x[4]*x[5] - x[3]*x[4]*x[5] +
    x[1]*x[5]^2 + x[3]*x[5]^2,
-x[1]*x[2]*x[4] + x[1]*x[3]*x[4] - x[2]*x[3]*x[4] + x[1]*x[4]^2 +
    x[2]*x[4]^2 + x[4]^3 + x[1]*x[4]*x[5],
-x[1]*x[2]*x[3] + x[1]*x[3]^2 - x[2]*x[3]^2 + x[1]*x[3]*x[4] +
    x[2]*x[3]*x[4] + x[3]*x[4]^2 + x[1]*x[3]*x[5],
-x[1]*x[2]*x[5] + x[1]*x[3]*x[5] - x[2]*x[3]*x[5] + x[1]*x[4]*x[5] +
    x[2]*x[4]*x[5] + x[4]^2*x[5] + x[1]*x[5]^2,
x[1]*x[2]*x[4] - x[2]^2*x[4] - x[2]*x[3]*x[4] + x[1]*x[2]*x[5] -
    x[1]*x[3]*x[5] + x[2]*x[3]*x[5] - x[1]*x[5]^2,
-x[1]^2*x[3] - x[1]*x[2]*x[3] + x[1]*x[3]^2 - x[2]*x[3]^2 -
    x[1]*x[2]*x[4] - x[3]^2*x[4] - x[2]*x[4]^2 + x[1]*x[3]*x[5] -
    x[2]*x[4]*x[5] - x[4]^2*x[5] - x[1]*x[5]^2,
-x[1]^2*x[2] + x[1]*x[2]^2 + x[2]^2*x[3] + x[1]^2*x[4] -
    x[1]*x[3]*x[4] + x[2]*x[3]*x[4] + x[3]*x[4]^2 + x[1]^2*x[5] -
    x[1]*x[3]*x[5] - x[3]^2*x[5] - x[1]*x[4]*x[5] - x[3]*x[4]*x[5] +
    x[2]*x[5]^2 + x[3]*x[5]^2 + x[4]*x[5]^2,
-x[1]^2*x[2] + x[1]^2*x[3] - x[1]*x[2]*x[3] + x[1]^2*x[4] +
    x[1]*x[2]*x[4] + x[1]*x[4]^2 + x[1]^2*x[5],
x[1]^2*x[2] + x[1]^2*x[3] - x[1]*x[2]*x[3] + x[2]^2*x[3] - x[1]*x[3]^2
    - x[3]^3 - x[1]*x[2]*x[4] + x[1]*x[3]*x[4] - x[2]*x[3]*x[4] -
    x[3]^2*x[4] + x[1]^2*x[5] + x[1]*x[3]*x[5] - x[2]*x[3]*x[5] +
    x[3]^2*x[5] - x[3]*x[4]*x[5] + x[1]*x[5]^2
> Genus(C);
5
```

128.10 Modular Curves of Given Level and Genus

The intrinsics described in this section are designed to construct curves C over $\mathbf{Q}$ of genus at least 2 which are images of $X_1(N)$ (or $X_0(N)$) under a morphism $\pi : X_1(N) \to C$ defined over $\mathbf{Q}$. Such curves will be referred to as modular curves. The code uses some of the ideas and methods from the papers [GJG03], [BGJGP05] and [GJO10].

The morphism π induces an isogeny π^* from the Jacobian of C, $Jac(C)$ onto a $\mathbf{Q}$-rational abelian subvariety B of $J_1(N)$ (or $J_0(N)$), i.e. a $\mathbf{Q}$-rational modular abelian variety of dimension g, the genus of C. The space of holomorphic differentials of $Jac(C)$ identified with those of C pull back under π^* to the holomorphic differentials of B which can be identified with the space of weight 2 cusp forms associated to B.

If $f_1, \ldots, f_g$ is a $\mathbf{Q}$-basis of this space, then these forms will satisfy the canonical relations for some canonical embedding of C in the non-hyperelliptic case. Conversely, if a basis of the space of cusp forms for a $\mathbf{Q}$-rational modular abelian variety B of dimension g satisfies the polynomial relations defining a canonical curve C of genus g, then C is a modular curve, but the pullback of the space of holomorphic differentials under the obvious map $\pi : z \mapsto [f_1(z) : f_2(z) : \ldots : f_g(z)]$ from $X_1(N)$ to C may NOT give the space of differentials $\langle f_1, \ldots, f_g \rangle$ and $Jac(C)$ may be isogenous to a different modular abelian subvariety of $J_1(N)$ to B.

In the hyperelliptic case, criteria for the pullback of the differentials of the curve C to give precisely the space of forms of a particular modular abelian variety B are given in [GJG03] and [BGJGP05], provided that B lies in the new part of $J_1(N)$. In the genus 3 non-hyperelliptic case, a simple explicit criterion for $Jac(C)$ to pull back to B is given in [GJO10].

Modular curves C such that the pullback $\pi^*(Jac(C))$ lies in the *new* part of $J_1(N)$ (or $J_0(N)$) are referred to as *new* modular curves.

Intrinsics are provided that determine all of the new modular hyperelliptic curves or all of the new modular genus 3 non-hyperelliptic curves of $X_1(N)$ (or $X_0(N)$) for a given level N. There are also intrinsics that determine whether a given modular abelian subvariety B corresponds to a hyperelliptic or genus 3 non-hyperelliptic curve C and give the equations of C (as a Weierstrass model or canonical model, respectively) in the affirmative case.

In future releases it is planned to add intrinsics to deal with non-hyperelliptic modular curves of genus greater than 3 and to add more functionality to cover non-new cases.

`SetVerbose("ModularCurve", v)`

Set the printing level for verbose output for the following intrinsics. Currently the legal values for v are `true`, `false`, 0, 1, 2 (`false` is the same as 0, and `true` is the same as 1).

`NewModularHyperellipticCurves(N, g)`

`check`	BoolElt	*Default :* `false`
`prec`	RngIntElt	*Default :* 100
`gamma`	RngIntElt	*Default :* 1

For level N, this function returns a list of all hyperelliptic curves of genus $g \geq 2$ which are new modular curves for $X_1(N)$ if parameter `gamma` equals 1 (the default) or $X_0(N)$, if `gamma` equals 0.

A hyperelliptic curve is returned as a univariate polynomial $f(x)$ such that $y^2 = f(x)$ is a Weierstrass equation of the actual curve.

The parameter `prec` (default 100) is the precision to which modular forms are expanded in the computations. If parameter `check` (default `false`) is `true`, the forms are actually computed to a precision slightly in excess of the precision $prec_1$ needed to guarantee that polynomial relations on them (of the degrees that are used in the computations) vanish if they vanish to precision $prec_1$.

`NewModularHyperellipticCurve(B)`

`check`	BoolElt	*Default :* `false`
`prec`	RngIntElt	*Default :* 100
`gamma`	RngIntElt	*Default :* 1

Given a sequence B of distinct modular abelian subvarieties of $J_1(N)^{new}$, presented as subspaces of modular symbols, this function `true` if the modular abelian variety M which is the direct sum of the B corresponds exactly to a hyperelliptic curve C as described in the introduction (M is isogenous to $Jac(C)$). If so, a univariate polynomial $f(x)$ is also returned such that $y^2 = f(x)$ is a Weierstrass equation for C.

The parameters `check` and `prec` have the same meaning as in the previous intrinsic. The parameter `gamma` is only relevant if `check` is `true`, is 1 by default, and should only be set to 0 if all of the B are abelian subvarieties of $J_0(N)$. It is then used to get sharper bounds for the precision required for the q-expansions.

`NewModularHyperellipticCurve(F)`

This is a variant on the intrinsic directly above where, instead of the sequence of modular abelian subvarieties as argument, the sequence of q-expansions of the basis of weight 2 forms for the modular abelian subvariety M is given instead.

`ModularHyperellipticCurve(B)`

`prec`	RngIntElt	*Default :* 100

Given a sequence B of distinct modular abelian subvarieties of $J_1(N)^{new}$, where the direct sum modular abelian variety M that B defines does not have to lie in the new part $J_1(N)^{new}$, the function determines whether the basis of the differentials (forms) of M satisfy the correct relations so as to arise from a hyperelliptic curve C of genus g. If so, the intrinsic also returns a univariate polynomial $f(x)$, such that $y^2 = f(x)$ is a Weierstrass equation for C. The difference between this and the new case is that, although C is a modular curve, it is not guaranteed that $Jac(C)$ is isogenous to M: it may be isogenous to a different modular abelian subvariety of $J_1(N)$.

The parameter `prec` (default 100) is the precision to which modular forms are expanded in the computations.

`ModularHyperellipticCurve(F)`

This is a variant on the intrinsic immediately above where, instead of a sequence of modular abelian varieties B with direct sum M passed as the argument, a sequence of q-expansions of the basis of weight 2 cusp forms of such an M is given instead.

`NewModularNonHyperellipticCurvesGenus3(N)`

`check`	BoolElt	*Default :* `false`
`prec`	RngIntElt	*Default :* 100
`gamma`	RngIntElt	*Default :* 1

Given an integer N, this function returns a list of all non-hyperelliptic curves of genus 3 which are new modular curves, for $X_1(N)$ if the parameter `gamma` equals 1 (the default), or for $X_0(N)$ if `gamma` equals 0.

The parameter `prec` (default 100) is the precision to which modular forms are computed to in the computations. If parameter `check` (default `false`) is `true`, the forms are actually computed to a little over the precision $prec_1$ needed to check that polynomial relations on them (of the degrees that are used in the computations) are definitely guaranteed to vanish if they vanish to precision $prec_1$.

`NewModularNonHyperellipticCurveGenus3(B)`

`check`	BoolElt	*Default :* `false`
`prec`	RngIntElt	*Default :* 100
`gamma`	RngIntElt	*Default :* 1

Given a sequence B of distinct modular abelian subvarieties of $J_1(N)^{new}$, where the direct sum modular abelian variety M that B defines does not have to lie in the new part $J_1(N)^{new}$, the function determines whether the basis of the differentials (forms) of M satisfy the correct relations so as to arise from a non-hyperelliptic curve C of genus 3. If so, the intrinsic also returns a a defining polynomial for the canonical image of C. The parameters `check` and `prec` have the same meaning as for the intrinsic `NewModularHyperellipticCurves`. The parameter `gamma`, which is only relevant if `check` is `true`, and is 1 by default, may be set to 0 if M is an abelian subvariety of $J_0(N)$ when sharper bounds for the required precision of q-expansions will be used. This intrinsic is similar to `NewModularHyperellipticCurve(B)`.

`NewModularNonHyperellipticCurveGenus3(F)`

This is a variant on the intrinsic `NewModularNonHyperellipticCurveGenus3(B)` above where, instead of the sequence of modular abelian varieties B with direct sum M passed as the argument, a sequence F of q-expansions of the basis of weight 2 cusp forms of such an M is given instead.

`ModularNonHyperellipticCurveGenus3(F)`

This is the same as the intrinsic `NewModularNonHyperellipticCurveGenus3(F)` immediately preceding except that it is not required that the modular abelian variety M which corresponds to F lies in the new part $J_1(N)^{new}$. This may be used to search for non-new non-hyperelliptic genus 3 curves.

Example H128E7

We give some examples of the use of these intrinsics.
The modular curve $X_1(13)$ is of genus 2, therefore hyperelliptic, and the space of modular forms of weight 2 for $\Gamma_1(13)$ is generated by an unique newform f of nebentype a Dirichlet character of order 6, such that the modular abelian variety attached to f, A_f, is $\mathbf{Q}$-isogenous to $J_1(13)$:

```
> chi := DirichletGroup(13,CyclotomicField(6)).1; //order 6 character mod 13
> A13:=Af(chi)[1];
> NewModularHyperellipticCurve([A13]);
true x^6 + 4*x^5 + 6*x^4 + 2*x^3 + x^2 + 2*x + 1
> f13:=qIntegralBasis(A13,100);
> NewModularHyperellipticCurve(f13);
true x^6 + 4*x^5 + 6*x^4 + 2*x^3 + x^2 + 2*x + 1
> SetVerbose("ModularCurve",1);
> NewModularHyperellipticCurve([A13]:check:=true);
Checking ...
          ... bound =113
true x^6 + 4*x^5 + 6*x^4 + 2*x^3 + x^2 + 2*x + 1
```

Let us compute all the new modular hyperelliptic curves parameterized by $X_0(80)$:

```
> NewModularHyperellipticCurves(80,0: gamma:=0);
     Candidates:=3
     All the curves
1 2 3
[
    x^5 + 2*x^4 - 26*x^3 - 132*x^2 - 231*x - 142
]
```

Finally, we calculate the new modular hyperelliptic curves parameterized by $X_1(80)$ (the default):

```
> SetVerbose("ModularCurve",0);
> NewModularHyperellipticCurves(80,0);
[
    x^5 + 2*x^4 - 26*x^3 - 132*x^2 - 231*x - 142,
    x^5 - 2*x^4 - 2*x^3 + 20*x^2 - 47*x + 30,
    x^7 - 4*x^6 - 4*x^5 + 39*x^4 - 64*x^3 + 40*x^2 - 8*x,
    x^7 + 2*x^5 + 7*x^4 - 4*x^3 - 20*x^2 - 16*x - 4
]
```

Example H128E8

In this exercise, an interesting example from [JK07] is presented.

```
> S:=CuspidalSubspace(ModularSymbolsH(21,[1,8,13,20],2,+1));
> S;
Modular symbols space of level 21, weight 2, and dimension 3 over Rational Field
(multi-character)
> ModularHyperellipticCurve([S]);
true x^8 - 6*x^6 + 4*x^5 + 11*x^4 - 24*x^3 + 22*x^2 - 8*x + 1
```

The above curve is the unique hyperelliptic intermediate modular curve $X_\Delta(N)$ between $X_1(N)$ and $X_0(N)$, where Δ is a subgroup of $(\mathbf{Z}/N\mathbf{Z})^*/\{\pm 1\}$ and $X_\Delta(N)$ is the modular curve associated to some congruence subgroup of $PSL_2(\mathbf{Z})$ attached to Δ. In fact, the above hyperelliptic curve is the new modular curve denoted by $C^A_{21A_{\{0,2\}}}$ in [BGJGP05].

Example H128E9

Given a modular abelian subvariety of $J_0(97)$, we determine whether it corresponds to a new modular non-hyperelliptic curve of genus 3 and level 97:

```
> M:=ModularSymbols(97,2,1);
> NN:=SortDecomposition(NewformDecomposition(NewSubspace(CuspidalSubspace(M))));
> NN;
[
    Modular symbols space for Gamma_0(97) of weight 2 and dimension 3 over
    Rational Field,
    Modular symbols space for Gamma_0(97) of weight 2 and dimension 4 over
    Rational Field
]
> A97:=NN[1];
> NewModularNonHyperellipticCurveGenus3([A97]);
true -x^2*y^2 + x*y^3 + x^3*z + x*y^2*z - 5*x^2*z^2 + 3*x*y*z^2 - 3*y^2*z^2 +
  6*x*z^3 - y*z^3 - 2*z^4
```

Example H128E10

We calculate all of the new modular non-hyperelliptic curves of genus 3 parameterized by $X_1(20)$:

```
> NewModularNonHyperellipticCurvesGenus3(20);
[
    -x^2*y^2 + x*y^3 + x^3*z - 3*x^2*z^2 + 4*x*z^3 - 2*z^4
]
```

We now construct an example of a non-new modular non-hyperelliptic curve of genus 3:

```
> S1:=ModularSymbols(178,2,1);
> N1:=SortDecomposition(NewformDecomposition(NewSubspace(CuspidalSubspace(S1))));
> A:=N1[3];A;
Modular symbols space for Gamma_0(178) of weight 2 and dimension 2 over
Rational Field
```

```
> S2:=ModularSymbols(89,2,1);
> N2:=SortDecomposition(NewformDecomposition(NewSubspace(CuspidalSubspace(S2))));
> B:=N2[1];B;
Modular symbols space for Gamma_0(89) of weight 2 and dimension 1 over
Rational Field
> fA:=qIntegralBasis(A,100);
> fB:=qIntegralBasis(B,100);
> q := Universe(fB).1;
> fB2:=&+[Coefficient(fB[1],k)*q^(2*k) : k in [1..99]];
> g:=fB[1]+2*fB2;
> ModularNonHyperellipticCurveGenus3([g,fA[1],fA[2]]);
true -x^4 + 2*x^2*y^2 - y^4 + 8*x^2*y*z + 8*y^3*z - 6*x^2*z^2 - 38*y^2*z^2 +
    24*y*z^3 + 7*z^4
>
```

The above curve is denoted by C_{178C}^{89A} in [GJO10]. Its jacobian is $\mathbf{Q}$-isogenous to $A \times B$.

128.11 Bibliography

[**BGJGP05**] M. Baker, E. Gonzalez-Jimenez, J. Gonzalez, and B. Poonen. Finiteness results for modular curves of genus at least 2. *Amer. J. Math.*, 127:1325–1387, 2005.

[**Elk98**] N. Elkies. Elliptic and modular curves over finite fields and related computational issues. In *Computational Perspectives on Number Theory A conference in honor of A.O.L. Atkin*, 1998.

[**GJG03**] E. Gonzalez-Jimenez and J. Gonzalez. Modular curves of genus 2. *Math. Comp.*, 72:397–418, 2003.

[**GJO10**] E. Gonzalez-Jimenez and Roger Oyono. Non-hyperelliptic modular curves of genus 3. *J. Number Th.*, 130:862–878, 2010.

[**JK07**] D. Jeon and C. H. Kim. On the arithmetic of certain modular curves. *Acta Arith.*, 130:181–193, 2007.

[**Mor95**] F. Morain. Calcul du nombre de points sur une courbe elliptique dans un corps fini: aspects algorithmiques. *J. Théorie des Nombres de Bordeaux*, 7:255–282, 1995.

[**Sch76**] R. Schertz. Die singularen Werte der Weberschen Funktionen $\mathbf{f}$,$\mathbf{f}_1$,$\mathbf{f}_2$,γ_2,γ_3. *J. reine angew. Math.*, 286/287:46–74, 1976.

[**YZ97**] N. Yui and D. Zagier. On the singular values of Weber modular functions. *Mathematics of Computation*, 66(220):1645–1662, 1997.

129 SMALL MODULAR CURVES

Chapter 129
SMALL MODULAR CURVES

129.1 Introduction

The Small Modular Curve database is a database of simple models over the rational numbers for the $X_0(N)$ modular curves along with the standard automorphisms, cusps and non-cuspidal rational points in the positive genus cases, the various projection maps from $X_0(N) \to X_0(M)$ when M divides N and expressions for standard functions $(j(z), j(Nz))$ and forms (e.g. E_4, E_6) in terms of rational functions or k-differentials on the model. We refer to them as "small", because the models are of low degree, are defined by reasonably sparse polynomials with small integer coefficients, and are non-singular or have only a few simple singularities.

Because the MAGMA type of the model may be `CrvEll`, `CrvHyp`, `CrvPln` or just general `Crv` and it is not possible for a type to extend different subtypes of `Crv`, we have not introduced a special type like `CrvMod` (or used this type) for small modular curves. The modular curve component of the arguments to most of the intrinsics consists of a projective curve, which should be a base change of the model over $\mathbf{Q}$ to a field of characteristic zero, and the level N. This is a little "clunky" but hopefully the user won't find it too awkward. For efficiency, there is currently no initial check that the curve argument really is a base change of the database model, which the user can obtain via the `SmallModularCurve` call. The call will almost certainly result in an error if this is not the case.

For simplicity, in the initial implementation, the intrinsics require the modular curve to be defined over a field of characteristic zero, although the models are all defined by polynomials with integral coefficients which reduce nicely for primes not dividing the level. Again, we do not check this condition in all intrinsics: the user may find that if he carries out a mod p reduction on the database model himself and uses it as the argument to some of the intrinsics, they may still work.

129.2 Small Modular Curve Models

The models are projective models of the complete curve $X_0(N)$ over $\mathbf{Q}$. In the genus 0 case, we simple take the projective line $\mathbf{P}^1$ with the point at infinity corresponding to the cusp at infinity ∞. In the genus 1 case, we take a `CrvEll` which is the standard minimal Weierstrass model of the elliptic curve $(X_0(N), \infty)$ with the cusp ∞ as the 0 point for the group law. In the hyperelliptic cases (see [Ogg74]), we take a minimal Weierstrass model with two rational points at infinity, the cusp ∞ (which is never a hyperelliptic Weierstrass point) and its image under the hyperelliptic involution.

In all other cases (non-subhyperelliptic), we have followed the rule of taking a plane model (in $\mathbf{P}^2$) or a non-singular model in $\mathbf{P}^3$. In fact, we have only used $\mathbf{P}^3$ models in the genus 4 cases. For genus 3 and 4, we take a canonical model (non-singular plane quartic

and non-singular complete intersection of a quadric and a cubic respectively). For genus 5 and 6, we find a smallest degree singular birational plane model. These plane curves are of degree 6 in all cases. The singularities are mainly nodes (type A_2) and simple cusps (type A_3) and often under cuspidal points, though there are some more complex ones of higher A_n type. All of the non-singular models that we have produced reduce mod p to non-singular models of the reduction of $X_0(N)$ for p not dividing the level N. The singular plane models reduce mod p (p not dividing N again) to singular plane models of the reduction of $X_0(N)$ with singularities of the same type, except for a few cases with small p where nodes become cusps or two singularities coalesce into a more complex one.

The initial version of the database contains data for all subhyperelliptic $X_0(N)$ and all other cases with genus ≤ 6 (with a few genus 5 and 6 cases not yet added). This covers all $N < 60$ along with about half of the N between 60 and 80 and $N = 81, 121$.

We briefly indicate how the models were arrived at. Equations in the genus 0 and elliptic cases and modular functions giving the coordinate generators are reasonably well-known (see [Lig75] for the elliptic cases), though we found them again anyway as part of our general procedure of searching for small degree rational functions on $X_0(N)$. We considered functions generated by Dedekind eta products and weight 2 integral forms coming from eta products, various types of theta series and Eisenstein series. For the hyperelliptic cases with genus g, functions x and y with poles at ∞ giving a $y^2 = f(x)$ type Weierstrass equation are determined from constructing weight two cusp forms G and F with q-expansions $q^{g-1} + ..$ and $q^g + ..$ respectively. These were found in terms of eta products and theta forms and the results were checked against the output of MAGMA's modular forms package.

In the non-subhyperelliptic cases, we started from a canonical image simplified by applying LLL as output by `ModularCurveQuotient`, occasionally slightly adapting this to get good reduction at 2. In the genus 5 and 6 cases, we determined minimal degree (singular) plane models from these. Genus 5 is fairly straightforward. For the method to find degree 6 plane images in the genus 6 case, see [Hara] where the genus 5 case is also discussed.

Having determined a model as the image of the mapping $X_0(N)$ into $\mathbf{P}^{r-1}$ by $z \mapsto [f_1(z) : \ldots : f_r(z)]$, $r = 3$ or 4, we then found expressions for the weight 2 forms f_i of the type described above (eta products etc.). For more information on this, see the subsection 129.8 where we also give intrinsics to return a symbolic description of the construction of the generating modular functions/forms and to return q-expansions up to a desired precision. We wished to have concrete expressions for these functions/forms in terms of certain basic types, independent of the generic modular symbol method of generating q-expansions for bases of forms. It also allows for slightly faster reconstruction of q-expansions.

`SmallModularCurve(N)`

`SmallModularCurve(N,K)`

The first intrinsic returns the model for $X_0(N)$ over the rationals from the small modular curve database. The second returns the base change of this to K, which should be a characteristic zero field.

If there is no database entry yet for level N, a runtime error results.

`IsInSmallModularCurveDatabase(N)`

Returns whether or not there is a data for level N in the small modular curves database.

Example H129E1

```
> IsInSmallModularCurveDatabase(79);
false
> IsInSmallModularCurveDatabase(35);
true
> SmallModularCurve(35);
Hyperelliptic Curve defined by y^2 + (-x^4 - x^2 - 1)*y = -x^7 - 2*x^6 - x^5 -
    3*x^4 + x^3 - 2*x^2 + x over Rational Field
> C<x,y,z> := SmallModularCurve(63);
> C;
Curve over Rational Field defined by
x^5*y - 2*x^4*y^2 + 3*x^3*y^3 - 2*x^2*y^4 + x*y^5 - 2*x^3*z^3 + x^2*y*z^3 +
    x*y^2*z^3 - 2*y^3*z^3 + z^6
```

129.3 Projection Maps

The database contains information allowing the reconstruction of the standard projection maps between the models of the level N and level M curves for any M dividing N.

`ProjectionMap(CN,N,CM,M)`

CN and CM should be base changes to the same characteristic zero field K of the small modular database curves of levels N and M with $M|N$.

Returns the natural projection map $CN \to CM$ that corresponds to $z \mapsto z$ in terms of the upper half-plane quotient models and to $(E, C) \mapsto (E, (N/M)C)$ in the moduli space interpretation where non-cuspidal points correspond to isomorphism classes of an elliptic curves E with a cyclic subgroup C of order N.

`ProjectionMap(CN,N,CM,M,r)`

CN and CM should be base changes to the same characteristic zero field K of the small modular database curves of levels N and M with $M|N$ and r should be a positive integer divisor of N/M.

Returns the projection map $CN \to CM$ that corresponds to $z \mapsto rz$ in terms of the upper half-plane quotient models and to

$$(E, C) \mapsto (E/(N/r)C, (N/(Mr))C/(N/r)C)$$

in the moduli space interpretation where non-cuspidal points correspond to isomorphism classes of an elliptic curves E with a cyclic subgroup C of order N.

Example H129E2

We use a 3-projection (case $r = 3$ in the above) in our extended example at the end of the section. Here, we just give some simple examples of the calls.

```
> C63<x,y,z> := SmallModularCurve(63);
> C63;
Curve over Rational Field defined by
x^5*y - 2*x^4*y^2 + 3*x^3*y^3 - 2*x^2*y^4 + x*y^5 - 2*x^3*z^3 + x^2*y*z^3 +
    x*y^2*z^3 - 2*y^3*z^3 + z^6
> C21 := SmallModularCurve(21);
> C21;
Elliptic Curve defined by y^2 + x*y = x^3 - 4*x - 1 over Rational Field
> C3 := SmallModularCurve(3);
> C3;
Curve over Rational Field defined by
0
> ProjectionMap(C63,63,C21,21);
Mapping from: Crv: C63 to CrvEll: C21
with equations :
8*x^3*y - 6*x^2*y^2 + 6*x*y^3 + 2*y^4 - 7*x^3*z + 15*x^2*y*z - 10*x*y^2*z +
    4*y^3*z - 7*x*y*z^2 + 7*y^2*z^2 + 5*x*z^3 - 15*y*z^3 - 2*z^4
-7*x^4 + 17*x^3*y - 18*x^2*y^2 + 11*x*y^3 - y^4 + 3*x^2*y*z + 5*x*y^2*z +
    5*y^3*z + 21*x^2*z^2 - 28*x*y*z^2 + 7*y^2*z^2 + x*z^3 - 3*y*z^3 - 13*z^4
-4*x^3*y + 3*x^2*y^2 - 3*x*y^3 - y^4 + 3*x^2*y*z - 2*x*y^2*z - 2*y^3*z +
    7*x*y*z^2 - 7*y^2*z^2 + x*z^3 + 4*y*z^3 + z^4
> ProjectionMap(C63,63,C3,3,7); //7-projection
Mapping from: Crv: C63 to Crv: C3
Composition of Mapping from: Crv: C63 to Crv: C
with equations :
x^2 - x*y + y^2 + 2*x*z - y*z - 2*z^2
x^2 - x*y + y^2 - x*z + 2*y*z - 2*z^2
x^2 - x*y + y^2 - x*z - y*z + z^2 and
Mapping from: Crv: C63 to Curve over Rational Field defined by
0
with equations :
x^3 + 3*x^2*y - 3*x*y^2 + 4*y^3 - 3*x^2*z - 2*z^3
x^2*z and
Mapping from: Curve over Rational Field defined by
0 to Curve over Rational Field defined by
0
with equations :
27*$.2
$.1 and
Mapping from: Curve over Rational Field defined by
0 to Crv: C3
with equations :
$.1^3
```

```
$.1^2*$.2 + 9*$.1*$.2^2 + 27*$.2^3
```

129.4 Automorphisms

For modular curves $X_0(N)$, there is a finite group of automorphisms $B_0(N)$ which come from matrices acting on the complex upper half-plane. This group is isomorphic to $\mathrm{Nm}_{\mathrm{SL}_2(\mathbf{R})}(\Gamma_0(\mathrm{N}))/\Gamma_0(\mathrm{N})$ where $\mathrm{Nm}_{\mathrm{SL}_2(\mathbf{R})}(\Gamma_0(\mathrm{N}))$ is the normaliser of $\Gamma_0(N)$ in $\mathrm{SL}_2(\mathbf{R})$. It is generated by Atkin-Lehner involutions and, if 4 or 9 divides N, a transformation of the form $z \mapsto z + (1/r)$ for some $1 < r \,|\, 24$ (see [AL70] and also [Bar08] for the correct structure in all cases). We denote the latter transformations by S_r and write w_d for the d-th Atkin-Lehner involution ($d|N$, $(d, N/d) = 1$). These transformations have a natural modular interpretation.

A famous result of Kenku and Momose [KM88], completed by Elkies for $N = 63$ [Elk90], says that $B_0(N)$ gives the *full* group of algebraic automorphisms of $X_0(N)$ (in characteristic zero) when the genus of $X_0(N)$ is at least 2, except for $N = 37$ and 63. In fact, there is a slight error in their analysis, and $N = 108$ is also an exception [Harb]. In these three cases $B_0(N)$ is of index two in the full automorphism group $A_0(N)$ of $X_0(N)$.

We have precomputed and stored the w_d and S_r curve isomorphisms in the database. Let $g(N)$ denote the genus of $X_0(N)$. There are intrinsics to return these individual isomorphisms and a special intrinsic to compute and return the automorphism group $B_0(N)$, when $g \leq 1$, or $A_0(N)$, when $g \geq 2$, over various fields as a `GrpAutCrv`. We sometimes refer to the elements of these groups as modular automorphisms to emphasise that they are not all of the curve automorphisms in the $g(N) = 0$ or 1 cases (when the automorphism group is infinite).

Except in the $N = 108$ case, all automorphisms are defined over the cyclotomic field $\mathbf{Q}(\mu_r)$ where r is the largest divisor of 24 such that $r^2|N$ (for this field of definition, r can be replaced by $r/2$ if r is twice an odd number). Specifically, the w_d are all defined over $\mathbf{Q}$ and S_r has $\mathbf{Q}(\mu_r)$ as its field of definition.

The automorphism group intrinsics include in various ways an integer n that specifies a cyclotomic level and they return the subgroup of $A_0(N)$, or $B_0(N)$ for $g \leq 1$, containing all elements defined over $\mathbf{Q}(\mu_n)$. In all cases, the group of automorphism returned is a semi-direct product of groups, one for each prime p dividing N. For $p > 3$ this group is just C_2, generated by an Atkin-Lehner involution. For $p = 2$ or 3, if $p^2|N$ and n is appropriately divisible, the p-component is more complex. We build up the automorphism group in the various cases from the known structure. This is much faster than using the generic algorithm to compute it. The only real time goes into building up the total set of function field automorphisms (starting from the stored data for the generators) that is cached in the `GrpAutCrv` object.

`AtkinLehnerInvolution(CN,N,d)`

CN should be a base change of the small modular database curve of level N to a field of characteristic 0. d is a divisor of N with $(d, N/d) = 1$. Returns the Atkin-Lehner involution w_d as a scheme automorphism of CN. If the full automorphism group G of CN has already been computed and cached with CN (see the intrinsics below and the section in the Algebraic Curves chapter on automorphism groups) then the intrinsic will return w_d as a `GrpAutCrvElt` in G. Note that the intrinsic should terminate immediately in the first case, when it is just retrieving the equations from the database file, but has to do a little bit of work in the second case.

`SrAutomorphism(CN,N,r,u)`

CN should be a base change of the small modular database curve of level N to a field K of characteristic 0. r is a divisor of 24 such that $r^2|N$. u is a primitive r-th root-of-unity in K. Returns S_r as a scheme automorphism of CN that corresponds to $z \mapsto z+(1/r)$ if K is embedded into $\mathbf{C}$ so that u maps to $e^{2\pi i/r}$. If the full automorphism group G of CN has already been computed and cached with CN (see the intrinsics below and the section in the Algebraic Curves chapter on automorphism groups) then the intrinsic will return S_r as a `GrpAutCrvElt` in G. Note that the intrinsic will usually terminate more quickly in the first case, when it is just retrieving equations from the database file and maybe performing a composition, whereas in the second case it is doing an extra little bit of work.

`ExtraAutomorphism(CN,N,u)`

CN should be a base change of the small modular database curve of level N to a field K of characteristic 0. N is 37, 63 or 108. u is a primitive r-th root-of-unity in K where $r = 1$ (so $u = 1$) when $N = 37$ and $r = 3$ in the other cases. Returns an automorphism of CN in $A_0(N)$ that is not in $B_0(N)$ (i.e. not given by the action of a matrix on the complex upper half-plane: see the introduction). This "extra" automorphism generates $A_0(N)$ over $B_0(N)$ and is an involution when $N = 37$ (the hyperelliptic involution) or $N = 108$ and is of order 4 when $N = 63$ with square equal to the Atkin-Lehner involution w_9. If the full automorphism group G of CN has already been computed and cached with CN (see the intrinsics below and the section in the Algebraic Curves chapter on automorphism groups) then the intrinsic will return the extra automorphism as a `GrpAutCrvElt` in G. Note that the intrinsic should terminate immediately in the first case, when it is just retrieving the equations from the database file, but has to do a little bit of work in the second case.

`AutomorphismGroupOverQ(CN,N)`

`Install` BoolElt *Default :* `true`

CN should be a base change of the small modular database curve of level N to a field K of characteristic 0. Computes and returns the full group of automorphisms of CN (automorphisms in $B_0(N)$ if the genus of $X_0(N)$ is ≤ 1: see the introduction)

defined over **Q**. If the parameter `Install` has the value `true`, which is the default, the return value is internally installed and cached as the (full) automorphism group of CN, so the user should be careful that K doesn't contains roots of unity which generate a field over which some extra modular automorphisms are defined (viz. roots of unity of exact order r where $2 < r|24$ and $r^2|N$).

`AutomorphismGroupOverCyclotomicExtension(CN,N,n)`

`Install` BoolElt *Default :* `true`

CN should be a base change of the small modular database curve of level N to a field K which is the cyclotomic extension of **Q** obtained by adjoining the n-th roots of unity (the precise condition is that $K.1$ should be a primitive n-th root of unity). Computes and returns the full group of automorphisms of CN (automorphisms in $B_0(N)$ if the genus of $X_0(N)$ is ≤ 1: see the introduction) defined over K. If the parameter `Install` has the value `true`, which is the default, the return value is internally installed and cached as the (full) automorphism group of CN.

`AutomorphismGroupOverExtension(CN,N,n,u)`

`Install` BoolElt *Default :* `true`

CN should be a base change of the small modular database curve of level N to a field K which is an extension of **Q** containing the n-th roots of unity. u should be a primitive n-th root of unity in K. Computes and returns the full group of automorphisms of CN (automorphisms in $B_0(N)$ if the genus of $X_0(N)$ is ≤ 1: see the introduction) defined over the cyclotomic field $\mathbf{Q}(\mu_n)$. If the parameter `Install` has the value `true`, which is the default, the return value is internally installed and cached as the (full) automorphism group of CN. In that case, the user should be careful that n is well-chosen so that K doesn't contain higher order roots of unity which generate a subfield over which some modular automorphisms are defined that are not defined over $\mathbf{Q}(\mu_n)$.

Example H129E3

```
> C := SmallModularCurve(37);
> C;
Hyperelliptic Curve defined by y^2 + (-x^3)*y = 2*x^5 - 5*x^4 + 7*x^3 - 6*x^2
    + 3*x - 1 over Rational Field
> AtkinLehnerInvolution(C,37,37);
Mapping from: CrvHyp: C to CrvHyp: C
with equations :
$.1
$.2
$.1 - $.3
and inverse
$.1
$.2
$.1 - $.3
```

```
> ExtraAutomorphism(C,37,1); //the hyperelliptic involution
Mapping from: CrvHyp: C to CrvHyp: C
with equations :
$.1
$.1^3 - $.2
$.3
and inverse
$.1
$.1^3 - $.2
$.3
> C<x,y,z> := SmallModularCurve(48);
> C;
Hyperelliptic Curve defined by y^2 + (-x^4 - 1)*y = 3*x^4 over Rational Field
> AtkinLehnerInvolution(C,48,3);
Mapping from: CrvHyp: C to CrvHyp: C
with equations :
-z
x^4 - y + z^4
x
and inverse
-z
x^4 - y + z^4
x
> Type($1);
MapAutSch
> G := AutomorphismGroupOverQ(C,48);
> #G;
16
> AtkinLehnerInvolution(C,48,3);
Mapping from: CrvHyp: C to CrvHyp: C
with equations :
-z
x^4 - y + z^4
x
and inverse
-z
x^4 - y + z^4
x
> Type($1);
GrpAutCrvElt
> PermutationRepresentation(G);
Permutation group acting on a set of cardinality 6
Order = 16 = 2^4
    (2, 3)
    (1, 2)(3, 4)
    (5, 6)
Mapping from: GrpAutCrv: G to GrpPerm: $, Degree 6, Order 2^4 given by a rule
> // D8 X C2
```

```
> K<i> := QuadraticField(-1);
> CK<x,y,z> := SmallModularCurve(48,K);
> G := AutomorphismGroupOverCyclotomicExtension(CK,48,4);
> #G;
48
> SrAutomorphism(CK,48,4,i); // z -> z+(1/4)
Mapping from: CrvHyp: CK to CrvHyp: CK
with equations :
-i*x
y
z
and inverse
i*x
y
z
```

129.5 Cusps and Rational Points

Letting H^* denote the extended complex upper half-plane $\{z \in C|\mathrm{Im(z)} > 0\} \cup \mathbf{Q} \cup \infty$, $X_0(N)(\mathbf{C})$ identifies with the quotient space $H^*/\Gamma_0(N)$ and the cusps are the images of $\mathbf{Q} \cup \infty$. A complete set of representatives of cusps are the images of the points a/d where d runs through positive divisors of N and, for each d, a runs through a set of integer representatives of $(\mathbf{Z}/(d, N/d)\mathbf{Z})^\times$ relatively prime to d. In particular, if $(d, N/d) \leq 2$, we just get $1/d$. The image of the cusp at ∞ identifies with the image of $1/N$.

With respect to the rational structure of $X_0(N)$, the cusps are all algebraic points defined over cyclotomic fields. Specifically, for a given $d|N$, the a/d are a set of Galois-conjugate points, each one having $\mathbf{Q}(\mu_{(d,N/d)})$ as the field of definition, and the Galois group of $\mathbf{Q}(\mu_{(d,N/d)})$ over $\mathbf{Q}$ acts simply transitively on this set. In particular, if $(d, N/d) \leq 2$, the cusp $1/d$ is a $\mathbf{Q}$-rational point on $X_0(N)$. Thus, $\infty \sim 1/N$ and $0 = 0/1 \sim 1/1$ are always rational cusps.

The cuspidal points on the models for small modular curves are stored in the database. More precisely, for each $d|N$, the database stores either a rational point for the cusp $1/d$, if $(d, N/d) \leq 2$, or a cluster that defines the $\mathbf{Q}$-conjugate set of cusps a/d, if $(d, N/d) > 2$. The user can access these through an intrinsic. If the model C for $X_0(N)$ is singular, some cusps may correspond to singular points on C. There may be two cusps that lie over the same node of C. In any case, the *place* over each conjugate class of cusps a/d is unique (different for different d) and the database contains information to determine which place belongs to which cusp for places above a singular cuspidal point on the model. There is an intrinsic to return the curve place above each $\mathbf{Q}$-conjugate set of cusps a/d.

Non-cuspidal rational points are of particular interest since they correspond to classes of elliptic curves defined over $\mathbf{Q}$ with a cyclic N-isogeny also defined over $\mathbf{Q}$ up to twist. On the complete set of curves $X_0(N)$ of genus > 0, there are very few non-cuspidal rational points and all such were determined by Mazur and others in the 1970s. The database

stores the list of non-cuspidal rational points for each level N. These are never singular points on the chosen models.

`Cusp(CN,N,d)`

CN should be a base change of the small modular database curve of level N to a field K of characteristic 0. d is a positive divisor of N. Returns the point on CN corresponding to the cusp $1/d$, if $(d, N/d) \leq 2$, or returns the cluster (zero-dimensional scheme) defined over K that is the reduced subscheme of CN consisting of the $\phi((d, N/d))$ cusps a/d, if $(d, N/d) > 2$ where ϕ is Euler's totient function.

`CuspIsSingular(N,d)`

Returns whether the points lying under the cusps a/d for $d|N$ on the small modular curve database model for $X_0(N)$ are singular. For a given d, they either all are or all are not.

`CuspPlaces(CN,N,d)`

CN should be a base change of the small modular database curve of level N to a field K of characteristic 0. d is a positive divisor of N. Returns the sequence of places of CN that correspond to the $\phi((d, N/d))$ cusps a/d. If $(d, N/d) \leq 2$ or $K = \mathbf{Q}$ there will only be one place. However, if K is a proper extension of $\mathbf{Q}$, the $\mathbf{Q}$-conjugate cusps a/d may split into several Galois orbits over K.

`NonCuspidalQRationalPoints(CN,N)`

CN should be a base change of the small modular database curve of level N of genus > 0 to a field K of characteristic 0. Returns the sequence of points on CN that correspond to non-cuspidal points in $X_0(N)(\mathbf{Q})$. There are only a small number of N for which this is a non-empty set and the rational points are non-singular ones on all of our models.

Example H129E4

```
> C := SmallModularCurve(32);
> C;
Elliptic Curve defined by y^2 = x^3 + 4*x over Rational Field
> Cusp(C,32,32); //cusp at infinity
(0 : 1 : 0)
> Cusp(C,32,8);
Cluster over Rational Field defined by
$.1^2 + 4*$.3^2,
$.2
> Degree($1);
2
> C<x,y,z> := SmallModularCurve(63);
> Cusp(C,63,1); // cusp at 0
(1 : 1 : 1)
```

```
> Cusp(C,63,7); // cusp at 1/7
(1 : 1 : 1)
> CuspIsSingular(63,1);
true
> CuspIsSingular(63,7);
true
> CuspPlaces(C,63,1);
[
    Place (2) at (1 : 1 : 1)
]
> CuspPlaces(C,63,7);
[
    Place (1) at (1 : 1 : 1)
]
> Cusp(C,63,3);
Cluster over Rational Field defined by
x - y,
y^2 + y*z + z^2
> Cusp(C,63,21);
Cluster over Rational Field defined by
x - y,
y^2 + y*z + z^2
> CuspPlaces(C,63,3)[1];
Place at ($.1 : $.1 : 1)
> CuspPlaces(C,63,21)[1];
Place at ($.1 : $.1 : 1)
> $1 eq $2;
false
```

129.6 Standard Functions and Forms

This section contains intrinsics that return the j-invariant as a rational function and normalised Eisenstein forms as meromorphic k-differentials on the small database models of $X_0(N)$. Standard variants of these can be obtained by pulling back by Atkin-Lehner involutions or pulling back the corresponding objects from lower level curves by the projection or r-projection maps.

The database actually only contains precomputed expressions for $E_2^{(N)}$ (see below), E_4 and E_6 for prime levels N and reconstructs everything else from these objects using projection maps.

`jInvariant(CN,N)`

`jFunction(CN,N)`

CN should be a base change of the small modular database curve of level N to a field of characteristic 0. These intrinsics return the j-invariant $j(z)$ as a rational function on CN. The second returns it as an element of the function field of CN. The first returns it as an element in the field of fractions of the coordinate ring of the ambient of CN, which is sometimes more convenient to use.

`jInvariant(p,N)`

p is a non-cuspidal point or a non-cuspidal place on a base change CN of the small modular database curve of level N to a field of characteristic 0. Returns the value of $j(z)$ at p, the value lying in L if p is a point in $CN(L)$ or in the residue class field of p if p is a place of CN. If p is a point, it should be non-singular. However, if the j function is defined at p, the intrinsic will still return a value.

`jNInvariant(p,N)`

Exactly as `jInvariant` above, except that the intrinsic gives the value of rational function $j(Nz)$ at the point or place. This is equivalent to computing the j-invariant value on the image of p under the Fricke involution w_N.

`E2NForm(CN,N)`

CN should be a base change of the small modular database curve of level N to a field of characteristic 0. $E_2^{(N)}(z) = NE_2(Nz) - E_2(z)$ is a weight 2 integral form for $\Gamma_0(N)$ where $E_2(z) = 1 - 24e^{2\pi iz} + \ldots$ is the normalised weight 2 Eisenstein series. $E_2^{(N)}(z)$ corresponds to a meromorphic differential (defined over $\mathbf{Q}$) on $X_0(N)$. The intrinsic returns $E_2^{(N)}(z)$ as a meromorphic differential in the function field of CN.

`E4Form(CN,N)`

CN should be a base change of the small modular database curve of level N to a field of characteristic 0. Returns a rational function f and a differential form ω in the function field of CN such that the Eisenstein series $E_4(z) = 1 + 240e^{2\pi iz} + \ldots$ as a meromorphic 2-differential on CN is given by $f\omega^2$.

`E6Form(CN,N)`

CN should be a base change of the small modular database curve of level N to a field of characteristic 0. Returns a rational function f and a differential form ω in the function field of CN such that the Eisenstein series $E_6(z) = 1 - 504e^{2\pi iz} + \ldots$ as a meromorphic 3-differential on CN is given by $f\omega^3$. Note that the same differential ω is returned by the `E4Form` intrinsic.

129.7 Parametrized Structures

As with the other modular equation databases in MAGMA there is functionality to explicitly compute a cyclic N-isogeny or cyclic subgroup of order N on an elliptic curve that is represented by a non-cuspidal point or place on the $X_0(N)$ models with the usual moduli space interpretation of the modular curves. The points represent equivalence classes of these structures (elliptic curve E with cyclic subgroup C, (E, C), or cyclic N-isogeny $\phi : E \to F$) up to isomorphism. In particular, if (E, C) is an elliptic curve with cyclic subgroup with both E and C defined over a field K, then any quadratic twist E_1 of E with the twisted subgroup C_1, which is the image of C under the isomorphism from E to E_1 over a quadratic extension, is represented by the same point on $X_0(N)$. If $j \neq 0, 1728$, any two elliptic curves over a field K with j-invariant j are quadratic twists of each other. Because of this, if $j(p) \neq 0, 1728$, any elliptic curve E over the field of definition K of the point p and j-invariant $j(p)$ will have a cyclic subgroup C defined over K such that (E, C) is represented by p or, equivalently, there is a cyclic N-isogeny defined over K and with domain E that is represented by p. We allow the user to pass in his choice of E for the base elliptic curve.

If z is a non-real complex number, let Λ_z be the two dimensional complex lattice $\mathbf{Z} \oplus \mathbf{Z}z$. Under the identification of the complex points of our model of $X_0(N)$ with $H^*/\Gamma_0(N)$ which follows from the identifcation of coordinate functions with specific modular functions or forms (see the next section), if a non-cuspidal point p corresponds to a point z in the upper halfplane H, the point p in the moduli intepretation is represented by (E_z, C), where $E_z = \mathbf{C}/\Lambda_z$ and C is the cyclic subgroup generated by $(1/N) \bmod \Lambda_z$, or the cyclic N-isogeny $\phi : E_z \to E_{Nz}$, $t \mapsto Nt$.

The computation of structures uses well-known methods. If d is a suitable differential on $X_0(N)$, we start with the data

$$p_1 = (N/12)(E_2^{(N)}/d)(p) \qquad E_4 = (E_4/d^2)(p) \qquad E_6 = (E_6/d^3)(p)$$

$$\tilde{E}_4 = N^2((E_4/d^2)(w_N(p))(w_N^*(d)/d)(p)^2 \qquad \tilde{E}_6 = N^3((E_6/d^3)(w_N(p))(w_N^*(d)/d)(p)^3$$

p is represented by (E, C) where E is given by $y^2 = x^3 - (E_4/48)x - (E_6/864)$ and the monic polynomial in x that defines C is computed from $p_1, E_4, E_6, \tilde{E}_4, \tilde{E}_6$ by the same algorithm as used in the SEA Elkies variant of Schoof's algorithm. The corresponding isogeny $\phi : E \to F$ with kernel C can then be computed using Velu's formulae.

```
SubgroupScheme(p,N)
```

```
SubgroupScheme(p,N,E)
```

Here p is a non-cuspidal point or place on C_N, a base change of the curve from the small modular curve database of level N to a field of characteristic zero. When p is given as a point, it should be non-singular point on C_N. The function returns an elliptic curve E_1 and a subgroup scheme G of E_1, both defined over the field of definition K of p (K if p is a point in $C_N(K)$; the residue class field of p if p is a place) such that G gives a cyclic subgroup of order N on E_1 and (E_1, G) represents p in the moduli space interpretation.

In the second version, when $j(p)$ should not be 0 or 1728, E should be an elliptic curve over a subfield of K with j-invariant $j(p)$. In this case, E_1 is taken as E or the base change of E to K, and only G is returned.

`Isogeny(p,N)`

`Isogeny(p,N,E)`

Here p is a non-cuspidal point or place on C_N, a base change of the curve from the small modular curve database of level N to a field of characteristic zero. When p is given as a point, it should be non-singular point on C_N. The function returns a cyclic N-isogeny of elliptic curves $\phi : E_1 \to F_1$, all defined over the field of definition K of p (K if p is a point in $C_N(K)$; the residue class field of p if p is a place), which represents p in the moduli space interpretation.

In the second version, when $j(p)$ should not be 0 or 1728, E should be an elliptic curve over a subfield of K with j-invariant $j(p)$. In this case, E_1 is taken as E or the base change of E to K.

Example H129E5

```
> C := SmallModularCurve(14);
> rats := NonCuspidalQRationalPoints(C,14);
> rats;
[ (2 : 2 : 1), (9 : -33 : 1) ]
> jInvariant(rats[1],14);
16581375
> jNInvariant(rats[1],14);
-3375
> jInvariant(rats[2],14);
-3375
> jNInvariant(rats[2],14);
16581375
> G,E := SubgroupScheme(rats[2],14);
> E;
Elliptic Curve defined by y^2 + x*y = x^3 - x^2 - 2*x - 1 over Rational Field
> G;
Subgroup scheme of E defined by x^7 + 7*x^6 - 7*x^5 - 35*x^4 + 7*x^3 + 35*x^2
    + 7*x - 2
> jInvariant(E);
-3375
> phi := Isogeny(rats[2],14);
> phi;
Elliptic curve isogeny from: Elliptic Curve defined by y^2 + x*y = x^3 - x^2 -
2*x - 1 over Rational Field to Elliptic Curve defined by y^2 + x*y = x^3 - x^2
- 1822*x + 30393 over Rational Field
taking (x : y : 1) to ((x^14 + 16*x^13 + 431*x^12 + 1604*x^11 - 768*x^10 -
    12344*x^9 - 7979*x^8 + 25044*x^7 + 29067*x^6 - 4796*x^5 - 12252*x^4 +
    3244*x^3 + 6789*x^2 + 2504*x + 371) / (x^13 + 16*x^12 + 67*x^11 - 34*x^10
```

```
    - 495*x^9 - 332*x^8 + 967*x^7 + 1048*x^6 - 431*x^5 - 834*x^4 - 205*x^3 +
    52*x^2 + 13*x - 2) : (x^20*y + 23*x^19*y - 364*x^19 - 192*x^18*y -
    2093*x^18 + 288*x^17*y - 2821*x^17 + 1248*x^16*y + 19201*x^16 -
    5484*x^15*y + 73430*x^15 + 5293*x^14*y - 59871*x^14 - 10840*x^13*y -
    398076*x^13 - 50135*x^12*y - 153384*x^12 - 107060*x^11*y + 723331*x^11 -
    175648*x^10*y + 619619*x^10 - 15884*x^9*y - 395549*x^9 + 540867*x^8*y -
    167643*x^8 + 1295892*x^7*y + 895937*x^7 + 1548467*x^6*y + 848974*x^6 +
    947584*x^5*y + 19075*x^5 + 32804*x^4*y - 348579*x^4 - 407788*x^3*y -
    218505*x^3 - 305738*x^2*y - 56084*x^2 - 95233*x*y - 3710*x - 9831*y + 371)
/ (x^20 + 23*x^19 + 172*x^18 + 288*x^17 - 1755*x^16 - 5757*x^15 + 4334*x^14 +
    29683*x^13 + 7538*x^12 - 65053*x^11 - 47954*x^10 + 59387*x^9 + 72098*x^8 -
    8621*x^7 - 37663*x^6 - 12228*x^5 + 2214*x^4 + 1215*x^3 - 83*x^2 - 40*x +
    4) : 1)
> F := Codomain($1);
> jInvariant(F);
16581375
```

129.8 Modular Generators and q-Expansions

In the genus 0 cases, the database model is just the projective line $\mathbf{P}^1$ and the uniformising parameter $t = x/y$ (where x, y are the projective coordinates of $\mathbf{P}^1$) is a modular function on $X_0(N)$. We have normalised so that this function has a simple zero at the cusp 0 and a simple pole at the cusp ∞ with q-expansion of the form $q^{-1} + \ldots$ when $N > 1$. For $N = 1$ we just take the j-invariant as our parameter.

In the elliptic and hyperelliptic cases, the model is a minimal Weierstrass equation $y^2 + h(x)y = f(x)$ and the x and y coordinate functions are modular functions on $X_0(N)$. We have normalised so that one of the points at infinity of the model is the cusp ∞, so x and y have poles there. In the elliptic cases, there is only one point at infinity. In the hyperelliptic cases, the other point at infinity is always the image of ∞ under the hyperelliptic involution: ∞ is never a Weierstrass point in these cases.

We sometimes refer to genus zero, elliptic and hyperelliptic curves by the semi-standard term subhyperelliptic. The models in the non-subhyperelliptic cases are ordinary projective and are sub-canonical images of $X_0(N)$. This means that there are linearly-independent weight 2 cusp forms $f_1, \ldots, f_r$ (which correspond to holomorphic differentials on $X_0(N)$) such that our model is the image of $X_0(N)$ under the map $z \mapsto [f_1(z) : \ldots : f_r(z)]$ into $\mathbf{P}^{r-1}$.

As stated earlier, although we sometimes used MAGMA's modular symbols machinery to compute a basis for forms when deriving models, we also worked out expressions for the modular functions t, x and y and the forms f_i in terms of certain basic types. This gives the models a degree of independence from the modular symbol machinery and also allows for faster reconstruction of q-expansions of the generating functions/forms. We describe the basic types here. There is an intrinsic that the user may call to retrieve the information as to how the forms are built up and one to return the actual q-expansions up to any desired precision. In some cases, we have used the cusp forms associated to an

elliptic curve of conductor N. We would have liked to avoid this and maybe with a bit more work we may be able to find alternative expressions that avoid using them. In fact, we should be using more of the theta series associated to quaternion algebras (this was an oversight only discovered recently: we are currently only using the theta series attached to one isomorphism class of maximal order for each quaternion algebra). Leaving aside the forms associated to elliptic curves, the basic types that we have looked at are all classical. We could also have tried to use some other basic types like the generalised eta products of Y. Yang.

The basic types fall into a number of categories:

(i) Dedekind eta products.

(ii) Theta series of binary quadratic forms of various kinds and theta series of quaternion algebras.

(iii) Weight 2 Eisenstein series of prime level.

(iv) Weight two cusp forms of elliptic curves.

Dedekind eta products for level N are of the form $\eta(d_1z)^{r_1}\eta(d_2z)^{r_2}\dots\eta(d_nz)^{r_n}$ where the d_i run over a subset of the positive divisors of N, the r_i are integers and $\eta(z)$ is the Dedekind η-function which has q-expansion $q^{1/24}\prod_{n=1}^{\infty}(1-q^n)$. If $\sum r_i$ is even and $\sum r_id_i$ and $\sum r_i(N/d_i)$ are divisible by 24, the product gives a modular form of weight $(1/2)\sum r_i$ for $\Gamma_0(N)$ with a certain character mod N. The form only has zeroes and poles at cusps with order given by a formula depending on N, r_i and d_i. We use these products to give modular functions and weight 2 forms and occasionally weight one forms to be used in combination with weight one forms coming from theta series.

The theta series associated to binary quadratic forms come in several flavours.

If $Q(x,y)=ax^2+bxy+cz^2$ (with $a,b,c\in\mathbf{Z}, a>0$) has discriminant $-N$, then the usual theta series $\theta_Q(z)=\sum_{(x,y)\in\mathbf{Z}^2}q^{Q(x,y)}$ is a weight one form of level N for the quadratic character given by the Jacobi symbol $(\frac{-N}{\cdot})$. The product of two of these gives a weight 2 integral form for $\Gamma_0(N)$.

If $N\equiv 1 \bmod 4$ and $Q(x,y)=ax^2+bxy+cz^2$ is a form with discriminant $-4N$ with a odd and b,c even, the function $\theta_Q^{(1)}(z)=\sum_{(x,y)\in\mathbf{Z}^2,xodd}(-1)^y q^{Q(x,y)/4}$ is a weight one form of level N for the character $(\frac{\cdot}{N})\chi^a$ where χ is the quartic character giving the action of $\Gamma_0(1)$ on $\eta(z)^6$. If Q_1,Q_2 are two such forms, one of which has $a\equiv 1 \bmod 4$ and $a\equiv 3 \bmod 4$ for the other, then $\theta_{Q_1}^{(1)}\theta_{Q_2}^{(1)}$ is a weight 2 integral form for $\Gamma_0(N)$. Additionally, the function $\theta_Q^{(2)}(z)=\sum_{(x,y)\in\mathbf{Z}^2,xodd}q^{Q(x,y)/4}$ is a weight one form of level $2N$ for the character $(\frac{\cdot}{N})\chi^a\delta$ where δ is the unique non-trivial character that factors through the quotient $\Gamma_0(2N)/\Gamma_0(4N)$. Again, $\theta_{Q_1}^{(2)}\theta_{Q_2}^{(2)}$ is a weight 2 form for $\Gamma_0(2N)$ if the as for Q_1 and Q_2 are non-congruent mod 4.

If $N\equiv 3 \bmod 8$ and $Q(x,y)=ax^2+bxy+cz^2$ is a form with discriminant $-N$ (here a,b,c must all be odd), the function $\theta_Q^{(3)}(z)=\sum_{(x,y)\in\mathbf{Z}^2,xodd}(-1)^y q^{Q(x,y)/2}$ is a weight one form of level N for the character $(\frac{-N}{\cdot})\chi$ where χ is the quadratic character giving the action of $\Gamma_0(1)$ on $\eta(z)^{12}$. The product of any two of these gives a weight 2 integral form for $\Gamma_0(N)$.

If $N = p_1 \dots p_r$ is a product of an odd number of distinct primes, let H_N be the quaternion algebra over $\mathbf{Q}$ which is ramified precisely at ∞ and the primes p_i. Let O_N be a maximal order of H_N, which is a $\mathbf{Z}$-lattice of rank 4, and Q_N the positive-definite quadratic form on it given by the reduced norm Nm of H_N. Then the theta function $\theta_{Q_N}(z) = \sum_{x \in O_N} q^{Nm(x)}$ is a weight two integral form for $\Gamma_0(N)$.

The weight 2 Eisenstein series of prime level p is the form $pE_2(pz) - E_2(z)$, which is an integral form of weight 2 for $\Gamma_0(p)$. Here $E_2(z)$ is the usual normalised weight 2 Eisenstein series (which isn't a modular form for any level) with q-expansion $1 - 24q - 72q^2 - \dots$. We usually normalise the prime level Eisenstein series by dividing by $GCD(24, p-1)$ which is the GCD of its integral coefficients.

If E is an elliptic curve over Q of conductor N, then the modular form $f_E(z)$ associated to E, which is returned by the call `ModularForm(E)`, is a weight 2 cusp form for $\Gamma_0(N)$.

We also use three standard operations on forms/functions.

If $f(z)$ is a modular function on $X_0(N)$ then $D(f) = (1/2\pi i)(df/dz)$ is a weight 2 form for $\Gamma_0(N)$.

If $f(z)$ is a form/function on $X_0(M)$ and d is a positive integer dividing N/M then $f(dz)$ is a form/function on $X_0(N)$.

If f is a form for $\Gamma_0(N)$ and χ is the quadratic character with conductor d where $|d|^2$ divides N, then the twisted form $f \otimes \chi$ is also a form for $\Gamma_0(N)$ where if f has the q-expansion $\sum a_n q^n$ then $f \otimes \chi$ has q-expansion $\sum a_n \chi(n) q^n$.

The basic types are forms or functions of types (i)-(iv), possibly operated on by one of the above 3 operations. Basic functions are functions of basic type or quotients of forms of basic type of the same weight. We build modular functions as rational functions of basic functions. We build modular forms of weight 2 as linear combinations of weight 2 forms of basic type or products of modular functions with weight 2 forms of basic type. We give some examples below.

`qExpansionExpressions(N)`

This procedure prints out a mini-program that is used to compute the modular functions that correspond to t or to x, y on $X_0(N)$ in the subhyperelliptic cases or the weight 2 cusp forms $f_1, \dots, f_r$ on $X_0(N)$ in the other cases. These give expressions for the forms/functions in terms of forms/functions of basic type. The format of the mini-program is as follows.

There are a sequence of statements. Each except the last is of the form $v = \langle expr \rangle$ where v is a variable name, $\langle expr \rangle$ is an expression and the statement assigns the value of the expression (a modular form or function on $X_0(N)$) to variable v. Each expression is a string of terms separated by binary arithmetic operators $\hat{} \; * \; / \; + \; -$ which have their usual meaning, are listed in precedence order and evaluate left to right for operators of the same precedence. There may also be subexpressions in () brackets. Subexpressions in brackets are evaluated first and replaced with their value, inner bracketed subexpressions being evaluated before outer ones. Terms are of one of the following types, possibly modified by an operator:

(i) An integer. This either multiplies a form/function, is an exponent for powering or gives a constant function.

(ii) q. This is just $e^{2\pi iz}$, the 'q' of the q-expansion.

(iii) A variable that has previously been assigned to.

(iv) An eta product. $e\{\langle d_1\ r_1\rangle\langle d_2\ r_2\rangle\dots\langle d_n\ r_n\rangle\}$ represents the eta product $\eta(d_1z)^{r_1}\eta(d_2z)^{r_2}\dots\eta(d_nz)^{r_n}$.

(v) Binary theta functions as described in the introduction. For a positive-definite binary quadratic form $Q(x,y)=Ax^2+Bxy+Cy^2$,
$th0\{A\ B\ C\}$ is the usual theta series of Q.
$th1\{A\ B\ C\}$ is $(1/2)q^{-a/4}\theta_Q^{(1)}$ where a is A mod 4. This has q-integral q-expansion.
$th2\{A\ B\ C\}$ is $(1/2)q^{-a/4}\theta_Q^{(2)}$ where a is A mod 4. This has q-integral q-expansion.
$th3\{A\ B\ C\}$ is $(1/2)q^{-1/2}\theta_Q^{(3)}$. This has q-integral q-expansion.

(vi) Theta functions of quaternion algebras as described in the introduction. $thQ\{N\}$ represents the weight 2 theta function attached to a maximal order of the quaternion algebra of level N, where N is a product of an odd number of distinct primes. The notation is deficient here. The theta series depends on the isomorphism class of the maximal order chosen. Currently, we are only using the maximal order returned by MAGMA in the call `MaximalOrder(QuaternionAlgebra(N))`.

(vii) Eisenstein series. $E\{p\}$ represents the weight 2 normalised Eisenstein series of prime level p, $(pE_2(pz)-E_2(z))/GCD(24,p-1)$.

(viii) Cusp forms from elliptic curves. $f\{a_1\ a_2\ a_3\ a_4\ a_6\}$ represents the weight two newform associated to the elliptic curve over $\mathbf{Q}$ with equation $y^2+a_1xy+a_3y=x^3+a_2x^2+a_4x+a_6$.

Modifying operators are

(i) D. $D(f)$ is $(1/2\pi i)(df/dz)$, where f is a modular function on $X_0(N)$.

(ii) $[d]$. d is a positive integer. If f represents a form or function $f(z)$, then $f[d]$ represents $f(dz)$.

(iii) $\langle d\rangle$. If f is a form, $f\langle d\rangle$ represents the form twisted by the quadratic character of conductor d.

The last line of the mini-program is of the form $[t,1]$ if $X_0(N)$ is of genus 0, $[x,y,1]$ if $X_0(N)$ is elliptic or hyperelliptic, and $[f_1,\dots,f_r]$ otherwise, where t,x,y,f_i are variables that were assigned to in earlier lines. These give the generating functions/forms for the $X_0(N)$ model.

`qExpansionsOfGenerators(N,R,r)`

Returns q-expansions to precision r as Laurent series in the Laurent series ring R over $\mathbf{Q}$ for the sequence of modular forms/functions of level N that define the small modular curve database model of $X_0(N)$. As described in the introduction, the

return sequence is of the form $[t]$ if $X_0(N)$ is of genus 0, $[x, y]$ if $X_0(N)$ is elliptic or hyperelliptic, and $[f_1, \ldots, f_r]$ otherwise. Here t, x and y are Laurent series giving the q-expansions of functions which will have poles at the cusp ∞, so have negative power terms, and the f_i will actually be power series giving the q-expansions of weight 2 cusp forms.

The q-expansions are computed from the expressions which are given in the mini-program output by `qExpansionExpressions(N)`, generating the forms/functions from ones of basic type for which there are fast routines to compute q-expansions.

Example H129E6

We give some examples of the mini-programs for various levels

```
> qExpansionExpressions(8); //genus 0 case
x=e{<1 4><2 -2><4 2><8 -4>}
[x,1]
> qExpansionExpressions(15); //genus 1 case
F=e{<1 1><3 1><5 1><15 1>}
t=e{<1 -3><3 3><5 3><15 -3>}
s=-D(t)/F
x=(t^2-5*t-3+s)/2
y=(t^3-8*t^2-4*t+3+s*(t-3))/2
[x,y,1]
> qExpansionExpressions(30); //hyperelliptic case
F=e{<1 -1><2 3><3 1><5 1><6 -1><10 -1><15 -1><30 3>}
x=e{<1 2><2 -2><3 -1><5 -1><6 1><10 1><15 2><30 -2>}
s=-D(x)/F
y=(s+x^4+x^3+x^2)/2
[x,y,1]
> qExpansionExpressions(64); //genus 3 case
F1=e{<4 2><8 2>}
F2=F1<-8>
[(F2-F1)/4,(F2+F1)/2,F1[2]]
> qExpansionExpressions(53); //genus 4 case
t1=th1{1 2 54}
t2=th1{3 2 18}
t3=th1{9 2 6}
F1=q*t1*t2
F2=q*t2*t3
F3=(13*thQ{53}-E{53})/18
F4=f{1 -1 1 0 0}
[F1,(5*F1-3*F2-F3-F4)/2,-4*F1+2*F2+F3,-4*F1+F2+F3]
> qExpansionExpressions(63); //genus 5 case
F1=(th0{1 1 16}-th0{4 1 4})*th0{2 1 8}/2
F2=q*th1{1 2 22}*th1{3 6 10}
F3=F2[3]
F4=e{<3 2><21 2>}
[(F1-F2-F3+F4)/2,(F1-F2-F3-F4)/2,F3]
```

```
> R<q> := LaurentSeriesRing(Rationals());
> qExpansionsOfGenerators(49,R,30); //genus 1 case
[
    q^-2 - q^-1 + 1 + q^3 + q^5 - q^10 - q^13 + q^20 + q^24 + O(q^31),
    q^-3 - 2*q^-2 + 2*q^-1 - 1 - q + q^2 - 2*q^3 + q^4 - 2*q^5 - q^9 + 2*q^10
        + q^11 + 2*q^13 + q^15 + q^16 - 2*q^20 - q^22 - q^23 - 2*q^24 - q^25 -
        q^29 + q^30 + O(q^31)
]
```

129.9 Extended Example

We give an example that combines functionality for small modular curves with MAGMA's machinery for curve quotients and elliptic curves to compute the j-invariant of a special class of elliptic curves over $\mathbf{Q}$ up to quadratic twist. This class of curves arose in Wiles' original paper on Fermat's Last Theorem and the Shimura-Tanayama-Weil (STW) conjecture.

Example H129E7

In the Breuil-Conrad-Diamond-Taylor-Wiles proof of the STW conjecture, a couple of special cases arise that lead to a finite number of classes of elliptic curves up to $\bar{\mathbf{Q}}$-isomorphism that have to be determined and then checked directly for modularity. If $E[p]$ is the p-torsion subgroup of E considered as a $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q})$-module, these are cases where $E[3]$ and $E[5]$ are both reducible, $E[3]$ is reducible and $E[5]$ is irreducible but absolutely reducible as a $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q}(\sqrt{5}))$-module, or $E[5]$ is reducible and $E[3]$ is irreducible but absolutely reducible as a $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q}(\sqrt{-3}))$-module. The first case corresponds to non-cuspidal rational points on $X_0(15)$ and leads, up to isogeny, to one class of twists with $j = -25/2$ as may easily be verified using the intrinsics here. The second case corresponds to non-cuspidal rational points on a quotient of $X_0(75)$ and leads to curves with $j = 0$. We consider the third case in this example, showing that, up to isogeny, there is one class (up to quadratic twist) with j-invariant $(11/2)^3$.

The curves we want are precisely those that have a cyclic subgroup of order 5 rational over $\mathbf{Q}$ and for which the image of $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q})$ in $GL_2(\mathbf{F}_3)$, giving the action on $E[3]$, is the normaliser of a split Cartan subgroup.

Curves just satisfying the $E[3]$ condition are classified by non-cuspidal rational points on $X_0(9)/w_9$ that are not the image of a rational point of $X_0(9)$. Such a point p lifts to a point p_1 on $X_0(9)$ defined over some quadratic field K such that $p_1^\sigma = w_9(p_1)$ where σ is the nontrivial automorphism of K over $\mathbf{Q}$. If (E, C) is an elliptic curve/K with cyclic subgroup $C = \langle P \rangle$ rational over K that is represented by the moduli point p_1, then $E_1 = E/\langle 3P \rangle$ can be defined over $\mathbf{Q}$ so that the order 3 subgroups $C/\langle 3P \rangle$ and $(C/\langle 3P \rangle)^\sigma$ generate $E_1[3]$, are rational over K and are swapped by σ.

In the same way, adding the extra condition that there is a rational subgroup of order 5 leads to curves classified by non-cuspidal rational points of $X_0(45)/w_9$. These do not lift to rational points because $X_0(45)$ has no non-cuspidal rational points. $X_0(45)/w_9$ is $\mathbf{Q}$-isogenous to the rank 0 elliptic curve $X_0(15)$ so only has finitely many rational points. If p_1 is a lift, the image of p_1 under the 3-projection map $X_0(45) \to X_0(5)$ ($z \mapsto 3z$ on complex points) is a rational point that corresponds to the desired curve E_1 (with its cyclic 5-subgroup). We could apply the 3-projection

all the way down to $X_0(1)$ of course, but it is slightly more efficient to just project to level 5 and remove duplicate images before computing j-invariants.

```
> X45<x,y,z> := SmallModularCurve(45);
> w9 := AtkinLehnerInvolution(X45,45,9);
> G := AutomorphismGroup(X45,[w9]);
> C,prjC := CurveQuotient(G);
> c_inf := Cusp(X45,45,45);
> ptE := prjC(c_inf);
> E1,mp1 := EllipticCurve(C,ptE);
> E,mp2 := MinimalModel(E1);
> prjE := Expand(prjC*mp1*mp2);
```

E is a minimal model for $X_0(45)/w_9$ and $prjE$ is the projection from $X_0(45)$ to E that takes the cusp ∞ to the zero point of E. We compute the image of the cusps under $prjE$ so as to discount these when we find all the rational points on E. Since p and $w_9(p)$ map to the same point under $prjE$, we only need consider cusps up to w_9 equivalence. The non-rational conjugate cusps $\pm 1/3$ are defined over $\mathbf{Q}(\sqrt{-3})$ and are swapped by w_9, so map to the same rational point. The same holds for the cusps $\pm 1/15$.

```
> i0 := prjE(Cusp(X45,45,1));
> K := QuadraticField(-3);
> c3 := Cusp(X45,45,3);
> c3p := X45(K)!Representative(Support(c3,K));
> i3 := E!(prjE(c3p));
> c15 := Cusp(X45,45,15);
> c15p := X45(K)!Representative(Support(c15,K));
> i15 := E!(prjE(c15p));
> Ecusps := [E!0,i0,i3,i15];
```

E has rank zero, so we can recover all of its rational points using `TorsionSubgroup`. There are 8 in all, 4 of which are images of cusps. We find the 4 non-cuspidal ones and compute the pullbacks of them to $X_0(45)$. In each case, the pull back is a pair of conjugate points defined over a quadratic field. It is easiest to work with places here (and we don't have to worry about the base scheme of $prjE$). The pullback of each point gives a single place corresponding to the two conjugate points.

```
> T,mp := TorsionSubgroup(E);
> Epts := [mp(g) : g in T];
> Eptsnc := [P : P in Epts | P notin Ecusps];
> plcs := [Support(Pullback(prjE,Place(p)))[1] : p in Eptsnc];
```

We now perform the 3-projection to $X_0(5)$ on these places and discard duplicate images.

```
> X5 := SmallModularCurve(5);
> prj3 := ProjectionMap(X45,45,X5,5,3);
> prj3 := Expand(prj3);
> plcs5 := [Support(Pushforward(prj3,p))[1] : p in plcs];
```

```
> plcs5 := Setseq(Seqset(plcs5));
```

We compute the j-invariants of the two images and verify that they represent isogenous curves under the cyclic 5-isogeny coming from the rational 5-subgroup (i.e. they are images of each other under w_5).

```
> js := [jInvariant(p,5) : p in plcs5];
> js;
[ -1680914269/32768, 1331/8 ]
> w5 := AtkinLehnerInvolution(X5,5,5);
> Pullback(w5,plcs5[1]) eq plcs5[2];
true
```

Finally, we can use MAGMA intrinsics to check that the elliptic curves with these j-invariants actually do satisfy the property that the image of the action of Galois on 3-torsion is the normaliser of a split Cartan subgroup (D_8). As this property remains true for quadratic twists and is unchanged by images under a rational 5-isogeny, we only need check it for one curve over $\mathbf{Q}$ with one of the j-invariants. We also check that a minimal twist has conductor 338 from which modularity can be checked explicitly.

```
> Ej := EllipticCurveWithjInvariant(js[2]);
> Ej := MinimalModel(MinimalTwist(Ej));
> Conductor(Ej);
338
> ThreeTorsionType(Ej);
Dihedral
```

129.10 Bibliography

[**AL70**] A.O.L. Atkin and J. Lehner. Hecke operators on $\Gamma_0(N)$. *Math. Annalen*, 185:134–160, 1970.

[**Bar08**] F. Bars. The group structure of the normaliser of $\Gamma_0(N)$ after Atkin-Lehner. *Communications in Algebra*, 36:2160–2170, 2008.

[**Elk90**] N. Elkies. The automorphism group of the modular curve $X_0(63)$. *Compositio Mathematica*, 74:203–208, 1990.

[**Hara**] M. C. Harrison. Explicit solution by radicals, gonal maps and plane models of algebraic curves of genus 5 or 6. Preprint: online at arXiv as arXiv:1103.4946v3[math. AG].

[**Harb**] M. C. Harrison. A new automorphism of $X_0(108)$. Preprint: online at arXiv as arXiv:1108.5595v2[math.NT].

[**KM88**] M.A. Kenku and F. Momose. Automorphism groups of the modular curves $X_0(N)$. *Compositio Mathematica*, 65:51–80, 1988.

[**Lig75**] G. Ligozat. Courbes modulaires de genre 1. *Bull. Soc. Math. France (Suppl.), Memoire 43*, 1975.

[**Ogg74**] A. Ogg. Hyperelliptic modular curves. *Bull. Soc. Math. France*, 228:449–462, 1974.

130 CONGRUENCE SUBGROUPS OF $\mathrm{PSL}_2(\mathbf{R})$

Chapter 130
CONGRUENCE SUBGROUPS OF $\mathrm{PSL}_2(\mathbf{R})$

130.1 Introduction

The group $GL_2^+(\mathbf{R})$ of 2 by 2 matrices defined over $\mathbf{R}$ with positive determinant acts on the upper half complex plane $\mathbf{H} = \{x \in \mathbf{C} | \mathrm{Im(x)} > 0\}$ by fractional linear transformation:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} : z \mapsto \frac{az+b}{cz+d}.$$

Any subgroup Γ of $GL_2^+(\mathbf{R})$ also acts on $\mathbf{H}$. To compactify the quotient $\Gamma\backslash\mathbf{H}$ it is necessary to adjoin the cusps, which are points in $\mathbf{P}^1(\mathbf{Q}) = \mathbf{Q} \cup \{\infty\}$. We set $\mathbf{H}^* := \mathbf{H} \cup \mathbf{P}^1(\mathbf{Q})$.

A fundamental domain for the action of Γ is a region of $\mathbf{H}^*$ containing a representative of each orbit of the action. The fundamental domain is most useful when it can be taken to be compact with finite area under the hyperbolic metric on $\mathbf{H}$. This is the situation for congruence subgroups.

This chapter gives a description of how to work in Magma with $\mathbf{H}^*$ and with congruence subgroups and their action on $\mathbf{H}^*$. This allows the computation of generators for congruence subgroups, and various other information, such as coset representatives. Farey symbols are used for this computation, and can be computed and manipulated by the user. Procedures are provided for producing graphical images, in the form of postscript files, which illustrate the fundamental domains. These procedures can be used to draw geodesics and polygons in the upper half plane.

This package has been written by Helena Verrill, and is partly based on a java program for drawing fundamental domains ([Ver00]). The Farey sequence algorithm used here is that described by Kulkarni [Kul91].

Example H130E1

An example of what can be computed with this package.

```
> G := CongruenceSubgroup(0,35);
> G;
Gamma_0(35)
> Generators(G);
[
    [1 1]
    [0 1],
    [ 11  -3]
    [ 70 -19],
    [ 13  -8]
    [ 70 -43],
    [  8  -3]
```

```
    [ 35 -13],
    [ 29 -21]
    [105 -76],
    [ 11  -6]
    [ 35 -19],
    [ 29 -17]
    [ 70 -41],
    [ 16 -11]
    [ 35 -24],
    [ 22 -17]
    [ 35 -27]
]
> C := CosetRepresentatives(G);
> H := UpperHalfPlaneWithCusps();
> triangle := [H|Infinity(),H.1,H.2];
> triangle_translates := [g*triangle : g in C];
>  DisplayPolygons(triangle_translates,"/tmp/Gamma035.ps": Show := false);
[ 0.E-57, 1.0000000000000000000000000000, 1.5000000000000000000000000000, 300 ]
```

In the example the command `DisplayPolygons` produces a postscript file "/tmp/Gamma035.ps" of a drawing of all the translates of the given triangle under the cosets of $\Gamma_0(35)$. If the `Show` option is set to `true`, a system command is issued and a window pops up for viewing the file created. Continuing with the above example, one can also find the vertices of a polygon defining the Fundamental domain of the group:

```
> FD := FundamentalDomain(G);
> // Print a FD in a reasonable format:
> &cat[Sprintf(" %o ",c) : c in FD];
 oo  0  1/6  1/5  1/4  3/11  2/7  1/3  2/5  3/7  1/2  4/7  3/5  5/8  2/3  5/7
3/4  1  oo
>  DisplayPolygons([FD],"/tmp/Gamma0_35_Domain.ps": Show := false);
[ 0.E-28, 1.0000000000000000000000000000, 1.5000000000000000000000000000, 300 ]
```

130.2 Congruence Subgroups

We denote by $\mathrm{SL}_2(\mathbf{Z})$ the group of 2 by 2 matrices with integer coefficients and determinant 1. The group $\mathrm{PSL}_2(\mathbf{Z})$ is the projectivization of $\mathrm{SL}_2(\mathbf{Z})$. For any integer N we have groups

$$\Gamma_0(N) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{SL}_2(\mathbf{Z}) \;\middle|\; \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} * & * \\ 0 & * \end{pmatrix} \bmod N \right\}$$

$$\Gamma_1(N) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{SL}_2(\mathbf{Z}) \;\middle|\; \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} 1 & * \\ 0 & 1 \end{pmatrix} \bmod N \right\}$$

$$\Gamma(N) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{SL}_2(\mathbf{Z}) \;\middle|\; \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \bmod N \right\}$$

$$\Gamma^1(N) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{SL}_2(\mathbf{Z}) \;\middle|\; \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 \\ * & 1 \end{pmatrix} \bmod N \right\}$$

$$\Gamma^0(N) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{SL}_2(\mathbf{Z}) \;\middle|\; \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} * & 0 \\ * & * \end{pmatrix} \bmod N \right\}$$

A *congruence subgroup* is any discrete subgroup Γ of $\mathrm{SL}_2(\mathbf{R})$ which is commensurable with $\mathrm{SL}_2(\mathbf{Z})$, that is, $\Gamma \cap \mathrm{SL}_2(\mathbf{Z})$ has finite index in Γ and in $\mathrm{SL}_2(\mathbf{Z})$, and such that $\Gamma(N)$ is contained in G for some N. The *level* N of a congruence subgroup G is the greatest integer N such that $\Gamma(N)$ is contained in Γ. We will abuse notation and also refer to the projectivizations of these groups by the same names.

130.2.1 Creation of Subgroups of $\mathrm{PSL}_2(\mathbf{R})$

`PSL2(R)`

Returns $\mathrm{PSL}_2(R)$, the projective linear group over the ring R.

`Gamma0(N)`

The group $\Gamma_0(N)$ for any positive integer N.

`Gamma1(N)`

The group $\Gamma_1(N)$ for any positive integer N.

`GammaUpper0(N)`

The group $\Gamma^0(N)$ for any positive integer N.

`GammaUpper1(N)`

The group $\Gamma^1(N)$ for any positive integer N.

`CongruenceSubgroup(N)`

The group $\Gamma(N)$ for any positive integer N.

`CongruenceSubgroup(i,N)`

For a positive integer N and $i = 0, 1, 2, 3$, or 4, this is the group $\Gamma_0(N)$, $\Gamma_1(N)$, $\Gamma(N)$, $\Gamma^1(N)$ or $\Gamma^0(N)$ respectively.

`CongruenceSubgroup([N,M,P])`

This is the congruence subgroup consisting of 2 by 2 matrices with integer coefficients $[a, b, c, d]$ with $b = 0 \pmod P$, $c = 0 \pmod N$, and $a = d = 1 \pmod M$. It is required that M divides NP.

`Intersection(G,H)`

`G meet H`

The intersection of congruence subgroups G and H.

Example H130E2

Examples of defining different congruence subgroups:

```
> G := PSL2(Integers());
> H := CongruenceSubgroup([2,3,6]);
> H;
Gamma_0(2) intersection Gamma^1(3) intersection Gamma^0(2)
> K := CongruenceSubgroup(0,5);
> K meet H;
Gamma_0(10) intersection Gamma^1(3) intersection Gamma^0(2)
```

130.2.2 Relations

`G eq H`

Returns `true` if and only if the congruence subgroups G and H are equal.

`H subset G`

For congruence subgroups G and H contained in $\mathrm{PSL}_2(\mathbf{Z})$, returns `true` if and only if H is a subgroup of G.

`Index(G,H)`

For congruence subgroups G and H, returns the index of G in H provided G is a subgroup of H.

`Index(G)`

For G a congruence subgroup in $\mathrm{PSL}_2(\mathbf{Z})$, returns the index in $\mathrm{PSL}_2(\mathbf{Z})$.

130.2.3 Basic Attributes

`Level(G)`

The level of a congruence subgroup G.

`IsCongruence(G)`

Returns `true` if and only if G is a congruence subgroup.

`IsGamma0(G)`

Returns `true` if and only if G is equal to $\Gamma_0(N)$ for some integer N.

`IsGamma1(G)`

Returns `true` if and only if G is equal to $\Gamma_1(N)$ for some integer N.

`BaseRing(G)`

Returns the base ring over which matrices of the congruence subgroup G are defined.

`Identity(G)`

Returns the identity matrix in the congruence subgroup G.

130.3 Structure of Congruence Subgroups

`CosetRepresentatives(G)`

If G is a subgroup of finite index in $\mathrm{PSL}_2(\mathbf{Z})$, then returns a sequence of coset representatives of G in $\mathrm{PSL}_2(\mathbf{Z})$.

`Generators(G)`

Returns a sequence of generators of the congruence subgroup G.

`FindWord(G,g)`

For a congruence subgroup G, and an element g of G, this function returns a sequence of integers corresponding to an expression for g in terms of a fixed set of generators for G. Let L be the list of generators for G output by the function `Generators`. Then the return sequence $[e_1n_1, e_2n_2, \ldots, e_mn_m]$, where n_i are positive integers, and $e_i = 1$ or -1, means that $g = L[n_1]^{e_1}L[n_2]^{e_2}\ldots L[n_m]^{e_m}$. Note that since the computation is in $PSL_2(\mathbf{R})$, this equality only holds up to multiplication by ± 1.

`Genus(G)`

The genus of the upper half plane quotiented by the congruence subgroup G.

`FundamentalDomain(G)`

For G a subgroup of $\mathrm{PSL}_2(\mathbf{Z})$ returns a sequence of points in the Upper Half plane which are the vertices of a fundamental domain for G.

Example H130E3

In this example we compute a set of generators for $\Gamma_0(12)$.

```
> G := CongruenceSubgroup(0,12);
> Generators(G);
[
    [1 1]
    [0 1],
    [ 5 -1]
    [36 -7],
    [  5  -4]
    [ 24 -19],
    [  7  -5]
    [ 24 -17],
    [ 5 -3]
    [12 -7]
]
> C := CosetRepresentatives(G);
> H<i,r> := UpperHalfPlaneWithCusps();
> triangle := [H|Infinity(),r,r-1];
> translates := [g*triangle : g in C];
```

Example H130E4

This example illustrates how any element of a congruence subgroup can be written in terms of the set of generators output by the `generators` function.

```
> N := 34;
> characters := DirichletGroup(N,CyclotomicField(EulerPhi(N)));
> char := Elements(characters)[5];
> G := CongruenceSubgroup([N,Conductor(char),1],char);
>
> // We can create a list of generators:
> gens := Generators(G);
> #gens;
21
> // given an element of G,
> g := G![ 21, 4, 68, 13 ];;
> // we can find g in terms of these generators:
> FindWord(G,g);
[ -8, 1 ]
> // This means that up to sign, g = gens[8]^(-1)*gens[1],
> // which can be verified as follows:
> gens[8]^(-1)*gens[1];
[-21  -4]
[-68 -13]
```

130.3.1 Cusps and Elliptic Points of Congruence Subgroups

`Cusps(G)`

Returns a sequence of inequivalent cusps of the congruence subgroup G.

`CuspWidth(G,x)`

Returns the width of x as a cusp of the congruence subgroup G.

`EllipticPoints(G)`

`EllipticPoints(G,H)`

Returns a list of inequivalent elliptic points for the congruence subgroup G. A second argument may be given to specify the upper half plane H containing these elliptic points.

Example H130E5

We can compute a set of representative cusps for $\Gamma_1(12)$, and their widths as follows:

```
> G := CongruenceSubgroup(0,12);
> Cusps(G);
[
    oo,
    0,
    1/6,
    1/4,
    1/3,
    1/2
]
> Widths(G);
[ 1, 12, 1, 3, 4, 3 ]
> // Note that the sum of the cusp widths is the same as the Index:
> &+Widths(G);
24
> Index(G);
24
```

In the following example we find which group $\Gamma_0(N)$ has the most elliptic points for N less than 20, and list the elliptic points in this case.

```
> H := UpperHalfPlaneWithCusps();
> [#EllipticPoints(Gamma0(N),H) : N in [1..20]];
[ 2, 1, 1, 0, 2, 0, 2, 0, 0, 2, 0, 0, 4, 0, 0, 0, 2, 0, 2, 0 ]
> // find the index where the maximal number of elliptic points is attained:
> Max($1);
4 13
> // find the elliptic points for Gamma0(13):
> EllipticPoints(Gamma0(13));
[
    5/13 + (1/13)*root(-1),
    8/13 + (1/13)*root(-1),
    7/26 + (1/26)*root(-3),
    19/26 + (1/26)*root(-3)
]
```

130.4 Elements of $\mathrm{PSL}_2(\mathbf{R})$

130.4.1 Creation

`G ! x`

If x is a sequence $x = [a, b, c, d]$ of elements in the base ring of G, this function returns $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$, provided this is an element of G. If x is an integer the identity matrix is returned. If x is a matrix, it is coerced into G if possible.

`Random(G,m)`

Returns a random element of the projective linear group G, with m determining the size of the coefficients.

130.4.2 Membership and Equality Testing

`g eq h`

For g and h elements of $\mathrm{PSL}_2(\mathbf{Z})$, returns `true` if g, h have compatible coefficient rings and if $g = h$, `false` otherwise. Since the group is projective, returns `true` if the matrices are equal up to a nonzero scalar multiple.

`IsEquivalent(g,h,G)`

For g and h elements of $\mathrm{PSL}_2(\mathbf{Z})$, returns `true` if g and h are defined of the same field, and if $Gg = Gh$, i.e. if $gh^{-1} \in G$.

`g in G`

For g an elements of $\mathrm{PSL}_2(\mathbf{Z})$, returns `true` if g is in the congruence subgroup G, `false` otherwise.

130.4.3 Basic Functions

For a matrix g in a congruence subgroup, and an integer n, returns g^n.

`Eltseq(g)`

Returns the sequence of four numbers which are the entries of the matrix g.

`g * h`

If g and h have the same parent then this returns their product.

`g ^ n`

For a matrix g and integer n returns g^n.

Example H130E6

Define congruence subgroups as in the following examples:

```
> // examples of defining matrix elements of congruence subgroups:
>
> G := PSL2(Integers());
> G![2,0,0,2];
[1 0]
[0 1]
> H := CongruenceSubgroup([2,3,6]);
> H![7,6,8,7];
[7 6]
[8 7]
```

130.5 The Upper Half Plane

The *upper half complex plane* is defined by $\mathbf{H} := \{z \in \mathbf{C} \mid \mathrm{Im(z)} > 0\}$. The group $\mathrm{SL}_2(\mathbf{Z})$ acts on H by fractional linear transformations. The space $\mathbf{H}/SL_2(\mathbf{Z})$ is not compact; it is compactified by adding the cusps, which are points of $\mathbf{Q}$, together with ∞. Thus we define $\mathbf{H}^*$ to be the upper half plane union the cusps. Then $\mathbf{H}^*/\mathrm{SL}_2(\mathbf{Z})$ is compact. Thus we define a function which will return the space of points in the upper half complex plane, together with the set of cusps.

In $\mathbf{H}^*$ we define two distinguished points, the elliptic points $\sqrt{-1}$ and $(1+\sqrt{-3})/2$. In general, points constructed in $\mathbf{H}^*$ are allowed to come from at most quadratic extensions of $\mathbf{Q}$, since in this case there is a canonical embedding in $\mathbf{C}$.

130.5.1 Creation

`UpperHalfPlane()`

Creates a copy of the upper half complex plane, with the cusps included. As a set this consists of all complex numbers with positive imaginary part, together with all rational numbers, and the point at infinity.

`H ! x`

Returns x as a point in $\mathbf{H}$. Here x can be a cusp, rational, integer, in a quadratic extension of $\mathbf{Q}$, or a complex number with positive imaginary part.

Example H130E7

Example of creating some points in the upper half plane.

```
> H := UpperHalfPlaneWithCusps();
> // coerce a cusp into H:
> c := Cusps()!(1/2);
> H!c;
1/2
> // coerce an element of a quadratic extension of Q into H
> K := QuadraticField(-7);
> K<u> := QuadraticField(-7);
> H!(u+5);
5 + root(-7)
> // refer to the two distinguished elliptic points:
> H.1;
root(-1)
> H.2;
1/2 + (1/2)*root(-3)
> // Defining the names of the elliptic points when constructing H:
> H<i,rho> := UpperHalfPlaneWithCusps();
> i;
root(-1)
> rho;
1/2 + (1/2)*root(-3)
```

130.5.2 Basic Attributes

`Imaginary(z)`

Returns the imaginary part of the argument as an element of `RealField`.

`Real(z)`

Returns the real part of the argument as an element of `RealField`.

`IsReal(z)`

Returns `true` if and only if the element z of the upper half plane lies on the real line (and is not the infinite cusp).

`IsCusp(z)`

Returns `true` if and only if the element z of the upper half plane is a cusp.

`IsInfinite(z)`

Returns `true` if and only if the element z of the upper half plane is the cusp at infinity.

`IsExact(z)`

Returns `true` if and only if the element z of the upper half plane is a cusp or has an exact value defined in a quadratic extension of the rationals.

`ExactValue(z)`

For x an element of the upper half plane, if x is a cusp, returns the value of x as an object of type `SetCspElt`; if x has an exact value in a quadratic extension, returns this value, as an object of type `FldQuadElt`; otherwise returns a complex value of type `FldComElt`.

`ComplexValue(x)`

`Precision`	RngIntElt	*Default :*
`MaxValue`	RngIntElt	*Default :* 600

For x an element of the upper half place, this returns x as a complex number. When x is the cusp at infinity, the value returned is `MaxValue + i*MaxValue`.

`x eq y`

Returns `true` if and only if the points x and y in the upper half plane are equal.

130.6 Action of $\mathrm{PSL}_2(\mathbf{R})$ on the Upper Half Plane

`g * z`

For z of type `SpcHypElt, SetCspElt` or `[SpcHypElt]`, and when g is an element of a projective linear group, returns the image of z under the action of g. The type of the image is the same as the type of z. If g is a positive integer, returns return az, which his is equivalent to acting on z with the matrix $\begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix} \in \mathrm{PGL}_2(\mathbf{R})$.

`FixedPoints(g,H)`

Returns a sequence of points in H fixed by the action of g.

`IsEquivalent(G,a,b)`

If points a, b in the upper half plane are equivalent under the action of the group G, returns `true`, and the matrix g in G such that $g \cdot a = b$. Otherwise returns `false` and the identity.

`EquivalentPoint(x)`

For the point x in the upper half plane, returns a point z in the region with $-1/2 < z \leq 1/2$ and $|z| \geq 1$, and a matrix g in $\mathrm{PSL}_2(\mathbf{Z})$ with g*x = z

`Stabilizer(a,G)`

Returns a generator of the subgroup of G stabilizing a.

`FixedArc(g,H)`

If g is an element of $\mathrm{PSL}_2(\mathbf{Z})$ which is an involution, this returns the end points in the real line of the arc fixed by g, with mid point of the arc also fixed by g. Note that for any point b, the arc from b to $g \cdot b$ is fixed by g.

130.6.1 Arithmetic

`z + a`
`z + a`
`z - a`
`z - a`

For any integer a, and element z in the upper half plane, this returns the element $z + a$ in the same copy of the upper half plane.

`a * z`
`z * a`
`a * z`
`a * z`
`a * seq`
`a * z`
`a * z`
`z * a`
`z * a`
`z * a`
`z * a`
`z * a`
`z / a`

Given an element z (or a sequence of elements) in the upper half plane, and a positive rational number a, this returns the product (or products) in the same copy of the upper half plane.

130.6.2 Distances, Angles and Geodesics

`Distance(z,w)`

`Precision` RngIntElt *Default :*

Returns the hyperbolic distance between z and w.

`TangentAngle(x,y)`

`Precision` RngIntElt *Default :*

Returns the angle of the tangent at x of the geodescic from x to y, with given precision.

`Angle(e1,e2)`

`Precision` RngIntElt *Default :*

Given two sequences $e_1 = [z_1, z_2]$ and $e_2 = [z_1, z_3]$, where $z1, z2, z3$ are elements of the upper half plane, this returns the angle between the geodesics at z_1.

`ExtendGeodesic([z1,z2], H)`

Given elements $z1, z2$ in the upper half plane H, this extends the geodesic between $z1$ and $z2$ to a semicircle with endpoints on the real line, and returns the two real endpoints as elements of H.

`GeodesicsIntersection(x1,x2)`

`GeodesicsIntersection(x1,x2)`

The intersection in the upper half plane of the two geodesics whose endpoints are given by the sequences x_1 and x_2. If the geodesics intersect along a line, the empty sequence is returned.

130.7 Farey Symbols and Fundamental Domains

One method of finding fundamental domains for congruence subgroups is the method of Farey Symbols, as described by Kulkarni [Kul91].

A *generalized Farey sequence* is a sequence of rationals

$$\frac{a_1}{b_1} < \frac{a_2}{b_2} < \ldots < \frac{a_n}{b_n}$$

such that for a consecutive pair of fractions $\frac{b}{d}, \frac{a}{c}$ in the sequence, written in lowest terms, we have $ad - bc = 1$. We extend the rationals to $\mathbf{Q} \cup \{-\infty, \infty\}$, where we use the convention $-\infty = \frac{-1}{0}$ and $\infty = \frac{1}{0}$.

A *Farey Symbol* is a Farey sequence of length n starting with $\frac{-1}{0}$, and ending with $\frac{1}{0}$, together with a sequence of $n-1$ labels. We use the convention that labels can be any elements of $\mathrm{N}_{>0} \cup \{-2, -3\}$. The sequence of labels must satisfy the condition that each element of $\mathrm{N}_{>0}$ appears in the sequence either exactly twice or not at all. For example, the sequences $\left[\frac{-1}{0}, \frac{0}{1}, \frac{1}{2}, \frac{2}{3}, \frac{5}{7}, \frac{3}{4}, \frac{1}{1}, \frac{1}{0}\right]$ and $[1, 2, 2, -3, -2, -2, 1]$ define a Farey symbol, which is generally written in the following format:

$$\left\{ \frac{-1}{0}, \overset{1}{\frac{0}{1}}, \overset{2}{\frac{1}{2}}, \overset{2}{\frac{2}{3}}, \overset{-3}{\frac{5}{7}}, \overset{-2}{\frac{3}{4}}, \overset{-3}{\frac{1}{1}}, \overset{1}{\frac{1}{0}} \right\}.$$

Farey symbols are used to define certain fundamental domains for congruence subgroups of $\mathrm{PSL}_2(\mathbf{Z})$. The sequence of fractions gives cusps which are vertices of the domain, and the labels give edge identifications. For a_i, a_{i+1} in the Farey sequence, with corresponding label l_i not -3, the corresponding edge of the domain is a geodesic between a_i and a_{i+1}. If the label is -3, there is an extra elliptic point of order 3 on the boundary of the domain between the two cusps, and the two edges between these cusps are identified. The label $l_i = -2$ indicates an elliptic point of order 2 on the boundary between the two cusps a_i and a_{i+1}. This point is on the geodesic between a_i and a_{i+1}, and the two halves of the geodesic are identified.

`FareySymbol(G)`

Computes the Farey Symbol of a congruence subgroup G in $\mathrm{PSL}_2(\mathbf{Z})$.

`Cusps(FS)`

Returns the cusp sequence of the Farey symbol FS. Note, this is not a sequence of inequivalent cusps of the corresponding group.

`Labels(FS)`

Returns the sequence of edge labels of a Farey symbol FS.

`Generators(FS)`

Returns the generators of the congruence subgroup corresponding to the Farey symbol FS.

`Group(FS)`

Returns the congruence subgroup corresponding to the Farey Symbol FS.

`Widths(FS)`

Returns the sequence of integers giving twice the widths of the cusp list of the Farey symbol FS.

`Index(FS)`

Returns the index of `Group(FS)` in $\mathrm{PSL}_2(\mathbf{Z})$.

`FundamentalDomain(FS)`
`FundamentalDomain(FS,H)`

Returns the vertices in the upper half plane of the fundamental domain described by the Farey Sequence FS. A second argument may be given to specify the upper half plane H.

`CosetRepresentatives(FS)`

Returns the coset representatives of the congruence subgroup of $\mathrm{PSL}_2(\mathbf{Z})$ corresponding to the Farey symbol FS.

`InternalEdges(FS)`

Returns a sequence of pairs of cusps which are cusps of the Farey Symbol FS, and which are not adjacent in FS but which are images of 0 and infinity under some matrix in $\mathrm{PSL}_2(\mathbf{Z})$.

130.8 Points and Geodesics

In this section we describe some functions involving points and geodesics. Geodesics in the upper half plane are given by circles or lines which intersect the real axes at right angles. A geodesic is defined by its two end points. Points and geodesics can be drawn using the graphics functions described in the next section.

`GeodesicsIntersection(x,y)`

Computes the intersection in the upper half plane of the two geodesics x, y, where x and y are specified by their end points, which must be cusps. If the geodesics intersect along a line the empty sequence is returned.

Example H130E8

In the following example we find and display the intersection points of some geodesics:

```
> H := UpperHalfPlaneWithCusps();
> g1 := [H|-1,2];
> g2 := [H|0,6];
> g3 := [H|1,5];
> g4 := [H|2,Infinity()];
> lines := [g1,g2,g3,g4];
> points := [GeodesicsIntersection(x,y,H) : x in lines,y in lines];
>  DisplayPolygons(lines cat points,"/tmp/pic.ps":
>  Labels := [-1,0,1,2,5,6],Colours := [1,0,0],Show := true);
[ -1.0000000000000000000000000000, 6.0000000000000000000000000000,
3.3284271247461900976033774484, 91 ]
```

130.9 Graphical Output

When working with a particular congruence subgroup it is often useful to be able to produce a picture of a fundamental domain of the group. Also when considering images of geodesics in $\mathbf{H}^*$ it can be useful to draw the images. The following graphics functions provide a useful tool for this purpose.

`DisplayPolygons(P,file)`

`Colours`	SeqEnum	*Default :* $[1,1,0]$
`Outline`	BoolElt	*Default :* `true`
`Fill`	BoolElt	*Default :* `true`
`Show`	BoolElt	*Default :* `false`
`Labels`	SeqEnum	*Default :* $[0,1]$
`Fontsize`	RngIntElt	*Default :* 2
`Size`	SeqEnum	*Default :* [ ]
`Pixels`	RngIntElt	*Default :* 300

`Overwrite`	BoolElt	*Default :* `false`
`Radius`	FldReElt	*Default :* 0.5
`PenColours`	SeEnum	*Default :* $[0, 0, 0]$

Given a sequence of polygons, each of which is defined by a sequence of points in the upper half plane, produce the postscript drawing of these polygons, and write the result to the named file. The function returns a sequence of 4 real numbers, `[x0,x1,h,S]`, where `(x0,0)` are the coordinates of the lower left corner, `(x1,h)` are the coordinates of the upper right corner, and the scale is such that there are `S` pixels per unit. These values are either given by the user, or computed automatically.

`Colours` : Either a sequence of 3 numbers, or a sequence of such sequences, one for each polygon in the sequence of polygons given. A colour is represented by 3 real numbers between 0 and 1, giving the red, green and blue components of the colour. This colour is used for the filling of the polygon.

`PenColours` : A sequence of 3 numbers, or a sequence of such sequences, one for each polygon in the sequence of polygons given. A colour is represented by 3 real numbers between 0 and 1, giving the red, green and blue components of the colour. This colour is used for drawing the outline of the polygon.

`Labels` : A sequence of elements of type `SetCspElt`, `FldRatElt` or `RngIntElt`, which will be labeled on the real axis in the diagram.

`Fontsize` : The size of the font used in labeling the diagram.

`Pixels` : When using Autoscale, the scale is computed so that the width of the resulting diagram is given by `Pixels` in pixels, plus a 20 pixel border. The minimal possible value for `Pixels` is 10, and if any smaller value is given it will be set to 10.

`Size` : If `Size` is not the empty sequence, the values in the sequence `Size` are used to determine the size of the image. When `Size` is given by `[x0,x1,y,S]`, the picture is draw with coordinates `(x0,0)` in the lower left corner, `(x1,h)` in the upper right corner, and the scale is such that there are `S` pixels per unit. If `Size` is not given, then an appropriate size for the image is computed automatically.

`Overwrite` : If `Overwrite` is set to `true`, then if the name of the file given names a file which already exists, it will be overwritten. Otherwise the user will be asked whether they want to overwrite the file or not.

`Show` : If `Show` is set to `true` a System command is issued to make a window pop up showing the file just created. The system command is `System("gv file &")`. In future other options may be possible. If `Show` is `false` then

`Fill` : A boolean or sequence of booleans, determining whether the polygon is drawn filled in with a colour, the colour coming from the sequence

`Colours`. If both `Fill` and `Outline` are set to `false` outline is reset to `true`.

`Outline` : A boolean or sequence of booleans, determining whether the polygon is drawn with an outline. If both `Fill` and `Outline` are set to `false` outline is reset to `true`.

`Radius` : A real number giving the radius of points marked on the diagram. A point will be marked for any polygon given by a single point.

`DisplayFareySymbolDomain(FS,file)`

`DisplayFareySymbolDomain(G,file)`

`Colour`	SeqEnum	*Default :* $[1, 1, 0]$
`Show`	BoolElt	*Default :* `false`
`Fontsize`	RngIntElt	*Default :* 2
`Labelsize`	RngIntElt	*Default :* 3
`Autoscale`	BoolElt	*Default :* `true`
`Size`	SeqEnum	*Default :* []
`Pixels`	RngIntElt	*Default :* 300
`Overwrite`	BoolElt	*Default :* `false`
`ShowInternalEdges`	BoolElt	*Default :* `false`

`Display` : a fundamental domain corresponding to the Farey Symbol FS, or a group G, for which the Farey symbol will be computed, with edge identifications and cusps labeled. This function returns a sequence of 4 real numbers, `[x0,x1,h,S]`, where `(x0,0)` are the coordinates of the lower left corner, `(x1,h)` are the coordinates of the upper right corner, and the scale is such that there are `S` pixels per unit. These values are either given by the user, or computed automatically. A polygon can be a sequence of any number of points of type `HypSpcElt` or `SetCspElt`. When at least 3 points are given a polygon is drawn. For two points, a geodesic between them is drawn. For one point a small circle is drawn to mark the point.

`Colour` : This colour is used in colouring the domain drawn. A colour is represented by a sequence of 3 real numbers between 0 and 1, giving the red, green and blue components of the colour.

`Fontsize` : The size of the font used in labeling the cusps on the diagram.

`Labelsize` : The size of the font used in labeling the identification labels on the diagram.

`Autoscale` : If `true`, the scale is computed and the file is created so that all the polygons listed can be seen in the diagram, if this is feasible. if `Autoscale` is set to `false`, then the variable `Size` must be given.

`Pixels` : When using `Autoscale`, the scale is computed so that the width of the resulting diagram is given by `Pixels` in pixels, plus a 20 pixel border. The minimal possible value for `Pixels` is 10, and if any smaller value is given it will be set to 10.

`Size` : When `Autoscale` is `false`, the values in the sequence `Size` are used to determine the size of the image. When `Size` is given by `[x0,x1,y,S]`, the picture is draw with coordinates `(x0,0)` in the lower left corner, `(x1,h)` in the upper right corner, and the scale is such that there are `S` pixels per unit.

`Overwrite` : If `Overwrite` is set to `true`, then if the name of the file given names a file which already exists, it will be overwritten. Otherwise the user will be asked whether they want to overwrite the file or not.

`Show` : If `Show` is set to `true` a System command is issued to make a window pop up showing the file just created. The system command is `System("gv file &")`. In future other options may be possible. If `Show` is `false` then

`ShowInternalEdges` : If this is set to `true` then internal edges are drawn dividing the domain into fundamental domains for $\Gamma_0(2)$.

Example H130E9

In the first example we draw a fundamental domain for $\Gamma_0(12)$.

```
> G := Gamma0(24);
> FS := FareySymbol(G);
> C := Cusps(FS);
> Domain := FundamentalDomain(G);
> red := [1,0,0];
>  DisplayPolygons(Domain,"/tmp/Gamma0N12.ps":
>      Colours := red, Labels := C, Show := true, Fontsize := 1);
[ 0.E-28, 1.0000000000000000000000000000, 1.5000000000000000000000000000, 300 ]
```

The user could write a procedure as follows for drawing the fundamental domain as a union of fundamental domains for $\mathrm{PSL}_2(\mathbf{Z})$.

```
> procedure draw_fundamental_domain(Group,file)
>     cosets := CosetRepresentatives(Group);
>     H<i,r> := UpperHalfPlaneWithCusps();
>     tri := [H|Infinity(),0,r];
```

```
>     red := [1,0,0];
>     yellow := [1,1,0];
>     cyan := [0,1,1];
>     cols := &cat[[red,yellow,cyan] : i in [1..(Ceiling(#cosets/3))]];
>     trans := [g*tri : g in cosets];
>     Outlines := [false : g in cosets];
>     DisplayPolygons(trans,file:
>         Colours := cols, Outline := Outlines, Show := true);
> end procedure;
>
> // use the procedure to draw a domain for Gamma_0(19):
>  draw_fundamental_domain(Gamma0(19),"/tmp/gamma019.ps");
[ 0.E-28, 1.0000000000000000000000000000, 1.5000000000000000000000000000, 300 ]
```

In the following example, we use the cosets of $\Gamma_0(11)$ to draw a fundamental domain for $\Gamma_0(11)$, divided up into fundamental domains for the group of automorphisms of $\mathbf{H}^*$ generated by $\mathrm{PSL}_2(\mathbf{Z})$ together with the map $z \mapsto -\overline{z}$.

```
> G := Gamma0(11);
> H<i,rho> := UpperHalfPlaneWithCusps();
> tri := [H|Infinity(),i,rho];
> tri1 := [H|0,i,rho];
> C11 := CosetRepresentatives(G);
> Colours := [[0.6,0.2,0.9] : i in [1..#C11]] cat [[1,1,0] : i in [1..#C11]];
> triangles := [g*tri : g in C11] cat [g*tri1 : g in C11];
>  DisplayPolygons(triangles,"/tmp/Gamma_0_11.ps":
>      Colours := Colours, Show := false);
[ 0.E-57, 1.0000000000000000000000000000, 1.5000000000000000000000000000, 300 ]
```

Continuing from the above example, we can take any other polygon we wish, and look at it's translates under the action of the cosets of $\Gamma_0(11)$.

```
> tri2 := [H|Infinity(),2*i + 1,i,rho];
>  DisplayPolygons([g*tri2 : g in C11],"/tmp/pic.ps":
>      Colours := Colours, Show := true);
[ 0.E-57, 1.0000000000000000000000000000, 2.5000000000000000000000000000, 300 ]
```

In the next example we define three different triangles. Each of these is a fundamental domain for $\Gamma^0(2)$, though MAGMA currently does not verify this. For each choice we use a set of generators of $\Gamma^0(2)$ to draw a partial tiling of the upper half plane with equivalent fundamental domains.

```
> G := CongruenceSubgroup(4,2);
> H<i,r> := UpperHalfPlaneWithCusps();
> generators := Generators(G);
> M := generators cat [g^(-1) : g in generators] cat [G!1];
> tri1 := [H|Infinity(),r-1,0,r,r+1];
> tri2 := [H|Infinity(),i-1,0,i+1];
> tri3 := FundamentalDomain(G);
> X := [1..#M];
> L := [G!m : m in Set([Matrix(M[i]*M[j]*M[k]) : i in X, j in X, k in X])];
>  DisplayPolygons([g*tri1 : g in L],"/tmp/picture1.ps": Show := true);
```

```
[ -4.000000000000000000000000000, 3.500000000000000000000000000,
1.500000000000000000000000000, 200 ]
>  DisplayPolygons([g*tri2 : g in L],"/tmp/picture2.ps": Show := true);
[ -4.000000000000000000000000000, 3.000000000000000000000000000,
1.500000000000000000000000000, 200 ]
>  DisplayPolygons([g*tri3 : g in L],"/tmp/picture3.ps": Show := true);
[ -4.000000000000000000000000000, 3.000000000000000000000000000,
1.500000000000000000000000000, 200 ]
```

Edges and Polygons with different numbers of sides can be included in the same picture, as in the following example.

```
> H<i,r> := UpperHalfPlaneWithCusps();
> cosets := CosetRepresentatives(Gamma0(19));
> polygons := [g*tri : g in cosets, tri in [[H|i,r],[H|i,2*r,3*r-1]]];
> outlines := [true : i in cosets] cat [false : i in cosets];
> cyan := [0,1,1];
>  DisplayPolygons(polygons,"/tmp/pic.ps":
>      Colours := cyan, Outline := outlines, Labels := [], Show := false);
[ 0.E-57, 1.000000000000000000000000000, 3.098076211353315940291169512, 300 ]
```

In the following example a function is defined to determine the colouring of the polygons in terms of which Farey sequence pairs of end points of the polygons belong to.

```
> frac := func<a | a[1]/a[2]>;
>
> function FareyValue(m)
>   mat := Matrix(m);
>   Denominators := [mat[2,1],mat[2,2],mat[2,1]+mat[2,2]];
>   values := [Abs(v) : v in Denominators];
>   return &+ContinuedFraction(frac(Sort(values)));
> end function;
>
> procedure drawDomain(cosets)
>    H<i,r> := UpperHalfPlaneWithCusps();
>    tri := [H|Infinity(),0,r];
>    cols := [[0.5,0.2*FareyValue(c),1-0.2*FareyValue(c)] : c  in cosets];
>    trans := [g*tri : g in cosets];
>    DisplayPolygons(trans,"/tmp/pic.ps":
>        Outline := false, Colours := cols, Show := true);
> end procedure;
>
> // Now we can use the above functions and procedure, for example
> // take the cosets as follows:
> C := CosetRepresentatives(Gamma0(41));
> // Then draw the picture:
>  drawDomain(C);
[ 0.E-28, 1.000000000000000000000000000, 1.500000000000000000000000000, 300 ]
```

Example H130E10

Functions are provided for drawing pictures coming from Farey symbols and fundamental domains. These functions provide short cuts for producing some of the pictures in the previous examples, but are not so flexible.

```
> G := CongruenceSubgroup(5);
> FS := FareySymbol(G);
> FS;
[ 2, 2, 7, 8, 8, 10, 9, 3, 3, 9, 4, 1, 1, 4, 5, 5, 10, 11, 11, 7, 6, 6 ]
[ oo, 0, 1/5, 2/9, 1/4, 3/11, 5/18, 2/7, 1/3, 3/8, 5/13, 2/5, 1/2, 3/5, 5/8, 2/3,
5/7, 8/11, 3/4, 7/9, 4/5, 1, oo ]
> // The following command graphically displays the information contained in FS:
>  DisplayFareySymbolDomain(FS,"/tmp/Gamma5b.ps": Show := false);
[ 0, 1.0000, 0.50000, 800 ]
> // This picture represents a modular curve with genus 0:
> Genus(G);
0
```

Example H130E11

In the following example we show how to create a postscript file and include the resulting file in a latex file.
The example in question is $\Gamma_0(37)$. A simple question is to find what is the fundamental domain, and to give a list of inequivalent elliptic points of the group, and display this information graphically. First we create the group and find the information we are interested in:

```
> G := Gamma0(37);
> // To draw a picture of the fundamental domain we need to define
> // the upper half plane.
> H<i,r> := UpperHalfPlaneWithCusps();
> D := FundamentalDomain(G,H);
> E := EllipticPoints(G,H);
```

We can now take a look at the picture:

```
> // We need to make sure the polygons we want to display all
> // have the same type, so we have t create the following object:
> HH:=Parent(D);
> // now we make a list of polygons and points:
> P1:=[HH|D] cat [HH|[e] : e in E];
> // we take a look at the default picture of the situation:
>  DisplayPolygons(P1,"/tmp/pic.ps": Show := true);
[ 0.E-28, 1.0000, 1.5000, 300 ]
```

The points have been displayed in yellow, the default colour, but we'd prefer a different colour. Also the picture is not quite the right size. It has been shown with the real axis shown from 0 to 1, which is reasonable, but we'd prefer less height, and the scale 300 could be changed, as in this example:

```
> // use different colours:
```

```
> yellow := [1,1,0];
> red := [1,0,0];
> green := [0,1,0];
> cyan := [0,1,1];
> black := [0,0,0];
> Colours := [yellow] cat [red : e in E];
> // use a different size and scale:
> Size := [0,1,0.2,400];
> // view the result:
>  DisplayPolygons(P1,"/tmp/pic.ps":
>      Show := true, Colours := Colours, Size := Size);
[ 0, 1, 0.19999999999999999999999999999, 400 ]
```

After more experimenting we reach the following:

```
> // We use a slightly different list of polygons, to show more
> // structure.
> tri := [H|Infinity(),0,H.2];
> cosets := CosetRepresentatives(G);
> P := [HH|g*tri : g in cosets] cat [HH|[e] : e in E] cat [HH|D];
> // Choosing some colours:
> LotsOfColours := &cat[[yellow,green,cyan] : c in cosets];
> Colours := [LotsOfColours[i] : i in [1..#cosets]]
>            cat [black]
>            cat [red : e in E];
> // choose which polygons are filled, and which have outlines:
> outlines := [false : i in cosets] cat [false : i in E] cat [true];
> fill := [true : i in cosets] cat [true : i in E] cat [false];
> labels := Cusps(FareySymbol(G));
> // Create a file of the result:
>  DisplayPolygons(P,"/tmp/pic.ps":
>      Colours := Colours, Show := false, Size := [0,1,1,200],
>      Labels := labels, Outline := outlines, Fill := fill,
>      Fontsize := 1, Radius := 0.3);
[ 0, 1, 1, 200 ]
```

If we are satisfied with the resulting picture then we can include the resulting picture in a latex file as in the following example, where we use the Latex `graphicx` package to include the diagram.

```
\documentclass{article}
\usepackage[dvips]{graphicx}
\begin{document}
Here is a picture of a
fundamental domain for $\Gamma_0(37)$:
$${\includegraphics{/tmp/pic.ps}}$$
In the picture we have marked $4$ inequivalent
elliptic points.
```

```
\end{document}
```

Note that currently, given a list of points to draw, the postscript file will include all of the points, even if they are not in the area of the bounding box, which can produce bad results some times. This should be changed in a future version. Sometimes using the Fundamental domain function can produce results in a more reliable way, as in the following example, which produces a similar picture to the above, and a latex file can be written in the same way.

```
> G := Gamma0(37);
> H<i,r> := UpperHalfPlaneWithCusps();
> D := FundamentalDomain(G,H);
> HH:=Parent(D);
> E := EllipticPoints(G,H);
> P:=[HH|D] cat [HH|[e] : e in E];
> labels := Cusps(FareySymbol(G));
> yellow := [1,1,0];
> red := [1,0,0];
> Colours := [yellow] cat [red : e in E];
>  DisplayPolygons(P,"/tmp/pic.ps":
>      Colours := Colours, Show := false, Size := [0,1,0.2,200],
>      Labels := labels,
>      Fontsize := 1, Radius := 0.3);
[ 0, 1, 0.19999999999999999999999999999, 200 ]
```

130.10 Bibliography

[**Kul91**] Ravi S. Kulkarni. An arithmetic-geometric method in the study of the subgroups of the modular group. *Amer. J. Math.*, 113(6):1053–1133, 1991.

[**Ver00**] H. A. Verrill. Fundamental domain drawing program. URL:http://hverrill.net/fundomain/, 2000.

131 ARITHMETIC FUCHSIAN GROUPS AND SHIMURA CURVES

Chapter 131
ARITHMETIC FUCHSIAN GROUPS AND SHIMURA CURVES

In this chapter, we document algorithms for arithmetic Fuchsian groups. Let F be a totally real number field. Let A be a *quaternion algebra* over F, a central simple algebra of dimension 4 (see Chapter 86), such that A is split at exactly one real place, corresponding to the injective map $\iota_\infty : A \to M_2(\mathbf{R})$. Let $\mathcal{O}$ be a maximal order in A (see also Section 81.4), let $\mathcal{O}^*$ denote the group of elements of $\mathcal{O}$ of reduced norm 1, and let $\Gamma(1) = \iota_\infty(\mathcal{O}^*)/\{\pm 1\} \subset PSL_2(\mathbf{R})$. An *arithmetic Fuchsian group* Γ is a discrete subgroup of $PSL_2(\mathbf{R})$ which is commensurable with $\Gamma(1)$ (for some choice of F and A).

The classical case of the modular groups corresponds to $F = \mathbf{Q}$, $A = M_2(\mathbf{Q})$, $\mathcal{O} = M_2(\mathbf{Z})$, and $\Gamma(1) = \mathrm{PSL}_2(\mathbf{Z})$. Specialized algorithms for this case apply, and they are treated in detail in Chapter 130. To exclude this case, we assume throughout that A is a division ring, or equivalently $A \ncong M_2(\mathbf{Q})$.

The group Γ acts properly and discontinuously on the upper half-plane $\mathbf{H}$, and the quotient $\Gamma\backslash\mathbf{H} = X(\Gamma)$ can be given the structure of a compact Riemann surface, called a *Shimura curve*.

We exhibit methods for computing with arithmetic Fuchsian groups Γ, including the basic invariants of Γ and a fundamental domain for $X(\Gamma)$. We provide further specialized algorithms for triangle groups. Along the way, we also present an interface for computing with the unit disc, parallel to that for the upper half-plane (see Chapter 130).

The algorithm used to compute fundamental domains is described in [Voi09]. We recommend the following additional reading concerning the algorithms in this section. For an introduction to Fuchsian groups, see Katok [Kat92] and Beardon [Bea77], and for their relationship to quaternion algebras, see also Vignéras [Vig80]. For a computational perspective on arithmetic Fuchsian groups and Shimura curves, see Alsina-Bayer [AB04], Elkies [Elk98], and Voight [Voi05], and for a discussion of triangle groups, see Voight [Voi06].

131.1 Arithmetic Fuchsian Groups

In this section, we computing the basic invariants of an arithmetic Fuchsian group Γ.

131.1.1 Creation

We begin by giving the basic constructors for arithmetic Fuchsian groups.

`FuchsianGroup(O)`

Returns the arithmetic Fuchsian group Γ corresponding to the group $\mathcal{O}^*$ of units of reduced norm 1 in the quaternion order $\mathcal{O}$.

`FuchsianGroup(A)`

Returns the arithmetic Fuchsian group Γ corresponding to the group $\mathcal{O}^*$ of units of reduced norm 1 in a maximal order $\mathcal{O}$ in the quaternion algebra A.

`FuchsianGroup(A, N)`

Returns the arithmetic Fuchsian group Γ corresponding to the group $\mathcal{O}^*$ of units of reduced norm 1 in an order $\mathcal{O}$ of level $\mathfrak{N}$ in the quaternion algebra A.

Example H131E1

In this example, we construct arithmetic Fuchsian groups in three ways. First, we construct the group associated to the quaternion algebra of discriminant 6 over $\mathbf{Q}$.

```
> A := QuaternionAlgebra(6);
> G := FuchsianGroup(A);
> G;
Arithmetic Fuchsian group arising from order of Quaternion Algebra with base
ring Rational Field
> O := BaseRing(G);
> O;
Order of Quaternion Algebra with base ring Rational Field
with coefficient ring Integer Ring
> Discriminant(O);
6
> Algebra(O) eq A;
true
```

Next, we construct a group "by hand", associated to the quaternion algebra over the totally real subfield $F = \mathbf{Q}(\zeta_9)^+$ of $\mathbf{Q}(\zeta_9)$ ramified only at two of the three real infinite places.

```
> K<z> := CyclotomicField(9);
> F := sub<K | z+1/z >;
> Degree(F);
3
> Z_F := MaximalOrder(F);
> Foo := InfinitePlaces(F);
> A := QuaternionAlgebra(ideal<Z_F | 1>, Foo[2..3]);
> Discriminant(A);
Principal Ideal of Z_F
Generator:
    [1, 0, 0]
[ 2nd place at infinity, 3rd place at infinity ]
> O := MaximalOrder(A);
> G := FuchsianGroup(O);
> G;
Arithmetic Fuchsian group arising from order of Quaternion Algebra with base
ring Field of Fractions of Maximal Equation Order with defining polynomial x^3 -
```

```
3*x - 1 over its ground order
```

Lastly, we construct the group of level 3 inside a quaternion algebra ramified at $2\infty_1$ over $\mathbf{Q}(\sqrt{5})$.

```
> F<x> := NumberField(Polynomial([-5,0,1]));
> Z_F := MaximalOrder(F);
> A := QuaternionAlgebra(ideal<Z_F | 2>, InfinitePlaces(F)[1..1]);
> G := FuchsianGroup(A, ideal<Z_F | 3>);
> O := BaseRing(G);
> O;
Order of Quaternion Algebra with base ring Field of Fractions of Z_F
with coefficient ring Maximal Order of Equation Order with defining polynomial
x^2 - 5 over its ground order
> Discriminant(O);
Principal Ideal of Z_F
Generator:
    6/1*Z_F.1
> Discriminant(A);
Principal Prime Ideal of Z_F
Generator:
    [2, 0]
[ 1st place at infinity ]
> Level(G);
Principal Ideal of Z_F
Generator:
    3/1*Z_F.1
```

131.1.2 Quaternionic Functions

In this section, we provide some functions related quaternion algebras which underlie the other functions in this chapter.

Let A be a quaternion algebra over a totally real field F, represented in standard form by $\alpha, \beta \in A$ satisfying $\alpha^2 = a$, $\beta^2 = b$, and $\beta\alpha = -\alpha\beta$. Suppose that A has a unique split real place v; we then take this place to be the identity and consider F as a subfield of $\mathbf{R}$. Then at least one of $a, b > 0$, and without loss of generality we may assume that $a > 0$.

We then define the embedding $\iota : A \to M_2(\mathbf{R})$, given by

$$\alpha \mapsto \begin{pmatrix} \sqrt{a} & 0 \\ 0 & -\sqrt{a} \end{pmatrix}, \quad \beta \mapsto \begin{pmatrix} 0 & \sqrt{|b|} \\ \mathrm{sgn}(b)\sqrt{|b|} & 0 \end{pmatrix}.$$

This particular choice of embedding is governed by the following. The reduced norm $\mathrm{nrd} : A \to \mathbf{R}$ given by

$$\gamma = x + y\alpha + z\beta + w\alpha\beta \mapsto \mathrm{nrd}(\gamma) = x^2 - ay^2 - bz^2 + abw^2$$

is not a Euclidean norm, so it is natural to instead take

$$\gamma \mapsto \mathrm{nrd}'(\gamma) = x^2 + |a|y^2 + |b|z^2 + |ab|w^2,$$

which corresponds to

$$\mathrm{nrd}'(\gamma) = x^2 + y^2 + z^2 + w^2 \text{ if } \iota(\gamma) = \begin{pmatrix} x & y \\ z & w \end{pmatrix}.$$

Combining this new norm, defined for the identity real place, with the ones corresponding to the nonidentity real places σ, we may then define the definite norm

$$N(\gamma) = \mathrm{nrd}'(\gamma) + \sum_{\sigma \neq \mathrm{id}} \sigma(\mathrm{nrd}(\gamma)).$$

This norm (and the corresponding Gram matrix) on A are used in the application to fundamental domains (see the next section).

`QuaternionOrder(G)`

`BaseRing(G)`

The order in some quaternion algebra that was used to define the Fuchsian group G.

`QuaternionAlgebra(G)`

The quaternion algebra used to define the Fuchsian group G.

`SplitRealPlace(A)`

Returns the unique real place at which A is split, if it exists.

`FuchsianMatrixRepresentation(A)`

Returns the map $A \to M_2(\mathbf{R})$ when A has a unique split place, as defined above.

`DefiniteNorm(gamma)`

Returns the definite norm of $\gamma \in A$ for A a quaternion algebra with a unique split real place, as defined above.

`DefiniteGramMatrix(B)`

Returns the definite Gram matrix for the basis B of a quaternion algebra A with a unique split real place, as defined above.

Example H131E2

We begin by defining the quaternion algebra over $F = \mathbf{Q}(\sqrt{5})$ which is ramified at the prime ideal 2 and the second real place ∞_2, as indexed by MAGMA.

```
> F<w> := QuadraticField(5);
> Z_F := MaximalOrder(F);
> Foo := InfinitePlaces(F);
> A<alpha,beta> := QuaternionAlgebra(ideal<Z_F | 2>, [Foo[2]]);
> a, b := StandardForm(A);
> F!a, F!b;
-1 w + 2
> Norm(a);
1
```

We see that A is isomorphic to the algebra represented in standard form by

$$\alpha^2 = a,\ \beta^2 = b,\ \beta\alpha = -\alpha\beta$$

where $a = -1$, $b = 2 + \sqrt{5}$. (Note that the algorithm used to construct A is probabilistic, so a and b are not canonically given.)
We now demonstrate the real embedding corresponding to the split real place ∞_1.

```
> v := SplitRealPlace(A);
> v;
1st place at infinity
> iota := FuchsianMatrixRepresentation(A);
> iota(alpha);
[0.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    00000000000000000000 1.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    00000000000000000000000000000000000000000000000000]
[-1.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    00000000000000000000 0.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    00000000000000000000000000000000000000000000000000]
> iota(beta);
[2.05817102727149225032198104758045042123873009967781948628154235264047962315837
    78925542297104146968394 0.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    00000000000000000000000000000000000000000000000000]
[0.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    00000000000000000000 -2.05817102727149225032198104758045042123873009967
    78194862815423526404796231583789255422971041469683941
> Sqrt(Evaluate(b, v));
2.05817102727149225032198104758045042123873009967781948628154235264047962315837
89255422971041469683941
> DefiniteNorm(alpha);
3.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000
```

We see indeed that

$$\alpha \mapsto \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad \beta \mapsto \begin{pmatrix} \sqrt{2+\sqrt{5}} & 0 \\ 0 & -\sqrt{2+\sqrt{5}} \end{pmatrix}$$

and

$$N(\alpha) = \mathrm{nrd}'(\alpha) + \sigma(\mathrm{nrd}(\alpha)) = (0^2 + 1^2 + (-1)^2 + 0^2) + 1 = 3.$$

`MultiplicativeOrder(gamma)`

`PlusMinus` BoolElt *Default :* `true`

Computes the order of the element γ of a quaternion algebra; either a finite number or 0 if the element has infinite order. If `PlusMinus` eq `true`, then compute the order in the group of units modulo ± 1.

`Quaternion(g)`

For g an element of an arithmetic Fuchsian group, return the underlying element of the quaternionic order.

131.1.3 Basic Invariants

In this section, we compute the basic invariants associated to an arithmetic Fuchsian group Γ.

The first is the hyperbolic volume $\mathrm{vol}(X)$ of the orbit space $X = \Gamma\backslash\mathbf{H}$, which can be computed via a formula of Shimizu:

$$\mathrm{vol}(X) = \left(\frac{1}{2}\right)^{d-2} \zeta_F(-1) \prod_{\mathfrak{p}|\mathrm{disc}A} (N(\mathfrak{p}) - 1)$$

where $d = [F : \mathbf{Q}]$. This volume is normalized so that an ideal triangle (i.e. a triangle with all vertices on the boundary of $\mathbf{H}$) has volume $1/2$. With this normalization, $\mathrm{vol}(X)$ takes rational values.

The next invariant is the signature of Γ. A point $z \in \mathbf{H}$ is an *elliptic point* of order $k \geq 2$ if the stabilizer of z under Γ, which is a finite cyclic group, has order k. There are only finitely many Γ-conjugacy classes of elliptic points, also known as *elliptic cycles*. If Γ has t elliptic cycles of order $m_1, \ldots, m_t$, and X has genus g, then we say that Γ has *signature* $(g; m_1, \ldots, m_t)$.

We compute the number of elliptic cycles of order k by a formula (too complicated to state here) which requires the computation of the relative class group of a CM extension of number fields. The genus g can then be calculated by the Riemann-Hurwitz formula:

$$\mathrm{vol}(X) = 2g - 2 + \sum_{i=1}^{t} \left(1 - \frac{1}{m_i}\right).$$

`ArithmeticVolume(G)`

Returns the hyperbolic volume of the quotient of the upper half-plane by G for an arithmetic Fuchsian group G. The volume is normalized arithmetic volume, so the "usual" volume is divided by 2π; this gives an ideal triangle volume $1/2$.

`EllipticInvariants(G)`

Returns the number of elliptic cycles of the arithmetic Fuchsian group G as a sequence of elliptic orders and their multiplicities.

`Signature(G)`

Returns the signature of the arithmetic Fuchsian group G.

131.1.4 Group Structure

We now compute a finite presentation for an arithmetic Fuchsian group Γ.

If Γ has signature $(g; m_1, \ldots, m_t)$, then there exists a standard presentation for Γ, generated (freely) by $\alpha_1, \beta_1, \ldots, \alpha_g, \beta_g, \gamma_1, \ldots, \gamma_t$ with the relations

$$\gamma_1^{m_1} = \ldots = \gamma_t^{m_t} = [\alpha_1, \beta_1] \cdots [\alpha_g, \beta_g] \gamma_1 \cdots \gamma_t = 1$$

where $[\alpha_i, \beta_i] = \alpha_i \beta_i \alpha_i^{-1} \beta_i^{-1}$ is the commutator. In the special case where $g = 0$, there exists an algorithm to reconstruct the standard presentation; if $g > 0$, we only obtain a finite presentation, and it is possible that the construction of a standard presentation may be algorithmically impractical.

Our algorithm uses a fundamental domain F for Γ (see section 131.3). In brief, we compute the set of elements which pair the sides of F; these elements are provably generators for the group Γ, and the relations between them are given as the set of minimal loops in a graph which records the side pairings.

`Group(G)`

Returns a presentation U for the arithmetic Fuchsian group G, a map $U \to G$, and a map $U \to \mathcal{O}$ where $\mathcal{O}$ is the quaternion order corresponding to G.

Example H131E3

In this example, we compute the basic invariants of the arithmetic Fuchsian group associated to a maximal order in the quaternion algebra of discriminant 6 over $\mathbf{Q}$.

```
> A := QuaternionAlgebra(6);
> O := MaximalOrder(A);
> G := FuchsianGroup(O);
> ArithmeticVolume(G);
1/3
> EllipticInvariants(G);
[ <2, 2>, <3, 2> ]
> Genus(G);
0
> Signature(G);
<0, [2, 2, 3, 3]>
```

We verify the Riemann-Hurwitz formula $1/3 = 2 \cdot 0 - 2 + 2(1 - \frac{1}{2}) + 2(1 - \frac{1}{3})$.

We can also compute a finite presentation of this group.

```
> U, m := Group(G);
> U;
Finitely presented group U on 3 generators
Relations
    U.1^2 = Id(U)
    U.2^3 = Id(U)
    U.3^3 = Id(U)
    (U.2 * U.1 * U.3)^2 = Id(U)
> [Matrix(m(U.i)) : i in [1..2]];
[
    [-0.44721359549995793928183473374634344441856698198020262470211905349191088
        22850343946302640398390408845 -1.09544511501033222691393956560182030636
        17511365128344804256651366192701912074123047253565164680093 74]
    [1.09544511501033222691393956560182030636175113651283448042566513661927019191
        20741230472535651646800937 4 0.44721359549995793928183473374634344441856
        69819802026247021190534919108822850343946302640398390408845],
    [0.17082039324993690892275210061973565995395874811504763667302759126238306632
        32123512881384895516677240064 0.41842280112390929127226731105032485970030
        73307330865863518420141891341904873780366272041685003642481]
    [-2.86791254390708738946955138575621625166625481138975671478453458144009456
        79446930631670636016050044833 -1.17082039324993690892275210061973565995395
        95874811504763667302759126238306632123512881384895516677240 1]
]
> [A!Quaternion(m(U.i)) : i in [1..2]];
[ -1/5*j + 1/5*k, -1/2 - 1/2*i + 3/10*j - 3/10*k ]
```

Next we compute a more complicated example, exhibiting an elliptic cycle of order 11. The group Γ corresponds to $F = \mathbf{Q}(\zeta_{11})^+$ and A the quaternion algebra only at real places. (See also the section on triangle groups.)

```
> K<z> := CyclotomicField(11);
> F := sub<K | z+1/z >;
> Foo := InfinitePlaces(F);
> Z_F := MaximalOrder(F);
> A := QuaternionAlgebra(ideal<Z_F | 1>, Foo[1..4]);
> G := FuchsianGroup(A);
> Signature(G);
<0, [ 2, 3, 11 ]>
> U, m := Group(G);
> U;
Finitely presented group U on 2 generators
Relations
    U.1^2 = Id(U)
    U.2^3 = Id(U)
```

```
    (U.1 * U.2^-1)^11 = Id(U)
```

As a final example, we show that the finite presentation of G may indeed be quite complicated. Here, we exmaine the (torsion-free) group associated to the quaternion algebra over $\mathbf{Q}$ with discriminant 35.

```
> G := FuchsianGroup(QuaternionOrder(35));
> Signature(G);
<3, []>;
> time U := Group(G);
Time: 78.870
> U;
Finitely presented group U on 6 generators
Relations
    U.5 * U.4^-1 * U.5^-1 * U.1 * U.3 * U.2^-1 * U.3^-1 * U.6 * U.4 *
    U.2 * U.1^-1 * U.6^-1 = Id(U)
```

131.2 Unit Disc

Let D denote the Poincaré unit disc,

$$D = \{z \in \mathbf{C} : |z| < 1\}$$

equipped with the hyperbolic metric

$$d(z, w) = \log\left(\frac{|1 - z\bar{w}| + |z - w|}{|1 - z\bar{w}| - |z - w|}\right).$$

We mimic the existing functionality for working with the upper half-plane (see Chapter 130) to compute in a similar way with the geometry of the unit disc, including computation of angles, intersections, areas, and so on.

131.2.1 Creation

`UnitDisc()`

`Center`	RngElt	*Default :* (9/10)i
`Precision`	RngElt	*Default :* 0

The hyperbolic unit disc, mapped conformally to the upper half-plane by mapping 0 in D to `Center`, with given `Precision`.

131.2.2 Basic Operations

`D ! x`

Coerces x into D, if possible.

`x eq y`

Returns `true` if and only if $x = y$.

`a * x`

Returns $a \cdot x$.

`x + y`

Returns $x + y$.

`x - y`

Returns $x - y$.

`x / a`

Returns $(1/a) \cdot x$.

131.2.3 Access Operations

`IsExact(z)`

Returns `true` (and the exact value of z) if and only if z is exact.

`ExactValue(z)`

Returns the exact value of z; if it does not exist, returns an error.

`ComplexValue(z)`

`Precision` RngIntElt *Default :* 0

Returns the complex value z, with given precision.

`Im(z)`
`Imaginary(z)`

`Precision` RngIntElt *Default :* 0

Returns the imaginary part of z, with given precision.

`Re(z)`
`Real(z)`

`Precision` RngIntElt *Default :* 0

Returns the real part of z, with given precision.

`Argument(z)`

`Precision` RngIntElt *Default :* 0

Returns the argument of z, with given precision.

`Abs(z)`

`AbsoluteValue(z)`

`Precision` RngIntElt *Default :* 0

Returns the absolute value of z, with given precision.

Example H131E4

In this example, we exhibit basic functions for the unit disc.

```
> D := UnitDisc();
> D;
Hyperbolic unit disc
> CC<I> := ComplexField();
> w := D ! ((1+I)/2);
> w;
0.50000000000000000000000000000 + 0.50000000000000000000000000000*I
> Re(w), Abs(w);
0.50000000000000000000000000000 0.70710678118654752440084436210 5
```

Note that since D does not form a ring, when binary operations are performed, we return elements according to the base ring in which they belong; if desired, they can be coerced back into D.

```
> z := D ! 0;
> z;
0
> IsExact(z);
true 0
> ComplexValue(z : Precision := 100);
0.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000
> 2*z;
0
> Type($1);
FldRatElt
> D!(z+w) eq w;
true
```

131.2.4 Distance and Angles

`Distance(z,w)`

`Precision` RngIntElt *Default :* 0

Returns the hyperbolic distance between z and w.

`Geodesic(z,w)`

`Precision` RngIntElt *Default :* 0

Returns the center and radius of the geodesic containing z and w, with given precision.

`TangentAngle(x,y)`

`Precision` RngIntElt *Default :* 0

Returns the angle of the tangent at x of the geodescic from x to y, with given precision.

`Angle(e1,e2)`

`Precision` RngIntElt *Default :* 0

Given two sequences $e_1 = [z_1, z_2]$ and $e_2 = [z_1, z_3]$, returns the angle between the geodesics at z_1.

`ArithmeticVolume(P)`

`Precision` RngIntElt *Default :* 0

The volume of the convex region specified the sequence of elements of the unit disc. The elements must be specified in counterclockwise order by their arguments. The volume is normalized "arithmetic" volume, so the "usual" volume is divided by 2π; this gives an ideal triangle volume $1/2$.

Example H131E5

In this example, we compute geodesics in the unit disc.

```
> D := UnitDisc();
> CC<I> := ComplexField();
> z0 := D!0;
> z1 := D!(1/2*I);
> z2 := D!(1/2);
> Distance(z0,z1);
1.0986122886681096913952452369 2
> Geodesic(z0,z1);
Infty 0.00000000000000000000000000000
> Geodesic(z1,z2);
1.25000000000000000000000000000 + 1.25000000000000000000000000000*I
1.4577379737113251177185382193 9
> TangentAngle(z0,z1);
```

```
1.5707963267948966192313216916 4
> TangentAngle(z1,z2);
-1.0303768265243124637877433270 3
> Angle([z1,z2],[z1,z0]);
2.6011731533192090830190650186 7
> ArithmeticVolume([D | 1/2+1/2*I, -1/2+1/2*I, -1/2-1/2*I, 1/2-1/2*I]);
0.5903344706017330967016043048 99
```

131.2.5 Structural Operations

`T * x`

Returns $T(x)$, using the identification of the unit disc with the upper half-plane.

`Center(D)`

Returns the element in the upper half-plane which maps to 0 in D.

`DiscToPlane(H,z)`

Maps z in a unit disc to $\mathbf{H}$.

`PlaneToDisc(D,z)`

Maps z in an upper half-plane to D.

`Matrix(g,D)`

Returns the matrix representation of g acting on the unit disc D.

`FixedPoints(g,D)`

Returns the fixed points of g acting on D.

`IsometricCircle(g)`

`IsometricCircle(g,H)`

Returns the center and radius of the set of points in the upper half-plane H where g acts as a Euclidean isometry.

`IsometricCircle(g,D)`

Returns the center and radius of the set of points in the unit disc D where g acts as a Euclidean isometry.

`GeodesicsIntersection(x1,x2)`

Returns the intersection in the unit disc of the two geodesics x_1, x_2, where x and y are specified by their end points. If more than one intersection exists, returns a sequence.

`BoundaryIntersection(x)`

Computes the intersection of the geodesic x with the boundary of the unit disc.

Example H131E6

In this example, we illustrate the use of the structural operations on the unit disc. We begin by declaring a Fuchsian group over the totally real subfield $F = \mathbf{Q}(\zeta_9)^+$ of $\mathbf{Q}(\zeta_9)$.

```
> K<z> := CyclotomicField(9);
> F := sub<K | z+1/z >;
> b := F! (z+1/z);
> A<alpha,beta> := QuaternionAlgebra<F | -3, -b>;
> G := FuchsianGroup(A);
> O := BaseRing(G);
> A<alpha,beta> := Algebra(O);
```

Next, we compute the fixed point of the element $g = (1 + \alpha)/2$.

```
> g := G!((1+alpha)/2);
> H := UpperHalfPlane();
> FixedPoints(g, H);
[
    root(-1)
]
> Matrix(g);
[0.500000000000000000000000000000000000000000000000000000000000000000000000000000000000
    000000000000000000000 0.866025403784438646763723170752936183471402626905
    190314027903489725966508454400018540573093378624287 8]
[-0.866025403784438646763723170752936183471402626905190314027903489725966508454
    400018540573093378624287 8 0.500000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000]
> g*H.1;
root(-1);
```

The fixed point is at $z = i$. Next, we compute the isometric circle $C(g)$ of g,

$$C(g) = \{z \in \mathbf{C} : |g(z)| = |z|\}.$$

(See also the next section on fundamental domains.)

```
> cH, rH := IsometricCircle(g);
> cH;
0.577350269189625764509148780501957455647601751270126876018602326483977672302933
3345693715395585749525 2
> rH;
1.154700538379251529018297561003914911295203502540253752037204652967955344605866
6691387430791171499050
```

The cirlce $C(g)$ has radius $r_H = 1.154\ldots$ and center $c_H = 0.577\ldots$. We verify that g acts by a Euclidean isometry at the point $c_H + r_H i \in C(g)$.

```
> CC<I> := ComplexField();
> Abs(Matrix(g)[2,1]*(cH+rH*I)+Matrix(g)[2,2]);
```

```
1.000000000000000000000000000
```

We now repeat these same steps, with g acting on the unit disc D.

```
> D := UnitDisc(: Center := 9/10*H.1);
> Matrix(g, D);
[0.500000000000000000000000000000000000000000000000000000000000000000000000000000000000
    000000000000000000000000 + 0.870836656027685528134632743923817879053886967119
    932875211895468244836121536336192488985662869809493679*$.1
    -0.091413792621690746047281890246448583363214116792146568962386184205317861
    463293527610523203343480504 49*$.1]
[0.091413792621690746047281890246448583363214116792146568962386184205317861463293
    527610523203343480504 49*$.1 0.500000000000000000000000000000000000000000000000
    000000000000000000000000000000000000000000000000000000000000000000000000 -
    0.870836656027685528134632743923817879053886967119328752118954682448361215363
    361924889856628698094936 79*$.1]
> cD, rD := IsometricCircle(g, D);
> Abs(Matrix(g, D)[2,1]*PlaneToDisc(D, H!(cH+rH*I))+Matrix(g, D)[2,2]);
1.000000000000000000000000000
```

131.3 Fundamental Domains

Let D denote the unit disc and Γ an arithmetic Fuchsian group. A *fundamental domain* in D for Γ is a closed, hyperbolically convex region $P \subset D$ such that the translates of P by Γ cover D and such that the interiors of all translates P are disjoint, i.e. $D = \bigcup_{\gamma\in\Gamma} \gamma P$ and $\mathrm{int}(P) \cap \mathrm{int}(\gamma P) = \emptyset$ for $\gamma \neq 1$, where $\mathrm{int}(P)$ denotes the interior of P.

Choosing a point $p \in D$ not fixed by any element of $\Gamma \setminus \{1\}$, we obtain a fundamental domain by letting

$$P = \{z \in D : d(z,p) \leq d(z,\gamma(p)) \text{ for all } \gamma \in \Gamma\}.$$

Typically, one takes $p = 0$.

One can similarly define a fundamental domain in the upper half-plane $\mathbf{H}$, and one can bijectively map fundamental domains in D to those in $\mathbf{H}$ via a choice of conformal map between them. The unit disc is a more natural setting for our algorithms.

For each $\gamma \in \Gamma \setminus \{1\}$, we define the *isometric circle* of γ to be the circle

$$C(\gamma) = \{z \in \mathbf{C} : |\gamma(z)| = |z|\}.$$

Then in fact P is the closure of the intersection of half-spaces given by

$$\bigcap_{\gamma\in\Gamma\setminus\{1\}} C(\gamma)^o$$

where $C(\gamma)^o$ denotes the exterior of $C(\gamma)$. Our algorithm recursively finds elements in Γ to decrease the hyperbolic volume of this intersection until the volume $\mathrm{vol}(X)$ is reached; it relies upon a "reduction theory" with respect to a set of generators of Γ.

`FundamentalDomain(G,D)`

Computes a fundamental domain in the unit disc D for the action of G.

`FundamentalDomain(G)`

Computes a fundamental domain in the upper half-plane for the action of G.

Example H131E7

We first compute a fundamental domain for a quaternion algebra defined over $\mathbf{Q}(\zeta_7)^+$ which turns out to be the $(2,3,7)$-triangle group.

```
> K<z> := CyclotomicField(7);
> F := sub<K | z+1/z >;
> b := F! (z+1/z);
> A<i,j,k> := QuaternionAlgebra<F | b, b>;
> O := MaximalOrder(A);
> G := FuchsianGroup(O);
> P := FundamentalDomain(G, UnitDisc());
> P;
[
    0.056346691761945477319557812463 9 -
      0.265288162495691167957067899257*$.1,
    0.056346691761945477319557812463 9 +
      0.265288162495691167957067899257*$.1,
    -0.088650485594770026461525429450 0 -
      4.571949565129099928863134193 22E-100*$.1
]
> P := FundamentalDomain(G);
> P;
[
    0.496970425395180896221180392445 +
      (0.867767478235116240951536665696)*root(-1),
    -0.496970425395180896221180392445 +
      (0.867767478235116240951536665696)*root(-1),
    6.943796662033680246332406840 73E-100 +
      (0.753423227948677598725236624130)*root(-1)
]
> ArithmeticVolume(P);
0.023809523809523809523809523809 2
> ArithmeticVolume(G);
1/42
> ($1)*1.0;
0.023809523809523809523809523809 5
```

We can visualize the domain P by using the postscript plotting tools of Chapter 130.

```
>  DisplayPolygons(P, "/tmp/quat237triang.ps" : Show := true);
[ -0.496970425395180896221180392445, 0.496970425395180896221180392445,
```

```
1.0500000000000000000000000000, 302.0000000000000000000000000000 ]
```

We repeat this with the quaternion algebra over $\mathbf{Q}$ of discriminant 10.

```
> G := FuchsianGroup(QuaternionOrder(10));
> P := FundamentalDomain(G);
>  DisplayPolygons(P, "/tmp/quat10.ps" : Show := true);
[ -1.3144841297278098202363616814O, 0.158113883008418966599944677206,
1.279124762273123946981795500S6, 235.0000000000000000000000000000 ]
> U, m := Group(G);
> U;
Finitely presented group U on 3 generators
Relations
    U.1^3 = Id(U)
    U.2^3 = Id(U)
    U.3^3 = Id(U)
    (U.1^-1 * U.2^-1 * U.3^-1)^3 = Id(U)
```

`ShimuraReduceUnit(delta, gammagens, G, D)`

`CreateWord`	BoolElt	*Default :* `false`
`NextSmallest`	BoolElt	*Default :* `false`
`z0`	SpcHydElt	*Default :* 0
`z1`	SpcHydElt	*Default :* 0

Reduce the unit delta moving z_0 to z_1 with respect to the generators in γ_{gens} by multiplying on the left or right by elements of γ_{gens} inside the arithmetic Fuchsian group G to minimize the distance to the origin in the unit disc D. Returns a sequence of triples containing reduced elements and the sequence on the left and right to obtain them. The argument `CreateWord` reduces δ even if δ is in γ_{gens}, disallowing the trivial word.

131.4 Triangle Groups

We now look at a certain class of arithmetic Fuchsian groups of particular interest. We assume that $\Gamma(1)$ is a *triangle group*, i.e. Γ has signature $(0; p, q, r)$ for $p, q, r \in \mathbf{Z}_{\geq} 2$. Hence there exists a presentation

$$\Gamma(1) \cong \langle \gamma_p, \gamma_q, \gamma_r \mid \gamma_p^p = \gamma_q^q = \gamma_r^r = \gamma_p\gamma_q\gamma_r = 1 \rangle.$$

Since $g = 0$, there exists a map j from $X(1)_{\mathbf{C}}$ to the projective line over $\mathbf{C}$ which is ramified only above the three *elliptic points*, the fixed points of $\gamma_p, \gamma_q, \gamma_r$, which we may take to be $0, 1, \infty$.

There are only finitely many such triples (p, q, r), and we have implemented a constructor for these triples which yields the group Γ as well as the arithmetic data including the maximal order $\mathcal{O}$ and the quaternion algebra A. We also compute the analytic map j for each of these groups, which includes methods for evaluating the ${}_2F_1$-hypergeometric series to extremely high precision at complex arguments.

131.4.1 Creation of Triangle Groups

We begin with the basic constructors.

`ArithmeticTriangleGroup(p,q,r)`

Returns the arithmetic triangle group of signature (p, q, r).

`AdmissableTriangleGroups()`

Returns the list of arithmetic triangle groups currently implemented.

`IsTriangleGroup(G)`

Returns `true` if and only if G was created as an arithmetic triangle group.

131.4.2 Fundamental Domain

`ReduceToTriangleVertices(G,z)`

Returns points in the G-orbit of z which are nearest to the vertices of the fundamental domain for G a triangle group.

131.4.3 CM Points

For triangle groups, we can compute CM points analytically. By work of Shimura, the curve $X(1)$ has an interpretation as a moduli space for certain abelian varieties and has a canonical model defined over $\mathbf{Q}$. Let K be a totally imaginary quadratic extension of F, and let $O_D \subset K$ be a quadratic order. Suppose that K splits A, i.e. $A \otimes_F K \cong M_2(K)$. Then there exists an embedding $\iota_K : K \hookrightarrow A$ such that $\iota_K(K) \cap \mathcal{O} = O_D$, given by an element $\mu \in \mathcal{O}$ such that $\mathbf{Z}_F[\mu]$ is isomorphic to O_D. The unique fixed point of $\iota_\infty(\mu)$ in $\mathbf{H}$ yields a *CM point* on $X(1)$ that is defined over an abelian extension H of F, and there is a reciprocity law which describes the action of $\mathrm{Gal}(H/K)$.

We exhibit algorithms for computing with these objects. Explicitly, given the data of a quadratic order O_D contained in a imaginary quadratic extension K of F, we compute the set of $\mathrm{Gal}(H/K)$-conjugates of a CM-point for O_D on $X(1)$. Our algorithm first computes representatives for the ring class group in terms of the Artin map. It then applies the Shimura reciprocity law for each of these representatives, which uses our architecture for principalization of ideals of quaternionic orders, to compute the conjugates. Given these points to high enough precision, we can recognize the CM point as a putative algebraic number.

`HypergeometricSeries2F1(A,B,C,z)`

Returns the value of the 2F1-hypergeometric function with parameters (A, B; C) at the complex number z.

Example H131E8

Although programmed for use by the hypergeometric reversion procedure, involved in the calculation of CM points, the `HypergeometricSeries2F1` function can also be called independently to evaluate the series at any complex number at any precision.

```
> HypergeometricSeries2F1(1/2, 1/2, 1, 1);
0.31830988618379067153776752674
> CC<I> := ComplexField(100);
> HypergeometricSeries2F1(1/2, 1/3, 1/4, 1+I);
0.18914889986718938009890989889270323251621296927695426144398500333164293730405
5659151692186899720481 9 + 1.3516500011026082918038421690042816399520000054582828
7705534621378149655804166701124643888726219267520*I
```

`ShimuraConjugates(mu)`

Returns the set of conjugates of the quaternion order element μ under the Shimura reciprocity law.

`jParameter(G,z)`

`Bound`	RngIntElt	*Default :* 200
`Precision`	RngIntElt	*Default :* 100

Returns the value of the map $j : X(G) \to \mathbf{P}^1$ at z, for G an arithmetic triangle group.

Example H131E9

We begin by constructing the $(2, 3, 9)$-triangle group.

```
> G := ArithmeticTriangleGroup(2,3,9);
> O := BaseRing(G);
> Z_F := BaseRing(O);
```

Next, we compute the CM point corresponding to the imaginary quadratic extension K/F of discriminant -7. First, we define the extension.

```
> Z_K := ext<Z_F | Polynomial([2,-1,1])>;
> Discriminant(Z_K);
Principal Ideal of Z_F
Generator:
    [-7, 0, 0]
> IsMaximal(Z_K);
true
> ClassNumber(AbsoluteOrder(Z_K));
1
```

Since $\mathbf{Z}_K$ has class number 1, the corresponding CM point will be a rational number. We now embed $\mathbf{Z}_K$ into $\mathcal{O}$.

```
> mu := Embed(Z_K, O);
```

```
> MinimalPolynomial(mu);
$.1^2 - $.1 + 2/1*Z_F.1
```

Finally, we compute the fixed point z of μ in the upper half-plane, and compute the value of the uniformizing parameter $j(z)$.

```
> z := FixedPoints(G!mu, UpperHalfPlane())[1];
> j := jParameter(G, z);
> j;
-9594.703124999999999999999999999999999999999999999999999999999999999999999999999
998082712686344925455 41 - 8.607499388211909042811249895070271789213361939086074
091584429508486426059712018492327398804638684615E-77*I
> BestApproximation(Re(j), 10^50);
-614061/64
> Factorization(Numerator($1)), Factorization(Denominator($1));
[ <3, 5>, <7, 1>, <19, 2> ]
[ <2, 6> ]
```

We recognize the value $j(z) = \frac{3^5 7^1 19^2}{2^6}$ as a smooth rational number.

`CMPoints(G,mu)`

Returns the minimal polynomial of the Galois conjugates of the CM points corresponding to μ and their complex values for G a triangle group. The element μ must be Galois-stable, i.e. $F(\mu)$ must be Galois over F, where F is the defining base field of G. The polynomial returned is the "best guess" rational approximation to the polynomial with the CM points as roots.

Example H131E10

We now compute the minimal polynomial for CM points which generate a nontrivial extension: we again take the $(2, 3, 9)$-triangle group and now consider K of discriminant -11.

```
> G := ArithmeticTriangleGroup(2,3,9);
> O := BaseRing(G);
> Z_F := BaseRing(O);
> Z_K := ext<Z_F | Polynomial([3,-1,1])>;
> Discriminant(Z_K);
Principal Ideal of Z_F
Generator:
    [-11, 0, 0]
> IsMaximal(Z_K);
true
> ClassNumber(AbsoluteOrder(Z_K));
3
> mu := Embed(Z_K, O);
> time f, js := CMPoints(G, mu);
Time: 18.120
> f;
```

```
-127332762814175510528*x^3 + 297798194745682427904*x^2 - 545543748833233386651*x
    + 443957770932571793051
```

The number field $H = K(\alpha)$, where α is a root of f, is the Hilbert class field of K, which we have computed analytically!

```
> D := Discriminant(f);
> Denominator(D);
1
> Factorization(Numerator(D));
[ <2, 32>, <3, 18>, <11, 1>, <17, 6>, <19, 12>, <73, 2>, <107, 6>, <109, 2>,
<197, 2>, <271, 2>, <431, 2>, <701, 2> ]
> K := AbsoluteField(NumberField(Z_K));
> H := ext<K | f>;
> Factorization(Discriminant(MaximalOrder(H)));
[]
> IsAbelian(GaloisGroup(H));
true
```

131.5 Bibliography

[**AB04**] Montserrat Alsina and Pilar Bayer. *Quaternion orders, quadratic forms, and Shimura curves*, volume 22 of *CRM Monograph Series*. American Mathematical Society, Providence, RI, 2004.

[**Bea77**] A. F. Beardon. The geometry of discrete groups. In *Discrete groups and automorphic functions (Proc. Conf., Cambridge, 1975)*, pages 47–72. Academic Press, London, 1977.

[**Elk98**] Noam D. Elkies. Shimura curve computations. In *Algorithmic number theory (Portland, OR, 1998)*, volume 1423 of *LNCS*, pages 1–47. Springer, Berlin, 1998.

[**Kat92**] Svetlana Katok. *Fuchsian groups*. Chicago Lectures in Mathematics. University of Chicago Press, Chicago, IL, 1992.

[**Vig80**] M.-F. Vignéras. *Arithmétique des Algèbres de Quaternions*, volume 800 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1980.

[**Voi05**] John Voight. *Quadratic forms and quaternion algebras: Algorithms and arithmetic*. Dissertation, University of California, Berkeley, 2005.

[**Voi06**] John Voight. Computing CM points on Shimura curves arising from cocompact arithmetic triangle groups. In *Algorithmic number theory*, volume 4076 of *LNCS*, pages 406–420. Springer, Berlin, 2006.

[**Voi09**] J. Voight. Computing fundamental domains for cofinite Fuchsian groups. *J. Théor. Nombres Bordeaux*, 21(2):469–491, 2009.

132 MODULAR FORMS

Chapter 132
MODULAR FORMS

132.1 Introduction

132.1.1 Modular Forms

This theoretically-oriented section serves as a guide to the rest of the chapter. We recall the definition of modular forms, then briefly discuss q-expansions, Hecke operators, eigenforms, congruences, and modular symbols.

Fix positive integers N and k, let $\mathbf{H}$ denote the complex upper half plane. Denote by $M_k(\Gamma_1(N))$ the space of modular forms on $\Gamma_1(N)$ of weight k. This is the complex vector space of holomorphic functions $f : \mathbf{H} \to \mathbf{C}$ such that

$$f\left(\frac{az+b}{cz+d}\right) = (cz+d)^k f(z) \qquad \text{for all} \qquad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \Gamma_1(N),$$

and $f(z)$ is holomorphic at each "cusp" in $\mathbf{P}^1(\mathbf{Q}) = \mathbf{Q} \cup \{\infty\}$ (see, e.g., [DI95] for a more precise definition.)

A *Dirichlet character* is a homomorphism $\varepsilon : (\mathbf{Z}/N\mathbf{Z})^* \to \mathbf{C}^*$ of abelian groups. Dirichlet characters are of interest because they decompose $M_k(\Gamma_1(N))$ into more manageable chunks. If V is any complex vector space equipped with an action $\rho : (\mathbf{Z}/N\mathbf{Z})^* \to \mathrm{Aut}(V)$ and ε is a Dirichlet character, we set

$$V(\varepsilon) = \{x \in V : \rho(a)x = \varepsilon(a)x \quad \text{all} \quad a \in (\mathbf{Z}/N\mathbf{Z})^*\}.$$

The space $M_k(\Gamma_1(N))$ is equipped with an action of $(\mathbf{Z}/N\mathbf{Z})^*$ by the diamond-bracket operators $\langle d \rangle$, which are defined as follows. Given $\overline{d} \in (\mathbf{Z}/N\mathbf{Z})^*$, choose a matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \Gamma_0(N)$ such that $d \bmod N = \overline{d}$. Then

$$\langle d \rangle f(z) = (cz+d)^{-k} f\left(\frac{az+b}{cz+d}\right).$$

We call $M_k(\Gamma_1(N))(\varepsilon)$ the space of modular forms of weight k, level N, and character ε. This is the complex vector space of holomorphic functions $f : \mathbf{H} \to \mathbf{C}$ such that

$$f\left(\frac{az+b}{cz+d}\right) = \varepsilon(a)(cz+d)^k f(z) \qquad \text{for all} \qquad \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \Gamma_0(N),$$

and which is holomorphic at the cusps. We let $M_k([\varepsilon])$ denote the direct sum of the spaces $M_k(\Gamma_1(N))(\varepsilon)$ *as* ε *varies over the* $\mathrm{Gal}(\overline{\mathbf{Q}}/\mathbf{Q})$*-conjugates* of ε. It is unnecessary to specify the level because it is built into ε.

To summarize, for any integer k and positive integer N, there is a finite-dimensional $\mathbf{C}$-vector space $M_k(\Gamma_1(N))$. Moreover,

$$M_k(\Gamma_1(N)) = \bigoplus_{\text{all } \varepsilon} M_k(\Gamma_1(N))(\varepsilon) = \bigoplus_{\text{Gal}(\overline{\mathbf{Q}}/\mathbf{Q})\text{-class reps. } \varepsilon} M_k([\varepsilon]).$$

In Section 139.2, we describe how to create the spaces $M_k(\Gamma_1(N))$ and $M_k([\varepsilon])$ in MAGMA, for any $k \geq 1$, $N \geq 1$, and character ε.

Let f be a modular form, and observe that since $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \in \Gamma_1(N)$, we have $f(z) = f(z+1)$. If we set $q = \exp(2\pi i z)$, there is a *q-expansion* representation for f:

$$f = a_0 + a_1 q + a_2 q^2 + a_3 q^3 + a_4 q^4 + \cdots.$$

The a_n are called the *Fourier coefficients* of f. MAGMA contains an algorithm for computing a basis of q-expansions for any space of modular forms of weight $k \geq 2$ (see Section 135.3).

Fix a positive integer N and let M be a sum of spaces $M_k(N, \varepsilon)$. Let $q{-}\exp : M \to \mathbf{C}[[q]]$ denote the map that associates to a modular form f its q-expansion. One can prove that there is a basis $f_1, \ldots, f_d$ of M that maps to a basis for the free $\mathbf{Z}$-module $q{-}\exp(M) \cap \mathbf{Z}[[q]]$. See Section 132.4 for how to compute such a basis in MAGMA. Let $M_{\mathbf{Z}}$ be the $\mathbf{Z}$-module spanned by $f_1, \ldots, f_d$. For any ring R, we define the *space of modular forms over* R to be $M_R = M_{\mathbf{Z}} \otimes_{\mathbf{Z}} R$. Thus M_R is a free R-module of rank d with basis the images of $f_1, \ldots, f_d$ in $M_{\mathbf{Z}} \otimes_{\mathbf{Z}} R$. The computation of M_R is discussed in Section 132.2.2.

Any space M of modular forms is equipped with an action of a commutative ring $\mathbf{T} = \mathbf{Z}[\ldots T_n \ldots]$ of Hecke operators. The computation of Hecke operators T_n and their characteristic polynomials is described in Section 136.14.

An *eigenform* is a simultaneous eigenvector for every element of the Hecke algebra $\mathbf{T}$. A *newform* is an eigenform that doesn't come from a space of lower level and is normalized so that the coefficient of q is 1. Section 132.13 describes how to find newforms. Computation of the mod p reductions and p-adic and complex embeddings of a newform is described in Section 132.14.

Computation of congruences is discussed in Section 132.15.

Modular symbols are closely related to modular forms. See Section 132.19 for the connection between the two.

132.1.2 About the Package

The modular forms package is in many ways an interface to the modular symbols machinery (Section 132.19). It also contains additional functionality (such as Eisenstein series), and features that are implemented independently (such as Hecke operators). In some situations however, it is better to work with modular symbols directly. In particular, spaces of modular forms are required to be Galois-stable over the rationals, while spaces of modular symbols are not. Moreover some features, such as Hecke operators of certain spaces, are

unavailable for a given space of modular forms but can be obtained using the corresponding modular symbols.

In MAGMA version 2.14 modular forms of weight one, and of half-integral weight, were added. Most of the existing functions now also work for these weights, however some are not implemented (for instance `Newforms`). Further functionality for half-integral weight will be added in future releases (including Hecke operators).

132.1.3 Categories

In MAGMA, spaces of modular forms belong to the category `ModFrm`, and the elements of spaces of modular forms belong to `ModFrmElt`.

132.1.4 Verbose Output

To set the verbosity level use the command `SetVerbose("ModularForms",n)`, where `n` is 0 (silent), 1 (verbose), or 2 (very verbose). The default verbose level is 0.

Example H132E1

In this example, we illustrate categories and verbosity for modular forms.

```
> M := ModularForms(11,2); M;
Space of modular forms on Gamma_0(11) of weight 2 and dimension 2 over
Integer Ring.
> Type(M);
ModFrm
> B := Basis(M); B;
[
    1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + O(q^8),
    q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
]
> Type(B[1]);
ModFrmElt
```

Using `SetVerbose`, we get some information about what is happening during computations.

```
> SetVerbose("ModularForms",2);
> M := ModularForms(30,4);
> EisensteinSubspace(M);
ModularForms: Computing eisenstein subspace.
ModularForms: Computing dimension.
Space of modular forms on Gamma_0(30) of weight 4 and dimension 8 over
Integer Ring.
> SetVerbose("ModularForms",0);   // turn off verbose mode
```

132.1.5 An Illustrative Overview

In this section, we give a longer example that serves as an overview of the modular forms package. It illustrates computations of modular forms of level 1, and illustrates the exceptional case of Serre's conjecture with a level 13 example.

Example H132E2

First, we compute the two-dimensional space of modular forms of weight 12 and level 1 over **Z**.

```
> M := ModularForms(Gamma0(1),12); M;
Space of modular forms on Gamma_0(1) of weight 12 and dimension 2 over
Integer Ring.
```

The default output precision is 8:

```
> Basis(M);
[
    1 + 196560*q^2 + 16773120*q^3 + 398034000*q^4 + 4629381120*q^5 +
    34417656000*q^6 + 187489935360*q^7 + O(q^8),
    q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7
    + O(q^8)
]
> [PowerSeries(f,10) : f in Basis(M)];
[
    1 + 196560*q^2 + 16773120*q^3 + 398034000*q^4 + 4629381120*q^5 +
        34417656000*q^6 + 187489935360*q^7 + 814879774800*q^8 +
        2975551488000*q^9 + O(q^10),
    q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7
        + 84480*q^8 - 113643*q^9 + O(q^10)
]
> f := Basis(M)[1];
> Coefficient(f,2);
196560
```

The `Newforms` command returns a list of the Galois-orbits of newforms.

```
> NumberOfNewformClasses(M);
2
> f := Newform(M,1); f;
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 +
O(q^8)
```

We can call the "q" in the q-expansion anything we want and also compute forms to higher precision.

```
> Mf<w> := Parent(f); Mf;
Space of modular forms on Gamma_0(1) of weight 12 and dimension 1 over
Rational Field.
> f + O(w^12);
w - 24*w^2 + 252*w^3 - 1472*w^4 + 4830*w^5 - 6048*w^6 - 16744*w^7 +
```

```
84480*w^8 - 113643*w^9 - 115920*w^10 + 534612*w^11 + O(w^12)
```

Typing `f + O(w^12)` is equivalent to typing `PowerSeries(f,12)`. The second newform orbit contains an Eisenstein series.

```
> E := Newform(M,2); PowerSeries(E,2);
691/65520 + q + O(q^2)
```

Next we compute the two conjugate newforms in $S_2(\Gamma_1(13))$:

```
> M := ModularForms(Gamma1(13),2);
> S := CuspidalSubspace(M); S;
Space of modular forms on Gamma_1(13) of weight 2 and dimension 2 over
Integer Ring.
> NumberOfNewformClasses(S);
1
> f := Newform(S, 1); f;
q + (-a - 1)*q^2 + (2*a - 2)*q^3 + a*q^4 + (-2*a + 1)*q^5 + (-2*a +
4)*q^6 + O(q^8)
```

Here a is a root of the polynomial $x^2 - x + 1$.

```
> Parent(f);
Space of modular forms on Gamma_1(13) of weight 2 and dimension 2 over
Number Field with defining polynomial x^2 - x + 1 over the Rational
Field.
```

The Galois-conjugacy class of f has two newforms in it.

```
> Degree(f);
2
> g := Newform(S, 1, 2); g;
q + (-b - 1)*q^2 + (2*b - 2)*q^3 + b*q^4 + (-2*b + 1)*q^5 + (-2*b +
4)*q^6 + O(q^8)
> BaseRing(Parent(g));
Number Field with defining polynomial x^2 - x + 1 over the Rational
Field
```

The parents of f and g are isomorphic (but distinct) abstract extensions of $\mathbf{Q}$ both isomorphic to $\mathbf{Q}(\sqrt{-3})$.

```
> Parent(f) eq Parent(g);
false
```

We can also list all of the newforms at once, gathered into Galois orbits, using the `Newforms` command:

```
> N := Newforms(M);
> #N;
8
> N[3];
[*
1/13*(-7*zeta_6 - 11) + q + (2*zeta_6 + 1)*q^2 + (-3*zeta_6 + 1)*q^3 +
```

```
(6*zeta_6 - 3)*q^4 - 4*q^5 + (-7*zeta_6 + 7)*q^6 + (-7*zeta_6 + 8)*q^7
+ O(q^8),
1/13*(7*zeta_6 - 18) + q + (-2*zeta_6 + 3)*q^2 + (3*zeta_6 - 2)*q^3 +
(-6*zeta_6 + 3)*q^4 - 4*q^5 + 7*zeta_6*q^6 + (7*zeta_6 + 1)*q^7 +
O(q^8)
*]
```

The "Nebentypus" or character of f has order 6.

```
> e := DirichletCharacter(f); Parent(e);
Group of Dirichlet characters of modulus 13 over Cyclotomic Field of
order 6 and degree 2
> Order(e);
6
```

This shows that there are no cuspidal newforms with Dirichlet character of order 2. As we see below, the mod-3 reduction of f has character $\overline{\varepsilon}$ that takes values in $\mathbf{F}_3$ and has order 2. The minimal lift of $\overline{\varepsilon}$ to characteristic 0 has order 2, while there are no newforms of level 13 with character of order 2. (This example illustrates the "exceptional case of Serre's conjecture".)

```
> f3 := Reductions(f,3); f3;
[* [*
q + 2*q^3 + 2*q^4 + O(q^8)
*] *]
> M3<q> := Parent(f3[1][1]);
> f3[1][1]+O(q^15);
q + 2*q^3 + 2*q^4 + q^9 + q^12 + 2*q^13 + O(q^15)
```

The modular forms package can also be used to quickly compute dimensions of spaces of modular forms.

```
> M := ModularForms(Gamma0(1),2048);
> Dimension(M);
171
> M := ModularForms(Gamma1(389),2);
> Dimension(M);
6499
> Dimension(CuspidalSubspace(M));
6112
```

Don't try to compute a basis of q-expansions for M!

```
> M := ModularForms(Gamma0(123456789),6);
> Dimension(CuspidalSubspace(M));
68624152
> Dimension(EisensteinSubspace(M));
16
```

132.2 Creation Functions

132.2.1 Ambient Spaces

The following are used for creating spaces of modular forms. For information on Dirichlet characters, see Section 19.8.

`ModularForms(N)`

The space $M_2(\Gamma_0(N), \mathbf{Z})$ of modular forms on $\Gamma_0(N)$ of weight 2. See the documentation for `ModularForms(N,k)` below, with $k = 2$.

`ModularForms(N, k)`

The space $M_k(\Gamma_0(N), \mathbf{Z})$ of weight k modular forms on $\Gamma_0(N)$ over $\mathbf{Z}$.

`ModularForms(eps, k)`

Given a Dirichlet character `eps` and an integer k, this returns a space of modular forms over the integers, of weight k, which under base extension becomes equal to the direct sum of the spaces $M_k(\Gamma_1(N), eps1)$ of weight k and nebentypus character `eps1`, where `eps1` runs over all Galois conjugates `eps`.

`ModularForms(chars, k)`

The space of modular forms of weight k over the integers, formed as the direct sum of spaces `ModularForms(eps,k)`, summing over all `eps` in the given sequence `chars` of Dirichlet characters.

`ModularForms(G)`

This is the same as `ModularForms(G,2)` (see below).

`ModularForms(G, k)`

The space $M_k(G, \mathbf{Z})$, where G is a congruence subgroup. The groups $\Gamma_0(N)$ and $\Gamma_1(N)$ are currently supported, and can be created using the commands `Gamma0(N)` and `Gamma1(N)`, respectively.

`CuspForms(x)`

`CuspForms(x,y)`

These commands are a shortcut, and return the `CuspidalSubspace` of the corresponding full space of modular forms.

Example H132E3

In this example, we illustrate each of the above constructors in turn. First we create $M_2(\Gamma_0(65))$.

```
> M := ModularForms(65); M;
Space of modular forms on Gamma_0(65) of weight 2 and dimension 8 over
Integer Ring.
> Dimension(M);
8
> Basis(CuspidalSubspace(M));
[
    q + q^5 + 2*q^6 + q^7 + O(q^8),
    q^2 + 2*q^5 + 3*q^6 + 2*q^7 + O(q^8),
    q^3 + 2*q^5 + 2*q^6 + 2*q^7 + O(q^8),
    q^4 + 2*q^5 + 3*q^6 + 3*q^7 + O(q^8),
    3*q^5 + 5*q^6 + 2*q^7 + O(q^8)
]
```

Next we create $M_4(\Gamma_0(8))$.

```
> M := ModularForms(8,4); M;
Space of modular forms on Gamma_0(8) of weight 4 and dimension 5 over
Integer Ring.
> Dimension(M);
5
> Basis(CuspidalSubspace(M));
[
    q - 4*q^3 - 2*q^5 + 24*q^7 + O(q^8)
]
```

Now we create the space $M_3(N, \varepsilon)$, where ε is a character of level 20, conductor 5 and order 4.

```
> G := DirichletGroup(20,CyclotomicField(EulerPhi(20)));
> chars := Elements(G); #chars;
8
> [Conductor(eps) : eps in chars];
[ 1, 4, 5, 20, 5, 20, 5, 20 ]
> eps := chars[3];
> IsEven(eps);
false
> M := ModularForms([eps],3); M;
Space of modular forms on Gamma_1(20) with character all conjugates of
[$.2], weight 3, and dimension 12 over Integer Ring.
> Dimension(EisensteinSubspace(M));
6
> Dimension(CuspidalSubspace(M));
6
```

Next we create the direct sum of the spaces $M_k(20, \varepsilon)$ as ε varies over the four mod 20 characters of order at most 2, for $k = 2$ and 3.

```
> G := DirichletGroup(20, RationalField());   // (Z/20Z)^* --> Q^*
```

```
> chars := Elements(G); #chars;
4
> M := ModularForms(chars,2); M;
Space of modular forms on Gamma_1(20) with characters all
conjugates of [1, $.1, $.2, $.1*$.2], weight 2, and dimension 12
over Integer Ring.
> M := ModularForms(chars,3); M;
Space of modular forms on Gamma_1(20) with characters all
conjugates of [1, $.1, $.2, $.1*$.2], weight 3, and dimension 16
over Integer Ring.
```

Now we create the spaces $M_k(\Gamma_1(20))$ for $k = 2, 3$.

```
> ModularForms(Gamma1(20));
Space of modular forms on Gamma_1(20) of weight 2 and dimension 22
over Integer Ring.
> ModularForms(Gamma1(20),3);
Space of modular forms on Gamma_1(20) of weight 3 and dimension 34
over Integer Ring.
```

We can also create the subspace of cuspforms directly:

```
> CuspForms(Gamma1(20));
Space of modular forms on Gamma_1(20) of weight 2 and dimension 3
over Integer Ring.
> CuspForms(Gamma1(20),3);
Space of modular forms on Gamma_1(20) of weight 3 and dimension 14
over Integer Ring.
```

132.2.1.1 Half-integral Weight Forms

Spaces of modular forms of half-integral weight can now be constructed. For these spaces, `CuspidalSubspace` and `qExpansionBasis` are available, as well as basic functionality such as element arithmetic. More functionality will be added in future releases.

The algorithm for determining the q-expansion basis involves computing those of related integral-weight spaces (of weight either one half smaller or one half larger, and appropriate level and character).

`HalfIntegralWeightForms(N, w)`

The space of half-integral weight forms on `Gamma0(N)` and weight w. Here N should be a multiple of 4 and w a positive element of $\mathbf{Z} + 1/2$.

`HalfIntegralWeightForms(chi, w)`

The space of half-integral weight forms on `Gamma1(N)` with character `chi` and weight w. The modulus of `chi` should be a multiple of 4, and w a positive element of $\mathbf{Z}+1/2$.

`HalfIntegralWeightForms(G, w)`

The space of half-integral weight forms on the congruence subgroup G and weight w. Here G must be contained in `Gamma0(4)`, and w is a positive element of $\mathbf{Z}+1/2$.

132.2.2 Base Extension

If M is a space of modular forms created using one of the constructors in Section 132.2.1, then the base ring of M is $\mathbf{Z}$. Thus we can base extend M to any ring R. The examples below illustrate some simple applications of `BaseExtend`.

`BaseExtend(M, R)`

The base extension of the space M of modular forms to the ring R and the induced map from M to `BaseExtend(M,R)`. The only requirement on R is that there is a natural coercion map from the base ring of M to R. For example, when `BaseRing(M)` is the integers, any ring R is allowed.

`BaseExtend(M, phi)`

The base extension of the space M of modular forms to the ring R using the map ϕ : `BaseRing(M)` $\to R$, and the induced map from M to `BaseExtend(M,R)`

Example H132E4

We first illustrate an Eisenstein series in $M_{12}(1)$ that is congruent to 1 modulo 3.

```
> M<q> := EisensteinSubspace(ModularForms(1,12));
> E12 := M.1; E12 + O(q^4);
691 + 65520*q + 134250480*q^2 + 11606736960*q^3 + O(q^4)
> M3<q3> := BaseExtend(M,GF(3));
> Dimension(M3);
1
> M3.1+O(q3^20);
1 + O(q3^20)
```

This congruence can be proved by noting that the coefficient of q^n in the q-expansion of $E_{12}/65520$, for any $n \geq 1$, is an eigenvalue of a Hecke operator, hence an integer, and that 65520 is divisible by 3. Because E_{12} is defined over $\mathbf{Z}$ the command `"E12Q/65520"` would result in an error, so we first base extend to $\mathbf{Q}$.

```
> MQ, phi := BaseExtend(M,RationalField());
> E12Q := phi(E12);
> E12Q/65520;
691/65520 + q + 2049*q^2 + 177148*q^3 + 4196353*q^4 + 48828126*q^5 +
362976252*q^6 + 1977326744*q^7 + O(q^8)
```

It is possible to base extend to almost any silly commutative ring.

```
> M := ModularForms(11,2);
> R := PolynomialRing(GF(17),3);
> MR<q> := BaseExtend(M,R); MR;
Space of modular forms on Gamma_0(11) of weight 2 and dimension 2 over
Polynomial ring of rank 3 over GF(17)
Lexicographical Order
Variables: $.1, $.2, $.3.
> f := MR.1;  f + O(q^5);
```

```
1 + 12*q^2 + 12*q^3 + 12*q^4 + O(q^5)
> f*(R.1+3*R.2) + O(q^4);
$.1 + 3*$.2 + (12*$.1 + 2*$.2)*q^2 + (12*$.1 + 2*$.2)*q^3 + O(q^4)
```

132.2.3 Elements

`M . i`

The ith basis vector of the space of modular forms M.

`M ! f`

The coercion of f into the space of modular forms M. Here f can be a modular form, a power series with absolute precision, or something that can be coerced into `RSpace(M)`.

`ModularForm(E)`

The modular form associated to the elliptic curve E over $\mathbf{Q}$. (See Section 132.18.)

Example H132E5

```
> M := ModularForms(Gamma0(11),2);
> M.1;
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + O(q^8)
> M.2;
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
> R<q> := PowerSeriesRing(Integers());
> f := M!(1 + q + 10*q^2 + O(q^3));
> f;
1 + q + 10*q^2 + 11*q^3 + 14*q^4 + 13*q^5 + 26*q^6 + 22*q^7 + O(q^8)
> Eltseq(f);
[ 1, 1 ]
```

`Eltseq` gives f as a linear combination of `M.1` and `M.2`. Next we coerce f into $M_2(\Gamma_0(22))$.

```
> M22 := ModularForms(Gamma0(22),2);
> g := M22!f; g;
1 + q + 10*q^2 + 11*q^3 + 14*q^4 + 13*q^5 + 26*q^6 + 22*q^7 + O(q^8)
> Eltseq(g);
[ 1, 1, 10, 11, 14 ]
```

The elliptic curve E below defines an element of $M_2(\Gamma_0(11))$.

```
> E := EllipticCurve([ 0, -1, 1, -10, -20 ]);
> Conductor(E);
11
> f := ModularForm(E);
> f;
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
```

```
> f + M.1;
1 + q + 10*q^2 + 11*q^3 + 14*q^4 + 13*q^5 + 26*q^6 + 22*q^7 + O(q^8)
```

Note, however, that the ambient space of the parent of f is not equal to M, despite the fact that both are isomorphic to $M_2(\Gamma_0(11))$. This is because they were created independently. The above operations are defined because there is a canonical way to coerce f into M using its q-expansion.

```
> f in M;
false
> AmbientSpace(Parent(f)) eq M;
false
```

132.3 Bases

Any space of modular forms that can be created in MAGMA is of the form $M_{\mathbf{Z}} \otimes_{\mathbf{Z}} R$ for some ring R and some space $M_{\mathbf{Z}}$ of modular forms defined over $\mathbf{Z}$. The basis of M is the image in M of `Basis(M_Z)`, and `Basis(M_Z)` is in Hermite normal form.

Note that in order to determine this basis over $\mathbf{Z}$, it is necessary to compute q-expansions up to a "Sturm bound" for M (see `PrecisionBound`). The precision used internally will therefore be at least as large as this bound. If desired, however, one can compute q-expansions to lower precision by working directly with spaces of modular symbols.

`Basis(M)`

The canonical basis of the space of modular forms or half-integral weight forms M.

`Basis(M, prec)`

`qExpansionBasis(M, prec)`

A sequence containing q-expansions (to the specified precision) of the elements of `Basis(M)`.

`PrecisionBound(M : parameters)`

`Exact` BOOLELT *Default :* `false`

An integer b such that $f+O(q^b)$ determines any modular form f in the given space M of modular forms or half-integral weight forms. If the optional parameter `Exact` is set to `true`, or if a q-expansion basis has already been computed for M, then the result is best-possible, ie the *smallest* integer b such that $f + O(q^b)$ determines any modular form f in M. Otherwise it is a "Sturm bound" similar to (although in some cases sharper than) the bounds given in section 9.4 of [Ste07].

Note: In some much older versions of MAGMA the default was `Exact := true`.

`RModule(M)`
`RSpace(M)`
`VectorSpace(M)`

`Ring` RNG *Default :*

An abstract free module isomorphic to the given space of modular forms M, over the same base ring (unless `Ring` is specified). The second returned object is a map from the abstract module to/from M.

This function is needed when one wants to use linear algebra functions on M (since in Magma, a space of modular forms is not a subtype of vector space).

Example H132E6

```
> M := ModularForms(Gamma1(16),3); M;
Space of modular forms on Gamma_1(16) of weight 3 and dimension 23
over Integer Ring.
> Dimension(CuspidalSubspace(M));
9
> SetPrecision(M,19);
> Basis(NewSubspace(CuspidalSubspace(M)))[1];
q - 76*q^8 + 39*q^9 + 132*q^10 - 44*q^11 + 84*q^12 - 144*q^13 -
232*q^14 + 120*q^15 + 160*q^16 + 158*q^17 - 76*q^18 + O(q^19)
```

We can print the whole basis to less precision:

```
> SetPrecision(M,10);
> Basis(NewSubspace(CuspidalSubspace(M)));
[
    q - 76*q^8 + 39*q^9 + O(q^10),
    q^2 + q^7 - 58*q^8 + 30*q^9 + O(q^10),
    q^3 + 2*q^7 - 42*q^8 + 18*q^9 + O(q^10),
    q^4 + q^7 - 26*q^8 + 13*q^9 + O(q^10),
    q^5 + 2*q^7 - 18*q^8 + 5*q^9 + O(q^10),
    q^6 + 2*q^7 - 12*q^8 + 3*q^9 + O(q^10),
    3*q^7 - 8*q^8 + O(q^10)
]
```

Note the coefficient 3 of q^7, which emphasizes that this is not the reduced echelon form over a field, but the image of a reduced form over the integers:

```
> MQ := BaseExtend(M,RationalField());
> Basis(NewSubspace(CuspidalSubspace(MQ)))[7];
3*q^7 - 8*q^8 + O(q^10)
```

132.4 q-Expansions

The following intrinsics give the q-expansion of a modular form (about the cusp ∞).

Note that q-expansions are printed by default only to precision $O(q^12)$. This may be adjusted using `SetPrecision` (see below), which should be used to control *printing only*; to control the amount of precision computed *internally*, instead use `qExpansion` or `qExpansionBasis` and specify the desired precision.

`qExpansion(f)`

`qExpansion(f, prec)`

`PowerSeries(f)`

`PowerSeries(f, prec)`

The q-expansion (at the cusp ∞) of the modular form (or half-integral weight form) f to absolute precision `prec`. This is an element of the power series ring over the base ring of the parent of f.

`Coefficient(f, n)`

The nth coefficient of the q-expansion of the modular form f.

`Precision(M)`

The default printing precision for elements of the space M of modular forms. This is the precision that is used when printing elements of M if the user does not specifically make a choice. The hard-coded default value is 12.

`SetPrecision(M, prec)`

Set the default printing precision for elements of the space M of modular forms (the hard-coded default value is 12).

Example H132E7

In this example, we compute the q-expansion of a modular form $f \in M_3(\Gamma_1(11))$ in several ways.

```
> M := ModularForms(Gamma1(11),3); M;
Space of modular forms on Gamma_1(11) of weight 3 and dimension 15
over Integer Ring.
> f := M.1;
> f;
1 + O(q^8)
> qExpansion(f);
1 + O(q^8)
> Coefficient(f,16);  // f is a modular form, so has infinite precision
-5457936
> qExpansion(f,17);
1 + 763774*q^15 - 5457936*q^16 + O(q^17)
> PowerSeries(f,20);   // same as qExpansion(f,20)
1 + 763774*q^15 - 5457936*q^16 + 14709156*q^17 - 12391258*q^18 -
```

```
    21614340*q^19 + O(q^20)
```

The "big-oh" notation is supported via addition of a modular form and a power series.

```
> M<q> := Parent(f);
> Parent(q);
Power series ring in q over Integer Ring
> f + O(q^17);
1 + 763774*q^15 - 5457936*q^16 + O(q^17)
> 5*q - O(q^17) + f;
1 + 5*q + 763774*q^15 - 5457936*q^16 + O(q^17)
> 5*q + f;
1 + 5*q + O(q^8)
```

Default printing precision can be set using the command `SetPrecision`.

```
> SetPrecision(M,16);
> f;
1 + 763774*q^15 + O(q^16)
```

Example H132E8

The `PrecisionBound` intrinsic is related to Weierstrass points on modular curves. Let N be a positive integer such that $S = S_2(\Gamma_0(N))$ has dimension at least 2. Then the point ∞ is a Weierstrass point on $X_0(N)$ if and only if `PrecisionBound(S : Exact := true)-1 ne Dimension(S)`.

```
> function InftyIsWP(N)
>    S := CuspidalSubspace(ModularForms(Gamma0(N),2));
>    assert Dimension(S) ge 2;
>    return (PrecisionBound(S : Exact := true)-1) ne Dimension(S);
> end function;
> [<N,InftyIsWP(N)> : N in [97..100]];
[ <97, false>, <98, true>, <99, false>, <100, true> ]
```

It is an open problem to give a simple characterization of the integers N such that ∞ is a Weierstrass point on $X_0(N)$, though Atkin and others have made significant progress on this problem (see, e.g., 1967 Annals paper [Atk67]). I verified that if $N < 3223$ is square free, then ∞ is not a Weierstrass point on $X_0(N)$, which suggests a nice conjecture.

132.5 Arithmetic

`f + g`

The sum of the modular forms f and g.

`f + g`

The sum of the modular form f and the power series g. The q-expansion of f must be coercible into the parent of g. The sum $g+f$ is also defined, as are the differences $f-g$ and $g-f$.

`f - g`

The difference of the modular forms f and g.

`a * f`

The product of the scalar a and the modular form f.

`f / a`

The product of the scalar $1/a$ and the modular form f.

`f ^ n`

The power f^n of the modular form f, where $n \geq 1$ is an integer.

`f * g`

The product of the modular forms f and g. The only condition is that the base fields of f and g be the same. The weight of `f*g` is the sum of the weights of f and g.

Example H132E9

```
> M2 := ModularForms(Gamma0(11), 2);
> f := M2.1;
> g := M2.2;
> f;
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + O(q^8)
> g;
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
> f+g;
1 + q + 10*q^2 + 11*q^3 + 14*q^4 + 13*q^5 + 26*q^6 + 22*q^7 + O(q^8)
> 2*f;
2 + 24*q^2 + 24*q^3 + 24*q^4 + 24*q^5 + 48*q^6 + 48*q^7 + O(q^8)
> MQ,phi := BaseExtend(M2, RationalField());
> phi(2*f)/2;
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + O(q^8)
> f^2;
1 + 24*q^2 + 24*q^3 + 168*q^4 + 312*q^5 + 480*q^6 + 624*q^7 + O(q^8)
> Parent($1);
Space of modular forms on Gamma_0(11) of weight 4 and dimension 4 over
```

```
Integer Ring.
> M3 := ModularForms([DirichletGroup(11).1], 3); M3;
Space of modular forms on Gamma_1(11) with character all conjugates of
[$.1], weight 3, and dimension 3 over Integer Ring.
> M3.1*f;
1 + 12*q^2 + 2*q^3 + 6*q^4 - 126*q^5 - 168*q^6 - 384*q^7 + O(q^8)
> Parent($1);
Space of modular forms on Gamma_1(11) of weight 5 and dimension 25
over Integer Ring.
```

132.6 Predicates

`IsAmbientSpace(M)`

Returns `true` if and only if M is an ambient space. Ambient spaces are those space constructed in Section 132.2.1.

`IsCuspidal(M)`

Returns `true` if M is contained in the cuspidal subspace of the ambient space.

`IsEisenstein(M)`

Returns `true` if M is contained in the Eisenstein subspace of the ambient space.

`IsEisensteinSeries(f)`

Returns `true` if f is an Eisenstein newform or was computed using the intrinsic `EisensteinSeries`. (See Section 132.10.)

`IsGamma0(M)`

Returns `true` if M is a space of modular forms for $\Gamma_0(N)$.

`IsGamma1(M)`

Returns `true` if M was created explicitly as a space of modular forms for $\Gamma_1(N)$, or if the `AmbientSpace` of M is such a space. (Note that `IsGamma1` will return `false` for any space `ModularForms(chars,k)`, even if `chars` consists of all mod N Dirichlet characters.)

`IsNew(M)`

Returns `true` if M is contained in the new subspace of its `AmbientSpace`.

`IsNewform(f)`

Returns `true` if f was created using `Newforms`. (Sometimes `true` in other cases in which f is obviously a newform. In number theory, "newform" means "normalized eigenform that lies in the new subspace".)

`IsRingOfAllModularForms(M)`

Returns `true` if and only if M is the ring of all modular forms over a given ring.

Example H132E10

We illustrate each of the above predicates with some simple computations in $M_3(\Gamma_1(11))$.

```
> M := ModularForms(Gamma1(11),3);
> f := Newform(M,1);
> IsAmbientSpace(M);
true
> IsAmbientSpace(CuspidalSubspace(M));
false
> IsCuspidal(M);
false
> IsCuspidal(CuspidalSubspace(M));
true
> IsEisenstein(CuspidalSubspace(M));
false
> IsEisenstein(EisensteinSubspace(M));
true
> IsGamma1(M);
true
> IsNew(M);
true
> IsNewform(M.1);
false
> IsNewform(f);
true
> IsRingOfAllModularForms(M);
false
> Level(f);
11
> Level(M);
11
> Weight(f);
3
> Weight(M);
3
> Weight(M.1);
3
```

132.7 Properties

`AmbientSpace(M)`

The full space of modular forms, in which the given space had been created as a subspace.

`BaseRing(M)`

`CoefficientRing(M)`

The ring over which the given space of modular forms was defined.

`Degree(f)`

The number of Galois-conjugates of the modular form f over the prime subfield of (the fraction field of) the base ring of f.

`Dimension(M)`

The dimension of the space M of modular forms or half-integral weight forms.

For spaces defined using the `ModularForms` constructors, the procedure used to obtain the dimension is to count the relevant Eisenstein series, and apply a formula giving the dimension of the relevant space of cusp forms.

`DimensionByFormula(M)`

The dimension of the given space of modular forms or half-integral weight forms (which must be either a full space or the cuspidal subspace of a full space), as given by the formulas in the paper by Cohen and Oesterle (in 'Modular Forms in One Variable, VI', Lecture Notes in Math. 627).

`DimensionByFormula(N, k)`

`DimensionByFormula(chi, k)`

`DimensionByFormula(N, chi, k)`

`Cuspidal` BoolElt *Default :* `false`

The dimension of the full space of the modular forms or half-integral weight forms with level N, character chi (taken to be trivial if not specified) and weight k, as given by the formulas in the paper by Cohen and Oesterle (in 'Modular Forms in One Variable, VI', Lecture Notes in Math. 627).

If `Cuspidal` is set to `true`, then the dimension of the space of cusp forms is returned.

`DirichletCharacters(M)`

A sequence containing exactly one representative from each Galois-conjugacy class of Dirichlet characters associated to the space of modular forms M.

`DirichletCharacter(f)`

Suppose f is a newform, created using the `Newform` command. This returns a Dirichlet character that is, up to Galois conjugacy, the Nebentypus character of f.

`Eltseq(f)`

The sequence $[a_1, \ldots, a_n]$ such that $f = a_1 g_1 + \cdots + a_n g_n$, where $g_1, \ldots, g_n$ is the basis of the parent of the modular form f.

`Level(f)`

The level of the modular form f.

`Level(M)`

The level of the space of modular forms M.

`Weight(f)`

The weight of the modular form f, if it is defined.

`Weight(M)`

The weight of the space M of modular forms.

`WeightOneHalfData(H)`

A list of tuples describing a basis of the given space of forms of weight 1/2. Each tuple is a pair $< f, t >$, where t is an integer and f is a Dirichlet character. The tuple $< f, t >$ designates the sum over all integers n of $f(n)q^{(tn^2)}$.

Example H132E11

We illustrate each of the above properties with some simple computations in $M_3(\Gamma_1(11))$.

```
> M := ModularForms(Gamma1(11),3);
> Degree(M.1);
1
> f := Newform(M,1);
> Degree(f);
4
> Dimension(M);
15
> DirichletCharacters(M);
[
    1,
    $.1,
    $.1^2,
    $.1^5
]
> Level(f);
11
> Level(M);
```

```
11
> Weight(f);
3
> Weight(M);
3
> Weight(M.1);
3
```

132.8 Subspaces

The following functions compute the cuspidal, Eisenstein, and new subspaces.

`ZeroSubspace(M)`

The trivial subspace of the space of modular forms M.

`CuspidalSubspace(M)`

The subspace of forms f in M such that the constant term of the Fourier expansion of f at every cusp is 0.

`EisensteinSubspace(M)`

The Eisenstein subspace of the space of modular forms M.

`EisensteinProjection(f)`

`CuspidalProjection(f)`

The projection of a given modular form to the `EisensteinSubspace` or to the `CuspidalSubspace`. The sum of the two projections equals the original form (after coercion).

The base ring of the given form must contain the rationals.

`NewSubspace(M)`

The new subspace of the space of modular forms M.

`DihedralSubspace(M)`

For a space M of weight 1 forms, this returns the subspace spanned by the cusp forms attached to dihedral Galois representations.

Example H132E12

We compute a basis of q-expansions for each of the above subspace of $M_2(\Gamma_0(33))$.

```
> M := ModularForms(Gamma0(33),2); M;
Space of modular forms on Gamma_0(33) of weight 2 and dimension 6 over
Integer Ring.
> Basis(M);
[
    1 + O(q^8),
    q - q^5 + 2*q^7 + O(q^8),
    q^2 + 2*q^7 + O(q^8),
    q^3 + O(q^8),
    q^4 + q^5 + O(q^8),
    q^6 + O(q^8)
]
> Basis(CuspidalSubspace(M));
[
    q - q^5 - 2*q^6 + 2*q^7 + O(q^8),
    q^2 - q^4 - q^5 - q^6 + 2*q^7 + O(q^8),
    q^3 - 2*q^6 + O(q^8)
]
> Basis(EisensteinSubspace(M));
[
    1 + O(q^8),
    q + 3*q^2 + 7*q^4 + 6*q^5 + 8*q^7 + O(q^8),
    q^3 + 3*q^6 + O(q^8)
]
> Basis(NewSubspace(M));
[
    q + q^2 - q^3 - q^4 - 2*q^5 - q^6 + 4*q^7 + O(q^8)
]
> Basis(NewSubspace(EisensteinSubspace(M)));
[]
> Basis(NewSubspace(CuspidalSubspace(M)));
[
    q + q^2 - q^3 - q^4 - 2*q^5 - q^6 + 4*q^7 + O(q^8)
]
> ZeroSubspace(M);
Space of modular forms on Gamma_0(33) of weight 2 and dimension 0 over
Integer Ring.
> MQ := BaseChange(M, Rationals()); SetPrecision(MQ, 20);
> b := Basis(MQ); b[5];
q^4 + q^5 + 2*q^8 - q^9 + 2*q^10 + 2*q^11 - q^12 + 2*q^13 +
 2*q^14 - q^15 + 3*q^16 + 2*q^17 - 2*q^18 + 2*q^19 + O(q^20)
> CuspidalProjection(b[5]);
-1/10*q - 3/10*q^2 + 1/10*q^3 + 3/10*q^4 + 2/5*q^5 + 3/10*q^6 - 4/5*q^7
+ 1/2*q^8 - 3/10*q^9 + 1/5*q^10 - 1/10*q^11 - 3/10*q^12 + 3/5*q^13 -
2/5*q^14 - 2/5*q^15 - 1/10*q^16 + 1/5*q^17 + 1/10*q^18 + O(q^20)
```

```
> EisensteinProjection(b[5]);
1/10*q + 3/10*q^2 - 1/10*q^3 + 7/10*q^4 + 3/5*q^5 - 3/10*q^6 + 4/5*q^7
+ 3/2*q^8 - 7/10*q^9 + 9/5*q^10 + 21/10*q^11 - 7/10*q^12 + 7/5*q^13 +
12/5*q^14 - 3/5*q^15 + 31/10*q^16 + 9/5*q^17 - 21/10*q^18 + 2*q^19 +
O(q^20)
> MQ! $1 + MQ! $2;  // Add the previous two answers, inside MQ
q^4 + q^5 + 2*q^8 - q^9 + 2*q^10 + 2*q^11 - q^12 + 2*q^13 +
 2*q^14 - q^15 + 3*q^16 + 2*q^17 - 2*q^18 + 2*q^19 + O(q^20)
```

The two projections sum to the original form.

132.9 Operators

Each space M of modular forms comes equipped with a commuting family $T_1, T_2, T_3, \ldots$ of linear operators acting on it called the *Hecke operators*. Unfortunately, at present, the computation of Hecke and other operators on spaces of modular forms with nontrivial character has not yet been implemented, though computation of characteristic polynomials of Hecke operators is supported.

`HeckeOperator(M, n)`

The matrix representing the nth Hecke operator T_n with respect to `Basis(M)`. (Currently M must be a space of modular forms with trivial character and integral weight ≥ 2.)

`HeckeOperator(n,f)`

The image under the Hecke operator T_n of the given modular form.

`HeckePolynomial(M, n` : *parameters*`)`

`Proof` BoolElt *Default :* `true`

The characteristic polynomial of the nth Hecke operator T_n. In some situations this is more efficient than `CharacteristicPolynomial(HeckeOperator(M,n))` or any of its variants. Note that M can be an arbitrary space of modular forms.

`AtkinLehnerOperator(M, q)`

The matrix representing the qth Atkin-Lehner involution W_q on M with respect to `Basis(M)`. (Currently M must be a cuspidal space of modular forms with trivial character and integral weight ≥ 2.)

`AtkinLehnerOperator(q,f)`

The image under the involution w_q of the given modular form.

Example H132E13

First we compute a characteristic polynomial on $S_2(\Gamma_1(13))$ over both $\mathbf{Z}$ and the finite field $\mathbf{F}_2$.

```
> R<x> := PolynomialRing(Integers());
> S := CuspForms(Gamma1(13),2);
> HeckePolynomial(S, 2);
x^2 + 3*x + 3
> S2 := BaseExtend(S, GF(2));
> R<y> := PolynomialRing(GF(2));
> Factorization(HeckePolynomial(S2,2));
[
    <y^2 + y + 1, 1>
]
```

Next we compute a Hecke operator on $M_4(\Gamma_0(14))$.

```
> M := ModularForms(Gamma0(14),4);
> T := HeckeOperator(M,2);
> T;
[  1   0   0   0   0   0   0 240]
[  0   0   0   0  18  12  50 100]
[  0   1   0   0  -2  18  12 -11]
[  0   0   0   0   1  22  25  46]
[  0   0   1   0  -1 -16 -20 -82]
[  0   0   0   0  -1  -6  -9 -38]
[  0   0   0   1   3   9  15  39]
[  0   0   0   0   0   0   0   8]
> Parent(T);
Full Matrix Algebra of degree 8 over Integer Ring
> Factorization(CharacteristicPolynomial(T));
[
    <x - 8, 2>,
    <x - 2, 1>,
    <x - 1, 2>,
    <x + 2, 1>,
    <x^2 + x + 8, 1>
]
> f := M.1;
> f*T;
1 + 240*q^7 + O(q^8)
> M.1 + 240*M.8;
1 + 240*q^7 + O(q^8)
```

This example demonstrates the Atkin-Lehner involution W_3 on $S_2(\Gamma_0(33))$.

```
> M := ModularForms(33,2);
> S := CuspidalSubspace(M);
> W3 := AtkinLehnerOperator(S, 3);
> W3;
[   1    0    0]
```

```
[ 1/3  1/3 -4/3]
[ 1/3 -2/3 -1/3]
> Factorization(CharacteristicPolynomial(W3));
[
    <x - 1, 2>,
    <x + 1, 1>
]
> f := S.2;
> f*W3;
1/3*q + 1/3*q^2 - 4/3*q^3 - 1/3*q^4 - 2/3*q^5 + 5/3*q^6
     + 4/3*q^7 + O(q^8)
```

The Atkin-Lehner and Hecke operators need not commute:

```
> T3 := HeckeOperator(S, 3);
> T3;
[ 0 -2 -1]
[ 0 -1  1]
[ 1 -2 -1]
> T3*W3 - W3*T3 eq 0;
false
```

132.10 Eisenstein Series

The intrinsics below require that the base ring of M has characteristic 0. To compute mod p eigenforms, use the `Reduction` intrinsic (see Section 132.14).

`EisensteinSeries(M)`

List of the Eisenstein series associated to the modular forms space M. By "associated to" we mean that the Eisenstein series lies in $M \otimes \mathbf{C}$.

`IsEisensteinSeries(f)`

Returns `true` if the modular form f was created using `EisensteinSeries`.

`EisensteinData(f)`

The data $< \chi, \psi, t, \chi', \psi' >$ that defines the Eisenstein series (modular form) f. Here χ is a primitive character of conductor S, ψ is primitive of conductor M, and MSt divides N, where N is the level of f. (The additional characters χ' and ψ' are equal to χ and ψ respectively, except they take values in the big field $\mathbf{Q}(\zeta_{\phi(N)})^*$ instead of $\mathbf{Q}(\zeta_n)^*$, where n is the order of χ or ψ.) The Eisenstein series associated to (χ, ψ, t) has q-expansion

$$c_0 + \sum_{m \geq 1} \left(\sum_{n|m} \psi(n) n^{k-1} \chi(m/n) \right) q^{mt},$$

where $c_0 = 0$ if $S > 1$ and $c_0 = L(1-k, \psi)/2$ if $S = 1$.

Example H132E14

We illustrate the above intrinsics by computing the Eisenstein series in $M_3(\Gamma_1(12))$.

```
> M := ModularForms(Gamma1(12),3); M;
Space of modular forms on Gamma_1(12) of weight 3 and dimension 13
over Integer Ring.
> E := EisensteinSubspace(M); E;
Space of modular forms on Gamma_1(12) of weight 3 and dimension 10
over Integer Ring.
> s := EisensteinSeries(E); s;
[*
-1/9 + q - 3*q^2 + q^3 + 13*q^4 - 24*q^5 - 3*q^6 + 50*q^7 + O(q^8),
-1/9 + q^2 - 3*q^4 + q^6 + O(q^8),
-1/9 + q^4 + O(q^8),
-1/4 + q + q^2 - 8*q^3 + q^4 + 26*q^5 - 8*q^6 - 48*q^7 + O(q^8),
-1/4 + q^3 + q^6 + O(q^8),
q + 3*q^2 + 9*q^3 + 13*q^4 + 24*q^5 + 27*q^6 + 50*q^7 + O(q^8),
q^2 + 3*q^4 + 9*q^6 + O(q^8),
q^4 + O(q^8),
q + 4*q^2 + 8*q^3 + 16*q^4 + 26*q^5 + 32*q^6 + 48*q^7 + O(q^8),
q^3 + 4*q^6 + O(q^8)
*]
> a := EisensteinData(s[1]); a;
<1, $.1, 1, 1, $.2>
> Parent(a[2]);
Group of Dirichlet characters of modulus 3 over Rational Field
> Order(a[2]);
2
> Parent(a[5]);
Group of Dirichlet characters of modulus 12 over Cyclotomic Field of
order 4 and degree 2
> Parent(s[1]);
Space of modular forms on Gamma_1(12) of weight 3 and dimension 10
over Rational Field.
> IsEisensteinSeries(s[1]);
true
```

132.11 Weight Half Forms

Modular forms of weight 1/2 are constructed directly as q-expansions following the explicit description given by Serre and Stark (see [SS77]).

`WeightOneHalfData(M)`

For a space M of modular forms of weight 1/2, this returns the basis of the space as described by Serre and Stark. A list of tuples is returned; each tuple contains a character ψ and an integer t, and the corresponding modular form is the theta series defined as the sum over all integers n of $\psi(n)q^{(tn^2)}$.

132.12 Weight One Forms

Modular forms of weight 1 can be defined using the usual constructors. For these spaces, the `Dimension`, the `CuspidalSubspace` and `EisensteinSubspace`, `EisensteinSeries`, a `qExpansionBasis`, and Hecke operators are available, as well as basic functionality such as element arithmetic.

The algorithm used to determine spaces of weight 1 forms is as follows. The Eisenstein series are constructed directly as q-expansions. The cuspidal eigenforms correspond to Galois representations; those corresponding to dihedral Galois representations are obtained explicitly (from characters on ray class groups of quadratic fields). If the dihedral forms span the full space of cusp forms, this is proved by comparing with suitable spaces of integral weight forms; if not, a q-expansion basis for the cuspidal space is obtained using the integral-weight spaces (this is the most time-consuming part of the process).

`DihedralForms(M)`

For a space of weight 1, this returns the cuspidal eigenforms in M corresponding to dihedral Galois representations, broken up according to character. A list of tuples is returned; each tuple contains an element of the `DirichletCharacters` of M, followed by a list of eigenforms.

132.13 Newforms

In this section we describe how to compute both cuspidal and Eisenstein newforms.

The intrinsics below require that the base ring of M has characteristic 0. To compute mod p eigenforms, use the `Reduction` intrinsic (see Section 132.14).

`NumberOfNewformClasses(M : parameters)`

`Proof` BoolElt *Default :* `true`

The number of Galois conjugacy-classes of newforms associate to the modular forms space M, which must have base ring $\mathbf{Z}$ or $\mathbf{Q}$. By "associated to" we mean that the newform lies in $M \otimes \mathbf{C}$.

`Newform(M, i, j : parameters)`

`Proof` BoolElt *Default :* `true`

The jth Galois-conjugate newform in the ith Galois-orbit of newforms in the space of modular forms M, which must have base ring $\mathbf{Z}$ or $\mathbf{Q}$.

`Newform(M, i : parameters)`

`Proof` BoolElt *Default :* `true`

The first Galois-conjugate newform in the ith orbit in the space of modular forms M, which must have base ring $\mathbf{Z}$ or $\mathbf{Q}$.

`Newforms(M : parameters)`

`Proof` BoolElt *Default :* `true`

Sort list of the newforms associated to the space of modular forms M divided up into Galois orbits.

`Newforms(I, M)`

Use this intrinsic to find the newforms associated to the space of modular forms M with prespecified eigenvalues. Here I is a sequence $[\langle p_1, f_1(x)\rangle, ..., \langle p_n, f_n(x)\rangle]$ of pairs. Each pair consists of a prime number that does not divide the level of M and a polynomial. This intrinsic returns the set of newforms $\sum a_n q^n$ in M such that $f_n(a_{p_n}) = 0$. (This intrinsic only works when M is cuspidal and defined over $\mathbf{Q}$ or $\mathbf{Z}$.)

Example H132E15

We compute the newforms in $M_5(\Gamma_1(8))$.

```
> M := ModularForms(Gamma1(8),5); M;
Space of modular forms on Gamma_1(8) of weight 5 and dimension 11 over
Integer Ring.
> NumberOfNewformClasses(M);
4
> Newforms(M);
[* [*
q + 4*q^2 - 14*q^3 + 16*q^4 - 56*q^6 + O(q^8)
*], [*
q + 1/24*(a - 30)*q^2 + 6*q^3 + 1/12*(-a - 162)*q^4 + 1/3*(-a + 6)*q^5
+ 1/4*(a - 30)*q^6 + 1/3*(2*a - 12)*q^7 + O(q^8),
q + 1/24*(b - 30)*q^2 + 6*q^3 + 1/12*(-b - 162)*q^4 + 1/3*(-b + 6)*q^5
+ 1/4*(b - 30)*q^6 + 1/3*(2*b - 12)*q^7 + O(q^8)
*], [*
57/2 + q + q^2 + 82*q^3 + q^4 - 624*q^5 + 82*q^6 - 2400*q^7 + O(q^8)
*], [*
q + 16*q^2 + 82*q^3 + 256*q^4 + 624*q^5 + 1312*q^6 + 2400*q^7 + O(q^8)
*] *]
> Newform(M,1);
```

```
q + 4*q^2 - 14*q^3 + 16*q^4 - 56*q^6 + O(q^8)
> Newform(M,2);
q + 1/24*(a - 30)*q^2 + 6*q^3 + 1/12*(-a - 162)*q^4 + 1/3*(-a + 6)*q^5
+ 1/4*(a - 30)*q^6 + 1/3*(2*a - 12)*q^7 + O(q^8)
> Parent(Newform(M,2));
Space of modular forms on Gamma_1(8) of weight 5 and dimension 2 over
Number Field with defining polynomial x^2 - 12*x + 8676 over the
Rational Field.
> Newform(M,2,2);
q + 1/24*(b - 30)*q^2 + 6*q^3 + 1/12*(-b - 162)*q^4 + 1/3*(-b + 6)*q^5
+ 1/4*(b - 30)*q^6 + 1/3*(2*b - 12)*q^7 + O(q^8)
> IsEisensteinSeries(Newform(M,1));
false
> IsEisensteinSeries(Newform(M,2));
false
> IsEisensteinSeries(Newform(M,3));
true
> IsEisensteinSeries(Newform(M,4));
true
```

The following example demonstrates picking out a newform in $S_2(\Gamma_0(65))$ with prespecified eigenvalues.

```
> S := CuspForms(65,2);
> R<x> := PolynomialRing(IntegerRing());
> I := [<3,x+2>];
> Newforms(I,S);
[* [*
q - q^2 - 2*q^3 - q^4 - q^5 + 2*q^6 - 4*q^7 + O(q^8)
*] *]
> Factorization(HeckePolynomial(S, 2));
[
    <x + 1, 1>,
    <x^2 - 3, 1>,
    <x^2 + 2*x - 1, 1>
]
> I := [<2,x^2-3>];
> Newforms(I,S);
[* [*
q + a*q^2 + (-a + 1)*q^3 + q^4 - q^5 + (a - 3)*q^6 + 2*q^7 + O(q^8),
q + b*q^2 + (-b + 1)*q^3 + q^4 - q^5 + (b - 3)*q^6 + 2*q^7 + O(q^8)
*] *]
```

132.13.1 Labels

It is possible to obtain the galois-conjugacy class of a newform by giving a descriptive label as an argument to `Newforms`. The format of the label is as follows:

`[G0N or G1N][Level]k[Weight][Isogeny Class].`

Some example labels are `"G0N11k2A"`, `"G0N1k12A"`, `"G1N17k2B"`, and `"G1N9k3B"`. If the string `"G0N"` or `"G1N"` is omitted, then the default is `"G0N"`. Thus the following are also valid: `"11k2A"`, `"1k12A"`, `"37k4A"`. If `k[Weight]` is omitted, then the default is weight 2, so the following are valid and all refer to weight 2 modular forms on some $\Gamma_0(N)$: `"11A"`, `"37A"`, `"65B"`. In order, possibilities for the isogeny class are as follows:

`A, B, C, ..., Y, Z, AA, BB, CC, ..., ZZ, AAA, BBB, CCC, ...`.

This is essentially the notation used in [Cre97] for isogeny classes, though sometimes for levels ≤ 450 the ordering differs from that in [Cre97].

Suppose s is a valid label, and let M be the space of modular forms that contains `ModularForm(s)`. Then `ModularForm(s)` is by definition `Newforms(M)[i]` where the isogeny class in the label s is the ith isogeny class. For example `C` corresponds to the 3rd isogeny class and `BB` corresponds to the 28th.

`Newforms(label)`

The Galois-conjugacy class(es) of newforms described by the string *label*. See the introduction for a description of the notation used for the label.

Example H132E16

We give many examples of constructing newforms using labels.

```
> Newforms("11A");
[*
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
*]
>  Newforms("G0N11k2A");
[*
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
*]
>  Newforms("G0N1k12A");
[*
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 +
O(q^8)
*]
>  Newforms("G1N17k2B");
[*
q + (-a^3 + a^2 - 1)*q^2 + (a^3 - a^2 - a - 1)*q^3 + (2*a^3 - a^2 +
2*a)*q^4 + (-a^3 - a^2)*q^5 + (-a^3 + a^2 - a + 1)*q^6 + (-a^3 + a^2 +
a - 1)*q^7 + O(q^8),
q + (-b^3 + b^2 - 1)*q^2 + (b^3 - b^2 - b - 1)*q^3 + (2*b^3 - b^2 +
2*b)*q^4 + (-b^3 - b^2)*q^5 + (-b^3 + b^2 - b + 1)*q^6 + (-b^3 + b^2 +
b - 1)*q^7 + O(q^8),
q + (-c^3 + c^2 - 1)*q^2 + (c^3 - c^2 - c - 1)*q^3 + (2*c^3 - c^2 +
```

```
2*c)*q^4 + (-c^3 - c^2)*q^5 + (-c^3 + c^2 - c + 1)*q^6 + (-c^3 + c^2 +
c - 1)*q^7 + O(q^8),
q + (-d^3 + d^2 - 1)*q^2 + (d^3 - d^2 - d - 1)*q^3 + (2*d^3 - d^2 +
2*d)*q^4 + (-d^3 - d^2)*q^5 + (-d^3 + d^2 - d + 1)*q^6 + (-d^3 + d^2 +
d - 1)*q^7 + O(q^8)
*]
> Newforms("G1N9k3B");
[*
1/3*(-5*zeta_6 - 2) + q + (4*zeta_6 + 1)*q^2 + q^3 + (20*zeta_6 -
15)*q^4 + (-25*zeta_6 + 26)*q^5 + (4*zeta_6 + 1)*q^6 + (-49*zeta_6 +
1)*q^7 + O(q^8),
1/3*(5*zeta_6 - 7) + q + (-4*zeta_6 + 5)*q^2 + q^3 + (-20*zeta_6 +
5)*q^4 + (25*zeta_6 + 1)*q^5 + (-4*zeta_6 + 5)*q^6 + (49*zeta_6 -
48)*q^7 + O(q^8)
*]
> Newforms("11k2A");
[*
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
*]
> Newforms("11A");
[*
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
*]
> Newforms("1k12A");
[*
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 +
O(q^8)
*]
> Newforms("37k4A");
[*
q + a*q^2 + 1/8*(-a^3 - 9*a^2 - 26*a - 22)*q^3 + (a^2 - 8)*q^4 +
1/8*(13*a^3 + 85*a^2 + 50*a - 186)*q^5 + 1/8*(-3*a^3 - 27*a^2 - 38*a +
6)*q^6 + 1/4*(-19*a^3 - 119*a^2 - 30*a + 170)*q^7 + O(q^8),
q + b*q^2 + 1/8*(-b^3 - 9*b^2 - 26*b - 22)*q^3 + (b^2 - 8)*q^4 +
1/8*(13*b^3 + 85*b^2 + 50*b - 186)*q^5 + 1/8*(-3*b^3 - 27*b^2 - 38*b +
6)*q^6 + 1/4*(-19*b^3 - 119*b^2 - 30*b + 170)*q^7 + O(q^8),
q + c*q^2 + 1/8*(-c^3 - 9*c^2 - 26*c - 22)*q^3 + (c^2 - 8)*q^4 +
1/8*(13*c^3 + 85*c^2 + 50*c - 186)*q^5 + 1/8*(-3*c^3 - 27*c^2 - 38*c +
6)*q^6 + 1/4*(-19*c^3 - 119*c^2 - 30*c + 170)*q^7 + O(q^8),
q + d*q^2 + 1/8*(-d^3 - 9*d^2 - 26*d - 22)*q^3 + (d^2 - 8)*q^4 +
1/8*(13*d^3 + 85*d^2 + 50*d - 186)*q^5 + 1/8*(-3*d^3 - 27*d^2 - 38*d +
6)*q^6 + 1/4*(-19*d^3 - 119*d^2 - 30*d + 170)*q^7 + O(q^8)
*]
> Newforms("37k2");
[* [*
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + 6*q^6 - q^7 + O(q^8)
*], [*
q + q^3 - 2*q^4 - q^7 + O(q^8)
```

```
*], [*
3/2 + q + 3*q^2 + 4*q^3 + 7*q^4 + 6*q^5 + 12*q^6 + 8*q^7 + O(q^8)
*] *]
```

132.14 Reductions and Embeddings

`Reductions(f, p)`

The mod p reductions of the modular forms f, where $p \in \mathbf{Z}$ is a prime number and f is a modular form over a number field (or the rationals or integers). Because of denominators, the list of reductions might be empty. (In some cases when f is defined over a field of large degree, the algorithm that I've implemented is definitely not close to optimal. I know a much better algorithm, but haven't implemented it yet.)

`pAdicEmbeddings(f, p)`

The p-adic embeddings of the modular form f.

`ComplexEmbeddings(f)`

The complex embeddings of the modular form f.

Example H132E17

We compute various reductions and embeddings of a degree 3 newform in $S_2(\Gamma_0(47))$.

```
> M := ModularForms(Gamma0(47),2);
> f := Newform(M,1);
> Degree(f);
4
> f;
q + a*q^2 + (a^3 - a^2 - 6*a + 4)*q^3 + (a^2 - 2)*q^4 + (-4*a^3 +
2*a^2 + 20*a - 10)*q^5 + (-a^2 - a + 1)*q^6 + (3*a^3 - a^2 - 16*a +
7)*q^7 + O(q^8)
> Parent(f);
Space of modular forms on Gamma_0(47) of weight 2 and dimension 4 over
Number Field with defining polynomial x^4 - x^3 - 5*x^2 + 5*x - 1 over
the Rational Field.
> Reductions(f,3);
[* [*
q + 2*q^2 + 2*q^3 + 2*q^4 + q^6 + q^7 + O(q^8)
*], [*
q + $.1^4*q^2 + $.1^25*q^3 + $.1^15*q^4 + $.1^20*q^5 + $.1^3*q^6 +
$.1^3*q^7 + O(q^8),
q + $.1^10*q^2 + $.1^17*q^3 + $.1^5*q^4 + $.1^24*q^5 + $.1*q^6 +
$.1*q^7 + O(q^8),
q + $.1^12*q^2 + $.1^23*q^3 + $.1^19*q^4 + $.1^8*q^5 + $.1^9*q^6 +
```

```
$.1^9*q^7 + O(q^8)
*] *]
```

The reductions are genuine modular forms, so we can compute them to higher precision later.

```
> f3 := Reductions(f,3)[1][1];
> Type(f3);
ModFrmElt
> Parent(f3);
Space of modular forms on Gamma_0(47) of weight 2 and dimension 4 over
Finite field of size 3.
> PowerSeries(f3,15);
q + 2*q^2 + 2*q^3 + 2*q^4 + q^6 + q^7 + q^9 + q^12 + 2*q^14 + O(q^15)
> f3^2;
q^2 + q^3 + 2*q^4 + q^7 + O(q^8)
> pAdicEmbeddings(f,3)[1][1];
q + (1383893738 + O(3^20))*q^2 + (288495368 + O(3^20))*q^3 +
(1448516780 + O(3^20))*q^4 + (291254407*3 + O(3^20))*q^5 + (654373882
+ O(3^20))*q^6 - (443261663 + O(3^20))*q^7 + O(q^8)
```

The p-adic precision can be increased by recreating p-adic field with higher precision.

```
> _ := pAdicField(3 : Precision := 30);
> pAdicEmbeddings(f,3)[1][1];
q + (27072777983102 + O(3^30))*q^2 - (7262683411915 + O(3^30))*q^3 +
(102359491392536 + O(3^30))*q^4 - (26022742991723*3 + O(3^30))*q^5 +
(76458862719010 + O(3^30))*q^6 + (31185356420881 + O(3^30))*q^7 +
O(q^8)
> ComplexEmbeddings(f)[1][1];
q + 0.28384129078391142728240587955711710388*q^2 +
2.23925429984775850028724717074345372615990081162*q^3 -
1.91943412164612303785432431586190778394853582709*q^4 -
4.25351411923443363456996356947846775230044737382*q^5 +
0.63559283086221161057191843630479068005905688171 2*q^6 +
2.44657723781885227971790463962077981836158867144*q^7 + O(q^8)
```

132.15 Congruences

`CongruenceGroup(M1, M2, prec)`

A group C that measures all possible congruences (to precision *prec*) between some modular form in M_1 and some modular form in M_2. The group C is defined as follows. Let W_1 be the finite-rank $\mathbf{Z}$-module q-exp$(M_1) \cap \mathbf{Z}[[q]]$ and let W_2 be q-exp$(M_2) \cap \mathbf{Z}[[q]]$. Let V be the saturation of $W_1 + W_2$ in $\mathbf{Z}[[q]]$. Then $C = V/(W_1 + W_2)$.

`CongruenceGroupAnemic(M1, M2, prec)`

Analogous to `CongruenceGroup`, but now considering congruences that hold for all q-expansion coefficients a_n with n coprime to the levels of both $M1$ and $M2$ (rather than for all q-expansion coefficients).

Example H132E18

We verify that the newform corresponding to the first elliptic curve of rank 2 is congruent modulo 5 to some Galois-conjugate newform corresponding to the winding quotient of $J_0(389)$ (there is also a congruence modulo 2 to some conjugate form).

```
> M := ModularForms(Gamma0(389),2);
> f := Newform(M,1);
> Degree(f);
1
> g := Newform(M,5);
> Degree(g);
20
> CongruenceGroup(Parent(f),Parent(g),30);
Abelian Group isomorphic to Z/20
Defined on 1 generator
Relations:
    20*$.1 = 0
```

The congruence can be seen directly by computing the reductions of f and g modulo 5:

```
> fmod5 := Reductions(f,5);
> gmod5 := Reductions(g,5);  // takes a few seconds.
> #gmod5;
7
> #fmod5;
1
> [gbar : gbar in gmod5 | #gbar eq 1];
[ [*
q + 4*q^2 + q^3 + 4*q^4 + q^5 + 4*q^6 + O(q^8)
*], [*
q + 3*q^2 + 3*q^3 + 2*q^4 + 2*q^5 + 4*q^6 + O(q^8)
*] ]
> fmod5[1][1];
q + 3*q^2 + 3*q^3 + 2*q^4 + 2*q^5 + 4*q^6 + O(q^8)
```

132.16 Overconvergent Modular Forms

These routines compute characteristic series of operators on overconvergent modular forms. While these are p-adic modular forms, the result also gives information about classical spaces: it determines the characteristic series up to a congruence (details are given below). The big advantage of this approach is that extremely large weights can be handled (the method works by indirectly computing a small part of the large space).

The algorithm has a running time which is linear in $\log(k)$, where k is the weight. The implementation for level 1 is well optimized.

The algorithm is given in Algorithms 1 and 2 of [Lau11]. Suggestions of David Loeffler and John Voight are used in generating certain spaces of classical modular forms for level $N \geq 2$.

`OverconvergentHeckeSeriesDegreeBound(p, N, k, m)`

This returns a bound on the degree of the characteristic series modulo p^m of the Atkin U_p operator on the space of overconvergent p-adic modular forms of (even) weight k and level $\Gamma_0(N)$. This bound is due to Daqing Wan and depends only on k modulo $p-1$ rather than k itself.

`OverconvergentHeckeSeries(p, N, k, m)`

`WeightBound` RngIntElt *Default :* 6

This returns the characteristic series $P(t)$ modulo p^m of the Atkin U_p operator acting on the space of overconvergent p-adic modular forms of level $\Gamma_0(N)$ and weight k. Here $p \geq 5$ is a prime not dividing N.

When $m \leq k-1$, by Coleman's theorem, $P(t)$ is also the reverse characteristic polynomial modulo p^m of the Atkin U_p operator on the space of classical modular forms of level $\Gamma_0(Np)$ and weight k, In addition, when $m \leq (k-2)/2$, P(t) is the reverse characteristic polynomial modulo p^m of the Hecke T_p operator on the space of classical modular forms of level $\Gamma_0(N)$ and weight k.

The optional parameter `WeightBound` gives an (even integer) bound on the weight of the generators which are chosen for certain spaces of classical modular forms required at one point in the algorithm (for level $N \geq 2$). The default value is 6; however, for most levels the algorithm terminates more quickly with `WeightBound := 4`. For some small levels, the algorithm may fail to terminate with these settings, however it will terminate when `WeightBound` is set to a sufficiently large value. There is some randomization in the algorithm, which may also cause variation in running time. The output is provably correct in all instances where the algorithm terminates.

`OverconvergentHeckeSeries(p, N, kseq, m)`

`WeightBound` RngIntElt *Default :* 6

This returns a sequence containing `OverconvergentHeckeSeries(p, N, k, m)`, as above, for each weight k in the given sequence `kseq`. These weights must be congruent to each other modulo $p-1$. Note: it is more efficient to compute several (congruent) weights together, because much of the computational work is the same.

Example H132E19

```
> _<t> := PolynomialRing(Integers()); // use t for printing
> time OverconvergentHeckeSeries(5,11,10000,5);
625*t^8 + 375*t^7 + 1275*t^6 - 875*t^5 - 661*t^4 + 335*t^3 + 1189*t^2 + 861*t + 1
Time: 0.830
> time OverconvergentHeckeSeries(5,11,10000,5 : WeightBound := 4);
625*t^8 + 375*t^7 + 1275*t^6 - 875*t^5 - 661*t^4 + 335*t^3 + 1189*t^2 + 861*t + 1
Time: 0.400
```

Note that the same result was obtained each time, but the weight bound reduced the computational work involved.

```
> time OverconvergentHeckeSeries(7,3,[1000,1006],5);
[
    7203*t^6 - 4116*t^5 + 6713*t^4 - 700*t^3 - 4675*t^2 - 4426*t + 1,
    2401*t^6 + 8134*t^4 - 3976*t^3 + 5160*t^2 + 5087*t + 1
]
Time: 0.110
> time Pseq := OverconvergentHeckeSeries(7,3,[1000,1006],5: WeightBound := 4);
Time: 0.090
> assert Pseq eq $1; // check that it is the same as the previous result
> OverconvergentHeckeSeriesDegreeBound(7,3,1000,5);
9
> time OverconvergentHeckeSeries(29,1,10000,10);
    134393786323419*t^8 - 174295724342741*t^7 - 174857545225000*t^6
    - 153412311426730*t^5 + 41820892464727*t^4 - 148803482429283*t^3
    - 111232323996568*t^2 + 165679475331974*t + 1
Time: 1.000
```

132.17 Algebraic Relations

`Relations(M, d, prec)`

The relations of degree d satisfied by the q-expansions of in the space M of modular forms. The q-expansions are computed to precision *prec*. If prec is too small, this intrinsic might return relations that are not really satisfied by the modular forms. To be sure of your result, `prec` must be at least as large as `PrecisionBound(M2)`, where M_2 has the same level as M and weight d times the weight of M.

Example H132E20

We compute an equation that defines the canonical embedding of $X_0(34)$.

```
> S := CuspidalSubspace(ModularForms(Gamma0(34)));
> Relations(S, 4, 20);
[
    a^3*c - a^2*b^2 - 3*a^2*c^2 + 2*a*b^3 + 3*a*b^2*c - 3*a*b*c^2 +
        4*a*c^3 - b^4 + 4*b^3*c - 6*b^2*c^2 + 4*b*c^3 - 2*c^4
]
[
    (0  0  1 -1  0 -3  2  3 -3  4 -1  4 -6  4 -2)
]
> // a, b, and c correspond to the cusp forms S.1, S.2 and S.3:
> S.1;
q - 2*q^4 - 2*q^5 + 4*q^7 + O(q^8)
> S.2;
q^2 - q^4 + O(q^8)
> S.3;
q^3 - 2*q^4 - q^5 + q^6 + 4*q^7 + O(q^8)
```

Next we compute the canonical embedding of $X_0(75)$.

```
> S := CuspidalSubspace(ModularForms(Gamma0(75)));
> R := Relations(S, 2, 20); R;
[
    a*c - b^2 - d^2 - 4*e^2,
    a*d - b*c + b*e + d*e - 3*e^2,
    a*e - b*d - c*e
]
> // NOTE: It is much faster to compute in the power
> // series ring than the ring of modular forms!
> a, b, c, d, e := Explode([PowerSeries(f,20) : f in Basis(S)]);
> a*c - b^2 - d^2 - 4*e^2;
O(q^21)
```

The connection between the above computations and models for modular curves is discussed in Steven Galbraith's Oxford Ph.D. thesis.

132.18 Elliptic Curves

Little has been implemented so far.

`ModularForm(E)`

`Newform(E)`

The modular form associated to the elliptic curve E (which must be defined over the rationals).

`Eigenform(E, prec)`

`qEigenform(E, prec)`

The q-expansion of the newform associated to E, to the specified precision.

Note: this is exactly the same as calling `qExpansion(ModularForm(E),prec)`.

`EllipticCurve(f)`

An elliptic curve E with associated modular form f, when f is a weight 2 newform on $\Gamma_0(N)$ with rational Fourier coefficients.

The Cremona database is used to identify the isogeny class. (A routine to compute the curve from scratch is implemented, and can be called with `EllipticCurve(M : Database:=false)` where M is the relevant space of modular symbols; however this is not optimized for large level.)

Example H132E21

```
> M := ModularForms(Gamma0(389),2);
> f := Newform(M,1);
> Degree(f);
1
> E := EllipticCurve(f);
> E;
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational
Field
> Conductor(E);
389
> time s := PowerSeries(f,200);  // faster because it knows the elliptic curve
Time: 0.509
```

132.19 Modular Symbols

`ModularSymbols(M)`

The sequence of characteristic 0 spaces of modular symbols with given sign associated to the space of modular forms M, when this makes sense.

`ModularSymbols(M, sign)`

The sequence of characteristic 0 spaces of modular symbols with given sign associated to the space of modular forms M, when this makes sense.

Example H132E22

```
> M := ModularForms(Gamma0(389),2);
> ModularSymbols(M,+1);
[
    Full Modular symbols space of level 389, weight 2, and dimension
    33
]
> ModularSymbols(M,-1);
[
    Full Modular symbols space of level 389, weight 2, and dimension
    32
]
> M := ModularForms(Gamma1(13),2);
> ModularSymbols(M);
[
    Full Modular symbols space of level 13, weight 2, and dimension 1,
    Full Modular symbols space of level 13, weight 2, character $.1,
    and dimension 0,
    Full Modular symbols space of level 13, weight 2, character $.1,
    and dimension 4,
    Full Modular symbols space of level 13, weight 2, character $.1,
    and dimension 0,
    Full Modular symbols space of level 13, weight 2, character $.1^2,
    and dimension 2,
    Full Modular symbols space of level 13, weight 2, character $.1,
    and dimension 2
]
> Basis($1)[1];
-1/6*{-7/15, -6/13} + -1/6*{-7/18, -5/13} + -1/6*{-5/16, -4/13} +
-1/6*{-4/17, -3/13} + -1/6*{-3/19, -2/13} + -1/6*{0, oo}
```

132.20 Bibliography

[**Atk67**] A. O. L. Atkin. Weierstrass points at cusps $\Gamma_o(N)$. *Ann. of Math. (2)*, 85:42–45, 1967.

[**Cre97**] J. E. Cremona. *Algorithms for modular elliptic curves*. Cambridge University Press, Cambridge, second edition, 1997.

[**DI95**] Fred Diamond and Ju Im. Modular forms and modular curves. In *Seminar on Fermat's Last Theorem*, pages 39–133. Amer. Math. Soc., Providence, RI, 1995.

[**Lau11**] A. Lauder. Computations with classical and p-adic modular forms. *LMS J. Comput. Math.*, 14:214–231, 2011.

[**SS77**] J.-P. Serre and H. M. Stark. Modular forms of weight 1/2. In *Modular functions of one variable, VI (Proc. Second Internat. Conf., Univ. Bonn, Bonn, 1976)*, pages 27–67. Lecture Notes in Math., Vol. 627. Springer, Berlin, 1977.

[**Ste07**] William A. Stein. *Modular forms: a computational approach*, volume 79 of *Graduate Studies in Mathematics*. Amer. Math. Soc., Providence, Rhode Island, 2007.

133 MODULAR SYMBOLS

Chapter 133
MODULAR SYMBOLS

133.1 Introduction

This chapter, which was written by William Stein (`was@math.harvard.edu`) with the help of feedback from Kevin Buzzard, describes how to compute with modular symbols using MAGMA. Modular symbols provide a presentation for certain homology groups, and as such they can be used to compute an eigenform basis for spaces of cusp forms $S_k(N, \varepsilon)$, where $k \geq 2$ is an integer and ε is an *arbitrary* Dirichlet character. Their generality makes modular symbols a natural tool in applications ranging from verification of modularity of Galois representations to elliptic curve computations.

Our implementation of modular symbols algorithms in MAGMA was deeply influenced by [Cre92, Cre97, Mer94]. The algorithms for computing arithmetic invariants of modular abelian varieties are based on [Ste00]. Those unfamiliar with modular symbols might wish to consult [Ste08] and the references contained therein and peruse [FM99].

133.1.1 Modular Symbols

The *modular group* $\mathrm{SL}_2(\mathbf{Z})$ is the group of 2×2 integer matrices with determinant 1. For each positive integer N let $\Gamma_0(N)$ denote the subgroup of $\mathrm{SL}_2(\mathbf{Z})$ of matrices that are upper triangular modulo N. As explained in the survey paper [DI95], there is an algebraic curve $X_0(N)$ over $\mathbf{Q}$ attached to $\Gamma_0(N)$. The Riemann surface attached to $X_0(N)$ is a compactified quotient of the upper half plane by the action of $\Gamma_0(N)$ via linear fractional transformations. Modular symbols provide an explicit computable presentation for certain "(co-)homology groups" attached to modular curves $X_0(N)$.

Let $\mathbf{P^1}(\mathbf{Q})$ denote the set $\mathbf{Q} \cup \{\infty\}$, and fix a field F. Let $\mathbf{M}$ denote the F-vector space generated by the formal symbols $\{a, b\}$, with $a, b \in \mathbf{P^1}(\mathbf{Q})$, modulo the relations $\{a, b\} + \{b, c\} + \{c, a\} = 0$ for all $a, b, c \in \mathbf{Q}$. (The symbol $\{a, b\}$ can be visualized as the homology class of a geodesic path from a to b in the upper half plane.) Fix a positive integer k. A weight-k symbol is a formal product $X^i Y^{k-2-i}\{a, b\}$, where $X^i Y^{k-2-i} \in F[X, Y]$. Denote by $\mathbf{M_k}$ the formal F-vector space with basis the set of all weight-k modular symbols (thus $\mathbf{M_k} \approx \mathbf{M} \otimes \mathrm{Sym}^{\mathbf{k-2}}(\mathbf{F} \times \mathbf{F})$). The group $\mathrm{GL}_2(\mathbf{Q})$ acts on the left on $\mathbf{M_k}$; the matrix $g = \begin{pmatrix} u & v \\ w & z \end{pmatrix}$ in $\mathrm{GL}_2(\mathbf{Q})$ acts by

$$g(X^i Y^{k-2-i}\{a, b\}) = (zX - vY)^i(-wX + uY)^{k-2-i} \left\{ \frac{ua + v}{wa + z}, \frac{ub + v}{wb + z} \right\}.$$

A *mod N Dirichlet character* ε is a homomorphism

$$\varepsilon : (\mathbf{Z}/N\mathbf{Z})^* \to F^*.$$

The vector space $\mathbf{M_k}(\mathbf{N}, \varepsilon; \mathbf{F})$ of *modular symbols of weight k, level N and character ε over F* is the quotient of $\mathbf{M_k}$ by the subspace generated by all $x - \varepsilon(u)g(x)$, for x in $\mathbf{M_k}$ and $g = \begin{pmatrix} u & v \\ w & z \end{pmatrix} \in \Gamma_0(N)$. We denote the equivalence class that defines a modular symbol by giving a representative element.

The space of modular symbols is a finite-dimensional vector space, and there is a natural finite presentation for it in terms of Manin symbols.

133.2 Basics

133.2.1 Verbose Output

The verbosity level for modular symbols computations can be set using the command `SetVerbose("ModularSymbols",n)`, where `n` is 0 (silent), 1 (verbose), or 2 (very verbose). (The verbose flag for modular symbols was called `ModularForms` in MAGMA version 2.7.)

133.2.2 Categories

Spaces of modular symbols belong to the category `ModSym`. The category `SetCsp` has exactly one object `Cusps()`, which is the set $\mathbf{P^1}(\mathbf{Q}) = \mathbf{Q} \cup \{\infty\}$ introduced above. The element ∞ of $\mathbf{P^1}(\mathbf{Q})$ is entered using the expression `Cusps()!Infinity()`.

Example H133E1

We compute a basis for the space of modular symbols of weight 2, level 11 and trivial character.

```
> M := ModularSymbols(11,2); M;
Full modular symbols space for Gamma_0(11) of weight 2 and dimension 3
over Rational Field
> Type(M);
ModSym
> Basis(M);
[
    {-1/7, 0},
    {-1/5, 0},
    {oo, 0}
]
> M!<1,[1/5,1]>;
{-1/5, 0}
> // the modular symbols {1/5,1} and {-1/5,0} are equal.
> Type(M!<1,[1/5,1]>);
ModSymElt
```

Using `SetVerbose`, we can see how the computation progresses.

```
> SetVerbose("ModularSymbols",2);
> M := ModularSymbols(11,2);
Computing space of modular symbols of level 11 and weight 2....
I.      Manin symbols list.
```

```
                (0 s)
II.     2-term relations.
                (0.019 s)
III.    3-term relations.
         Computing quotient by 4 relations.
                (0.009 s)
                (total time to create space = 0.029 s)
> SetVerbose("ModularSymbols",0);
```

Modular symbols can be input using `Cusps()`.

```
> M := ModularSymbols(11,2);
> P := Cusps(); P;
Set of all cusps
> Type(P);
SetCsp
> oo := P!Infinity();
> M!<1,[oo,P!0]>;       // note that 0 must be coerced into P.
{oo, 0}
> M!<1,[1/5,1]> + M!<1,[oo,P!0]>;
{-1/5, 0} + {oo, 0}
```

Modular symbols are also defined over finite fields.

```
> M := ModularSymbols(11,2,GF(7)); M;
Full modular symbols space for Gamma_0(11) of weight 2 and dimension 3
over Finite field of size 7
> BaseField(M);
Finite field of size 7
> 7*M!<1,[1/5,1]>;
0
```

133.3 Creation Functions

133.3.1 Ambient Spaces

An ambient space of modular symbols is created by specifying a base ring, character, weight, and optional sign. Note that spaces of modular symbols may be defined directly over any base ring (unlike `ModularForms`, which can only be base extensions of spaces over the integers). The most general signature is `ModularSymbols(eps, k, sign)`, where `eps` is a Dirichlet character (the level is taken to be the modulus of `eps`, and the base ring is taken to be the base ring of `eps`).

For information on Dirichlet characters, see Section 19.8.

Warning: Certain functions, such as `DualVectorSpace`, may fail when given as input a space of modular symbols over a field of positive characteristic, because the Hecke operators T_p, with p prime to the level, need not be semisimple.

`ModularSymbols(N)`

The space of modular symbols of level N, weight 2, and trivial character over the rational numbers.

`ModularSymbols(N, k)`

The space of modular symbols of level N, weight k, and trivial character over the rational numbers.

`ModularSymbols(N, k, F)`

The space of modular symbols of level N, weight k, and trivial character over the field F.

`ModularSymbols(N, k, sign)`

The space of modular symbols of level N, weight k, trivial character, and given *sign* (as an integer) over the rational numbers.

`ModularSymbols(N, k, F, sign)`

The space of modular symbols of level N, weight k, trivial character, and given *sign* (as an integer), over the field F.

`ModularSymbols(eps, k)`

The space of modular symbols of weight k and character ε (as an element of a Dirichlet group). Note that ε determines the level and the base field, so they do not need to be specified.

`ModularSymbols(eps, k, sign)`

The space of modular symbols of weight k and character ε (as an element of a Dirichlet group). The level and base field are specified as part of ε. The third argument "sign" allows for working in certain quotients. The possible values are -1, 0, and $+1$, which correspond to the -1 quotient, full space, and $+1$ quotient, respectively. The $+1$ quotient of M is $M/(*-1)M$, where $*$ is `StarInvolution(M)`.

Example H133E2

We create spaces of modular symbols in several different ways.

```
> M37 := ModularSymbols(37); M37;
Full modular symbols space for Gamma_0(37) of weight 2 and dimension 5
over Rational Field
> Basis(M37);
[
    {-1/29, 0},
    {-1/22, 0},
    {-1/12, 0},
    {-1/18, 0},
    {oo, 0}
```

```
]
```

As M37 is a space of modular symbols, it is not incorrect that its dimension is different than that of the three-dimensional space of modular forms $M_2(\Gamma_0(37))$. We have

$$\dim \mathbf{M_2}(\mathbf{\Gamma_0}(\mathbf{37})) = \mathbf{2} \times (\text{dim cusp forms}) + \mathbf{1} \times (\text{dim Eisenstein series}) = \mathbf{5}.$$

```
> MF := ModularForms(Gamma0(37),2);
> 2*Dimension(CuspidalSubspace(MF)) + Dimension(EisensteinSubspace(MF));
5
```

Next we decompose M37 with respect to the Hecke operators T_2, T_3, and T_5.

```
> D := Decomposition(M37,5); D;
[
     Modular symbols space for Gamma_0(37) of weight 2 and dimension 1
     over Rational Field,
     Modular symbols space for Gamma_0(37) of weight 2 and dimension 2
     over Rational Field,
     Modular symbols space for Gamma_0(37) of weight 2 and dimension 2
     over Rational Field
]
```

The first factor corresponds to the standard Eisenstein series, and the second corresponds to an elliptic curve:

```
> E := EllipticCurve(D[2]); E;
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 23*x - 50 over
Rational Field
> Rank(E);
0
```

We now create the space $\mathbf{M_{12}}(\mathbf{1})$ of weight 12 modular symbols of level 1.

```
> M12 := ModularSymbols(1,12); M12;
Full modular symbols space for Gamma_0(1) of weight 12 and dimension 3
over Rational Field
> Basis(M12);
[
    X^10*{0, oo},
    X^8*Y^2*{0, oo},
    X^9*Y*{0, oo}
]

> DimensionCuspFormsGamma0(1,12);
1
> R<z>:=PowerSeriesRing(Rationals());
> Delta(z)+ O(z^7);
z - 24*z^2 + 252*z^3 - 1472*z^4 + 4830*z^5 - 6048*z^6 + O(z^7)
```

As a module, cuspidal modular symbols equal cuspforms with multiplicity two.

```
> M12 := ModularSymbols(1,12);
```

```
> HeckeOperator(CuspidalSubspace(M12),2);
[-24   0]
[  0 -24]
>  qExpansionBasis(CuspidalSubspace(M12),7);
[
    q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 + O(q^7)
]
```

For efficiency purposes, since one is often interested only in q-expansion, it is possible to work in the quotient of the space of modular symbols by all relations $*x = x$ (or $*x = -x$), where $*$ is `StarInvolution(M)`. In either of these quotients (except possibly in characteristic $p > 0$) the cusp forms appear with multiplicity one instead of two.

```
> M12plus := ModularSymbols(1,12,+1);
> Basis(M12plus);
[
    X^10*{0, oo},
    X^8*Y^2*{0, oo}
]
> CuspidalSubspace(M12plus);
Modular symbols space for Gamma_0(1) of weight 12 and dimension 1 over
Rational Field
> qExpansionBasis(CuspidalSubspace(M12),7);
[
    q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 + O(q^7)
]
```

The following is an example of how to create Dirichlet characters in MAGMA, and how to create a space of modular symbols with nontrivial character. For more details, see Section 19.8.

```
> G<a,b,c> := DirichletGroup(16*7,CyclotomicField(EulerPhi(16*7)));
> Order(a);
2
> Conductor(a);
4
> Order(b);
4
> Conductor(b);
16
> Order(c);
6
> Conductor(c);
7
> eps := a*b*c;
> M := ModularSymbols(eps,2); M;
Full modular symbols space of level 112, weight 2, character a*b*c,
and dimension 32 over Cyclotomic Field of order 48 and degree 16
> BaseField(M);
Cyclotomic Field of order 48 and degree 16
```

133.3.2 Labels

It is also possible to create many spaces of modular symbols for $\Gamma_0(N)$ by passing a "descriptive label" as an argument to `ModularSymbols`. The most specific label is a string of the form `[Level]k[Weight][IsogenyClass]`, where `[Level]` is the level, `[Weight]` is the weight, `[IsogenyClass]` is a letter code: `A`, `B`, ..., `Z`, `AA`, `BB`, ..., `ZZ`, `AAA`, ..., and `k` is a place holder to separate the level and the weight. If the label is `[Level][IsogenyClass]`, then the weight k is assumed equal to 2. If the label is `[Level]k[Weight]` then the cuspidal subspace of the full ambient space of modular symbols of weight k and level N is returned. The following are valid labels: `11A`, `37B`, `3k12A`, `11k4`. The ordering used on isogenies classes is `lt`; see the documentation for `SortDecomposition`.

Note: There is currently no intrinsic that, given a space of modular symbols, returns its label.

Warning: For 146 of the levels between 56 and 450, our ordering of weight 2 rational newforms disagrees with the ordering used in [Cre97]. Fortunately, it is easy to create a space of modular symbols from one of Cremona's labels using the associated elliptic curve.

```
> E  := EllipticCurve(CremonaDatabase(),"56A");
> M1 := ModularSymbols(E);
```

Observe that Cremona's `"56A"` is different from ours.

```
> M2 := ModularSymbols("56A");
> M2 eq M1;
false
```

`ModularSymbols(s, sign)`

`ModularSymbols(s)`

The space of modular symbols described by a label, the string s, and given *sign*.

Example H133E3

The cusp form $\Delta(q)$ is related to the space of modular symbols whose label is `"1k12A"`.

```
> Del := ModularSymbols("1k12A"); Del;
Modular symbols space for Gamma_0(1) of weight 12 and dimension 2 over
Rational Field
> qEigenform(Del,5);
q - 24*q^2 + 252*q^3 - 1472*q^4 + O(q^5)
```

Next, we create the space corresponding to the first newform on $\Gamma_0(11)$ of weight 4.

```
> M := ModularSymbols("11k4A"); M;
Modular symbols space for Gamma_0(11) of weight 4 and dimension 4 over
Rational Field
> AmbientSpace(M);
Full modular symbols space for Gamma_0(11) of weight 4 and dimension 6
over Rational Field
```

```
> qEigenform(M,5);
q + a*q^2 + (-4*a + 3)*q^3 + (2*a - 6)*q^4 + O(q^5)
> Parent($1);
Power series ring in q over Univariate Quotient Polynomial Algebra in
a over Rational Field with modulus a^2 - 2*a - 2
```

We next create the $+1$ quotient of the cuspidal subspace of weight-4 modular symbols of level 37.

```
> M := ModularSymbols("37k4",+1); M;
Modular symbols space for Gamma_0(37) of weight 4 and dimension 9 over
Rational Field
> AmbientSpace(M);
Full modular symbols space for Gamma_0(37) of weight 4 and dimension
11 over Rational Field
> Factorization(CharacteristicPolynomial(HeckeOperator(M,2)));
[
    <x^4 + 6*x^3 - x^2 - 16*x + 6, 1>,
    <x^5 - 4*x^4 - 21*x^3 + 74*x^2 + 102*x - 296, 1>
]
```

133.3.3 Creation of Elements

Suppose M is a space of weight k modular symbols over a field F. A modular symbol $P(X,Y)\{\alpha,\beta\}$ is input as `M!<P(X,Y),[alpha,beta]>`, where $P(X,Y) \in F[X,Y]$ is homogeneous of degree $k-2$, and $\alpha, \beta \in \mathbf{P^1}(\mathbf{Q})$. Here is an example:

Example H133E4

First create the space $M = \mathbf{M_4}(\mathbf{\Gamma_0}(\mathbf{3}); \mathbf{F_7})$.

```
> F7 := GF(7);
> M := ModularSymbols(3,4,F7);
> R<X,Y> := PolynomialRing(F7,2);
```

Now we input $(X^2 - 2XY)\{0,1\}$.

```
> M!<X^2-2*X*Y,[Cusps()|0,1]>;
6*Y^2*{oo, 0}
```

Note that $(X^2 - 2XY)\{0,1\} = 6Y^2\{\infty, 0\}$ in M.
When $k = 2$, simply enter `M!<1,[alpha,beta]>`.

```
> M := ModularSymbols(11,2);
> M!<1,[Cusps()|0,Infinity()]>;
-1*{oo, 0}
> M![<1,[Cusps()|0,Infinity()]>, <1,[Cusps()|0,1/11]>];
-2*{oo, 0}
```

Any space M of modular symbols is finitely generated. One proof of this uses that every modular symbol is a linear combination of *Manin symbols*. Let $\mathbf{P^1}(\mathbf{Z}/\mathbf{NZ})$ be the set of pairs $(\overline{u},\overline{v}) \in \mathbf{Z}/N\mathbf{Z} \times \mathbf{Z}N\mathbf{Z}$ such that $\mathrm{GCD}(u,v,N) = 1$, where u and v are lifts of $\overline{u}$ and $\overline{v}$ to $\mathbf{Z}$. A Manin symbols $\langle P(X,Y),(\overline{u},\overline{v})\rangle$ is a pair consisting of a homogeneous polynomial $P(X,Y) \in F[X,Y]$ of degree $k-2$ and an element $(\overline{u},\overline{v}) \in \mathbf{P^1}(\mathbf{Z}/\mathbf{NZ})$. The modular symbol associated to $\langle P(X,Y),(\overline{u},\overline{v})\rangle$ is constructed as follows. Choose lifts u,v of $\overline{u},\overline{v}$ such that $\mathrm{GCD}(u,v) = 1$. Then there is a matrix $g = \begin{pmatrix} w & z \\ u & v \end{pmatrix}$ in $\mathrm{SL}_2(\mathbf{Z})$ whose lower two entries are u and v. The modular symbol is then $g(P(X,Y)\{0,\infty\})$. The intrinsic `ConvertFromManinSymbol` computes the modular symbol attached to a Manin symbol. Ever modular symbol can be written as a linear combination of Manin symbols using the intrinsic `ManinSymbol`.

Example H133E5

In this example, we convert between Manin and modular symbols representations of a few elements of a space of modular symbols of weight 4 over $\mathbf{F}_5$.

```
> F5 := GF(5);
> M := ModularSymbols(6,4,F5);
> R<X,Y> := PolynomialRing(F5,2);
> ConvertFromManinSymbol(M,<X^2+Y^2,[1,4]>);
(3*X^2 + 3*X*Y + 2*Y^2)*{-1/2, 0} + (X^2 + 4*X*Y + 4*Y^2)*{-1/3, 0} +
(X^2 + X*Y + 4*Y^2)*{1/3, 1/2}
> ManinSymbol(M.1-3*M.2);
[
    <X^2, (0 1)>,
    <2*X^2, (1 2)>
]
```

Thus the element `M.1-3*M.2` of M corresponds to the sum of Manin symbols $\langle X^2,(0,1)\rangle + 2\langle X^2,(1,2)\rangle$.

`M ! x`

The coercion of x into the space of modular symbols M. Here x can be either a modular symbol that lies in a subspace of M, a 2-tuple that describes a modular symbol, a sequence of such 2-tuples, or anything that can be coerced into `VectorSpace(M)`. If x is a valid sequence of such 2-tuples, then `M!x` is the sum of the coercions into M of the elements of the sequence x.

`ConvertFromManinSymbol(M, x)`

The modular symbol associated to the 2-tuple $x = \langle P(X,Y),[u,v]\rangle$, where $P(X,Y) \in F[X,Y]$ is homogeneous of degree $k-1$, F is the base field of the space of modular symbols M, and $[u,v]$ is a sequence of 2 integers that defines an element of $\mathbf{P^1}(\mathbf{Z}/\mathbf{NZ})$, where N is the level of M.

`ManinSymbol(x)`

An expression for the modular symbol x in terms of Manin symbols, which are represented as 2-tuples $\langle P(X,Y), [u,v] \rangle$.

Example H133E6

```
> M := ModularSymbols(14,2); M;
Full modular symbols space for Gamma_0(14) of weight 2 and dimension 5
over Rational Field
> Basis(M);
[
    {oo, 0},
    {-1/8, 0},
    {-1/10, 0},
    {-1/12, 0},
    {-1/2, -3/7}
]
> M!<1,[1,0]>;
0
> M!<1,[0,1/11]>;
{-1/10, 0} + -1*{-1/12, 0}
> M![<1,[0,1/2]>, <-1,[0,1/7]>];              // sequences are added
{-1/8, 0} + -1*{-1/12, 0} + -1*{-1/2, -3/7}
> M!<1,[0,1/2]> - M!<1,[0,1/7]>;
{-1/8, 0} + -1*{-1/12, 0} + -1*{-1/2, -3/7}
> M!<1,[Cusps()|Infinity(),0]>;               // Infinity() is in Cusps().
{oo, 0}
```

We can also coerce sequences into the underlying vector space of M.

```
> VectorSpace(M);
Full Vector space of degree 5 over Rational Field
Mapping from: Full Vector space of degree 5 over Rational Field to
ModSym: M given by a rule [no inverse]
Mapping from: ModSym: M to Full Vector space of degree 5 over Rational
Field given by a rule [no inverse]
> Eltseq(M.3);
[ 0, 0, 1, 0, 0 ]
> M![ 0, 0, 1, 0, 0 ];
{-1/10, 0}
> M.3;
{-1/10, 0}
```

The "polynomial coefficients" of the modular symbols are homogeneous polynomials in 2 variables of degree $k-2$.

```
> M := ModularSymbols(1,12);
> Basis(M);
[
```

```
    X^10*{0, oo},
    X^8*Y^2*{0, oo},
    X^9*Y*{0, oo}
]
> R<X,Y> := PolynomialRing(Rationals(),2);
> M!<X^9*Y,[Cusps()|0,Infinity()]>;
X^9*Y*{0, oo}
> M!<X^7*Y^3,[Cusps()|0,Infinity()]>;
-25/48*X^9*Y*{0, oo}
> Eltseq(M!<X*Y^9,[1/3,1/2]>);
[ -19171, -58050, -30970 ]
> M![1,2,3];
X^10*{0, oo} + 2*X^8*Y^2*{0, oo} + 3*X^9*Y*{0, oo}
> ManinSymbol(M![1,2,3]);
[
    <X^10, (0 1)>,
    <2*X^8*Y^2, (0 1)>,
    <3*X^9*Y, (0 1)>
]
```

133.4 Bases

`Basis(M)`

Basis for the space of modular symbols M.

`IntegralBasis(M)`

First suppose that the space of modular symbols M equals `AmbientSpace(M)`. Then this intrinsic returns a basis $x_1, \ldots, x_n$ for M such that $\mathbf{Z}x_1 + \cdots + \mathbf{Z}x_n$ is the $\mathbf{Z}$-submodule of M generated by all modular symbols $X^i \cdot Y^{k-2-i}\{\alpha, \beta\}$ with $i = 0, \ldots, k-2$ and $\alpha, \beta \in \mathbf{P^1}(\mathbf{Q})$. If M is not `AmbientSpace(M)`, then this intrinsic returns a $\mathbf{Z}$-basis for $M \cap (\mathbf{Z}x_1 + \cdots + \mathbf{Z}x_n)$, where $x_1, \ldots, x_n$ is an integral basis for `AmbientSpace(M)`. The base field of M must be $\mathbf{Q}$.

Example H133E7

```
> M := ModularSymbols(1,12);
> Basis(M);
[
    X^10*{0, oo},
    X^8*Y^2*{0, oo},
    X^9*Y*{0, oo}
]
> IntegralBasis(M);
[
```

```
    1/48*X^9*Y*{0, oo},
    1/14*X^8*Y^2*{0, oo},
    X^10*{0, oo}
]
```

`IntegralBasis(M)` is a basis for the **Z**-module spanned by the following symbols:

```
> R<X,Y> := PolynomialRing(Rationals(),2);
> [M!<X^i*Y^(10-i),[Cusps()|0,Infinity()]> : i in [0..10]];
[
    -X^10*{0, oo},
    X^9*Y*{0, oo},
    -X^8*Y^2*{0, oo},
    -25/48*X^9*Y*{0, oo},
    9/14*X^8*Y^2*{0, oo},
    5/12*X^9*Y*{0, oo},
    -9/14*X^8*Y^2*{0, oo},
    -25/48*X^9*Y*{0, oo},
    X^8*Y^2*{0, oo},
    X^9*Y*{0, oo},
    X^10*{0, oo}
]
```

We can also compute an integral basis of a subspace.

```
> C := CuspidalSubspace(M);
> IntegralBasis(C);
[
    1/48*X^9*Y*{0, oo},
    1/14*X^8*Y^2*{0, oo}
]
```

In Remark 3 on page 69 of [Mer94], Merel says "it would be interesting to find a basis in terms of Manin symbols" for the **Z**-module of Eisenstein symbols (see Section 133.8 for the definition of `EisensteinSubspace`). Here are the first few examples in the case of level 1:

```
> M := ModularSymbols(1,12);
> E := EisensteinSubspace(M);
> IntegralBasis(E);
[
    691*X^10*{0, oo} + 1620*X^8*Y^2*{0, oo}
]
> ManinSymbol(IntegralBasis(E)[1]);
[
    <691*X^10, (0 1)>,
    <1620*X^8*Y^2, (0 1)>
]
```

To more easily compute several examples, we define a function:

```
> function EisZ(k)
```

```
>    E := EisensteinSubspace(ModularSymbols(1,k));
>    B := IntegralBasis(E);
>    return [ManinSymbol(z) : z in B];
> end function;
> EisZ(12);
[
    [
        <691*X^10, (0 1)>,
        <1620*X^8*Y^2, (0 1)>
    ]
]
> EisZ(16);
[
    [
        <16380*X^12*Y^2, (0 1)>,
        <3617*X^14, (0 1)>
    ]
]
> EisZ(18);
[
    [
        <43867*X^16, (0 1)>,
        <270000*X^14*Y^2, (0 1)>
    ]
]
> EisZ(20);
[
    [
        <174611*X^18, (0 1)>,
        <1349460*X^16*Y^2, (0 1)>
    ]
]
> EisZ(22);
[
    [
        <748125*X^18*Y^2, (0 1)>,
        <77683*X^20, (0 1)>
    ]
]
```

Send me an email if you determine the basis in general. In each example above the coefficient of X^{k-2} is, up to sign, `Numerator(Bernoulli(k)/k)`.

133.5 Associated Vector Space

The functions `VectorSpace`, `DualVectorSpace`, and `Lattice` return the underlying vector space, dual vector space, and lattice associated to a space of modular symbols. A space of modular symbols is represented internally as a subspace of a vector space, and a subspace of the linear dual of the vector space. To carry along the subspace of the linear dual is useful in many computations; one example is efficient computation of Hecke operators. When the base field is $\mathbf{Q}$, the lattice comes from the natural integral structure on modular symbols.

`VectorSpace(M)`

The vector space V underlying the space of modular symbols M, the map $V \to M$, and the map $M \to V$.

`DualVectorSpace(M)`

The subspace of the linear dual of `VectorSpace(AmbientSpace(M))` that is isomorphic to the space of modular symbols M as a module over the Hecke algebra.

`Lattice(M)`

The lattice generated by the integral modular symbols in the vector space representation of the space of modular symbols M. This is the lattice generated by all modular symbols $X^i Y^{k-2-i}\{a, b\}$. The base field of M must be `RationalField()`.

Example H133E8

```
> M := ModularSymbols(DirichletGroup(11).1,3); M;
Full modular symbols space of level 11, weight 3, character $.1, and
dimension 4 over Rational Field
> VectorSpace(M);
Full Vector space of degree 4 over Rational Field
Mapping from: Full Vector space of degree 4 over Rational Field to
ModSym: M given by a rule [no inverse]
Mapping from: ModSym: M to Full Vector space of degree 4 over Rational
Field given by a rule [no inverse]
> Basis(VectorSpace(CuspidalSubspace(M)));
[
    ( 0  1  0 -1),
    ( 0  0  1 -1)
]
> Basis(VectorSpace(EisensteinSubspace(M)));
[
    (   1    0 -2/3 -1/3),
    ( 0  1 -5 -2)
]
> Lattice(CuspidalSubspace(M));
Lattice of rank 2 and degree 4
Basis:
```

```
( 0  1 -1  0)
( 0  1  1 -2)
Basis Denominator: 2
Mapping from: Lattice of rank 2 and degree 4 to Modular symbols space
of level 11, weight 3, character $.1, and dimension 2 over Rational
Field given by a rule [no inverse]
> Basis(Lattice(EisensteinSubspace(M)));
[
    (   0  1/2 -5/2   -1),
    (   3 -1/2  1/2    0)
]
```

133.6 Degeneracy Maps

Consider an ambient space M_1 of modular symbols of level N_1, and suppose M_2 is an ambient space of modular symbols of level a multiple N_2 of N_1 whose weight equals the weight of M_1 and whose character is induced by the character of M_1. Then for each divisor d of N_2/N_1 there are natural maps $\alpha_d : M_1 \to M_2$ and $\beta_d : M_2 \to M_1$ such that $\beta_d \circ \alpha_d$ is multiplication by $d^{k-2} \cdot [\Gamma_0(N_1) : \Gamma_0(N_2)]$, where k is the common weight of M_1 and M_2. On cuspidal parts, the map β_d is dual to the map $f(q) \to f(q^d)$ on modular forms. Use the function `DegeneracyMap` to compute the maps α_d and β_d.

Given a space M of modular symbols and a positive integer N that is a multiple of the level of M, the images of M under the degeneracy maps generate a modular symbols space of level N. The constructor `ModularSymbols(M,N)` computes this space.

Let M be a space of modular symbols of level N, and let N' be a multiple of N. The subspace

$$\sum_{d \mid \frac{N'}{N}} \alpha_d(M) \subset \mathbf{M_k}(\mathbf{N}', \varepsilon)$$

is stable under the Hecke operators. Here is how to create this subspace using MAGMA:

```
> M := ModularSymbols(11,2); M;
Full modular symbols space for Gamma_0(11) of weight 2 and dimension 3
over Rational Field
> M33 := ModularSymbols(M,33); M33;
Modular symbols space for Gamma_0(33) of weight 2 and dimension 6 over
Rational Field
```

`DegeneracyMap(M1, M2, d)`

The degeneracy map $M_1 \to M_2$ of spaces of modular symbols associated to d. Let N_i be the level of M_i for $i = 1, 2$. Suppose that d is a divisor of either the numerator or denominator of the rational number N_1/N_2, written in reduced form. If $N_1 \mid N_2$, then this intrinsic returns $\alpha_d : M_1 \to M_2$, or if $N_2 \mid N_1$, then this intrinsic returns $\beta_d : M_1 \to M_2$. It is an error if neither divisibility holds.

`DegeneracyMatrix(M1, M2, d)`

Given spaces of modular symbols $M1$ and $M2$ and an integer d, return the matrix of `DegeneracyMap(M1,M2,d)` with respect to `Basis(M1)` and `Basis(M2)`. Both `IsAmbient(M1)` and `IsAmbient(M2)` must be true.

`ModularSymbols(M, N')`

The modular symbols space of level N' associated to M. Let N be the level of M. If $N \mid N'$, then this intrinsic returns the modular symbols space

$$\sum_{d \mid \frac{N'}{N}} \alpha_d(M).$$

If $N' \mid N$, then this intrinsic returns the modular symbols space

$$\sum_{d \mid \frac{N}{N'}} \beta_d(M).$$

In this latter case, if `Conductor(DirichletCharacter(M))` does not divide N', then the 0 space is returned.

`M1 !! M2`

The modular symbols subspace of M_1 associated to M_2. Let N_1 be the level of M_1. If `ModularSymbols(M2,N1)` is defined, let M_3 be this modular symbols space, otherwise terminate with an error. If M_3 is contained in M_1, return M_3, otherwise terminate with an error.

Example H133E9

We compute degeneracy maps α_2 and β_2.

```
> M15 := ModularSymbols(15);
> M30 := ModularSymbols(30);
> alp_2 := DegeneracyMap(M15,M30,2);
> alp_2(M15.1);
2*{oo, 0} + -1*{-1/28, 0} + -1*{-1/2, -7/15}
> beta_2 := DegeneracyMap(M30,M15,2);
> beta_2(alp_2(M15.1));
3*{oo, 0}
> M15.1;
{oo, 0}
```

We can consider the space generated by the image of a space of modular symbols of level 11 in spaces of higher level.

```
> X11 := ModularSymbols("11k2A");
> qEigenform(X11,6);
q - 2*q^2 - q^3 + 2*q^4 + q^5 + O(q^6)
```

```
> ModularSymbols(X11,33);
Modular symbols space for Gamma_0(33) of weight 2 and dimension 4 over
Rational Field
> X33 := ModularSymbols(X11,33);
> qExpansionBasis(X33,6);
[
    q - 2*q^2 + 2*q^4 + q^5 + O(q^6),
    q^3 + O(q^6)
]
> Factorization(CharacteristicPolynomial(HeckeOperator(X33,3)));
[
    <x^2 + x + 3, 2>
]
> ModularDegree(X33);
3
```

We can also construct the space generated by the images of `X11` at higher level using the `!!` operator.

```
> M44 := ModularSymbols(44,2);
> A := M44!!X11; A;
Modular symbols space for Gamma_0(44) of weight 2 and dimension 6 over
Rational Field
> X11!!A;       // back to the original space
Modular symbols space for Gamma_0(11) of weight 2 and dimension 2 over
Rational Field
```

133.7 Decomposition

The functions `Decomposition` and `NewformDecomposition` express a space of modular symbols as a direct sum of Hecke-stable subspaces.

In the intrinsics below, the `Proof` parameter affects the internal characteristic polynomial computations. If `Proof` is set to `false` and this causes a characteristic polynomial computation to fail, then the sum of the dimensions of the spaces returned by `Decomposition` will be less than the dimension of `M`. Thus setting `Proof` equal to `false` is usually safe.

`Decomposition(M, bound : parameters)`

`Proof` BoolElt *Default :* `true`

The decomposition of the space of modular symbols M with respect to the Hecke operators T_p with p coprime to the level of M and $p \leq$ `bound`. If `bound` is too small, the constituents of the decomposition are not guaranteed to be "irreducible", in the sense that they can not be decomposed further into kernels and images of Hecke operators T_p with p prime to the level of M. When `Decomposition` is called, the result is cached, so each successive call results in a possibly more refined decomposition.

Important Note: In some cases `NewformDecomposition` is significantly faster than `Decomposition`.

`NewformDecomposition(M : parameters)`

`Proof` BoolElt *Default :* `true`

Unsorted decomposition of the space of modular symbols M into factors corresponding to the Galois conjugacy classes of newforms of level some divisor of the level of M. We require that `IsCuspidal(M)` is `true`.

`AssociatedNewSpace(M)`

The space of modular symbols corresponding to the Galois-conjugacy class of newforms associated to the space of modular symbols M. The level of the newforms is allowed to be a proper divisor of the level of M. The space M must have been created using `NewformDecomposition`.

`SortDecomposition(D)`

Sort the sequence D of spaces of modular symbols with respect to the `lt` comparison operator.

`IsIrreducible(M)`

Returns `true` if and only if `Decomposition(M)` has cardinality 1.

`M1 lt M2`

The ordering on spaces of modular symbols is determined as follows:

(1) This rule applies only if `NewformDecomposition` was used to construct both of M_1 and M_2: If `Level(AssociatedNewSpace(M1))` is not equal to that of $M2$ then the M_i with larger associated level is first.

(2) The smaller dimension is first.

(3) The following applies when the weight is 2 and the character is trivial: Order by W_q eigenvalues, starting with the *smallest* $p \mid N$, with the eigenvalue $+1$ being less than the eigenvalue -1.

(4) Order by abs(trace(a_p)), with p not dividing the level, and with positive trace being smaller in the event that the two absolute values are equal.

Rule (3) is included so that our ordering extends the one used in (most of!) [Cre97].

Example H133E10

First, we compute the decomposition of the space of modular symbols of weight 2 and level 37.

```
> M := ModularSymbols(37,2); M;
Full modular symbols space for Gamma_0(37) of weight 2 and dimension 5
over Rational Field
> D := Decomposition(M,2); D;
[
     Modular symbols space for Gamma_0(37) of weight 2 and dimension 1
     over Rational Field,
     Modular symbols space for Gamma_0(37) of weight 2 and dimension 2
     over Rational Field,
     Modular symbols space for Gamma_0(37) of weight 2 and dimension 2
     over Rational Field
]
> IsIrreducible(D[2]);
true
> C := CuspidalSubspace(M); C;
Modular symbols space for Gamma_0(37) of weight 2 and dimension 4 over
Rational Field
> N := NewformDecomposition(C); N;
[
     Modular symbols space for Gamma_0(37) of weight 2 and dimension 2
     over Rational Field,
     Modular symbols space for Gamma_0(37) of weight 2 and dimension 2
     over Rational Field
]
```

Next, we use `NewformDecomposition` to decompose a space having plentiful old subspaces.

```
> M := ModularSymbols(90,2); M;
Full modular symbols space for Gamma_0(90) of weight 2 and dimension
37 over Rational Field
> D := Decomposition(M,11); D;
[
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 11
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 4
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 2
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 2
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 4
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 8
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 6
     over Rational Field
```

```
]
> C := CuspidalSubspace(M); C;
Modular symbols space for Gamma_0(90) of weight 2 and dimension 22
over Rational Field
> N := NewformDecomposition(C); N;
[
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 2
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 2
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 2
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 4
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 4
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 8
     over Rational Field
]
```

The above decomposition uses all of the Hecke operator; it suggests that the decomposition `D` is not as fine as possible. Indeed, `D[7]` breaks up further:

```
> Decomposition(D[7],11);
[
    Modular symbols space for Gamma_0(90) of weight 2 and dimension 6
    over Rational Field
]
> Decomposition(D[7],19);
[
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 4
     over Rational Field,
     Modular symbols space for Gamma_0(90) of weight 2 and dimension 2
     over Rational Field
]
```

The function `AssociatedNewSpace` allows us to see where each of these subspace comes from. By definition they each arise by taking images under the degeneracy maps from a single Galois-conjugacy class of newforms of *some* level dividing 90.

```
> [Level(AssociatedNewSpace(A)) : A in N];
[ 90, 90, 90, 45, 30, 15 ]
> A := N[4];
> qEigenform(AssociatedNewSpace(A),7);
q + q^2 - q^4 - q^5 + O(q^7)
> qExpansionBasis(A,7);
[
    q - 2*q^4 - q^5 + O(q^7),
    q^2 + q^4 + O(q^7)
```

```
]
```

133.8 Subspaces

The following functions compute the cuspidal, Eisenstein, and new subspaces, along with the complement of a subspace.

`CuspidalSubspace(M)`

The cuspidal subspace of the space of modular symbols M. This is the kernel of `BoundaryMap(M)`.

`IsCuspidal(M)`

Returns `true` if and only if the space of modular symbols M is contained in the cuspidal subspace of the ambient space.

`EisensteinSubspace(M)`

The Eisenstein subspace of the space of modular symbols M. This is the complement in M of the cuspidal subspace of M.

`IsEisenstein(M)`

Returns `true` if and only if the space of modular symbols M is contained in the Eisenstein subspace of the ambient space.

`NewSubspace(M)`

The new subspace of the space of modular symbols M. This is the intersection of `NewSubspace(M,p)` as p varies over all prime divisors of the level of M. Note that M is required to be cuspidal.

`IsNew(M)`

Returns `true` if and only if the space of modular symbols M is contained in the new cuspidal subspace of the ambient space.

`NewSubspace(M, p)`

The p-new subspace of the space of modular symbols M. This is the kernel of the degeneracy map from M to the space of modular symbols of level equal to the level of M divided by p and character the restriction of the character of M. If the character of M does not restrict, then `NewSubspace(M,p)` is equal to M. Note that M is required to be cuspidal.

`Kernel(I, M)`

The kernel of I on the space of modular symbols M. Let T_p denote the pth Hecke operator (see Section 136.14). This is the subspace of M obtained by intersecting the kernels of the operators $f_n(T_{p_n})$, where I is a sequence $[\langle p_1, f_1(x)\rangle, ..., \langle p_n, f_n(x)\rangle]$ of pairs consisting of a prime number and a polynomial. Only primes p_i which do not divide the level of M are used.

`Complement(M)`

The space of modular symbols complementary to the space of modular symbols M in the ambient space of M. Thus the ambient space of M is equal to the direct sum of M and `Complement(M)`.

`BoundaryMap(M)`

A matrix that represents the boundary map from the space of modular symbols M to the vector space whose basis consists of the weight k cusps. (Note: At present there is no intrinsic that lists these cusps.)

Example H133E11

First we compute the cuspidal subspace of the space of modular symbols for $\Gamma_0(11)$.

```
> M := ModularSymbols(11,2); M;
Full modular symbols space for Gamma_0(11) of weight 2 and dimension 3
over Rational Field
> IsCuspidal(M);
false
> C := CuspidalSubspace(M); C;
Modular symbols space for Gamma_0(11) of weight 2 and dimension 2 over
Rational Field
> IsCuspidal(C);
true
```

Next we compute the Eisenstein subspace.

```
> IsEisenstein(C);
false
> E := EisensteinSubspace(M); E;
Modular symbols space for Gamma_0(11) of weight 2 and dimension 1 over
Rational Field
> IsEisenstein(E);
true
> E + C eq M;
true
```

The Eisenstein subspace is the complement of the cuspidal subspace, and conversely.

```
> E eq Complement(C);
true
> C eq Complement(E);
```

```
true
```

Example H133E12

```
> M := ModularSymbols("37B"); M;
Modular symbols space for Gamma_0(37) of weight 2 and dimension 2 over
Rational Field
> BoundaryMap(M);
[0 0]
[0 0]
> A := AmbientSpace(M);
> BoundaryMap(A);
[ 0  0]
[ 0  0]
[ 0  0]
[ 0  0]
[ 1 -1]
```

Observe that the Eisenstein subspace of A is not in the kernel of the boundary map.

```
> Basis(VectorSpace(EisensteinSubspace(A)));
[
    (0 0 0 1 3)
]
```

133.9 Twists

This section is about twists of newforms by Dirichlet characters. A newform is specified by giving a space of modular symbols that contains a single Galois-orbit (over $\mathbf{Q}$ or some extension of $\mathbf{Q}$) of newforms. Spaces of this kind are obtained using `NewformDecomposition`.

To prove that $f_2 = f_1^\chi$ holds for two newforms, the program compares their Hecke eigenvalues up to an appropriate Sturm bound. (For instance, a bound of this kind is given in Lemma 1.4 of [BS02]).

`IsTwist(M1, M2, p)`

Given two spaces `M1` and `M2` of modular symbols that specify newforms f_1 and f_2 as above, and a prime p, this determines whether some Galois conjugate (over $\mathbf{Q}$) of f_2 is the twist of f_1 by a nontrivial Dirichlet character of p-power conductor. If so, a character χ such that $f_2 = f_1^\chi$ is also returned.

`IsMinimalTwist(M, p)`

Given a space `M` of modular symbols that specifies a newform f as above, and a prime p, this determines whether some Galois conjugate (over $\mathbf{Q}$) of f is a twist of some newform of lower level by some Dirichlet character of p-power conductor. If so, it returns `false`, together with the newform of lower level (specified by a space of modular symbols), and the Dirichlet character.

Example H133E13

We exhibit a newform that is a twist of itself, namely the only newform of level 9 and weight 4. The newform is specified by the space of modular symbols on $\Gamma_0(9)$ of weight 4 (with sign 1).

```
> M9 := CuspidalSubspace(ModularSymbols(9, 4, 1));
> newforms := NewformDecomposition(NewSubspace(M9));
> newforms;
[
    Modular symbols space for Gamma_0(9) of weight 4 and dimension 1
    over Rational Field
]
> f := newforms[1];
> Eigenform(f, 20);
q - 8*q^4 + 20*q^7 - 70*q^13 + 64*q^16 + 56*q^19 + O(q^20)
```

Note that here the coefficients for primes congruent to 2 mod 3 are all zero.

```
> bool, chi := IsTwist(f, f, 3);
> bool;
true
> Parent(chi);
Group of Dirichlet characters of modulus 3 over Rational Field
> Conductor(chi), Order(chi);
3 2
```

However, f is not a twist of any newform with lower level:

```
> bool := IsMinimalTwist(f, 3);
> bool;
true
```

133.10 Operators

Each space $\mathbf{M}$ of modular symbols comes equipped with a commuting family $T_1, T_2, T_3, \ldots$ of linear operators acting on it called the Hecke operators.

The Hecke operators are defined recursively, as follows. First, $T_1 = 1$. When $n = p$ is prime,

$$T_p(x) = \left[\begin{pmatrix} p & 0 \\ 0 & 1 \end{pmatrix} + \sum_{r \bmod p} \begin{pmatrix} 1 & r \\ 0 & p \end{pmatrix} \right] x,$$

where the first matrix is omitted if p divides the level N of M. If m and n are coprime, then $T_{mn} = T_m T_n$. If p is a prime, $r \geq 2$ is an integer, ε is the Dirichlet character associated to M, and k is the weight of M, then

$$T_{p^r} = T_p T_{p^{r-1}} - \varepsilon(p) p^{k-1} T_{p^{r-2}}.$$

Example H133E14

In MAGMA, Hecke operators are represented as $n \times n$-matrices, acting from the right, with respect to the basis `Basis(M)`. For example

```
> M := ModularSymbols(12);
> T2 := HeckeOperator(M,2);
> M.1;
{oo, 0}
> T2;
[ 2  0 -1  0  0]
[ 2  0 -1  0  0]
[ 0  0  1 -2 -2]
[ 0 -1  1 -1 -2]
[ 0  1 -1  1  2]
> M.1*T2;
2*{oo, 0} + -1*{-1/10, 0}
```

`HeckeOperator(M, n)`

Compute a matrix representing the nth Hecke operator T_n with respect to `Basis(M)` where M is a space of modular symbols.

`HeckePolynomial(M, n)`

Compute the characteristic polynomial of the Hecke operator T_n with respect to the space of modular symbols M. When n is prime, the Deligne bound on the sizes of Hecke eigenvalues is used, so `HeckePolynomial` is frequently much faster than `CharacteristicPolynomial(HeckeOperator(M,n))`.

`IntegralHeckeOperator(M, n)`

A matrix representing the nth Hecke operator with respect to `Basis(Lattice(M))` where M is a space of modular symbols.

`DualHeckeOperator(M, n)`

Compute a matrix representing the Hecke operator T_n on the dual vector space representation of the space of modular symbols M. This function is much more efficient than `HeckeOperator(M,n)` when the dimension of M is small relative to the dimension of the `AmbientSpace(M)`. Note that `DualHeckeOperator(M,n)` is not guaranteed to be the transpose of `HeckeOperator(M,n)` because `DualHeckeOperator(M,n)` is computed with respect to `Basis(DualVectorSpace(M))`.

`AtkinLehner(M, q)`

A matrix representing the qth Atkin-Lehner involution W_q on the space of modular symbols M, when it is defined. The involution W_q is defined when M has trivial character and even weight. When possible, the Atkin-Lehner map is normalized so that it is an involution; such normalization may not be possible when $k > 2$ and the characteristic of the base field of M divides q.

To each divisor q of N such that $\gcd(q, N/q) = 1$ there is an *Atkin-Lehner involution* W_q on M, which is defined as follows. Using the Euclidean algorithm, choose integers x, y, z, w such that $qxw - (N/q)yz = 1$; let $g = \begin{pmatrix} dx & y \\ Nz & qw \end{pmatrix}$ and define

$$W_q(x) = g(x)/q^{\frac{k-2}{2}}.$$

For example, when $q = N$ we have $g = \begin{pmatrix} 0 & -1 \\ N & 0 \end{pmatrix}$.

`DualAtkinLehner(M, q)`

The action of the Atkin-Lehner involution on the dual representation of the space of modular symbols M, when it is defined.

`StarInvolution(M)`

The conjugation involution $*$ on the space of modular symbols M that sends the modular symbol $X^iY^j\{u, v\}$ to $(-1)^j X^iY^j\{-u, -v\}$.

`DualStarInvolution(M)`

The conjugation involution $*$ on the dual representation of the space of modular symbols M (see the documentation for `StarInvolution`.)

`ThetaOperator(M1, M2)`

Multiplication by $X^pY - XY^p$, which is a possible analogue of the θ-operator. (On mod p modular forms, the θ-operator is the map given by $f \mapsto q\frac{df}{dq}$.) Both M_1 and M_2 must be spaces of modular symbols over a field of positive characteristic p; they must have the same level and character, and the weight of M_2 must equal the weight of M_1 plus $p + 1$.

Example H133E15

```
> M := ModularSymbols(11,4,+1); M;
Full modular symbols space for Gamma_0(11) of weight 4 and dimension 4
over Rational Field
> HeckeOperator(M,2);
[   9    0  2/5 -2/5]
[   0    5  9/5 11/5]
[   0    5  7/5 13/5]
```

```
[   0    0 22/5 23/5]
```

The entries of T_2 are not guaranteed to be integers because `Basis(M)` is just a basis of a $\mathbf{Q}$-vector space. The entries will be integers if we compute T_2 with respect to an integral basis.

```
> IntegralHeckeOperator(M,2);
[ 0  2  0  0]
[ 1  2  0  0]
[-5  6  9  0]
[ 2  0  0  9]
```

The matrix for the Hecke operator on the dual of M is the transpose of T_2. However, the chosen basis for the cuspidal subspace of the dual of M need not satisfy any compatibility with `CuspidalSubspace(M)`.

```
> DualHeckeOperator(M,2);
[   9    0    0    0]
[   0    5    5    0]
[ 2/5  9/5  7/5 22/5]
[-2/5 11/5 13/5 23/5]
> S := CuspidalSubspace(M);
> HeckeOperator(S, 2);
[    5 -13/5]
[    5    -3]
> DualHeckeOperator(S, 2);
[-3/4  1/8]
[-1/2 11/4]
> // NOT the transpose!
```

We can also compute the Atkin-Lehner and the $*$-involution. The $*$-involution is the identity because we are working in the $+1$-quotient, which is the largest quotient of `ModularSymbols(11,4)` where $*$ acts as $+1$.

```
> AtkinLehner(S, 11);
[1 0]
[0 1]
> StarInvolution(S);
[1 0]
[0 1]
```

On the -1 quotient the Atkin-Lehner involution is the same, but $*$ acts as -1:

```
> M := ModularSymbols(11,4,-1); M;
Full modular symbols space for Gamma_0(11) of weight 4 and dimension 2
over Rational Field
> S := CuspidalSubspace(M);
> AtkinLehner(S, 11);
[1 0]
[0 1]
> StarInvolution(S);
[-1  0]
```

```
[ 0 -1]
```

Example H133E16

We compute an example of our analogue of the θ-operator on modular symbols.

```
> N := 11; p := 3;
> k1 := 2; k2 := k1 + (p+1);
> M1 := ModularSymbols(11,k1,GF(p));
> M2 := ModularSymbols(11,k2,GF(p));
> theta  := ThetaOperator(M1,M2); theta;
Mapping from: ModSym: M1 to ModSym: M2 given by a rule [no inverse]
```

Now that we have computed `theta`, we can apply it to one of the modular symbols corresponding to the newform in $S_2(\Gamma_0(11))$.

```
> D := Decomposition(M1,2);
> f := qEigenform(D[2],10); f;
q + q^2 + 2*q^3 + 2*q^4 + q^5 + 2*q^6 + q^7 + q^9 + O(q^10)
> x := D[2].1;
> y := theta(x); y;
(X^4 + X*Y^3)*{-1/7, 0} + (X^4 + X^3*Y + X*Y^3 + Y^4)*{-1/7, 0} + (X^4
+ 2*X^3*Y + 2*X*Y^3 + Y^4)*{-1/5, 0} + Y^4*{oo, 0}
```

Finally, we verify for $n < 10$ that the nth Hecke eigenvalue of $y = \theta(x)$ equals $n \cdot a_n(f)$, where f is as above.

```
> [y*HeckeOperator(M2,n) - n*Coefficient(f,n)*y : n in [1..9]];
[
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0
]
```

133.11 The Hecke Algebra

`HeckeBound(M)`

A positive integer n such that the Hecke operators $T_1, \ldots, T_n$ generate the Hecke algebra as a $\mathbf{Z}$-module. When the character is trivial, the default bound is $(k/12) \cdot [\mathrm{SL}_2(\mathbf{Z}) : \Gamma_0(N)]$. That this suffices follows from [Stu87], as is explained in [AS02]. When the character of the space of modular symbols M is nontrivial, the default bound is twice the above bound; however, *it is not known that this bound is large enough in all cases in which the character is nontrivial*, so one may wish to increase the bound using `SetHeckeBound`.

`SetHeckeBound(M, n)`

Many computations require a bound n such that $T_1, \ldots, T_n$ generate the Hecke algebra associated to the space of modular symbols M as a $\mathbf{Z}$-module. This command allows you to set the bound that is used internally. Setting it too low can result in functions quickly producing incorrect results.

`HeckeAlgebra(M : Bound)`

The Hecke algebra associated to the space of modular symbols M. This is an algebra `TQ` over $\mathbf{Q}$, such that `Generators(TQ)` is a set that generates the ring $\mathbf{Z}[T_1, T_2, T_3, \ldots]$, as a $\mathbf{Z}$-module. If the optional integer parameter `Bound` is set, then `HeckeAlgebra` only computes the algebra generated by those T_n, with $n \leq$ `Bound`.

`DiscriminantOfHeckeAlgebra(M : Bound)`

The discriminant of the Hecke algebra associated to the space of modular symbols M. If the optional parameter `Bound` is set, then the discriminant of the algebra generated by only those T_n, with $n \leq$ `Bound`, is computed instead.

`HeckeEigenvalueRing(M : parameters)`

`Bound` RngIntElt *Default :* -1

The order generated by the Fourier coefficients of one of the q-expansions of a newform corresponding to the space of modular symbols M, along with a map from the ring containing the coefficients of `qExpansion(A)` to the order. If the optional parameter `Bound` is set, then the order generated only by those a_n, with $n \leq$ `Bound`, is computed.

`HeckeEigenvalueField(M)`

The number field generated by the Fourier coefficients of one of the q-expansions of a newform corresponding to the space of modular symbols M, along with a map from the ring containing the coefficients of `qExpansion(M)` to the number field. We require that M be defined over $\mathbf{Q}$.

Example H133E17

In this example, we compute the discriminant of the Hecke algebra of prime level 389.

```
> M := ModularSymbols(389,2,+1);
> C := CuspidalSubspace(M);
> DiscriminantOfHeckeAlgebra(C);
629670054720061882880174736321392595498204931550235108311040000000
> Factorization($1);
[ <2, 53>, <3, 4>, <5, 6>, <31, 2>, <37, 1>, <389, 1>, <3881, 1>,
<215517113148241, 1>, <477439237737571441, 1> ]
```

The prime 389 is the only prime $p < 10000$ such that p divides the discriminant of the Hecke algebra associated to $S_2(\Gamma_0(p))$. It is an open problem to decide whether or not there are any other such primes. Are there infinitely many?

133.12 The Intersection Pairing

MAGMA can compute the intersection pairing

$$H_1(X_0(N), \mathbf{Q}) \times H_1(X_0(N), \mathbf{Q}) \to \mathbf{Q}$$

on the homology of the modular curve $X_0(N)$. The algorithm that we implemented is essentially the one given in [Mer93]. (Warning: There is a typo in Proposition 4 of [Mer93]; W_i should be replaced by $W_i^{\varepsilon_i}$.)

`IntersectionPairing(x, y)`

The intersection pairing of the homology classes corresponding to the weight-2 cuspidal modular symbols x and y. The symbols x and y must have the same parent, which must have trivial character and not be a $+1$ or -1 quotient.

Example H133E18

In this example, we illustrate several basic properties of the intersection pairing on $H_1(X_0(37), \mathbf{Z})$. First, let `H37` be the space of modular symbols that corresponds to $H_1(X_0(37), \mathbf{Z})$, and compute a basis for `H37`.

```
> M37 := ModularSymbols(37,2);
> H37 := CuspidalSubspace(M37);
> Z := IntegralBasis(H37); Z;
[
    {-1/29, 0},
    {-1/22, 0},
    {-1/12, 0},
    {-1/18, 0}
]
```

Now we compute some intersection numbers.

```
> IntersectionPairing(Z[1],Z[2]);
```

```
-1
> IntersectionPairing(Z[3],Z[4]);
0
```

The intersection pairing is perfect and skew-symmetric, so the matrix that defines it is skew-symmetric and has determinant ± 1 (in fact, it has determinant $+1$).

```
> A := MatrixAlgebra(RationalField(),4);
> I := A![IntersectionPairing(x,y) : x in Z, y in Z]; I;
[ 0  1  0  1]
[-1  0  1  1]
[ 0 -1  0  0]
[-1 -1  0  0]
> I + Transpose(I) eq 0;
true
> Determinant(I);
1
```

The Hecke operators are compatible with the intersection pairing in the sense that $(T_n x, y) = (x, T_n y)$.

```
> T2 := HeckeOperator(M37,2);
> IntersectionPairing(Z[1]*T2,Z[2]);
1
> IntersectionPairing(Z[1],Z[2]*T2);
1
```

It is note the case $(T_n x, T_n y) = (x, y)$ for all n, x, and y.

```
> IntersectionPairing(Z[1]*T2,Z[2]*T2);
-2
```

The existence of the intersection pairing implies that $H_1(X_0(N), \mathbf{Z})$ is isomorphic, as a module over the Hecke algebra, to its linear dual $\mathrm{Hom}(H_1(X_0(N), \mathbf{Z}), \mathbf{Z})$.

133.13 q-Expansions

The following functions should only be called on modular symbols spaces that are cuspidal. For q-expansions of Eisenstein series, use the modular forms functions instead (see the example below).

`Eigenform(M, prec)`

`Eigenform(M)`

`qEigenform(M, prec)`

`qEigenform(M)`

`PowerSeries(M, prec)`

`PowerSeries(M)`

The q-expansion of one of the Galois-conjugate newforms associated to the irreducible cuspidal space M of modular symbols, computed to absolute precision *prec* (which defaults to the highest precision computed in previous calls to this intrinsic, or 8 if none have been computed). The coefficients of the q-expansion lie in a quotient of a polynomial extension of the base field of M. In most cases, it is necessary for M to have been defined using `NewformDecomposition`.

`qExpansionBasis(M, prec : parameters)`

`Al` MonStgElt *Default :* "$Newform$"

The reduced row-echelon basis of q-expansions for the space of modular forms associated to M, where K is the base field of M. The absolute precision of the q-expansions is *prec*.

The optional parameter `Al` can take the values `"Newform"` and `"Universal"`. The default is `"Newform"`, which computes a basis of q-expansions by finding a decomposition of M into subspaces corresponding to newforms, computing their q-expansions, and then taking all of their images under the degeneracy maps. If `Al := "Universal"` then the algorithm of Section 4.3 of [Mer94] is used. This latter algorithm does not require computing a newform decomposition of M, but requires computing the action of many more Hecke operators. Consequently, in practice, our implementation of Merel's algorithm is usually less efficient than our implementation of the newform algorithm.

`qIntegralBasis(M)`

`qIntegralBasis(M, prec)`

`qIntegralBasis(seq, prec)`

`Al` MonStgElt *Default :* "$Newform$"

The reduced integral basis of q-expansions for the space of modular forms associated to the given space M of modular symbols (or the given sequence of spaces). The q-expansions are computed to absolute precision *prec*. The base field of M must be either the rationals or a cyclotomic field.

`SystemOfEigenvalues(M, prec)`

The sequence of Hecke eigenvalues $[a_2, a_3, a_5, a_7, \ldots, a_p]$ attached to the space of modular symbols M, where p is the largest prime less than or equal to *prec*. Let K be the base field of M. Then the a_ℓ either lie in K or an extension of K (which may be constructed either as a number field or as a quotient of $K[x]$). We assume that M corresponds to a single Galois-conjugacy class of newforms.

Example H133E19

First we compute a q-basis and a representative newform for the two-dimensional space $S_2(\Gamma_0(23))$. We work in the $+1$ quotient of modular symbols since, for the purpose of computing q-expansions, nothing is lost and many algorithms are more efficient.

```
> M := CuspidalSubspace(ModularSymbols(23,2, +1));
> qExpansionBasis(M);
[
    q - q^3 - q^4 - 2*q^6 + 2*q^7 + O(q^8),
    q^2 - 2*q^3 - q^4 + 2*q^5 + q^6 + 2*q^7 + O(q^8)
]
> f := qEigenform(M,6); f;
q + a*q^2 + (-2*a - 1)*q^3 + (-a - 1)*q^4 + 2*a*q^5 + O(q^6)
> Parent(f);
Power series ring in q over Univariate Quotient Polynomial Algebra
in a over Rational Field with modulus a^2 + a - 1
> PowerSeries(M);
q + a*q^2 + (-2*a - 1)*q^3 + (-a - 1)*q^4 + 2*a*q^5 + (a - 2)*q^6 +
    (2*a + 2)*q^7 + O(q^8)
> SystemOfEigenvalues(M, 7);
[
    a,
    -2*a - 1,
    2*a,
    2*a + 2
]
```

Next we compare an integral and rational basis of q-expansions for $S_2(\Gamma_0(65))$, computed using modular symbols.

```
> S := CuspidalSubspace(ModularSymbols(65,2,+1));
> qExpansionBasis(S);
[
    q + 1/3*q^6 + 1/3*q^7 + O(q^8),
    q^2 - 1/3*q^6 + 2/3*q^7 + O(q^8),
    q^3 - 4/3*q^6 + 2/3*q^7 + O(q^8),
    q^4 - 1/3*q^6 + 5/3*q^7 + O(q^8),
    q^5 + 5/3*q^6 + 2/3*q^7 + O(q^8)
]
> qIntegralBasis(S);
[
    q + q^5 + 2*q^6 + q^7 + O(q^8),
    q^2 + 2*q^5 + 3*q^6 + 2*q^7 + O(q^8),
    q^3 + 2*q^5 + 2*q^6 + 2*q^7 + O(q^8),
    q^4 + 2*q^5 + 3*q^6 + 3*q^7 + O(q^8),
    3*q^5 + 5*q^6 + 2*q^7 + O(q^8)
```

```
]
```

If you're interested in q-expansions of Eisenstein series, see the chapter on modular forms. For example:

```
> E := EisensteinSubspace(ModularForms(65,2));
> Basis(E);
[
    1 + O(q^8),
    q + 3*q^2 + 4*q^3 + 7*q^4 + 12*q^6 + 8*q^7 + O(q^8),
    q^5 + O(q^8)
]
```

133.14 Special Values of L-functions

Let M be an irreducible space of cuspidal modular symbols defined over $\mathbf{Q}$, irreducible in the sense that M corresponds to a single Galois-conjugacy class of cuspidal newforms. Such an M can be computed using `NewformDecomposition`. Let $f^{(1)}, \ldots, f^{(d)}$ be the $\mathrm{Gal}(\overline{\mathbf{Q}}/\mathbf{Q})$-conjugate newforms that correspond to M, and write $f^{(d)} = \sum_{n=1}^{\infty} a_n^{(d)} q^n$. By a theorem of Hecke, the Dirichlet series

$$L(f^{(i)}, s) = \sum_{n=1}^{\infty} \frac{a_n^{(i)}}{n^s}$$

extends (uniquely) to a holomorphic function on the whole complex plane. Of particular interest is the special value

$$L(M, j) = L(f^{(1)}, j) \cdots L(f^{(d)}, j),$$

for any $j \in \{1, 2, \ldots, k-1\}$.

In this section we describe how to approximate the complex numbers $L(M, j)$ in Magma. If you are interested in computing individual special values $L(f^{(i)}, j)$, then you should use the modular forms package instead of the modular symbols package for this.

The variable `prec` below refers to the number of terms of the q-expansion of each $f^{(i)}$ that are used in the computation, and not to the number of decimals of the answer that are correct. Thus, for example, to get a heuristic idea of the quality of an answer, you can increase `prec`, make another call to `LSeries`, and observe the difference between the two answers. If the difference is "small", then the approximation is probably "good".

`LSeries(M, j, prec)`

The special value $L(M, j)$, where j is an integer that lies in the critical strip, so $1 \leq j \leq k-1$ with k the weight of M. Here M is a space of modular symbols with sign 0, and *prec* is a positive integer which specifies the numbers of terms of q-expansions to use in the computation.

`LSeriesLeadingCoefficient(M, j, prec)`

The leading coefficient of Taylor expansion about the critical integer j and order of vanishing of $L(M,s)$ at $s=1$. Thus if the series expansion of $L(M,s)$ about $s=1$ is

$$L(M,s) = a_r(s-1)^r + a_{r+1}(s-1)^{r+1} + a_{r+2}(s-1)^{r+2} + \cdots,$$

then the leading coefficient of $L(M,s)$ is a_r and the order of vanishing is r.

`RealVolume(M, prec)`

The volume of $A_M(\mathbf{R})$, which is defined as follows. Let $S \subset \mathbf{C}[[q]]$ be the space of cusp forms associated to M. Choose a basis $f_1, \ldots, f_d$ for the free $\mathbf{Z}$-module $S \cap \mathbf{Z}[[q]]$; one can prove that $f_1, \ldots, f_d$ is also a basis for S. There is a period map Φ from integral cuspidal modular symbols H to $\mathbf{C}^d$ that sends a modular symbol $x \in H$ to the d-tuple of integrals $(\langle f_1, x\rangle, \ldots, \langle f_d, x\rangle) \in \mathbf{C}^d$. The cokernel of Φ is isomorphic to $A_M(\mathbf{C})$. Moreover, the standard measure on the Euclidean space $\mathbf{C}^d$ induces a measure on $A_M(\mathbf{R})$. It is with respect to this measure that we compute the volume. For more details, see Section 3.12.16 of [Ste00].

`MinusVolume(M, prec)`

The volume of the subgroup of $A_M(\mathbf{C})$ on which complex conjugation acts as -1.

`LRatio(M, j : parameters)`

`Bound` RngIntElt *Default :* -1

The rational number

$$\frac{L(A,j)\cdot(j-1)!}{(2\pi)^{j-1}\cdot\Omega,}$$

where j is a "critical integer", so $1 \le j \le k-1$, and Ω is `RealVolume(M)` when j is odd and `MinusVolume(M)` when j is even. If the optional parameter Bound is set, then `LRatio` is only a divisibility upper bound on the above rational number. If `Sign(M)` is not 0, then `LRatio(M,j)` is only correct up to a power of 2.

`LRatioOddPart(M, j)`

The odd part of the rational number `LRatio(M,j)`. Hopefully, computing `LRatioOddPart(M,j)` takes less time than finding the odd part of `LRatio(M,j)`.

Example H133E20

```
> M := ModularSymbols(11,2);
> C := CuspidalSubspace(M);
> LSeries(C,1,100);
0.25384186085591068433775892 33
> A := ModularSymbols("65B"); A;  // <--> dimension two abelian variety
Modular symbols space of level 65, weight 2, and dimension 4
> LSeries(A,1,100);
0.91225158869818984109351402 11 +  0.E-29*i
```

133.14.1 Winding Elements

Let $\mathbf{M_k(N)}$ be a space of modular symbols over $\mathbf{Q}$. For $i = 1, \ldots, k$, the *ith winding element*

$$\mathbf{e}_i = X^{i-1}Y^{k-2-(i-1)}\{0, \infty\} \in \mathbf{M_k(N)}$$

is of importance for the computation of special values. For any modular form $f \in S_k(N)$ and homogeneous polynomial $P(X, Y)$ of degree $k-2$, let

$$\langle f, P(X,Y)\{0,\infty\}\rangle = -2\pi i \cdot \int_0^{i\infty} f(z)P(z,1)\mathrm{dz}.$$

Fix a newform $f \in S_k(N)$ corresponding to a space M of modular symbols, and let j be a integer in $\{0, 1, \ldots, k-1\}$. The winding element is significant because

$$L(f, j) = \frac{(2\pi)^{j-1}}{i^{j+1}(j-1)!} \cdot \langle f, X^{j-1}Y^{k-2-(j-1)}\{0,\infty\}\rangle.$$

Moreover, the submodule that is generated by the winding element is used in the formula for a canonical rational part of the number $L(M, j)$ (see `LRatio`, above).

`WindingElement(M)`

The winding element $Y^{k-2}\{0, \infty\}$.

`WindingElement(M, i)`

The winding element $X^{i-1}Y^{k-2-(i-1)}\{0, \infty\}$.

`TwistedWindingElement(M, i, eps)`

The element $\sum_{a\in(\mathbf{Z}/m\mathbf{Z})^*} \varepsilon(a)X^{i-1}Y^{k-2-(i-1)}\{0, \frac{a}{m}\}$.

`WindingLattice(M, j :` *parameters*`)`

`Bound` RngIntElt *Default :* -1

The image under `RationalMapping(M)` of the lattice generated by the images of the jth winding element under all Hecke operators T_n. If M is the ambient space, then the image under `RationalMapping(M)` is not taken.

`WindingSubmodule(M, j :` *parameters*`)`

`Bound` RngIntElt *Default :* -1

The image under `RationalMapping(M)` of the vector space generated by all images of `WindingElement(M,j)` under all Hecke operators T_n. If M is the ambient space, then the image under the rational period mapping is not taken.

`TwistedWindingSubmodule(M, j, eps)`

The Hecke submodule of the vector space $\Phi(M)$ generated by the image of the jth ε-twisted modular winding element, where Φ is `RationalMapping(M)`. This module is useful, for example, because in characteristic 0, if M is new of weight 2, has sign $+1$ or -1, and corresponds to a collection $\{f_i\}$ of Galois-conjugate newforms, then the dimension of the twisted winding submodule equals the cardinality of the subset of f_i such that $L(f_i, eps, 1) \neq 0$.

133.15 The Associated Complex Torus

Let M be a space of cuspidal modular symbols, which is the kernel of an ideal in the Hecke algebra. When M has weight 2 there is an abelian variety A_M attached to M; more generally, there is a complex torus $A_M(\mathbf{C})$ attached to M. The associated complex torus $A_M(\mathbf{C})$ is constructed as follows. Let S be the space of modular forms corresponding to M. The integration pairing gives rise to a natural map $M \to \mathrm{Hom}(S, \mathbf{C})$, and the cokernel of this map is $A_M(\mathbf{C})$.

`SubgroupOfTorus(M, x)`

The cyclic subgroup of the complex torus attached to the space of modular symbols M that is generated by the image under the period map of the modular symbol x.

`SubgroupOfTorus(M, s)`

An abelian group that is isomorphic to the finite group generated by the sequence of images $\pi(s[i])$ in the complex torus attached to M, where π is `PeriodMapping(M)`.

Example H133E21

The cuspidal subgroup of $J_0(N)$ is the subgroup generated by the degree 0 divisors on $X_0(N)$ of the form $(\alpha) - (\beta)$, where α and β are cusps. The following examples illustrate how to use the above functions to compute the cuspidal subgroup, as an abstract abelian group.
The modular symbols approach has the advantage that it is essentially no more complicated for N highly composite than for N prime. However, it is only applicable when the corresponding space of modular symbols can be computed in a reasonable amount of time, which at present means that N should have less than 5 decimal digits. There are other methods which may be much more efficient in special cases. For example, when p is prime Andrew Ogg showed that the cuspidal subgroup of $J_0(p)$ is cyclic of order equal to the numerator of $(p-1)/12$. More generally, he gave a simple formula for the order of the cuspidal subgroup when $N = pq$ is the product of two primes. See also papers of Ligozat.

```
> M := ModularSymbols(20); M;
Full Modular symbols space of level 20, weight 2, and dimension 7
> e := M ! <1, [Cusps()|0,Infinity()] >;   // the path from 0 to infinity
> e;
-1*{oo, 0}
> J0of20 := CuspidalSubspace(M);
```

```
> A := SubgroupOfTorus(J0of20, e); A;
Abelian Group isomorphic to Z/6
Defined on 1 generator
Relations:
    6*A.1 = 0
> // Next, the subgroup generated by all cusps
> A := SubgroupOfTorus(J0of20, IntegralBasis(M));  A;
Abelian Group isomorphic to Z/6
Defined on 1 generator
Relations:
    6*A.1 = 0
> // Let's do another example.
> M := ModularSymbols(100);
> J0of100 := CuspidalSubspace(M);
> A := SubgroupOfTorus(J0of100, IntegralBasis(M));  A;
Abelian Group isomorphic to Z/6 + Z/30 + Z/30 + Z/30 + Z/30
Defined on 5 generators
Relations:
    6*A.1 = 0
    30*A.2 = 0
    30*A.3 = 0
    30*A.4 = 0
    30*A.5 = 0
> M := ModularSymbols(77);
> J0of77 := CuspidalSubspace(M);
> A := SubgroupOfTorus(J0of77, IntegralBasis(M)); A;
Abelian Group isomorphic to Z/10 + Z/60
Defined on 2 generators
Relations:
    10*A.1 = 0
    60*A.2 = 0
> M := ModularSymbols(97);
> A := SubgroupOfTorus(CuspidalSubspace(M), IntegralBasis(M)); A;
Abelian Group isomorphic to Z/8
Defined on 1 generator
Relations:
    8*A.1 = 0
> Numerator((97-1)/12);
8
```

Example H133E22

The following code creates a file that contains a table which lists, for each integer N in some range, the abstract group structure of the subgroup of $J_0(N) = \mathrm{Jac}(X_0(N))$ generated by the cusps $(\alpha) - (\infty)$, with $\alpha \in \mathbf{Q} \cup \{\infty\}$.

```
> function CuspidalSubgroup(N)
>    M := ModularSymbols(N);
```

```
>    J := CuspidalSubspace(M);
>    G := SubgroupOfTorus(J,IntegralBasis(M));
>    return G;
> end function;
> // Test the function
> CuspidalSubgroup(65);
Abelian Group isomorphic to Z/2 + Z/84
Defined on 2 generators
Relations:
    2*$.1 = 0
    84*$.2 = 0
> procedure CuspidalTable(start, stop)
>    fname := Sprintf("cuspidal_subgroup_%o-%o.m", start, stop);
>    file  := Open(fname,"w");
>       for N in [start..stop] do
>       G := Invariants(CuspidalSubgroup(N));
>       fprintf file, "C[%o] := \t%o;\n\n", N, G;
>       printf "C[%o] := \t%o;\n\n", N, G;
>       Flush(file);
>    end for;
> end procedure;
```

`ModularKernel(M)`

The kernel of the modular isogeny. Let T be the complex torus attached to the space of modular symbols M. Then the modular isogeny is the natural map from the dual of T into T induced by autoduality of `CuspidalSubspace(AmbientSpace(M))`.

`CongruenceGroup(M` : *parameters*`)`

`Bound` RngIntElt *Default :* -1

The congruence group of the space of cusp forms corresponding to the space of cuspidal modular symbols M. Let $S = S_k(\Gamma_0(N), \mathbf{Z})$, let V be the sub $\mathbf{Z}$-module corresponding to M, and W be its orthogonal complement. Then the congruence group is $S/(V + W)$. This group encodes information about congruences between forms in V and forms in the complement of V.

The optional parameter `Bound` is a positive integer b such that the q-expansions of cusp forms are computed to absolute precision b. If the bound is too small, then `CongruenceGroup` will give only an upper bound on the correct answer. The default is `HeckeBound(M) + 1`, which gives a provably correct answer.

`IntersectionGroup(M1, M2)`

An abelian group G that encodes information about the intersection of the complex tori corresponding to the spaces of modular symbols M_1 and M_2. We require that M_1 and M_2 lie in a common ambient space. When the `IntersectionGroup(M1,M2)` is finite, it is isomorphic to $A_{M_1}(\mathbf{C}) \cap A_{M_2}(\mathbf{C})$.

`IntersectionGroup(S)`

An abelian group G that encodes information about the intersection of the collection of complex tori corresponding to the sequence S of spaces of modular symbols.

Example H133E23

In this example, we investigate a 2-dimensional abelian variety B, which is a quotient of $J_0(43)$. The purpose of this example is to show how numerical computation with modular symbols suggests interesting arithmetic questions about familiar abelian varieties. In the following example, we find that the conjecture of Birch and Swinnerton-Dyer (plus the Manin $c = 1$ conjecture) implies that the first nontrivial Shafarevich-Tate group of an (optimal) modular abelian variety has order *two*. Thus the surprising existence of an abelian varieties with non-square order could have been (but was not) hinted at long ago by somebody playing around with a modular symbols package (in fact, it was discovered by B. Poonen and M. Stoll [PS99] while they were designing and implementing algorithms for computing with Jacobians of genus-two curves).

```
> M43 := ModularSymbols(43,2);     // Level 43, weight 2.
> H1  := CuspidalSubspace(M43);    // H_1(X_0(43),Q)
> D   := NewformDecomposition(H1); // factors corresponding to newforms
> A,B := Explode(D);
> A;     // The homology of the elliptic curve "43A"
Modular symbols space of level 43, weight 2, and dimension 2
> B;     // The homology of the 2-dimensional abelian variety "43B"
Modular symbols space of level 43, weight 2, and dimension 4
> LRatio(B,1);        // L(B,1)/Omega_B
2/7
```

The Birch and Swinnerton-Dyer conjecture predicts that the Shafarevich-Tate group of B has order as given by the formula for `ShaAn` in the code below. To compute this value, it remains to compute $\#B(\mathbf{Q})$ and the Tamagawa number c_{43}.

```
> T := TorsionBound(B,11);  T; // #B(Q) divides this number
7
> // Compute the subgroup of B(Q) generated by (0)-(oo).
> C := SubgroupOfTorus(B,WindingElement(M43));  C;
Abelian Group isomorphic to Z/7
Defined on 1 generator
Relations:
    7*C.1 = 0
> TamagawaNumber(B,43);
7
> ShaAn := LRatio(B,1)*TorsionBound(B,11)^2/TamagawaNumber(B,43);
```

`ShaAn` is the Birch and Swinnerton-Dyer conjectural order of the Shafarevich-Tate group of B, under the assumption that the Manin constant of B is 1.

```
> ShaAn;
2
```

One of the Galois conjugate newforms associated to B is given below.

```
> qEigenform(B,12);
```

```
q + a*q^2 - a*q^3 + (-a + 2)*q^5 - 2*q^6 + (a - 2)*q^7 - 2*a*q^8 - q^9
    + (2*a - 2)*q^10 + (2*a - 1)*q^11 + O(q^12)
> BaseRing(Parent(qEigenform(B,12)));
Univariate Quotient Polynomial Algebra in a over Rational Field
with modulus a^2 - 2
> qIntegralBasis(B,12);
[
    q + 2*q^5 - 2*q^6 - 2*q^7 - q^9 - 2*q^10 - q^11 + O(q^12),
    q^2 - q^3 - q^5 + q^7 - 2*q^8 + 2*q^10 + 2*q^11 + O(q^12)
]
```

By integrating homology against the differentials corresponding to the two modular forms above, we obtain a lattice that defines the complex torus $A_B(\mathbf{C})$:

```
> Periods(B,97);
[
    (-0.22594995830676421187395192 -
        1.76664467629959953227333314 0*i
        0.52502811591322194337294916 48 +
        0.80660185770293072302831423 71*i),
    (0.59815631622412229864757672 20 -
        1.92008563861211949327648563 2*i
        0.82410627422619603486491720 82 -
        0.15344096225717705687483549 95*i),
    (-0.82410627453088651052152864 45 -
        0.15344096231251996100315249 20*i
        -0.29907815831297409149196804 34 -
        0.96004281996010777990314973 67*i),
    (-0.59815631622412229864757672 20 -
        1.92008563861211949327648563 2*i
        -0.82410627422619603486491720 83 -
        0.15344096225717705687483549 95*i)
]
```

Finally, it is tempting to ask whether or not the (conjectural) two-torsion element of the Shafarevich-Tate group of B suggested above is "visible" in the sense that it is "explained by a jump in the rank of the Mordell-Weil group of A" (see [CM00]). The following computation suggests, but does not prove, that this is the case.

```
> G := MordellWeilGroup(EllipticCurve(A)); G;
Abelian Group isomorphic to Z
Defined on 1 generator (free)
> IntersectionGroup(A,B);
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
```

133.15.1 The Period Map

Let M be a space of modular symbols the corresponds to a Galois-conjugacy class of newforms. The *period map* attached to M is a linear map

$$\text{AmbientSpace(M)} \to \mathbf{C}^d,$$

where d is the dimension of the space of modular forms associated to M. The cokernel of the period map is a complex torus $A_M(\mathbf{C})$. The terminology "period mapping" comes from the fact that there are (often?) mereomorphic functions on $\mathbf{C}^d$ whose periods are the image of the integral cuspidal modular symbols under the period mapping.

In the functions below, M must not be a $+1$ or -1 quotient and must be cuspidal.

`PeriodMapping(M, prec)`

The period mapping attached to the space of modular symbols M, computed using `prec` terms of the q-expansions of modular forms associated to M.

`Periods(M, prec)`

The complex period lattice associated to the space of modular symbols M, computed using `prec` terms of the q-expansions of modular forms associated to M.

`ClassicalPeriod(M, j, prec)`

The value

$$r_j(f) = \int_0^{i\infty} f(z) z^j dz.$$

133.15.2 Projection Mappings

Let M be a space of modular symbols over a field K. For many purposes it is useful to have a surjective map

$$\pi : \text{AmbientSpace(M)} \to V,$$

where V is a vector space over K and $\ker(\pi)$ is the same as the kernel of the period mapping.

`RationalMapping(M)`

A surjective linear map from the ambient space of the space of modular symbols M to a vector space, such that the kernel of this map is the same as the kernel of the period mapping.

`IntegralMapping(M)`

A surjective linear map from the ambient space of the space of modular symbols M to a vector space, such that the kernel of this map is the same as the kernel of the period mapping. This map is chosen in such a way that the image of

`IntegralBasis(CuspidalSubspace(AmbientSpace(M)))`

is the standard $\mathbf{Z}$-lattice. (Note that M must be defined over $\mathbf{Q}$.)

Example H133E24

```
> M := ModularSymbols(33); M;
Full Modular symbols space of level 33, weight 2, and dimension 9
> S := CuspidalSubspace(M);
> N := NewSubspace(S);
> phi := RationalMapping(N);
> [phi(x) : x in IntegralBasis(S)];
[
    ( -2 4/3),
    ( -4 2/3),
    ( -2 2/3),
    (-2  0),
    (  -2 -2/3),
    (-4  0)
]
```

Notice that the image of the basis `IntegralBasis(S)` for $H_1(X_0(33), \mathbf{Z})$ is not $\mathbf{Z} \times \mathbf{Z}$. However, `IntegralMapping(N)` is normalized so that the image is $\mathbf{Z} \times \mathbf{Z}$:

```
> int := IntegralMapping(N);
> [int(S.i) : i in [1..Dimension(S)]];
[
    ( 2 -1),
    ( 1 -2),
    ( 1 -1),
    ( 0 -2),
    ( 0 -1),
    (-1 -1)
]
```

Consider a quotient A_f of $J_0(N)$ attached to a newform $f \in S_2(\Gamma_0(N))$. Using `IntegralMapping` and the Abel-Jacobi theorem, we can see the image in $A_f(\mathbf{Q})$ of the point $(0) - (\infty) \in J_0(N)(\mathbf{Q})$. In the level 97 example below, this image has order 8, which is the numerator of $(97-1)/12$.

```
> Af := ModularSymbols("97B"); Af;
Modular symbols space of level 97, weight 2, and dimension 8
> int := IntegralMapping(Af);
> // Let x be the modular symbol {0,oo}
> x := AmbientSpace(Af)!<1,[Cusps()|0,Infinity()]>;
> int(x);
(-5/8  1/4 -1/4    0    0  1/4  3/8  1/4)
> Numerator((97-1)/12);
8
```

133.16 Modular Abelian Varieties

Let M be a space of weight 2 cuspidal modular symbols with trivial character that corresponds to a Galois-conjugacy class of newforms, and let $A_M(\mathbf{C})$ be the cokernel of the period map. G. Shimura proved that $A_M(\mathbf{C})$ is the set of complex points of an abelian variety A_M defined over $\mathbf{Q}$. Let N be the level of M and let $J_0(N)$ be the Jacobian of the modular curve $X_0(N)$. Shimura constructed A_M as a quotient of $J_0(N)$ by an abelian subvariety. More precisely, if I is the annihilator of M in the Hecke algebra, then $A_M = J_0(N)/IJ_0(N)$.

When A_M has dimension 1 it is an elliptic curve, and the theory of computing with A_M is well developed, though many interesting problems remain. In the contrary case, when A_M has dimension greater than 1, the theory of computation with A_M is still in its infancy. Fortunately, it is possible to compute a number of interesting quantities about A_M using algorithms that rely on our extensive knowledge of $J_0(N)$.

MAGMA contains functions for computing the modular degree, congruence modulus, upper and lower bounds on the order of the torsion subgroup, and the order of the component group of the closed fiber of the Néron model of A_M at primes that exactly divide the level of M.

133.16.1 Modular Degree and Torsion

`ModularDegree(M)`

The modular degree of the space of modular symbols M, which is defined as follows. Let M be a space of modular symbols of weight 2 and trivial character. The modular degree of M is the square root of `#ModularKernel(M)`. When M corresponds to an elliptic curve $E = A_M$, then the modular degree of M is the degree of induced map $X_0(N) \to E$.

`CongruenceModulus(M : parameters)`

`Bound` RNGINTELT *Default :* -1

The congruence number r of the space of modular symbols M. This is the index in $S_k(\Gamma_0(N), \mathbf{Z})$ of the sum $L + W$ of the lattice W of cusp forms L corresponding to M and the lattice of cusp forms corresponding to the complement of L in S.

`TorsionBound(M, maxp)`

The following upper bound on the order of the torsion subgroup of the abelian variety A attached to the space of modular symbols M:

$$\gcd\{\#A(\mathbf{F}_p) : 3 \leq p \leq \text{maxp},\ \ p \not| N\},$$

where N is the level of M. This bound is an isogeny invariant, so it is also a bound on the order of the torsion subgroup of the dual abelian variety $A^\vee$ of A.

To compute a lower bound, use `#SubgroupOfTorus(M,WindingElement(M))`.

Example H133E25

We compute the first example of an optimal elliptic curve over $\mathbf{Q}$ such that the congruence modulus does not equal the modular degree. (See [FM99] for further discussion of this problem. We warn the reader that the divisibility $r \mid \deg(\phi) \mid rN^i$ cited there is incorrect, as our `54B` example shows.)

```
> E := ModularSymbols("54B");
> ModularDegree(E);
2
> CongruenceModulus(E);
6
```

We next verify directly that the congruence modulus is divisible by 3.

```
> A := ModularSymbols("27A"); A;   // 27=54/2.
Modular symbols space of level 27, weight 2, and dimension 2
> A54 := ModularSymbols(A,54); A54;  // all images of A at level 54.
Modular symbols space of level 54, weight 2, and dimension 4
> qE := qIntegralBasis(E,17);
> qA54 := qIntegralBasis(A54,17);
> &+qA54 - &+qE;
-3*q^4 + 3*q^5 - 3*q^8 + 3*q^10 - 3*q^11 + 9*q^13 + 3*q^16 + O(q^17)
> IntersectionGroup(E,A54);   // however, the intersection is trivial.
Abelian Group of order 1
```

Ken Ribet proved that if E is an optimal elliptic curve quotient of $J_0(N)$, with N prime, and if f_E is the corresponding newform, then the congruence modulus of f_E equals the modular degree of E. The author is aware of no counterexamples to the following more general statement: "If E is an optimal elliptic curve of square-free conductor, then the congruence modulus of the newform f_E attached to E equals the modular degree of E." An analogous statement for abelian varieties is false, even at prime level. The first counterexample is `ModularSymbols("431F")`, which corresponds to an abelian variety of dimension 24. In this case, the modular degree is $2^{11} \cdot 6947$, whereas the congruence modulus is $2^{10} \cdot 6947$.
The following code makes a table of congruence moduli and modular degrees for the elliptic curves of conductor near 54. Notice the counterexample at level 54.

```
> for N in [53..55] do
>    C := CuspidalSubspace(ModularSymbols(N,2));
>    newforms := NewSubspace(C);
>    D := EllipticFactors(newforms,19);
>    for E in D do
>       printf "%o:\t%o,\t%o\n", N, ModularDegree(E), CongruenceModulus(E);
>    end for;
> end for;
53:     2,      2
54:     2,      6
54:     6,      6
55:     2,      2
```

`ModularKernel` makes sense even for spaces of modular symbols of weight greater than 2. As in the case of weight 2, this number gives information about congruences between modular forms.

The following example illustrates how `ModularKernel` suggest a congruence between a form of level 10 and weight 4 with a form of level 5.

```
> M := ModularSymbols(10,4);
> S := CuspidalSubspace(M);
> D := NewformDecomposition(S); D;
[
    Modular symbols space of level 10, weight 4, and dimension 2,
    Modular symbols space of level 10, weight 4, and dimension 4
]
> #ModularKernel(D[1]);
10
> f := qEigenform(D[1],8);
> g := qEigenform(D[2],8);
> g2 := Evaluate(g,Parent(g).1^2);
> f-(g+6*g2);   // a congruence modulo 10!
-10*q^3 + 20*q^4 + 10*q^5 - 20*q^6 - 10*q^7 + O(q^8)
```

133.16.2 Tamagawa Numbers and Orders of Component Groups

We provide several functions for computing the orders of component groups of optimal quotients of $J_0(N)$ at primes p that exactly divide N. Our algorithm involves Grothendieck's monodromy pairing on the character group of the toric part of the closed fiber at p of the Néron model of $J_0(N)$; the theory behind this algorithm is described in [Ste01] (or [Ste00]); see [KS00] for a computationally-oriented introduction to the algorithm. When N is prime, we use the Mestre and Oesterlé method to construct the character group of the torus, as described in [Mes86]. In general, the ideal theory of quaternion algebras is used.

Note: In the appendix to [Maz77], Mazur and Rapaport give an explicit formula for the order of the component group of $J_0(N)$ at primes $p \geq 5$ that exactly divide N. Their formula is not currently used by the `ComponentGroupOrder` function.

The `RealTamagawaNumber` function computes the order of the "component group at infinity".

`ComponentGroupOrder(M, p)`

The order of the component group at p. This is the order of the group of $\overline{\mathbf{F}}_p$-points of the component group of the reduction modulo p of the Néron model of the abelian variety attached to the space of modular symbols M. At present, it is necessary that p exactly divides the level. If `Sign(M)` is not equal to 0, then only the odd part of the order is returned.

`TamagawaNumber(M, p)`

The order of the group of $\mathbf{F}_p$-rational points of the component group of the space of modular symbols M. We require M to be associated to a single Galois-conjugacy class of newforms.

`RealTamagawaNumber(M)`

The number of connected components of $A_M(\mathbf{R})$.

`MinusTamagawaNumber(M)`

The number of connected components of the subgroup $A_M(\mathbf{C})^-$ of $A_M(\mathbf{C})$ on which complex conjugation acts as -1

Example H133E26

We compute the orders of the component groups of some abelian varieties.

```
> X11 := ModularSymbols("11A");     // corresponds to X_0(11).
> ComponentGroupOrder(X11,11);
5
> TamagawaNumber(X11,11);
5
> RealTamagawaNumber(X11);
1
> MinusTamagawaNumber(X11);
1
> J37 := ModularSymbols("37"); J37;
Modular symbols space of level 37, weight 2, and dimension 4
> ComponentGroupOrder(J37,37);
3
> A, B := Explode(NewformDecomposition(J37));
> ComponentGroupOrder(A,37);
3
> ComponentGroupOrder(B,37);
1
```

We can also compute component groups of optimal quotients whose dimension is greater than 1. The abelian varieties `B` and `C` below correspond to the Jacobians labeled $65B$ and $65A$ in [FLS$^+$01], respectively.

```
> J65 := ModularSymbols("65");
> A,B,C := Explode(SortDecomposition(NewformDecomposition(J65)));
> B;
Modular symbols space of level 65, weight 2, and dimension 4
> C;
Modular symbols space of level 65, weight 2, and dimension 4
> ComponentGroupOrder(B,5);      // not the Tamagawa number
3
> ComponentGroupOrder(B,13);
3
> ComponentGroupOrder(C,5);
7
> ComponentGroupOrder(C,13);
1
```

```
> HeckeEigenvalueField(C);
Number Field with defining polynomial x^2 + 2*x - 1 over the
Rational Field
Mapping from: Univariate Quotient Polynomial Algebra in a over
Rational Field
with modulus a^2 + 2*a - 1 to Number Field with defining
polynomial x^2 + 2*x - 1 over the Rational Field given by a rule
[no inverse]
> ComponentGroupOrder(J65,5);
42
```

When the Atkin-Lehner involution W_p acts as $+1$ on a modular abelian variety A, the order of the component group can be larger than the Tamagawa number $c_p = [A(\mathbf{Q}_p) : A_0(\mathbf{Q}_p)]$ that appears in the conjecture of Birch and Swinnerton-Dyer.

```
> AtkinLehner(B,5);
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
> ComponentGroupOrder(B,5);
3
> TamagawaNumber(B,5);
1
```

The real and minus Tamagawa numbers are defined for spaces of modular symbols of any weight over the rationals.

```
> Del := ModularSymbols("1k12A");
> Del;
Modular symbols space of level 1, weight 12, and dimension 2
```

Next we see that the period lattice associated to Δ is rectangular.

```
> RealTamagawaNumber(Del);
2
> MinusTamagawaNumber(Del);
2
> Periods(Del,40);
[
    (-0.00048533816492995160492413044 29*i),
    (0.00114073744958307 9336044545337)
]
```

133.17 Elliptic Curves

Let E be an elliptic curve. By the modularity theorem, which was recently proved by Breuil, Conrad, Diamond, Taylor, and Wiles there is a two-dimensional space M of modular symbols attached to E. Let N be the conductor of E; then M is obtained from `ModularSymbols(N,2)` by intersecting the kernels of $T_p - a_p(E)$ for sufficiently many p.

Warning: The computation of M can already be very resource intensive for elliptic curves for which `Conductor(E)` is on the order of 5000. For example, the seemingly harmless expression `ModularSymbols(EllipticCurve([0,6]))` would bring my computer to its knees.

`ModularSymbols(E)`

`ModularSymbols(E, sign)`

The space M of modular symbols associated to the elliptic curve E.

`EllipticCurve(M)`

`Database`	BoolElt	*Default :* `true`

An elliptic curve over the rational numbers that lies in the isogeny class of elliptic curves associated to M.

By default, the Cremona database is used. To compute the curve from scratch, set the optional parameter `Database` to `false`. (Note however that this is not optimized for large level.)

`pAdicLSeries(E, p)`

`CoefficientPrecision`	RngIntElt	*Default :* 3
`SeriesPrecision`	RngIntElt	*Default :* 5
`QuadraticTwist`	RngIntElt	*Default :* 1
`ProperScaling`	BoolElt	*Default :* `true`

Given an elliptic curve over the rationals and a prime of multiplicative or good ordinary reduction, compute the p-adic L-series. For technical reasons involving caching, the curve is required to be given by a (global) minimal model.

This computation can be rather time-consuming for large conductors and/or primes. The algorithm needs to compute the modular symbols (taking N^3 time or so), and typically needs to evaluate $2(p-1)p^n$ of them, where n is the desired coefficient precision. Also, the desired coefficient precision does not always apply to every term in the series.

The `QuadraticTwist` vararg computes the p-adic L-series for a twist by a fundamental discriminant D. This requires $|D|$ as many modular symbols to be computed, but in most cases is much faster than computing the modular symbols for the twisted curve. The twisting discriminant is required to be coprime to the conductor of the curve.

The `ProperScaling` vararg indicates whether to ensure that the scaling of the p-adic L-series is correct. For many applications, only the valuation of the coefficients

is needed. The computation of the proper scaling induces a bit of extra overhead, but is still typically faster than computing the modular symbols for examples of interest.

Example H133E27

We use the elliptic curve functions to numerically compute the Birch and Swinnerton-Dyer conjectural order of the Shafarevich-Tate group of the elliptic curve **389A**, which is the curve of rank 2 with smallest conductor. The Birch and Swinnerton-Dyer conjecture asserts that

$$\frac{L^{(r)}(E,1)}{r!} = \frac{\prod c_p \cdot \mathrm{Sha} \cdot \mathrm{Reg}}{|E(\mathbf{Q})_{\mathrm{tor}}|^2},$$

where r is the order of vanishing of $L(E,s)$ at $s=1$.

```
> E := EllipticCurve(CremonaDatabase(),"389A");
> M := ModularSymbols(E);
> M;
Modular symbols space of level 389, weight 2, and dimension 2
> LRatio(M,1);
0
```

Next we compute the analytic rank and the leading coefficient of the L-series at $s=1$. (If your computer is very slow, use a number smaller than 300 below.)

```
> L1, r := LSeriesLeadingCoefficient(M,1,300);
> L1;
0.75931650029224679065762600 31
> r;        // The analytic rank is 2.
2
```

Finally we check that the rank conjecture is true in this case, and compute the conjectural order of the Shafarevich-Tate group.

```
> Rank(E);  // The algebraic rank is 2.
2
> Omega := Periods(M,300)[2][1] * 2; Omega;
4.980435433609741580582713757
> Reg := Regulator(E); Reg;
0.1524601779431437875
> #TorsionSubgroup(E);
1
> TamagawaNumber(E,389);
1
> TamagawaNumber(M,389);        // entirely different algorithm
1
> Sha := L1/(Omega*Reg); Sha;
0.9999979295234896211
> f := pAdicLSeries(E,3); _<T> := Parent(f); f;
O(3^11) + O(3^3)*T - (1 + O(3^3))*T^2 + (2 + O(3^2))*T^3
 - (2 + O(3^2))*T^4 + (1 + O(3^2))*T^5 + O(T^6)
```

133.18 Dimension Formulas

`DimensionCuspFormsGamma0(N, k)`

The dimension of the space $S_k(\Gamma_0(N))$ of weight k cusp forms for $\Gamma_0(N)$.

`DimensionNewCuspFormsGamma0(N, k)`

The dimension of the new subspace of the space $S_k(\Gamma_0(N))$ of weight k cusp forms for $\Gamma_0(N)$.

`DimensionCuspFormsGamma1(N, k)`

The dimension of the space $S_k(\Gamma_1(N))$ of weight k cusp forms for $\Gamma_1(N)$.

`DimensionNewCuspFormsGamma1(N, k)`

The dimension of the new subspace of the space $S_k(\Gamma_1(N))$ of weight k cusp forms for $\Gamma_1(N)$.

`DimensionCuspForms(eps, k)`

The dimension of the space $S_k(\Gamma_1(N))(\varepsilon)$ of cusp forms of weight k and Dirichlet character `eps`. The level N is the modulus of `eps`. The dimension is computed using the formula of Cohen and Oesterlè (see [CO77]).

Example H133E28

```
> DimensionCuspFormsGamma0(11,2);
1
> DimensionCuspFormsGamma0(1,12);
1
> DimensionCuspFormsGamma0(5077,2);
422
> DimensionCuspFormsGamma1(5077,2);
1071460
> G := DirichletGroup(5*7);
> eps := G.1*G.2;
> IsOdd(eps);
true
> DimensionCuspForms(eps,2);
0
> DimensionCuspForms(eps,3);
6
```

The dimension of the space of cuspidal modular symbols is twice the dimension of the space of cusp forms.

```
> Dimension(CuspidalSubspace(ModularSymbols(eps,3)));
12
```

133.19 Bibliography

[**AS02**] Amod Agashe and William A. Stein. *Appendix to Joan-C. Lario and René Schoof:* Some computations with Hecke rings and deformation rings. *submitted*, 2002.

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**BS02**] Kevin Buzzard and William A. Stein. A mod five approach to modularity of icosahedral Galois representations. *Pacific J. Math.*, 203(2):265–282, 2002.

[**CM00**] J. E. Cremona and Barry Mazur. Visualizing elements in the Shafarevich-Tate group. *Experiment. Math.*, 9(1):13–28, 2000.

[**CO77**] Henri Cohen and J. Oesterlé. *Dimensions des espaces de formes modulaires*, volume 627 of *Lecture Notes in Math.*, pages 69–78. Springer, Berlin, 1977.

[**Cre92**] J. E. Cremona. Modular symbols for $\Gamma_1(N)$ and elliptic curves with everywhere good reduction. *Math. Proc. Cambridge Philos. Soc.*, 111(2):199–218, 1992.

[**Cre97**] J. E. Cremona. *Algorithms for modular elliptic curves*. Cambridge University Press, Cambridge, second edition, 1997.

[**DI95**] Fred Diamond and Ju Im. Modular forms and modular curves. In *Seminar on Fermat's Last Theorem*, pages 39–133. Amer. Math. Soc., Providence, RI, 1995.

[**FLS$^+$01**] E. V. Flynn, F. Leprévost, E. F. Schaefer, W. A. Stein, M. Stoll, and J. L. Wetherell. Empirical evidence for the Birch and Swinnerton-Dyer conjectures for modular Jacobians of genus 2 curves. *Math. of Comp.*, 70(236):1675–1697, 2001.

[**FM99**] Gerhard Frey and Michael Muller. Arithmetic of modular curves and applications. In *Algorithmic algebra and number theory (Heidelberg, 1997)*, pages 11–48. Springer, Berlin, 1999.

[**KS00**] David R. Kohel and William A. Stein. Component Groups of Quotients of $J_0(N)$. In Bosma [Bos00].

[**Maz77**] Barry Mazur. Modular curves and the Eisenstein ideal. *Inst. Hautes Études Sci. Publ. Math.*, 47:33–186 (1978), 1977.

[**Mer93**] Loic Merel. Intersections sur des courbes modulaires. *Manuscripta Math.*, 80(3):283–289, 1993.

[**Mer94**] Loic Merel. Universal Fourier expansions of modular forms. In *On Artin's conjecture for odd 2-dimensional representations*, pages 59–94. Springer, 1994.

[**Mes86**] Jean-Francois Mestre. La méthode des graphes. Exemples et applications. *Proceedings of the international conference on class numbers and fundamental units of algebraic number fields (Katata)*, pages 217–242, 1986.

[**PS99**] Bjorn Poonen and Michael Stoll. The Cassels-Tate pairing on polarized abelian varieties. *Ann. of Math. (2)*, 150(3):1109–1149, 1999.

[**Ste00**] William A. Stein. Explicit approaches to modular abelian varieties. *Ph.D. thesis, University of California, Berkeley*, 2000.

[**Ste01**] William A. Stein. Component Groups of Purely Toric Quotients of Semistable Jacobians. *submitted*, 2001.

[**Ste08**] William A. Stein. An introduction to computing modular forms using modular symbols. In *Algorithmic number theory: lattices, number fields, curves and cryptography*, volume 44 of *Math. Sci. Res. Inst. Publ.*, pages 641–652. Cambridge Univ. Press, Cambridge, 2008.

[**Stu87**] Jacob Sturm. On the congruence of modular forms. In *Number theory (New York, 1984–1985)*, pages 275–280. Springer, Berlin, 1987.

134 BRANDT MODULES

Chapter 134
BRANDT MODULES

134.1 Introduction

Brandt modules provide a representation in terms of quaternion ideals of certain cohomology subgroups associated to Shimura curves $X_0^D(N)$ which generalize the classical modular curves $X_0(N)$. The Brandt module datatype is that of a Hecke module – a free module of finite rank with the action of a ring of Hecke operators – which is equipped with a canonical basis (identified with left quaternion ideal classes) and an inner product which is adjoint with respect to the Hecke operators. The machinery of modular symbols, Brandt modules, and, in a future release, a module of singular elliptic curves, form the computational machinery underlying modular forms in MAGMA.

Brandt modules were implemented by David Kohel, motivated by the article of Mestre and Oesterlé [Mes86] on the method of graphs for supersingular elliptic curves, the article of Pizer [Piz80] on computing spaces of modular forms using quaternion arithmetic, and grew out of research in the author's thesis [Koh96] on endomorphism ring structure of elliptic curves over finite fields. The Brandt module machinery is described in the article [Koh01] and has been used, together with modular symbols, in the computation of component groups of quotients of the Jacobians $J_0(N)$ of classical modular curves [KS00].

134.2 Brandt Module Creation

We first describe the various constructors for Brandt modules and their elements.

`BrandtModule(D)`

`BrandtModule(D, m)`

`ComputeGrams` BOOLELT *Default :* `true`

Given a product D of an odd number of primes, and a integer m which has valuation at most one at each prime divisor p of D, return a Brandt module of level (D, m) over the integers. If not specified, then the conductor m is taken to be 1.

The parameter `ComputeGrams` can be set to `false` in order to not compute the $h \times h$ array, where h is the left class number of level (D, m), of reduced Gram matrices of the quaternion ideal norm forms. Instead the basis of quaternion ideals is stored and the collection of degree p ideal homomorphisms is then computed in order to find the Hecke operator T_p for each prime p.

For very large levels, setting `ComputeGrams` to `false` is more space efficient. For moderate sized levels for which one wants to compute many Hecke operators, it is preferable to compute the Gram matrices and determine the Hecke operators using theta series.

`BrandtModule(A)`

`BrandtModule(A, R)`

`ComputeGrams` BoolElt *Default :* `true`

Given a definite order A in a quaternion algebra over $\mathbf{Q}$, returns the Brandt module on the left ideals classes for A, as a module over R. If not specified, the ring R is taken to be the integers. The parameter `ComputeGrams` is as previously described.

`BaseExtend(M, R)`

Forms the Brandt module with coefficient ring base extended to R.

`BrandtModule(M, N)`

This constructor is an alternative to `BrandtModule(D, N)` above, and *uses a different algorithm* which is preferable in the case where N is not very small.

It constructs the Brandt module attached to an Eichler order of level N inside the maximal order M. The algorithm avoids explicitly working with the Eichler order.

Example H134E1

In the following example we create the Brandt module of level 101 over the field of 7 elements and decompose it into its invariant subspaces.

```
> A := QuaternionOrder(101);
> FF := FiniteField(7);
> M := BrandtModule(A,FF);
> Decomposition(M,13);
[
    Brandt module of level (101,1), dimension 1, and degree 9 over Finite
    field of size 7,
    Brandt module of level (101,1), dimension 1, and degree 9 over Finite
    field of size 7,
    Brandt module of level (101,1), dimension 1, and degree 9 over Finite
    field of size 7,
    Brandt module of level (101,1), dimension 6, and degree 9 over Finite
    field of size 7
]
```

We note that Brandt modules of non-prime discriminant can be useful for studying isogeny factors of modular curves, since it is possible to describe exactly the piece of cohomology of interest, without first computing a much larger space. In this example we see that the space of weight 2 cusp forms for $\Gamma_0(1491)$, where $1491 = 3 \cdot 7 \cdot 71$, is of dimension 189 (plus an Eisenstein space of dimension 7), while the newspace has dimension 71. The Jacobian of the Shimura curve $X_0^{1491}(1)$ is isogenous to the new factor of $J_0(1491)$, so that we can study the newspace directly via the Brandt module.

```
> DimensionCuspFormsGamma0(3*7*71,2);
189
> DimensionNewCuspFormsGamma0(3*7*71,2);
```

```
71
> BrandtModuleDimension(3*7*71,1);
72
> M := BrandtModule(3*7*71 : ComputeGrams := false);
> S := CuspidalSubspace(M);
> Dimension(S);
71
> [ Dimension(N) : N in Decomposition(S,13 : Sort := true) ];
[ 6, 6, 6, 6, 11, 12, 12, 12 ]
```

In this example by setting `ComputeGrams` equal to `false` we obtain the Brandt module much faster, but the decomposition is much more expensive. For most applications the default computation of Gram matrices is preferable.

134.2.1 Creation of Elements

`M ! x`

Given a sequence or module element x compatible with the Brandt module M, forms the corresponding element in M.

`M . i`

For a Brandt module M and integer i, returns the i-th basis element.

134.2.2 Operations on Elements

Brandt module elements support standard operations.

`a * x`
`x * a`

The scalar multiplication of a Brandt module element x by an element a in the base ring.

`x * T`

Given a Brandt module element x and an element T of the algebra of Hecke operators of degree compatible with the parent of x or of its ambient module, returns the image of x under T.

`x + y`

Returns the sum of two Brandt module elements.

`x - y`

Returns the difference of two Brandt module elements.

`x eq y`

Returns `true` if x and y are equal elements of the same Brandt module.

`Eltseq(x)`

Returns the sequence of coefficients of the Brandt module element x.

`InnerProduct(x, y)`

Returns the inner product of the Brandt module elements x and y with respect to the canonical pairing on their common parent.

`Norm(x)`

Returns the inner product of the Brandt module element x with itself.

134.2.3 Categories and Parent

Brandt modules belong to the category `ModBrdt`, with elements of type `ModBrdtElt`, involved in the type checking of arguments in MAGMA programming. The `Parent` of an element is the space to which it belongs.

`Category(M)` `Type(M)`
`Category(x)` `Type(x)`

The category, `ModBrdt` or `ModBrdtElt`, of the Brandt module M or of the Brandt module element x.

`Parent(x)`

The parent module M of a Brandt module element x.

`x in M`

Returns `true` if M is the parent of x.

134.2.4 Elementary Invariants

Here we describe the elementary invariants of the Brandt module, defined with respect to a definite quaternion order A in a quaternion algebra $\mathbf{H}$ over $\mathbf{Q}$. The *level* of M is defined to be the reduced discriminant of A, the *discriminant* is defined to be the discriminant of the algebra $\mathbf{H}$, and the *conductor* to be the index of A in any maximal order of $\mathbf{H}$ which contains it. We note that the discriminant of M is just the product of the ramified primes of $\mathbf{H}$, and the product of the conductor and discriminant of M is the reduced discriminant of A.

`Level(M)`

Returns the level of the Brandt module, which is the product of the discriminant and the conductor, and equal to the reduced discriminant of its defining quaternion order.

`Discriminant(M)`

Returns the discriminant of the quaternion algebra **H** with respect to which the Brandt module M is defined (equal to the product of the primes which ramify in **H**).

`Conductor(M)`

Returns the conductor or index of the defining quaternion order of the Brandt module M in a maximal order of its quaternion algebra.

`BaseRing(M)`

The ring over which the Brandt module M is defined.

`Basis(M)`

Returns the basis of the Brandt module M.

134.2.5 Associated Structures

The following give structures associated to Brandt modules. In particular we note the definition of the `AmbientModule`, which is the full module containing a given Brandt module whose basis corresponds to the left quaternion ideals. Elements of every submodule of the ambient module are displayed with respect to the basis of the ambient module.

`AmbientModule(M)`

The full module of level (D, m) containing a given module of this level.

`IsAmbient(M)`

Returns `true` if and only if the Brandt module M is its own ambient module.

`Dimension(M)`
`Rank(M)`

Returns the rank of the Brandt module M over its base ring.

`Degree(M)`

Returns the degree of the Brandt module M, defined to be the dimension of its ambient module.

`GramMatrix(M)`

The matrix $(\langle u_i, u_j \rangle)$ defined with respect to the basis $\{u_i\}$ of the Brandt module M.

`InnerProductMatrix(M)`

Returns the Gram matrix of the ambient module of the Brandt module M.

Example H134E2

The following example demonstrates the use of `AmbientModule` to get back to the original Brandt module.

```
> M := BrandtModule(3,17);
> S := CuspidalSubspace(M);
> M eq AmbientModule(M);
true
```

`Ideals(M)`

This constructs the quaternionic ideals which correspond to the basis of the Brandt module M. It is only implemented when M was constructed using `BrandtModule(M, N)` – the case where the new algorithm is used, which avoids constructing these ideals explicitly.

134.2.6 Verbose Output

The verbose level for Brandt modules is set with the command `SetVerbose("Brandt",n)`. Since the construction of a Brandt module requires intensive quaternion algebra machinery for ideal enumeration, the `Quaternion` verbose flag is also relevant. In both cases, the value of n can be 0 (silent), 1 (verbose), or 2 (very verbose).

Example H134E3

In the following example we show the verbose output from the quaternion ideal enumeration in the creation of the Brandt module of level $(37, 1)$.

```
> SetVerbose("Quaternion",2);
> BrandtModule(37);
Ideal number 1, right order module
Full RSpace of degree 4 over Integer Ring
Inner Product Matrix:
[ 2  0  1  1]
[ 0  4 -1  2]
[ 1 -1 10  0]
[ 1  2  0 20]
Frontier at 2-depth 1 has 3 elements.
Number of ideals = 1
Ideal number 2, new right order module
Full RSpace of degree 4 over Integer Ring
Inner Product Matrix:
[ 2 -1  0  1]
[-1  8 -1 -4]
[ 0 -1 10 -2]
[ 1 -4 -2 12]
Ideal number 3, new right order module
Full RSpace of degree 4 over Integer Ring
```

```
Inner Product Matrix:
[ 2  1  0 -1]
[ 1  8  1  3]
[ 0  1 10 -2]
[-1  3 -2 12]
Frontier at 2-depth 2 has 4 elements.
Number of ideals = 3
Brandt module of level (37,1), dimension 3, and degree 3 over Integer Ring
```

134.3 Subspaces and Decomposition

`EisensteinSubspace(M)`

Returns the Eisenstein subspace of the Brandt module M. When the level of M is square-free this will be the submodule generated by a vector of the form $(w/w_1, \ldots, w/w_n)$, if it exists in M, where w_i is the number of automorphisms of the i-th basis ideal and $w = \mathrm{LCM}(\{w_i\})$.

`CuspidalSubspace(M)`

Returns the cuspidal subspace, defined to be the orthogonal complement of the Eisenstein subspace of the Brandt module M. If the discriminant of M is coprime to the conductor, then the cuspidal subspace consists of the vectors in M of the form $(a_1, \ldots, a_n)$, where $\sum_i a_i = 0$.

`OrthogonalComplement(M)`

The Brandt module orthogonal to the given module M in the ambient module of M.

`M meet N`

Returns the intersection of the Brandt modules M and N.

`Decomposition(M, B)`

`Sort` BoolElt *Default :* `true`

Returns a decomposition of the Brandt module with respect to the Atkin–Lehner operators and Hecke operators up to the bound B. The parameter `Sort` can be set to `true` to return a sequence sorted under the operator `lt` as defined below.

`SortDecomposition(D)`

Sort the sequence D of spaces of Brandt modules with respect to the `lt` comparison operator.

Example H134E4

```
> M := BrandtModule(2*3*17);
> Decomp := Decomposition(M,11 : Sort := true);
> Decomp;
[
    Brandt module of level (102,1), dimension 1, and degree 4 over Integer Ring,
    Brandt module of level (102,1), dimension 1, and degree 4 over Integer Ring,
    Brandt module of level (102,1), dimension 1, and degree 4 over Integer Ring,
    Brandt module of level (102,1), dimension 1, and degree 4 over Integer Ring
]
> [ IsEisenstein(N) : N in Decomp ];
[ true, false, false, false ]
```

134.3.1 Boolean Tests on Subspaces

`IsEisenstein(M)`

Returns `true` if and only if the Brandt module M is contained in the Eisenstein subspace of the ambient module.

`IsCuspidal(M)`

Returns `true` if and only if the Brandt module M is contained in the cuspidal subspace of the ambient module.

`IsIndecomposable(M, B)`

Returns `true` if an only if the Brandt module M does not decompose into complementary Hecke-invariant submodules under the Atkin-Lehner operators, nor under the Hecke operators T_n, for $n \leq B$.

`M1 subset M2`

Returns `true` if and only if $M1$ is contained in the module $M2$.

`M1 lt M2`

`Bound` RngIntElt *Default :* 101

Given two indecomposable subspaces, M_1 and M_2, returns `true` if and only if $M_1 < M_2$ under the following ordering:

(1) Order by dimension, with smaller dimension being less.

(2) An Eisenstein subspace is less than a cuspidal subspace of the same dimension.

(3) Order by Atkin–Lehner eigenvalues, starting with *smallest* prime dividing the level and with '+' being less than '–'.

(4) Order by $|\operatorname{Tr}(T_{p^i}(M_j))|$, p not dividing the level, and $1 \leq i \leq g$, where g is `Dimension`(M_1), with the positive one being smaller in the event of equality.

Condition (4) differs from the similar one for modular symbols, but permits the comparison of arbitrary Brandt modules. The algorithm returns `false` if all primes up to value of the parameter `Bound` fail to differentiate the arguments.

`M1 gt M2`

`Bound` RngIntElt *Default :* 101

Returns the complement of `lt` for Brandt modules $M1$ and $M2$.

Example H134E5

```
> M := BrandtModule(7,7);
> E := EisensteinSubspace(M);
> Basis(E);
[
    (1 1 0 0),
    (0 0 1 1)
]
> S := CuspidalSubspace(M);
> Basis(S);
[
    ( 1 -1  0  0),
    ( 0  0  1 -1)
]
> PS<q> := LaurentSeriesRing(RationalField());
> qExpansionBasis(S,100);
[
    q + q^2 - q^4 - 3*q^8 - 3*q^9 + 4*q^11 - q^16 - 3*q^18 + 4*q^22 + 8*q^23
        - 5*q^25 + 2*q^29 + 5*q^32 + 3*q^36 - 6*q^37 - 12*q^43 - 4*q^44 +
        8*q^46 - 5*q^50 - 10*q^53 + 2*q^58 + 7*q^64 + 4*q^67 + 16*q^71 +
        9*q^72 - 6*q^74 + 8*q^79 + 9*q^81 - 12*q^86 - 12*q^88 - 8*q^92 -
        12*q^99 + 5*q^100 + O(q^101)
]
```

134.4 Hecke Operators

A Brandt module M is equipped with a family of linear Hecke operators which act on it. These are returned as matrices, which act on the right with respect to the basis `Basis(M)`, but the Hecke operators of the ambient module may also be applied to elements of submodules. The system of Hecke operators are computed by default using the theta series which define the classical Brandt matrices. If a module is created with the `ComputeGram` parameter set to false, the Hecke operators are determined by means of enumeration of ideals in a p-neighbouring operation analogous to the method of graphs approach of Mestre and Oesterlé [Mes86].

`HeckeOperator(M, n)`

Compute a matrix representing the nth Hecke operator T_n with respect to `Basis(M)` for the Brandt module M.

`AtkinLehnerOperator(M, p)`

Computes the Atkin–Lehner operator on the Brandt module M, where p is a prime dividing the discriminant of M.

134.5 q-Expansions

We can associate a theta series to any pair of elements of a Brandt module, which give embeddings (with respect to any fixed module element) of the Brandt module in a space of weight 2 modular forms.

`ThetaSeries(x, y, prec)`

Returns the theta series associated to the pair (x, y) of elements of a Brandt module, as an element of a power series ring.

`qExpansionBasis(M, prec)`

A sequence of power series elements, to precision *prec*, spanning the image of the theta functions associated to pairs in the Brandt module M.

134.6 Dimensions of Spaces

`BrandtModuleDimension(D, N)`

The dimension of the Brandt module of level (D, N), computed by means of standard formulas.

Example H134E6

In the following example we demonstrate the computation of the dimensions of the Brandt modules for the sequence of Eichler orders of index 3^i in a maximal order in the quaternion algebra of discriminant $2 \cdot 3 \cdot 7$.

```
> D := 2*5*7;
> for i in [0..10] do
>     BrandtModuleDimension(D,3^i);
> end for;
2
8
24
72
216
648
1944
5832
17496
52488
157464
```

134.7 Brandt Modules Over $F_q[t]$

This section concerns the case of quaternion orders whose base ring is $F_q[t]$. The definitions and constructions are similar to the case of quaternion orders over the integers. The implementation follows the new implementation over the integers (avoiding working explicitly in terms of ideals in Eichler orders), and makes use of techniques for quadratic forms over $F_q[t]$ (developed by Markus Kirschmer).

The following intrinsics are provided. Where no description is given, the arguments and return values are similar to the corresponding intrinsics over the integers.

```
BrandtModuleDimension(D, N)
```

```
BrandtModuleDimensionOfNewSubspace(D, N)
```

```
BrandtModule(M, N)
```

This constructs the Brandt module attached to an Eichler order of level N in the maximal order M.

```
QuaternionOrder(M)
```

```
Level(M)
```

```
Discriminant(M)
```

```
Conductor(M)
```

```
Ideals(M)
```

```
InnerProductMatrix(M)
```

```
HeckeOperator(M, n)
```

```
HeckeEigenvectors(M)
```

This returns the common eigenvectors for the Hecke operators on the Brandt module M, as elements of M.

```
HeckeEigenvalue(f, p)
```

For a Hecke eigenform f in a Brandt module, this returns the eigenvalue for the Hecke operator at the prime p.

134.8 Bibliography

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**Koh96**] D. Kohel. *Endomorphism rings of elliptic curves over finite fields*. PhD thesis, University of California, Berkeley, 1996.

[**Koh01**] D. Kohel. Hecke module structure of quaternions. In K. Miyake, editor, *Class Field Theory – its Centenary and Prospect*, 2001.

[**KS00**] D. Kohel and W. Stein. Component groups of quotients of $J_0(N)$. In Bosma [Bos00].

[**Mes86**] J.-F. Mestre. Sur la méthode des graphes, Exemples et applications. In *Proceedings of the international conference on class numbers and fundamental units of algebraic number fields*, pages 217–242. Nagoya University, 1986.

[**Piz80**] A. Pizer. An Algorithm for Computing Modular Forms on $\Gamma_0(N)$. *Journal of Algebra*, 64:340–390, 1980.

135 SUPERSINGULAR DIVISORS ON MODULAR CURVES

Chapter 135
SUPERSINGULAR DIVISORS ON MODULAR CURVES

135.1 Introduction

This chapter is about how to use MAGMA to compute with the Hecke module $D(N,p)$ of divisors on the supersingular points on $X_0(N)$ in characteristic p.

Let p be a prime. A divisor on the supersingular points of $X_0(1)$ in characteristic p is a finite formal linear combination of supersingular j-invariants $j \in \mathbf{F}_{p^2}$. More generally, suppose N is an integer that is not divisible by p. The module $D(N,p)$ of divisors on the supersingular points of $X_0(N)$ in characteristic p is the free abelian group generated by isomorphism classes of pairs (E,C) where E is a supersingular elliptic curve over $\mathbf{F}_{p^2}$ and C is a cyclic subgroup of E of order N. (We call such a pair (E,C) an elliptic curve enhanced with level N structure.) The abelian group $D(N,p)$ of divisors is equipped in a natural way with an action of Hecke operators T_n.

The module of supersingular points is a special case of the Brandt module construction (see Chapter 134) because of the following equivalence between objects:

- pairs (E,C) as above,
- isomorphism classes of left ideals of an Eichler order of level N in the quaternion algebra over $\mathbf{Q}$ ramified at, and p and ∞,
- supersingular points on $X_0(N)/\mathbf{F}_p$ over $\overline{\mathbf{F}}_p$.

The supersingular points of $X_0(N)/\mathbf{F}_p$ correspond to singularities of the special fiber of a minimal model of $X_0(Np)$ at p. This special fiber, $X_0(Np)/\mathbf{F}_p$, is isomorphic to two copies of $X_0(N)/\mathbf{F}_p$ joined at the supersingular points as simple double points. The structure of the multiplicative part of the Jacobian $J_0(Np)$ at p is captured by the behavior of the supersingular points of $X_0(N)/\mathbf{F}_p$ (see Grothendieck [Gro72] and Deligne-Rapoport [DR73]). The system of Hecke operators on the supersingular divisor group gives a representation of the p-new subspace of modular forms of level Np. We therefore refer to the *level* of the supersingular module as Np, and use the term *auxiliary level* to refer to N.

There are many reasons why one might be interested in computing with $D(N,p)$. Foremost, $D(N,p)$ is isomorphic as a module over the Hecke algebra to a subspace of the modular forms for $\Gamma_0(N)$ of weight 2 and level Np (more precisely, $D(N,p)\otimes\mathbf{C}$ is isomorphic to the p-new subspace of $M_2(\Gamma_0(Np))$). If $N = 1,2,3,5,7,13$, then MAGMA computes $D(N,p)$ using the highly-efficient method of graphs, which for small q quickly produce very sparse matrices that represent Hecke operators T_q. There are also formulas of Gross, Kudla, Merel, and others that involve the explicit representation of an eigenform in terms of a

basis of supersingular j-invariants, and Stein has a formula, which involves $D(N,p)$, for orders of component groups of modular abelian varieties (see the `ComponentGroupOrder` command in the modular symbols package).

MAGMA computes the module of supersingular divisors using either the method of graphs of Mestre–Oesterlé when $N = 1, 2, 3, 5, 7, 13$ or Brandt modules in general. When it is applicable, the method of graphs is much faster (in MAGMA) than Brandt modules, but the Brandt modules method works in general.

The modular curve approach computes correspondences on the modular curves $X_0(N)$ by means of pre-computed models for the system of covering maps $X_0(N\ell) \to X_0(N)$. Such correspondences give rise to the Hecke operators T_ℓ as the adjacency matrices of the graphs of ℓ-isogenies of the basis of $D(N,p)$. For the alternative approach through Brandt modules, the reader should consult Chapter 134 and the articles of Pizer [Piz80] and Kohel [Koh01].

This package still has some unnecessary limitations. When using Brandt modules to compute the module of supersingular divisor in MAGMA, no facility is currently provided for describing the divisors in terms of supersingular elliptic curves. Also, it is currently only possible to work with the module of supersingular divisors over the integers.

135.1.1 Categories

Modules of supersingular points belong to the category `ModSS`, and the elements of these modules belong to `ModSSElt`.

135.1.2 Verbose Output

To set the verbosity level use the command `SetVerbose("SupersingularModule",n)`, where `n` is 0 (silent), 1 (verbose), or 2 (very verbose).

135.2 Creation Functions

135.2.1 Ambient Spaces

An ambient supersingular divisors module is specified by giving an integer N (the level) and a prime p (the characteristic).

`SupersingularModule(p,N : parameters)`

`Brandt` BOOLELT *Default :* `false`

The module M of supersingular points on $X_0(N)$ over $\overline{\mathbf{F}}_p$. Equivalently, this is the free abelian group on the supersingular elliptic curves in characteristic p enhanced with level N structure. We require that N and p are coprime.

`SupersingularModule(p)`

The Hecke module M of divisors of degree 0 on the supersingular points on $X_0(1)$ over $\overline{\mathbf{F}}_p$. Equivalently, this is the free abelian group on the supersingular j-invariants in characteristic p.

Example H135E1

```
> SupersingularModule(11);
Supersingular module associated to X_0(1)/GF(11) of dimension 2
> SupersingularModule(11,3);
Supersingular module associated to X_0(3)/GF(11) of dimension 4
> SupersingularModule(3,11);
Supersingular module associated to X_0(11)/GF(3) of dimension 2
```

The optional parameter `Brandt` forces computation of the supersingular module using quaternion arithmetic, even if this will be slower. (It's not clear why anyone would want to do use this parameter except to compute the same thing using two different algorithms.)

```
> SupersingularModule(97);
Supersingular module associated to X_0(1)/GF(97) of dimension 8
> SupersingularModule(97 : Brandt := true);
Supersingular module associated to X_0(1)/GF(97) of dimension 8
```

135.2.2 Elements

`M . i`

The ith basis element of the module M.

`M ! x`

The coercion of x into the module M.

Example H135E2

First we create the supersingular module attached to $p = 11$, $N = 3$. This is the free abelian group generated by the supersingular points on $X_0(3)$ in characteristic 11, equipped with the structure of module over the Hecke algebra.

```
> X := SupersingularModule(11,3);
> P := X.1;
> P;
(5, 5)
> Eltseq(P);
[ 1, 0, 0, 0 ]
> X![ 1, 0, 0, 0 ];
(5, 5)
```

Note that the module associated to $p = 3$, $N = 11$ is computed using Brandt matrices (since $X_0(11)$ has positive genus), so elements are printed in a less informative way.

```
> Z := SupersingularModule(3,11); Z;
Supersingular module associated to X_0(11)/GF(3) of dimension 2
> P := Z.1;
> P;
```

```
[E1]
> Eltseq(P);
[ 1, 0 ]
> Z![1,0];
[E1]
```

135.2.3 Subspaces

`CuspidalSubspace(M)`

The cuspidal submodule X of M. Thus X is the submodule of divisors of degree 0 on the supersingular points. It is "cuspidal" in the sense that $X \otimes \mathbf{Q}$ is isomorphic as a Hecke module to the space $S_2(\Gamma_0(Np); \mathbf{Q})^{p\text{-new}}$ of p-new cuspforms with Fourier coefficients in $\mathbf{Q}$. Explicitly, the cuspidal subspace is the subspace of elements such that the sum of the coefficients is 0 (i.e., the subspace of divisors of degree 0).

`EisensteinSubspace(M)`

The Eisenstein submodule of M, i.e., the orthogonal complement of the cuspidal subspace of M with respect to the monodromy pairing.

`OrthogonalComplement(M)`

The orthogonal complement of the module M in the ambient space with respect to the monodromy pairing.

`Kernel(I, M)`

The kernel of I on the module M. This is the subspace of M obtained by intersecting the kernels of the operators $f_n(T_{p_n})$, where I is a sequence $[\langle p_1, f_1(x)\rangle, ..., \langle p_n, f_n(x)\rangle]$ of pairs consisting of a prime number and a polynomial.

`Decomposition(M, n)`

Decomposition of the module M with respect to the Hecke operators $T_1, T_2, \ldots, T_n$.

Example H135E3

We compute bases for the cuspidal and eisenstein subspaces when $p = 11$ and $N = 1$.

```
> M := SupersingularModule(11); Basis(M);
[
    (1, 1),
    (0, 0)
]
> S := CuspidalSubspace(M);
> E := EisensteinSubspace(M);
> Basis(S);
[
    (1, 1) - (0, 0)
```

```
]
> Basis(E);
[
    3*(1, 1) + 2*(0, 0)
]
```

Next we compute the orthogonal complement of each subspace.

```
> Basis(OrthogonalComplement(E));
[
    (1, 1) - (0, 0)
]
> Basis(OrthogonalComplement(S));
[
    3*(1, 1) + 2*(0, 0)
]
> S eq OrthogonalComplement(E);
true
```

Note that the Hecke operator T_2 acts as -2 on the cuspidal subspace. Using the `Kernel` command, we compute the subspace of M on which T_2 acts as -2, and recover the cuspidal subspace.

```
> R<x> := PolynomialRing(Integers());
> I := [<2, x + 2>];
> K := Kernel(I,M);
> Basis(K);
[
    (1, 1) - (0, 0)
]
```

We can also compute the decomposition of M into submodules for the action of the first few Hecke operators (typically a few Hecke operators are enough to give a complete decomposition with respect to all Hecke operators).

```
> Decomposition(M,5);
[
    Supersingular module associated to X_0(1)/GF(11) of dimension 1,
    Supersingular module associated to X_0(1)/GF(11) of dimension 1
]
```

135.3 Basis

`Basis(M)`

A basis for M as a $\mathbf{Z}$-module.

135.4 Properties

`AuxiliaryLevel(M)`

The level of the module M, where the auxiliary level of `SupersingularModule(N,p)` is, by definition, N.

`BaseRing(M)`

The base ring of the module M. (Currently this is always $\mathbf{Z}$.)

`Degree(P)`

The sum of the coefficients of the module element P, where P is written with respect the basis of the ambient space of the parent of M.

`Dimension(M)`

The dimension of the module M.

`Eltseq(P)`

A sequence of integers that defines the module element P.

`Level(M)`

The level of the module M, where the level of `SupersingularModule(N,p)` is, by definition, Np.

`ModularEquation(M)`

The equation of $X_0(N)$ that we use when using the Mestre method to compute with the module M of supersingular points.

`Prime(M)`

The prime of the module M, where the prime of `SupersingularModule(N,p)` is, by definition, p.

Example H135E4

```
> M := SupersingularModule(3,11);
> AuxiliaryLevel(M);
11
> BaseRing(M);
Integer Ring
> Degree(M.1+7*M.2);
8
> Dimension(M);
2
> Eltseq(M.1+7*M.2);
[ 1, 7 ]
> Level(M);
```

```
33
> Prime(M);
3
> M := SupersingularModule(11,3); M;
Supersingular module associated to X_0(3)/GF(11) of dimension 4
> ModularEquation(M);
x*y + 8
```

135.5 Associated Spaces

`BrandtModule(M)`

The Brandt module associated to the supersingular module M.

`ModularSymbols(M : parameters)`

`Proof` BoolElt *Default :* `true`

The space of modular symbols corresponding to the supersingular module M.

`ModularSymbols(M, sign : parameters)`

`Proof` BoolElt *Default :* `true`

The $+1$ or -1 quotient of the space of modular symbols corresponding to the supersingular module M.

`RSpace(M)`

The $\mathbf{Z}$-module V underlying the supersingular module M along with an invertible map $V \to M$.

Example H135E5

We compute the Brandt module and modular symbols spaces associated to the supersingular module for $p = 3$, $N = 11$, and verify that T_2 has the same characteristic polynomial on each.

```
> M := SupersingularModule(3,11);
> B := BrandtModule(M); B;
Brandt module of level (3,11), dimension 2, and degree 2 over
Integer Ring
> MS := ModularSymbols(M); MS;
Modular symbols space for Gamma_0(33) of weight 2 and dimension 4
over Rational Field
> Factorization(CharacteristicPolynomial(HeckeOperator(B, 2)));
[
    <$.1 - 3, 1>,
    <$.1 - 1, 1>
]
> Factorization(CharacteristicPolynomial(HeckeOperator(MS, 2)));
```

```
[
    <$.1 - 3, 2>,
    <$.1 - 1, 2>
]
```

There is an associated Brandt module even if the underlying computations on M are done using the Mestre-Oesterle graph method.

```
> M := SupersingularModule(11);
> UsesMestre(M);
true
> B := BrandtModule(M); B;   // takes a while
Brandt module of level (11,1), dimension 2, and degree 2 over
Integer Ring
```

135.6 Predicates

`IsAmbientSpace(M)`

Returns `true` if the supersingular module M is the full module of supersingular points and not a submodule.

`M1 eq M2`

Returns `true` if the supersingular module M_1 equals M_2.

`P eq Q`

Returns `true` if the module element P equals Q.

`M1 subset M2`

Returns `true` if the supersingular module M_1 is a subset of M_2.

`UsesBrandt(M)`

Returns `true` if the underlying computations on the supersingular module M are done using Brandt modules.

`UsesMestre(M)`

Returns `true` if the underlying computations on the supersingular module M are done using the Mestre-Oesterle method of graphs.

Example H135E6

In this example we illustrate each of the above predicates.

```
> M := SupersingularModule(11);
> S := CuspidalSubspace(M);
> IsAmbientSpace(S);
false
> IsAmbientSpace(M);
true
> S eq M;
false
> S eq S;
true
> S.1 eq S.1;
true
> S.1 eq M.1 - M.2;
true
> S.1 eq M.1;
false
> S subset M;
true
> UsesBrandt(S);
false
> UsesMestre(S);
true
> M := SupersingularModule(11 : Brandt := true);
> UsesBrandt(M);
true
> UsesMestre(M);
false
```

135.7 Arithmetic

The standard arithmetic operations + and − and left scalar multiplication are defined for elements of supersingular modules. Also one can add and intersect two submodules of an ambient supersingular module.

`P + Q` `P - Q` `a * P`

`M1 + M2`

The submodule generated by all sums of elements in the supersingular modules M_1 and M_2.

`M1 meet M2`

The intersection of the supersingular modules M_1 and M_2.

Example H135E7

First we illustrate some arithmetic on elements.

```
> M := SupersingularModule(11);
> P := M.1; P;
(1, 1)
> Q := M.2; Q;
(0, 0)
> P + Q;
(1, 1) + (0, 0)
> P - Q;
(1, 1) - (0, 0)
> 3*P;
3*(1, 1)
```

Next we illustrate some arithmetic on submodules.

```
> E := EisensteinSubspace(M);
> S := CuspidalSubspace(M);
> V := E + S;
> V;
Supersingular module associated to X_0(1)/GF(11) of dimension 2
> Basis(V);
[
    (1, 1) + 4*(0, 0),
    5*(0, 0)
]
```

The index of $E + S$ in M is of interest since it is related to congruences between Eisenstein series and cusp forms. Upon converting each of E and S to an `RSpace`, we find that the index is 5.

```
> RSpace(M)/RSpace(V);
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 5 ]
```

The intersection of E and S is the zero module.

```
> W := E meet S; W;
Supersingular module associated to X_0(1)/GF(11) of dimension 0
> Basis(W);
[]
```

135.8 Operators

`HeckeOperator(M, n)`

Compute a matrix representing the nth Hecke operator T_n with respect to `Basis(M)` for the supersingular module M.

`AtkinLehnerOperator(M, q)`

A matrix representing the Atkin-Lehner involution W_q on the supersingular module M. The number q must equal either `Prime(M)` or `AuxiliaryLevel(M)`.

Example H135E8

In this example we observe that T_2 and W_3 have the same characteristic polynomial on $S_2(\Gamma_0(33))$ as on the cuspidal subspace of the supersingular module with $p = 11$, $N = 3$.

```
> SS := CuspidalSubspace(SupersingularModule(11, 3));
> MF := CuspForms(33, 2);
> Factorization(CharacteristicPolynomial(HeckeOperator(SS, 2)));
[
    <$.1 - 1, 1>,
    <$.1 + 2, 2>
]
> Factorization(CharacteristicPolynomial(HeckeOperator(MF, 2)));
[
    <$.1 - 1, 1>,
    <$.1 + 2, 2>
]
> Factorization(CharacteristicPolynomial(AtkinLehnerOperator(SS, 3)));
[
    <$.1 - 1, 1>,
    <$.1 + 1, 2>
]
> Factorization(CharacteristicPolynomial(AtkinLehnerOperator(MF, 3)));
[
    <$.1 - 1, 2>,
    <$.1 + 1, 1>
]
```

The supersingular module with $p = 3$ and $N = 11$ is isomorphic as a module to the subspace of 3-new cuspforms in $S_2(\Gamma_0(33))$.

```
> SS := CuspidalSubspace(SupersingularModule(3, 11));
> MF := NewSubspace(CuspForms(33,2), 3);
> HeckeOperator(SS, 17);
[-2]
> HeckeOperator(MF, 17);
[-2]
> AtkinLehnerOperator(SS, 11);
[-1]
```

```
> AtkinLehnerOperator(MF, 11);
[-1]
```

135.9 The Monodromy Pairing

The monodromy pairing is a nondegenerate Hecke equivariant integer valued pairing on the module of supersingular points. (Note that it is not perfect, in general.) This pairing is simple to describe in terms of the basis for the supersingular module given by the enhanced supersingular elliptic curves. If E and F are two supersingular elliptic curve in characteristic p equipped with level N structure, then E and F pair to 0 unless $E \cong F$, in which case they pair to half the number of automorphisms of E.

`MonodromyPairing(P, Q)`

The monodromy pairing of elements P and Q of a supersingular module.

`MonodromyWeights(M)`

The diagonal entries that define the monodromy pairing on the ambient space.

Example H135E9

We compute the Brandt module and modular symbols spaces associated to the supersingular module for $p = 3$, $N = 11$, and verify that T_2 acts in a compatible way on them.

```
> M := SupersingularModule(11);
> MonodromyWeights(M);
[ 2, 3 ]
> P := Basis(CuspidalSubspace(M))[1]; P;
(1, 1) - (0, 0)
> Q := Basis(EisensteinSubspace(M))[1]; Q;
3*(1, 1) + 2*(0, 0)
> Basis(M);
[
    (1, 1),
    (0, 0)
]
> MonodromyPairing(P,Q);
0
> MonodromyPairing(P,P);
5
```

135.10 Bibliography

[**DR73**] P. Deligne and M. Rapoport. Les schémas de modules de courbes elliptiques. In *Lecture Notes in Mathematics*, volume 349, pages 143–316. Springer-Verlag, 1973.

[**Gro72**] A. Grothendieck. *Groupes de monodromie en géométrie algébrique. I.* Springer-Verlag, Berlin, 1972. Séminaire de Géométrie Algébrique du Bois-Marie 1967–1969 (SGA 7 I), Dirigé par A. Grothendieck. Avec la collaboration de M. Raynaud et D. S. Rim, Lecture Notes in Mathematics, Vol. 288.

[**Koh01**] D. Kohel. Hecke module structure of quaternions. In K. Miyake, editor, *Class Field Theory – its Centenary and Prospect*, 2001.

[**Piz80**] A. Pizer. An Algorithm for Computing Modular Forms on $\Gamma_0(N)$. *Journal of Algebra*, 64:340–390, 1980.

136 MODULAR ABELIAN VARIETIES

Chapter 136
MODULAR ABELIAN VARIETIES

136.1 Introduction

This chapter is the reference for the modular abelian varieties package in MAGMA. A *modular abelian variety* is an abelian variety that is a quotient of the modular Jacobian $J_1(N)$, for some integer N. This package provides extensive functionality for computing with such abelian varieties, including functions for enumerating and decomposing modular abelian varieties, isomorphism testing, computing exact endomorphism and homomorphism rings, doing arithmetic with finite subgroups and computing information about torsion subgroups, special values of L-functions and Tamagawa numbers.

Essentially none of the algorithms in this package use explicit defining equations for varieties, and as such work in a great degree of generality. For example, many even make sense for Grothendieck motives attached to modular forms, and we have included the corresponding functionality, when it makes sense.

MAGMA V2.11 was the first release of the modular abelian varieties package. The major drawback of the current version is that complete decomposition into simples is only implemented over the rational numbers. Thus the interesting behavior over number fields, involving extra inner twists, which leads to $\mathbf{Q}$-curves and associated questions, is not available (much of it is implemented, but there are some fundamental theoretical obstructions to overcome).

Our philosophy for representing modular abelian varieties is perhaps different than what you might expect, so we describe how we view an abelian subvariety A over $\mathbf{Q}$ contained in the modular Jacobian $J_0(N)$. By the Abel-Jacobi theorem we may view $J_0(N)$ over the complex numbers as a complex vector space V modulo the lattice $H_1(J_0(N), \mathbf{Z}) = H_1(X_0(N), \mathbf{Z})$. An abelian subvariety $A \subset J_0(N)$ and the map $i : A \to J_0(N)$ is completely determined by giving the image of $H_1(A, \mathbf{Q})$ in the vector space $H_1(X_0(N), \mathbf{Q})$. At this point, it might appear that we have to compute lots of floating point numbers and approximate lattices in the complex numbers, but this is not the case. Instead, we use modular symbols to compute $H_1(X_0(N), \mathbf{Z})$ as an abstract abelian group, and use everything we can from the extensive theory of modular forms to compute things about the abelian varieties determined by subgroups of $H_1(X_0(N), \mathbf{Z})$ and other related abelian varieties. Note that even though we work with homology, which is associated to complex tori, the abelian variety A over $\mathbf{Q}$ is still determined by our defining data (a certain subgroup of $H_1(X_0(N), \mathbf{Z})$), and our algorithms can often take advantage of this.

136.1.1 Categories

Modular abelian varieties belong to the category `ModAbVar`, and the elements of modular abelian varieties belong to `ModAbVarElt`. The category `MapModAbVar` consists of homomorphisms between modular abelian varieties (sometimes only up to isogeny, i.e., with a denominator). Spaces of homomorphisms between modular abelian varieties form the category `HomModAbVar`. Finitely generated subgroups of modular abelian varieties form the category `ModAbVarSubGrp`. Homology of a modular abelian variety is in the category `ModAbVarHomol`. The L-series of modular abelian varieties are in `ModAbVarLSer`.

Example H136E1

We create an object of each category.

```
> A := JZero(11);
> Type(A);
ModAbVar
> Type(A!0);
ModAbVarElt
> Type(nIsogeny(A,2));
MapModAbVar
> Type(nTorsionSubgroup(A,2));
ModAbVarSubGrp
> Type(End(A));
HomModAbVar
> Type(Homology(A));
ModAbVarHomol
> Type(LSeries(A));
ModAbVarLSer
```

136.1.2 Verbose Output

The verbosity level is set using the command `SetVerbose("ModAbVar",n)`, where `n` is 0 (silent), 1 (verbose), or 2 (very verbose). The default verbose level is 0.

Two additional verbose levels are included in this package. Level 3 is exactly like level 1, except instead of displaying to the screen, verbose output is appended to the file `ModAbVar-verbose.log` in the directory that MAGMA was run from. Verbose level 4 is exactly like level 2, except verbose output is appended to `ModAbVar-verbose.log`. On a UNIX-like system, use the shell command `tail -f ModAbVar-verbose.log` to watch the verbose log in another terminal.

Example H136E2

Using `SetVerbose`, we get some information about what is happening during computations.

```
> SetVerbose("ModAbVar",1);   // some verbose output
> SetVerbose("ModAbVar",2);   // tons of verbose output
> SetVerbose("ModAbVar",3);   // some verbose output to ModAbVar-verbose.log
> SetVerbose("ModAbVar",4);   // tons of verbose output to ModAbVar-verbose.log
```

136.2 Creation and Basic Functions

The functions described below create modular abelian varieties, combine them together in various ways, and obtain simple information about them.

Modular abelian varieties are much less restricted than spaces of modular symbols as one can take arbitrary finite direct sums.

136.2.1 Creating the Modular Jacobian $J_0(N)$

The `JZero` command will create the Jacobian $J_0(N)$ of the modular curve $X_0(N)$ (which parameterizes isomorphism classes of pairs of elliptic curves and cyclic subgroups of order N). Higher weight motivic analogues of this Jacobian can be created. Computations can be carried out in the $+1$ or -1 quotient of homology for efficiency, though certain results will be off by factors of 2.

`JZero(N : parameters)`

`JZero(N, k : parameters)`

`Sign` RngIntElt *Default :* 0

Create the modular abelian variety $J_0(N)$ of level N and weight 2 or weight $k \geq 2$, i.e., the Jacobian of the modular curve $X_0(N)$. The parameter `Sign` determines whether computations will be carried out in the $+1$ or -1 quotient of homology.

Example H136E3

```
> JZero(23);
Modular abelian variety JZero(23) of dimension 2 and level 23 over Q
> JZero(23 : Sign := +1);
Modular abelian variety JZero(23) of dimension 2 and level 23 over Q
with sign 1
> JZero(23,4);
Modular motive JZero(23,4) of dimension 5 and level 23 over Q
> JZero(23,4 : Sign := -1);
Modular motive JZero(23,4) of dimension 5 and level 23 over Q with
sign -1
> JZero(389,2 : Sign := +1);
Modular abelian variety JZero(389) of dimension 32 and level 389
over Q with sign 1
```

136.2.2 Creating the Modular Jacobians $J_1(N)$ and $J_H(N)$

The `JOne` command creates the Jacobian of the modular curve $X_1(N)$ (which parameterizes isomorphism classes of pairs of elliptic curves and cyclic subgroups of order N). The command `Js` creates an abelian variety isogenous to $J_1(N)$; more precisely it is the product of abelian varieties $J_\varepsilon(N)$, where $J_\varepsilon(N)$ is the abelian variety attached to all modular forms whose character is a Galois conjugate of ε. Creating $J_s(N)$ is much faster than creating $J_1(N)$, since less time is spent finding the integral structure on homology.

The `JH` command creates the Jacobian $J_H(N)$ of the curve $X_H(N)$, which is the quotient of $X_1(N)$ by the subgroup H of the integers modulo N.

`JOne(N : parameters)`

`JOne(N, k : parameters)`

`Sign` RngIntElt *Default :* 0

Create the modular abelian variety $J_1(N)$ of level N and weight k, if given, or 2, i.e., the Jacobian of the modular curve $X_1(N)$. Note that creating and finding the integral structure on $J_s(N)$, which is isogenous to $J_1(N)$, is much faster. Computing with $J_1(N)$ may be expensive.

`Js(N : parameters)`

`Js(N, k : parameters)`

`Sign` RngIntElt *Default :* 0

A modular abelian variety that is $\mathbf{Q}$-isogenous to the weight k (or 2) version of $J_1(N)$. More precisely, the direct sum of the modular abelian varieties attached to modular symbols spaces with Nebentypus.

`JH(N, d : parameters)`

`JH(N, k, d : parameters)`

`Sign` RngIntElt *Default :* 0

Suppose H is some (cyclic) subgroup of $G = (\mathbf{Z}/N\mathbf{Z})^*$ such that G/H has order d. Create the modular abelian variety $J_H(N)$ of level N and weight k (or 2 if k not given), where $J_H(N)$ is *isogenous* to the Jacobian of the modular curve $X_H(N)$ associated to the subgroup of $\mathrm{SL}_2(\mathbf{Z})$ of matrices $[a, b; c, d]$ with c divisible by N and a in H modulo N. It is the product of modular symbols varieties $J(\epsilon)$ for all Dirichlet characters ϵ that are trivial on H.

The parameter `Sign` determines whether computations will be carried out in the $+1$ or -1 quotient of homology.

`JH(N, gens : parameters)`

`JH(N, k, gens : parameters)`

`Sign` RngIntElt *Default :* 0

Let H be the subgroup of $(\mathbf{Z}/N\mathbf{Z})^*$ generated by the sequence of integers, *gens*. Create the modular abelian variety $J_H(N)$ of level N and weight k (or 2 if k not given), where $J_H(N)$ is isogenous to the Jacobian of the modular curve $X_H(N)$ associated to the subgroup of $\mathrm{SL}_2(\mathbf{Z})$ of matrices $[a, b; c, d]$ with c divisible by N and a in H modulo N. It is the product of modular symbols variety $J(\epsilon)$ for all Dirichlet characters ϵ that are trivial on H.

The parameter `Sign` determines whether computations will be carried out in the $+1$ or -1 quotient of homology.

Example H136E4

```
> JOne(13);
Modular abelian variety JOne(13) of dimension 2 and level 13 over Q
> JOne(13,4);
Modular motive JOne(13,4) of dimension 15 and level 13 over Q
> JOne(13,4 : Sign := 1);
Modular motive JOne(13,4) of dimension 15 and level 13 over Q
> JH(13,6);
Modular abelian variety J_H(13) of dimension 2 and level 13 over Q
> JH(13,3);
Modular abelian variety J_H(13) of dimension 0 and level 13 over Q
> JH(13,[-1]);
Modular abelian variety J_H(13) of dimension 2 and level 13 over Q
> JOne(17);
Modular abelian variety JOne(17) of dimension 5 and level 17 over Q
> Js(17);
Modular abelian variety Js(17) of dimension 5 and level 17 over Q
> IsIsogenous(JOne(17),Js(17));
true
> Degree(NaturalMap(JOne(17),Js(17)));
16
> JH(17,2);
Modular abelian variety J_H(17) of dimension 1 and level 17 over Q
> JH(17,4);
Modular abelian variety J_H(17) of dimension 1 and level 17 over Q
> JH(17,8);
Modular abelian variety J_H(17) of dimension 5 and level 17 over Q
```

136.2.3 Abelian Varieties Attached to Modular Forms

The following commands create abelian varieties attached to spaces of modular forms, sequences of spaces of forms, newforms, and characters. If an input space of modular forms is not cuspidal, MAGMA automatically replaces it with its cuspidal subspace.

`ModularAbelianVariety(M : parameters)`

`Sign` RNGINTELT *Default :* 0

The abelian variety attached to the modular forms space M.

`ModularAbelianVariety(X : parameters)`

`Sign` RNGINTELT *Default :* 0

The abelian variety attached to the sequence X of modular forms spaces. This is the direct sum of the spaces attached to each element of the sequence.

`ModularAbelianVariety(eps : parameters)`

`ModularAbelianVariety(eps, k : parameters)`

`Sign` RNGINTELT *Default :* 0

The abelian variety associated to the dirichlet character ϵ. This corresponds to the space of modular forms of weight k and character any Galois conjugate of ϵ. We include all Galois conjugates in order to obtain an abelian variety that is defined over $\mathbf{Q}$.

`ModularAbelianVariety(f)`

The abelian variety attached to the newform f.

`Newform(A)`

A newform f such that the modular abelian variety A is isogenous to the newform abelian variety A_f. An error occurs if A is not attached to a newform.

Example H136E5

We first create the modular abelian variety attached to the spaces $S_2(\Gamma_0(11))$ and $S_2(\Gamma_1(13))$. This is the direct sum of $J_0(11)$ with $J_1(13)$.

```
> X := [ModularForms(11,2), ModularForms(Gamma1(13),2)];
> A := ModularAbelianVariety(X); A;
Modular abelian variety of dimension 3 and level 11*13 over Q
> IsIsomorphic(A, JZero(11)*JOne(13));
true Homomorphism N(1) from modular abelian variety of dimension
3 to JZero(11) x JOne(13) (not printing 6x6 matrix)
```

Next we create the modular abelian variety A attached to $S_2(\Gamma_1(17))$ along with $J_1(17)$ and $J_s(17)$. We then note that A is isomorphic to $J_1(17)$, but there is no reason that A should be isomorphic to $J_s(17)$ (they are probably only isogenous). This example illustrates the fact that the abelian variety computed by MAGMA attached to $S_2(\Gamma_1(17))$ is $J_1(17)$ rather than $J_s(17)$.

(Recall that $J_s(N)$ is a product of copies of abelian varieties corresponding to conjugacy classes of characters.)

```
> A := ModularAbelianVariety(ModularForms(Gamma1(17),2)); A;
Modular abelian variety of dimension 5 and level 17 over Q
> B := JOne(17); B;
Modular abelian variety JOne(17) of dimension 5 and level 17 over Q
> C := Js(17); C;
Modular abelian variety Js(17) of dimension 5 and level 17 over Q
> IsIsomorphic(A,B);
true Homomorphism from modular abelian variety of dimension 5 to
JOne(17) (not printing 10x10 matrix)
> Degree(NaturalMap(A,C));
16
```

Example H136E6

If ε is a Dirichlet character and $k \geq 2$ is an integer, let S be the space of modular forms with weight k and character a Galois conjugate of ε. The command `ModularAbelianVariety(eps,k)` computes the modular abelian variety attached to S.

```
> G<eps> := DirichletGroup(13,CyclotomicField(12));
> Order(eps^2);
6
> ModularAbelianVariety(eps^2);
Modular abelian variety of dimension 2 and level 13 over Q
> ModularAbelianVariety(eps,3);
Modular motive of dimension 4 and level 13 over Q
```

Next we compute the modular abelian variety attached to a newform in $S_2(\Gamma_1(25))$.

```
> S := CuspForms(Gamma1(25),2);
> N := Newforms(S);
> #N;
2
> f := N[1][1];
> PowerSeries(f,4);
q + a*q^2 + 1/1355*(941*a^7 + 4820*a^6 + 11150*a^5 + 11522*a^4 +
    3582*a^3 + 10041*a^2 + 24432*a - 5718)*q^3 + O(q^4)
> A_f := ModularAbelianVariety(f);
> A_f;
Modular abelian variety Af of dimension 8 and level 5^2 over Q
```

The abelian variety A_f also determines the newform:

```
> PowerSeries(Newform(A_f),4);
q + a*q^2 + 1/1355*(941*a^7 + 4820*a^6 + 11150*a^5 + 11522*a^4 +
    3582*a^3 + 10041*a^2 + 24432*a - 5718)*q^3 + O(q^4)
```

Example H136E7

The `Newform` command works even if A wasn't explicitly created using a newform.

```
> A := Decomposition(JZero(37))[1];
> Newform(A);
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + 6*q^6 - q^7 + O(q^8)
```

136.2.4 Abelian Varieties Attached to Modular Symbols

The commands below associate modular abelian varieties to spaces of modular symbols and to sequences of spaces of modular symbols. Conversely, they associate spaces of modular symbols to modular abelian varieties. If an input space of modular symbols is not cuspidal, it is replaced by its cuspidal subspace.

`ModularAbelianVariety(M)`

The abelian variety attached to the modular symbols space M.

`ModularAbelianVariety(X)`

The abelian variety attached to the sequence X of modular symbols spaces.

`ModularSymbols(A)`

A sequence of spaces of modular symbols associated to the abelian variety A.

Example H136E8

We create modular abelian varieties attached to several spaces of modular symbols.

```
> M := ModularSymbols(37,2);
> ModularAbelianVariety(M);
Modular abelian variety of dimension 2 and level 37 over Q
> M := ModularSymbols(Gamma1(17));
> ModularAbelianVariety(M);
Modular abelian variety of dimension 5 and level 17 over Q
```

Note that the sign of the space of modular symbols determines the sign of the corresponding abelian variety.

```
> M := ModularSymbols(Gamma1(17),2,+1);
> A := ModularAbelianVariety(M); A;
Modular abelian variety of dimension 5 and level 17 over Q with sign 1
```

We can also create an abelian variety attached to any sequence of modular symbols spaces.

```
> ModularAbelianVariety([ModularSymbols(11), ModularSymbols(Gamma1(13))]);
```

```
Modular abelian variety of dimension 3 and level 11*13 over Q
```

Conversely, there is a sequence of modular symbols spaces associated to any abelian variety defined over $\mathbf{Q}$. These need not be the same as the spaces used to define the modular abelian variety; instead they are what is used internally in computations on that abelian variety.

```
> ModularSymbols(JOne(13));
[
    Modular symbols space of level 13, weight 2, character $.1,
    and dimension 2 over Cyclotomic Field of order 6 and degree 2
]
> ModularSymbols(JZero(37));
[
    Modular symbols space for Gamma_0(37) of weight 2 and
    dimension 4 over Rational Field
]
> A := JOne(17);
> ModularSymbols(A);
[
    Modular symbols space for Gamma_0(17) of weight 2 and
    dimension 2 over Rational Field,
    Modular symbols space of level 17, weight 2, character $.1,
    and dimension 2 over Cyclotomic Field of order 8 and degree 4
]
```

136.2.5 Creation of Abelian Subvarieties

Suppose A is an abelian variety and V is a vector subspace of the rational homology $H_1(A, \mathbf{Q})$. Then it can be determined whether or not V is the rational homology of an abelian subvariety of A, and if so that abelian subvariety can be computed.

The other commands below are used to create the zero-dimensional abelian variety.

`DefinesAbelianSubvariety(A, V)`

Return `true` if and only if the subspace V of rational homology defines an abelian subvariety of the variety A. If so, also returns the abelian subvariety.

This intrinsic relies on knowing a complete decomposition of A as a product of simple abelian varieties (so it is currently restricted to abelian varieties for which such a decomposition can be computed in Magma, e.g., modular abelian varieties over $\mathbf{Q}$).

`ZeroModularAbelianVariety()`

The zero-dimensional abelian variety.

`ZeroModularAbelianVariety(k)`

The zero-dimensional abelian variety of weight k.

`ZeroSubvariety(A)`

Returns the zero subvariety of the abelian variety A.

Example H136E9

We define two subspaces of the rational homology of $J_0(33)$; one defines an abelian subvariety and the other does not.

```
> A := JZero(33);
> w3 := AtkinLehnerOperator(A,3);
> W := Kernel(Matrix(w3)+1);
> DefinesAbelianSubvariety(A,W);
true Modular abelian variety of dimension 1 and level 3*11 over Q
> V := RationalHomology(A);
> DefinesAbelianSubvariety(A,W + sub<V|[V.1]>);
false
```

We create several zero-dimensional abelian varieties.

```
> ZeroModularAbelianVariety();
Modular abelian variety ZERO of dimension 0 and level 1 over Q
> ZeroModularAbelianVariety(2);
Modular abelian variety ZERO of dimension 0 and level 1 over Q
> ZeroSubvariety(JZero(11));
Modular abelian variety ZERO of dimension 0 and level 11 over Q
```

136.2.6 Creation Using a Label

As a useful shorthand, it is sometimes possible to create modular abelian varieties by giving a short string. If the string contains a single integer N, e.g., 37, then the corresponding abelian variety is $J_0(N)$. If it is of the form `"<level>k<weight>"`, then it is the possibly motivic $J_0(N)$ of weight k. If it is of the form `"<level>k<weight><isogeny code>"`, where `<isogeny code>` is one of `"A"`, `"B"`, ... `"Z"`, `"AA"`, `"BB"`, ... `"ZZ"`, `"AAA"`, `"BBB"`, ... then the corresponding abelian variety is `JZero(N,k)(iso)`, where iso is a positive integer, and `"A"` corresponds to iso=1, `"Z"` to iso=26, `"AA"` to iso=27, `"ZZ"` to iso=52, `"AAA"` to iso=53, etc.

This labeling convention is the same as the one used for modular symbols, and extends the one used for Cremona's database of elliptic curves, except that Cremona's database contains some random scrambling for levels between 56 and 450. If the weight part of the label is omitted, the weight is assumed to be 2. To get the optimal quotient of $J_0(N)$ with Cremona label s, set the optional parameter `Cremona` equal to true.

`ModularAbelianVariety(s : parameters)`

`Sign`	RngIntElt	*Default :* 0
`Cremona`	BoolElt	*Default :* `false`

The abelian variety defined by the string s. The parameter `Sign` determines whether computations will be carried out in the $+1$ or -1 quotient of homology. If the parameter `Cremona` is set to `true`, the optimal quotient of $J_0(N)$ with Cremona label s will be returned.

Example H136E10

```
> ModularAbelianVariety("37");
Modular abelian variety 37 of dimension 2 and level 37 over Q
> ModularAbelianVariety("37A");
Modular abelian variety 37A of dimension 1 and level 37 over Q
> ModularAbelianVariety("11k4A");
Modular motive 11k4A of dimension 2 and level 11 over Q
> ModularAbelianVariety("65C");
Modular abelian variety 65C of dimension 2 and level 5*13 over Q
> ModularDegree(ModularAbelianVariety("56A"));
4
> ModularDegree(ModularAbelianVariety("56A" : Cremona := true));
2
```

136.2.7 Invariants

Commands are available which can retrieve the base ring, dimension, character of the defining modular form, a field of definition, the level, the sign, the weights, and a short name of a modular abelian variety.

`BaseRing(A)`

The ring that the modular abelian variety A is defined over.

`Dimension(A)`

The dimension of the modular abelian variety A.

`DirichletCharacter(A)`

If the modular abelian variety $A = A_f$ is attached to a newform, returns the Nebentypus character of f. Since f is only well-defined up to $\mathrm{Gal}(\overline{\mathbf{Q}}/\mathbf{Q})$ conjugacy, the character is also only well-defined up to $\mathrm{Gal}(\overline{\mathbf{Q}}/\mathbf{Q})$ conjugacy.

For information on Dirichlet characters, see 19.8.

`DirichletCharacters(A)`

List of all Dirichlet characters of spaces of modular symbols associated with the modular symbols abelian variety which parameterizes A.

`FieldOfDefinition(A)`

The best known field of definition of the modular abelian variety A.

`Level(A)`

An integer N such that the modular abelian variety A is a quotient of a power of $J_1(N)$. Note that N need not be minimal. It is determined by how A is explicitly represented as a quotient of modular Jacobians.

`Sign(A)`

The sign of the modular abelian variety A, which is either 0, -1, or $+1$. If the sign is $+1$ or -1, then only the corresponding complex-conjugation eigenspace of the homology of A is computed, so various computations will be off by a factor of 2.

`Weights(A)`

The set of weights of the modular abelian variety A. The weight need not be unique since direct sums of modular symbols spaces of different weights are allowed.

Example H136E11

We illustrate all the commands for $J_0(23)$.

```
> A := JZero(23);
> BaseRing(A);
Rational Field
> Dimension(A);
2
> DirichletCharacter(A);
1
> FieldOfDefinition(A);
Rational Field
> Level(A);
23
> Sign(A);
0
> Weights(A);
{ 2 }
```

Example H136E12

This is an example of a nontrivial Dirichlet character.

```
> eps := DirichletCharacter(JOne(23)(2)); eps;
$.1^2
> Order(eps);
11
```

Example H136E13

We illustrate the `Weights` command in several cases.

```
> Weights(JZero(11));
{ 2 }
> Weights(JZero(11,4));
{ 4 }
> Weights(JOne(13,3));
{ 3 }
> Weights(DirectSum(JZero(11),JOne(13,3)));
{ 2, 3 }
> Weights(DirectSum(JZero(11),JZero(13,3)));
{ 2 }
```

Example H136E14

We display a few fields of definition.

```
> FieldOfDefinition(JOne(13));
Rational Field
> FieldOfDefinition(BaseExtend(JZero(11),QuadraticField(7)));
Rational Field
> FieldOfDefinition(ChangeRing(JZero(11),GF(7)));
Finite field of size 7
```

Example H136E15

In the following example we quotient $J_0(11)$ out by a 5-torsion point. The resulting abelian variety might not be defined over $\mathbf{Q}$, and the `FieldOfDefinition` command currently plays it safe and returns $\overline{\mathbf{Q}}$.

```
> A := JZero(11);
> G := nTorsionSubgroup(A,5);
> H := Subgroup([G.1]);
> H;
Finitely generated subgroup of abelian variety with invariants [ 5 ]
> FieldOfDefinition(A/H);
Algebraically closed field with no variables
```

136.2.8 Conductor

Let A be a modular abelian variety over $\mathbf{Q}$. The conductor command computes the conductor of A by factoring A into newform abelian varieties A_f whose conductor is N^d, where N is the level of f and d is the dimension of A_f.

`Conductor(A)`

The conductor of the abelian variety A. We require that A is defined over $\mathbf{Q}$. When $A = A_f$ is attached to a newform of level N, then the conductor of A is N^d, where d is the dimension of A.

Example H136E16

```
> Factorization(Conductor(JZero(33)));
[ <3, 1>, <11, 3> ]
> Factorization(Conductor(JZero(11)^5));
[ <11, 5> ]
> Factorization(Conductor(OldSubvariety(JZero(46))));
[ <23, 4> ]
> Factorization(Conductor(JOne(25)));
[ <5, 24> ]
```

136.2.9 Number of Points

Given an abelian variety A over a field K a divisor and a multiple of $\#A(K)$ can be computed. When finite, the multiple of the number of rational points is computed using reduction mod primes up to 100. Currently the lower bound is nontrivial only when A is a quotient of $J_0(N)$.

`NumberOfRationalPoints(A)`

A divisor and a multiple of the cardinality of $A(K)$, where A is a modular abelian variety defined over a field K. If K is an abelian number field, then we assume the Birch and Swinnerton-Dyer conjecture.

`#A`

The cardinality of $A(K)$ where A is a modular abelian variety defined over the field K if an exact value for this cardinality is known.

Example H136E17

```
> #JZero(11);
5
> #JZero(23);
11
> NumberOfRationalPoints(JZero(37));
Infinity Infinity
> NumberOfRationalPoints(JOne(13));
1 19
> NumberOfRationalPoints(JOne(23));
1 408991
> Factorization(408991);
[ <11, 1>, <37181, 1> ]
> NumberOfRationalPoints(ModularAbelianVariety("43B"));
7 7
```

136.2.10 Inner Twists and Complex Multiplication

If f is a newform then an *inner twist* of f is a Dirichlet character χ such that the twist of f by χ equals a Galois conjugate of f, at least at Fourier coefficients whose subscript is coprime to some fixed integer. A *CM twist* is a nontrivial character χ such that f twisted by χ equals f, at least at Fourier coefficients whose subscript is coprime to some fixed integer. The commands below find the CM and inner twists of the newform corresponding to a newform abelian variety.

The optional parameter `Proof` to each command is by default `false`. If `true`, it uses a huge number of terms of q-expansions. If `false`, it uses far less (and is hence very quick), and in practice this should be fine. The computation of inner and CM twists is not provably correct, even if the `Proof := true` option is set. It's very likely to be correct when `Proof := true`, but not provably correct. (More precisely, MAGMA only checks that each twist is a twist to precision 10^{-5}, but does not prove that this precision is sufficient.)

`CMTwists(A : parameters)`

`Proof` BOOLELT *Default :* `false`

A sequence of the CM inner twist characters of the newform abelian variety $A = A_f$ that are defined over the base ring of A. To gain all CM twists, base extend to `AlgebraicClosure(RationalField())` first.

`InnerTwists(A : parameters)`

`Proof` BOOLELT *Default :* `false`

A sequence of the inner twist characters of the newform abelian variety $A = A_f$ that are defined over the base ring of A. To gain all inner twists, first base extend to `AlgebraicClosure(RationalField())`.

Example H136E18

We compute the inner twists for $J_1(13)$.

```
> A := JOne(13); A;
Modular abelian variety JOne(13) of dimension 2 and level 13 over Q
> CMTwists(A);
[]
> A2 := BaseExtend(A,AlgebraicClosure(RationalField()));
> CMTwists(A2);
[]
> InnerTwists(A2);
[
    1,
    $.1^5
]
> Parent($1[2]);
Group of Dirichlet characters of modulus 13 over Cyclotomic Field
of order 6 and degree 2
```

We compute the inner twists for the second newform factor of $J_1(23)$.

```
> A := Decomposition(JOne(23))[2]; A;
Modular abelian variety image(23A[2]) of dimension 10, level 23
and conductor 23^10 over Q
> InnerTwists(BaseExtend(A,AlgebraicClosure(RationalField())));
[
    1,
    $.1^20
]
```

The CM elliptic curve $J_0(32)$ has a nontrivial CM inner twist.

```
> A := JZero(32);
> InnerTwists(BaseExtend(A,AlgebraicClosure(RationalField())));
[
    1,
    $.1
]
> CMTwists(BaseExtend(A,AlgebraicClosure(RationalField())));
[
    $.1
]
```

Quotients of $J_0(N)$ can also have nontrivial inner twists, which are not CM twists.

```
> J := JZero(81);
> A := Decomposition(J)[1];
> InnerTwists(BaseExtend(A,AlgebraicClosure(RationalField())));
[
    1,
```

```
    $.1
]
> CMTwists(BaseExtend(A,AlgebraicClosure(RationalField())));
[]
> Newform(A);
q + a*q^2 + q^4 - a*q^5 + 2*q^7 + O(q^8)
```

Example H136E19

The following is an example of a 4-dimensional abelian variety $A = A_f$ in $J_0(512)$ that has four inner twists, none of which are CM twists. One can use this fact to prove that if a_p is a prime-indexed Fourier coefficient of f, then $a_p^2 \in \mathbf{Z}$. Thus no single a_p generates the degree 4 field generated by all a_n.

```
> J := JZero(512, 2, +1);
> A := Decomposition(J)[7]; A;
Modular abelian variety 512G of dimension 4, level 2^9 and
conductor 2^36 over Q with sign 1
> f := Newform(A); f;
q + 1/12*(a^3 - 30*a)*q^3 + 1/12*(-a^3 + 42*a)*q^5 +
     1/6*(-a^2 + 18)*q^7 + O(q^8)
> Coefficient(f,3)^2;
6
> Coefficient(f,5)^2;
12
> Coefficient(f,7)^2;
8
> Abar := BaseExtend(JZero(512,2,+1)(7),AlgebraicClosure(RationalField()));
> InnerTwists(Abar);
[
    1,
    $.1,
    $.2,
    $.1*$.2
]
> CMTwists(Abar);
[]
```

136.2.11 Predicates

Most of the predicates below work in full generality. The ones whose domain of applicability is somewhat limited are `IsIsomorphic`, `IsQuaternionic`, and `IsSelfDual`. In theory, it is possible to determine isomorphism for more general classes of modular abelian varieties, but this has not yet been implemented.

`CanDetermineIsomorphism(A, B)`

Return `true` if we can determine whether or not the modular abelian varieties A and B are isomorphic. If we can determine isomorphism, also returns `true` if A and B are isomorphic and an explicit isomorphism, or `false` if they are not isomorphic. If we can not determine isomorphism, also returns the reason why we cannot as a string. If A and B are simple and defined over $\mathbf{Q}$, then there is an algorithm to determine whether or not A and B are isomorphic; it has been implemented in most (but not all) cases in MAGMA. If one of A or B has simple factors of multiplicity one, then in principal it is possible, but the algorithm has not been programmed.

`HasMultiplicityOne(A)`

Return `true` if the simple factors of the modular abelian variety A appear with multiplicity one.

`IsAbelianVariety(A)`

Return `true` if A is an abelian variety, i.e., defined over a ring of characteristic 0 in which the conductor is invertible, or a finite field whose characteristic does not divide the conductor of A. For example, if A has positive dimension and is defined over $\mathbf{Z}$, then Raynaud's theorem implies that A is not an abelian variety.

`IsAttachedToModularSymbols(A)`

Return `true` if the underlying homology of the modular abelian variety A is being computed using a space of modular symbols. For example, this will be true for $J_0(N)$ and for newform abelian varieties.

`IsAttachedToNewform(A)`

Return `true` if the modular abelian variety A is isogenous to a newform abelian variety A_f. This intrinsic also returns the abelian variety A_f and an explicit isogeny from A_f to A.

`IsIsogenous(A, B)`

Return `true` if the modular abelian varieties A and B are isogenous. If this can *not* be determined, an error occurs. It is always possible to determine whether or not A and B are isogenous when both are defined over $\mathbf{Q}$.

`IsIsomorphic(A, B)`

Return `true` if the modular abelian varieties A and B are isomorphic. If so, also returns an explicit isomorphism. This command will work if A and B are defined over $\mathbf{Q}$ and the simple factors occur with multiplicity one, and may work otherwise, but an error may occur in the general case. Use the command `CanDetermineIsomorphism` to avoid getting an error.

`IsOnlyMotivic(A)`

Return `true` if any of the modular forms attached to the modular abelian variety A have weight bigger than 2.

`IsQuaternionic(A)`

Return `true` if and only if some simple factor of the modular abelian variety A over the base ring has quaternionic multiplication.

`IsSelfDual(A)`

Return `true` if the modular abelian variety A is known to be isomorphic to its dual. An error occurs if MAGMA is unable to decide.

`IsSimple(A)`

Return `true` if and only if the modular abelian variety A has no proper abelian subvarieties over `BaseRing(A)`.

Example H136E20

We test whether a few objects are actually abelian varieties.

```
> A := JZero(11);
> A11 := ChangeRing(A,GF(11));
> IsAbelianVariety(A11);
false
> AZ := ChangeRing(A,Integers());
> IsAbelianVariety(AZ);
false
> A3 := ChangeRing(A,pAdicRing(3));
> IsAbelianVariety(A3);
true
```

Example H136E21

A modular motive is sometimes also an abelian variety, but only over the complex numbers.

```
> A := JZero(11,4); A;
Modular motive JZero(11,4) of dimension 2 and level 11 over Q
> IsAbelianVariety(A);
false
> IsOnlyMotivic(A);
true
> IsAbelianVariety(BaseExtend(A,ComplexField()));
true
```

Example H136E22

Abelian varieties $J_0(N)$ and $J_s(N)$ are attached to modular symbols, as are newform abelian varieties.

```
> J := JZero(37);
> IsAttachedToModularSymbols(J);
true
> A := Decomposition(J)[1];
> IsAttachedToModularSymbols(A);
false
> t, Af := IsAttachedToNewform(A);
> IsAttachedToModularSymbols(Af);
true
> IsIsomorphic(A,Af);
true Homomorphism from 37A to 37A given on integral homology by:
[1 0]
[0 1]
> IsAttachedToModularSymbols(Js(17));
true
> IsAttachedToModularSymbols(JOne(17));
false
```

Example H136E23

We test isogeny between a few abelian varieties.

```
> IsIsogenous(JZero(11),JOne(11));
true
> IsIsogenous(JZero(11)*JZero(11),JZero(22));
true
> IsIsogenous(JZero(11)*JZero(11),JZero(33));
false
> IsIsogenous(JZero(11),JZero(14));
false
> IsIsogenous(JZero(11)^2,JZero(22));
```

```
true
> A := JZero(37)(2); B := JOne(13);
> IsIsogenous(A*B*A, A*A*B);
true
> A := JZero(43);
> G := RationalCuspidalSubgroup(A);
> IsIsogenous(A,A/G);
true
```

Next we test isomorphism between some abelian varieties.

```
> A := JZero(43);
> G := RationalCuspidalSubgroup(A);
> B := A/G;
> CanDetermineIsomorphism(A,B);
true false
> IsIsomorphic(A,B);
false
> IsIsomorphic(JZero(11),JOne(11));
false
> IsIsomorphic(JZero(13),JOne(13));
false
> IsIsomorphic(Js(13),JOne(13));
true Homomorphism from Js(13) to JOne(13) given on integral
homology by:
[ 0  0 -1  0]
[ 0  0  0 -1]
[ 1  0 -1  0]
[ 0  1  0 -1]
> CanDetermineIsomorphism(Js(17),JOne(17));
false All tests failed to decide whether A and B are isomorphic.
```

Example H136E24

We test whether certain Jacobians have simple factors with multiplicity one.

```
> HasMultiplicityOne(JZero(43));
true
> HasMultiplicityOne(JZero(33));
false
> Decomposition(JZero(33));
[
    Modular abelian variety 33A of dimension 1, level 3*11 and
    conductor 3*11 over Q,
    Modular abelian variety N(11,33,1)(11A) of dimension 1, level
    3*11 and conductor 11 over Q,
    Modular abelian variety N(11,33,3)(11A) of dimension 1, level
    3*11 and conductor 11 over Q
```

```
]
```

Example H136E25

We give an example of a non-quaternionic surface.

```
> IsQuaternionic(JOne(13));
false
```

Example H136E26

Jacobians are self dual, and there is a surface of level 43 that is self dual and a surface of level 69 that is not.

```
> IsSelfDual(JOne(13));
true
> IsSelfDual(JZero(69)(2));
false
```

The surface A below is isomorphic to its dual. The natural polarization has kernel that is the kernel of multiplication by 2.

```
> A  := JZero(43)(2);
> A;
Modular abelian variety 43B of dimension 2, level 43 and
conductor 43^2 over Q
> IsSelfDual(A);
true
> phi := ModularPolarization(A);
> Invariants(Kernel(phi));
[ 2, 2, 2, 2 ]
```

Example H136E27

We test a few abelian varieties for simplicity.

```
> IsSimple(JOne(25));
false
> IsSimple(JOne(13));
true
> IsSimple(JZero(11)^10);
false
> IsSimple(NewSubvariety(JZero(100)));
true
```

136.2.12 Equality and Inclusion Testing

These functions test whether two abelian varieties are exactly equal or if one is a subset of another.

`A eq B`

Return `true` if the modular abelian varieties A and B are equal.

`A subset B`

Return `true` if the modular abelian variety A is a subset of the modular abelian variety B.

Example H136E28

This example illustrates that taking the direct product of abelian varieties is not commutative, as measured by equality (though it is as measured by isomorphism).

```
> A := JZero(11);
> B := JZero(14);
> A*B eq A*B;
true
> A*B eq B*A;
false
> IsIsomorphic(A*B, B*A);
true Homomorphism N(1) from JZero(11) x JZero(14) to JZero(14) x JZero(11)
given on integral homology by:
[0 0 1 0]
[0 0 0 1]
[1 0 0 0]
[0 1 0 0]
```

Example H136E29

The first inclusion below is as expected, but the second non-inclusion might be surprising. We do not consider $J_0(11)$ as a subset of $J_0(22)$, even though there is an injective map from one to the other, since to be a subset is much stronger than just the existence of an inclusion map.

```
> JZero(37) subset JZero(37);
true
> JZero(11) subset JZero(22);
false
> IsInjective(NaturalMap(JZero(11),JZero(22)));
true
```

136.2.13 Modular Embedding and Parameterization

Every modular abelian variety A is equipped with a modular parameterization and a modular embedding. The modular parameterization is a surjective homomorphism from a modular symbols abelian variety, such as $J_0(N)$. The modular embedding is a homomorphism to a modular symbols abelian variety, which is only guaranteed to be *injective in the category of abelian varieties up to isogeny*, i.e., to have finite kernel as a morphism of abelian varieties. The structure of these two homomorphisms is extremely important as they completely define A.

`CommonModularStructure(X)`

This intrinsic finds modular abelian varieties J_e and J_p associated to modular symbols and returns a list of finite-kernel maps from the abelian varieties in the sequence X to J_e and a list of modular parameterizations from J_p to the abelian varieties in X.

`ModularEmbedding(A)`

A morphism with finite kernel from the modular abelian variety A to a modular abelian variety attached to modular symbols. This is only guaranteed to be an embedding in the category of abelian varieties up to isogeny.

`ModularParameterization(A)`

A surjective morphism to the modular abelian variety A from an abelian variety attached to modular symbols.

Example H136E30

```
> X := [JZero(11),ModularAbelianVariety("37B")];
> CommonModularStructure(X);
[*
Homomorphism from JZero(11) to JZero(11) x JZero(37) given on integral
homology by:
[1 0 0 0 0 0]
[0 1 0 0 0 0],
Homomorphism from 37B to JZero(11) x JZero(37) given on integral
homology by:
[0 0 1 1 1 0]
[0 0 0 0 0 1]
*] [*
Homomorphism from JZero(11) x JZero(37) to JZero(11) (not printing 6x2
matrix),
Homomorphism from JZero(11) x JZero(37) to 37B (not printing 6x2
matrix)
*]
```

Example H136E31

This example illustrates that the modular "embedding" need only be an embedding in the category of abelian varieties up to isogeny.

```
> A := JZero(37)(1);
> x := A![1/2,1];
> B := A/Subgroup([x]);
> e := ModularEmbedding(B);
> e;
Homomorphism from modular abelian variety of dimension 1 to
JZero(37)_Qbar given on integral homology by:
[ 1 -1  1  0]
[ 2 -2 -2  2]
> IsInjective(e);
false
```

Moreover, the modular parameterization is surjective, but it need be optimal (have connected kernel).

```
> pi := ModularParameterization(B);
> IsSurjective(pi);
true
> ComponentGroupOfKernel(pi);
Finitely generated subgroup of abelian variety with invariants [ 2 ]
> IsOptimal(pi);
false
```

136.2.14 Coercion

Coercion can be used to create points on modular abelian varieties from vectors on a basis of integral homology, from other elements of modular abelian varieties, or from modular symbols. See the examples below, especially the last one, to understand some of the subtleties of coercion that arise because we view an abelian variety as a vector space modulo a lattice, and that lattice can be embedded in any way in $\mathbf{Q}^n$.

`A ! x`

Coerce x into the modular abelian variety A. The argument x can be an element of a modular abelian variety, the integer 0, a sequence like that obtained by from `Eltseq` of element of A (i.e., a linear combination of *integral* homology), a vector on the basis for *rational* homology, or a tuple of the form $\langle P(X,Y), [u,v]\rangle$ defining a modular symbol.

Example H136E32

If you coerce a *sequence* of rationals or reals into an abelian variety A, then MAGMA computes the corresponding linear combination of a basis of integral homology and returns the point it defines. The sequence must have length the rank of the integral homology.

```
> JZero(11)![1/2,1/5];
Element of abelian variety defined by [1/2 1/5] modulo homology
```

If you coerce exactly two cusps (or extended reals) into A, then MAGMA computes the point corresponding to that modular symbol.

```
> JZero(11)![Cusps()|1/2,1/5];
0
> JZero(11)![Sqrt(2),0];
Element of abelian variety defined by
[1.41421356237309504880168872419 8] modulo homology
> JZero(11)![Cusps()|0,Infinity()];     // cusps
Element of abelian variety defined by [0 1/5] modulo homology
> JZero(11)![0,Infinity()];             // extended reals
Element of abelian variety defined by [0 1/5] modulo homology
```

Coercion of modular symbols also works for higher weight.

```
> JZero(11,4)![0,Infinity()];
Element of abelian variety defined by [-4/61 5/61 1/61 -1/61]
modulo homology
> R<x,y> := PolynomialRing(RationalField(),2);
> JZero(11,4)!<x^2,[0,Infinity()]>;
Element of abelian variety defined by [-4/61 5/61 1/61 -1/61]
modulo homology
> JZero(11,4)!<y^2,[0,Infinity()]>;
Element of abelian variety defined by [44/61 -55/61 -11/61 11/61]
modulo homology
```

You can also coerce elements from abelian subvarieties into an ambient abelian variety.

```
> J := JZero(37); A := Decomposition(J)[1];
> x := A![1/5,0];
> Parent(x);
Modular abelian variety 37A of dimension 1, level 37 and
conductor 37 over Q
> x in J;
false
> y := J!x; y;
Element of abelian variety defined by [1/5 -1/5 1/5 0] modulo
homology
> y in J;
true
> Parent(y);
```

```
Modular abelian variety JZero(37) of dimension 2 and level 37 over Q
```

Coercion also provides an easy way to create the 0 element.

```
> JZero(37)!0;
0
```

Example H136E33

The following example illustrates the subtlety of coercion when the element being coerced in is a vector instead of a sequence. We create the quotient of $J_0(11)$ by a cyclic subgroup of order 10. The lattice that defines $J_0(11)$ is $\mathbf{Z} \times \mathbf{Z}$, but the lattice that defines this quotient is $(1/10)\mathbf{Z} \times \mathbf{Z}$. Thus the natural quotient map on vector spaces is defined by the identity matrix. On the other hand, the matrix of the quotient with respect to a basis for integral homology has determinant 10.

```
> A := JZero(11);
> x := A![1/10,0]; x;
Element of abelian variety defined by [1/10 0] modulo homology
> Order(x);
10
> B,pi := A/Subgroup([x]);
> B;
Modular abelian variety of dimension 1 and level 11 over Qbar
> pi;
Homomorphism from JZero(11)_Qbar to modular abelian variety of
dimension 1 given on integral homology by:
[10  0]
[ 0  1]
> Matrix(pi);
[1 0]
[0 1]
> IntegralMatrix(pi);
[10  0]
[ 0  1]
> base := Basis(IntegralHomology(B)); base;
[
    (1/10    0),
    (0 1)
]
```

If we coerce in the sequence [1/10,0] we get the point in B that is represented by 1/10th of the first generator for homology. If we coerce in the vector $(1/10, 0)$, we instead get the element of B represented by that element of the rational homology, which is 0, since the lattice that defines B is embedded in such a way that it contains $(1/10, 0)$.

```
> y := B![1/10,0]; y;
Element of abelian variety defined by [1/10 0] modulo homology
> Order(y);
10
```

```
> z := B!base[1]; z;
0
```

136.2.15 Modular Symbols to Homology

Modular symbols determine elements of the rational homology of $J_0(N)$, $J_1(N)$, etc., and hence of arbitrary modular abelian varieties, by using the modular parameterization. The commands below convert from modular symbols, which are represented in various ways, to vectors on the basis for rational or integral homology.

`ModularSymbolToIntegralHomology(A, x)`

The element of integral homology of the abelian variety A, naturally associated to the (formal) modular symbol $x = P(X,Y)\{\alpha,\beta\}$, where α, β are in $P^1(\mathbf{Q})$ and P is a homogeneous polynomial of degree 2. The returned vector is written with respect to the basis of integral homology. The argument x may be given as a sequence $[\alpha,\beta]$ or as a tuple $\langle P(X,Y),[\alpha,\beta]\rangle$ where α and β are in `Cusps()`.

`ModularSymbolToRationalHomology(A, x)`

The element of rational homology of the abelian variety A naturally associated to the (formal) modular symbol $x = P(X,Y)\{\alpha,\beta\}$, where α, β are in $P^1(\mathbf{Q})$ and P is a homogeneous polynomial of degree 2. The returned vector is written with respect to the basis of rational homology. The argument x may be given as a modular symbol, a sequence $[\alpha,\beta]$ or as a tuple $\langle P(X,Y),[\alpha,\beta]\rangle$ where α and β are in `Cusps()`.

Example H136E34

```
> A := JZero(11);
> x := ModularSymbolToIntegralHomology(A,[0,Infinity()]); x;
(  0 1/5)
> z := A!x; z;
Element of abelian variety defined by [0 1/5] modulo homology
> Order(z);
5
> A := JZero(47);
> x := ModularSymbolToIntegralHomology(A,[0,Infinity()]); x;
(-1/23  3/23 -4/23  4/23 -3/23  1/23  2/23  8/23)
> z := A!x;
> Order(z);
23
```

Example H136E35

```
> J := JZero(11,4);
> IntegralHomology(J);
Lattice of rank 4 and degree 4
Basis:
( 3  1 -1 -1)
( 1 -1 -3  1)
( 2 -2  2  2)
( 2 -2  2 -2)
Basis Denominator: 8
> ModularSymbolToIntegralHomology(J,[0,Infinity()]);
(-4/61  5/61  1/61 -1/61)
```

Notice that for weight greater than 2 the homogeneous polynomial part of the modular symbol may be omitted. If so, it defaults to x^{k-2}.

```
> R<x,y> := PolynomialRing(RationalField(),2);
> ModularSymbolToIntegralHomology(J,<x^2,[0,Infinity()]>);
(-4/61  5/61  1/61 -1/61)
> ModularSymbolToIntegralHomology(J,<y^2,[0,Infinity()]>);
( 44/61 -55/61 -11/61  11/61)
```

The result of coercion to rational homology is different because it is written in terms of the basis for rational homology instead of the basis for integral homology, and in this example the two basis differ.

```
> ModularSymbolToRationalHomology(J,[0,Infinity()]);
( -7/488  -9/488 -11/488  13/488)
```

Coercion is also a way to define torsion points on abelian varieties.

```
> JZero(37)![1/5,0,0,0];
Element of abelian variety defined by [1/5 0 0 0] modulo homology
```

136.2.16 Embeddings

The `Embeddings` command contains a list of embeddings (up to isogeny) from A to other abelian varieties. The `AssertEmbedding` command allows you to add an embedding to the beginning of the list. The embeddings are used for computing intersections, sums, etc., with the embedding at the front of the list having highest priority.

`Embeddings(A)`

A list of morphisms from the modular abelian variety A into abelian varieties, which are used in making sense of intersections, sums, etc. The embeddings at the beginning of the list take precedence over those that occur later. Note that these maps might not really be injective; e.g., the modular embedding, which need only be injective on homology, is at the end of this list.

`AssertEmbedding(~A, phi)`

Place the homomorphism ϕ of abelian varieties at the beginning of `Embeddings(A)` where A is a modular abelian variety. The morphism ϕ must have finite kernel.

Example H136E36

Every modular abelian variety comes equipped with at least one embedding, the "modular embedding".

```
> Embeddings(JZero(11));
[*
Homomorphism from JZero(11) to JZero(11) given on integral homology by:
[1 0]
[0 1]
*]
> A := JZero(37)(1);
> Embeddings(A);
[*
Homomorphism from 37A to JZero(37) given on integral homology by:
[ 1 -1  1  0]
[ 1 -1 -1  1]
*]
```

We add another embedding to the list of embeddings for A.

```
> phi := NaturalMap(A,JZero(37*2));
> AssertEmbedding(~A,phi);
> Embeddings(A);
[*
Homomorphism N(1) from 37A to JZero(74) given on integral homology
by:
[-1  1 -1  0  2 -1  1  0 -2  1 -2  2 -1  2 -1  1]
[-1  0  0  0  2 -1  1 -2  0  0 -1  1 -1  2  1 -1],
Homomorphism from 37A to JZero(37) given on integral homology by:
[ 1 -1  1  0]
[ 1 -1 -1  1]
*]
```

The following intersection would not make sense if we hadn't chosen an embedding of A into $J_0(74)$.

```
> B := Codomain(phi)(1); B;
Modular abelian variety 74A of dimension 2, level 2*37 and
conductor 2^2*37^2 over Q
> #(A meet B);
1
```

136.2.17 Base Change

The `BaseExtend` and `ChangeRing` commands allow you to change the base ring of a modular abelian variety. The `BaseExtend` command is the same `ChangeRing`, but is more restrictive. For example, if A is over $\mathbf{Q}$ then `BaseExtend(A,GF(2))` is not allowed, but `ChangeRing(A,GF(2))` is.

Abelian varieties can have their base ring set to a finite field, but there is very little that is implemented for abelian varieties over finite fields. Computing the number of points is implemented, but creation of actual points or homomorphisms is not.

`CanChangeRing(A, R)`

Return `true` if it is possible to change the base ring of the modular abelian variety A to the ring R, and the modular abelian variety over R when possible.

`ChangeRing(A, R)`

Return the modular abelian variety obtained from the modular abelian variety A having base ring R.

`BaseExtend(A, R)`

Extend the base ring of the modular abelian variety A to the ring R, if possible. This is a more restrictive version of `ChangeRing`.

Example H136E37

We consider $J_1(13)$ over many fields and rings.

```
> A := JOne(13);
> BaseExtend(A,CyclotomicField(7));
Modular abelian variety JOne(13) of dimension 2 and level 13 over
Q(zeta_7)
> BaseExtend(A,AlgebraicClosure(RationalField()));
Modular abelian variety JOne(13)_Qbar of dimension 2 and level 13
over Qbar
> BaseExtend(A,RealField());
Modular abelian variety JOne(13)_R of dimension 2 and level 13 over R
> BaseExtend(A,ComplexField());
Modular abelian variety JOne(13)_C of dimension 2 and level 13 over C
> ChangeRing(A,GF(3));
Modular abelian variety JOne(13)_GF(3) of dimension 2 and level 13
over GF(3)
> #ChangeRing(A,GF(3));
19 19
> B := ChangeRing(A,GF(13)); B;
Modular abelian variety JOne(13)_GF(13) of dimension 2 and level 13
over GF(13)
> IsAbelianVariety(B);
false
> ChangeRing(A,Integers());
```

```
Modular abelian variety JOne(13)_Z of dimension 2 and level 13 over
Z
> ChangeRing(A,PolynomialRing(RationalField(),10));
Modular abelian variety JOne(13) of dimension 2 and level 13 over
Polynomial ring of rank 10 over Rational Field
Lexicographical Order
Variables: $.1, $.2, $.3, $.4, $.5, $.6, $.7, $.8, $.9, $.10
```

136.2.18 Additional Examples

Example H136E38

The simplest abelian variety is an abelian variety of dimension 0, i.e., a point.

```
> A := ZeroModularAbelianVariety(); A;
Modular abelian variety ZERO of dimension 0 and level 1 over Q
```

We create the Jacobian $J_0(22)$ of the modular curve $X_0(22)$ as follows:

```
> J := JZero(22); J;
Modular abelian variety JZero(22) of dimension 2 and level 2*11 over Q
```

Notice that A is a subset of $J_0(22)$.

```
> A subset J;
true
```

We can also create the higher weight analogues $J_0(N, k)$ of $J_0(N)$, which are motives defined over $\mathbf{Q}$. Many computations that make sense for $J_0(N)$ also make sense for these higher weight analogues.

```
> J4 := JZero(22,4); J4;
Modular motive JZero(22,4) of dimension 7 and level 2*11 over Q
> IsOnlyMotivic(J4);
true
```

One can also create $J_1(N)$:

```
> JOne(22);
Modular abelian variety JOne(22) of dimension 6 and level 2*11 over Q
```

For efficiency purposes, it is often much quicker to do computations working only with the $+1$ quotient of $H_1(J_0(N), \mathbf{Z})$, or using only the -1 quotient. These computations will be off by powers of 2, or subgroups will be halved and off by a power of 2. Nonetheless, if you know what you are doing, such computations can be very useful. Create $J_0(N)$, but working only with the $+1$ quotient of homology as follows:

```
> Jplus := JZero(22,2,+1); Jplus;
Modular abelian variety JZero(22) of dimension 2 and level 2*11 over Q
with sign 1
> Sign(Jplus);
1
```

```
> Sign(JZero(22));
0
```

Notice that the sign is printed out when it is 1 or -1. Also, sign 0 is shorthand for "no sign", i.e., working with the full homology.

Example H136E39

Let $\varepsilon : (\mathbf{Z}/N\mathbf{Z})^* \to \mathbf{Q}(\zeta_n)^*$ be Dirichlet character. The command `ModularAbelianVariety(eps)` creates the modular abelian variety *over* $\mathbf{Q}$ corresponding to the cusp forms with Dirichlet character any Galois conjugate of ε.

```
> G<eps> := DirichletGroup(22,CyclotomicField(EulerPhi(22)));
> Order(eps);
10
> Conductor(eps);
11
> A := ModularAbelianVariety(eps); A;
Modular abelian variety of dimension 0 and level 2*11 over Q
> A := ModularAbelianVariety(eps^2); A;
Modular abelian variety of dimension 4 and level 2*11 over Q
```

Example H136E40

Let H be a subgroup of $G = (\mathbf{Z}/N\mathbf{Z})^*$. The group $(\mathbf{Z}/N\mathbf{Z})^*$ acts by diamond bracket operators as a group of automorphisms on $X_1(N)$, and the modular curve $X_H(N)$ is the quotient of $X_1(N)$ by the action of H. Thus if $H = 1$, then $X_H(N) = X_1(N)$, and if $H = G$, then $X_H(N) = X_0(N)$. The command `JH(N,d)` creates an abelian variety *isogenous to* the Jacobian of $X_H(N)$, where d is the index of H in G (thus $d = 1$ corresponds to $J_0(N)$). A weight k and sign (either 0 or ± 1) can be provided. More precisely, `JH(N,d)` is the product of $J(\varepsilon)$, where ε varies over Dirichlet characters such that $\varepsilon(H) = \{1\}$.

```
> JH(22,1);
Modular abelian variety JZero(22) of dimension 2 and level 2*11 over Q
> JH(22,10);
Modular abelian variety Js(22) of dimension 6 and level 2*11 over Q
> JH(22,2);
Modular abelian variety J_H(22) of dimension 2 and level 2*11 over Q
```

As a shortcut, the command `Js(N)` creates a modular abelian variety that is isogenous to $J_1(N)$. Thus `Js(N)` is the same as `JH(N,d)`, where d is the order of $(\mathbf{Z}/N\mathbf{Z})^*$.

```
> Js(22);
Modular abelian variety Js(22) of dimension 6 and level 2*11 over Q
> JH(22,10) eq Js(22);
true
```

Example H136E41

We can also create modular abelian varieties attached to spaces of modular forms and spaces of modular symbols:

```
> ModularAbelianVariety(ModularForms(22));
Modular abelian variety of dimension 2 and level 2*11 over Q
> ModularAbelianVariety(ModularSymbols(22));
Modular abelian variety of dimension 2 and level 2*11 over Q
```

Example H136E42

Here is another example, in which the space of modular forms is on $\Gamma_1(25)$.

```
> M := ModularForms(Gamma1(25)); M;
Space of modular forms on Gamma_1(25) of weight 2 and dimension
39 over Integer Ring.
> S := CuspidalSubspace(M); S;
Space of modular forms on Gamma_1(25) of weight 2 and dimension
12 over Integer Ring.
> A := ModularAbelianVariety(S); A;
Modular abelian variety of dimension 12 and level 5^2 over Q
```

Example H136E43

We can also construct abelian varieties attached to newforms.

```
> S := CuspForms(43);
> N := Newforms(S); N;
[* [*
q - 2*q^2 - 2*q^3 + 2*q^4 - 4*q^5 + 4*q^6 + O(q^8)
*], [*
q + a*q^2 - a*q^3 + (-a + 2)*q^5 - 2*q^6 + (a - 2)*q^7 + O(q^8),
q + b*q^2 - b*q^3 + (-b + 2)*q^5 - 2*q^6 + (b - 2)*q^7 + O(q^8)
*] *]
> f := N[2][1]; f;
q + a*q^2 - a*q^3 + (-a + 2)*q^5 - 2*q^6 + (a - 2)*q^7 + O(q^8)
> A := ModularAbelianVariety(f); A;
Modular abelian variety Af of dimension 2 and level 43 over Q
> Newform(A);
q + a*q^2 - a*q^3 + (-a + 2)*q^5 - 2*q^6 + (a - 2)*q^7 + O(q^8)
```

When possible, we can also obtain a newform that gives rise to an abelian variety.

```
> J := JZero(43);
> D := Decomposition(J); D;
[
    Modular abelian variety 43A of dimension 1, level 43 and
    conductor 43 over Q,
    Modular abelian variety 43B of dimension 2, level 43 and
```

```
    conductor 43^2 over Q
]
> Newform(D[2]);
q + a*q^2 - a*q^3 + (-a + 2)*q^5 - 2*q^6 + (a - 2)*q^7 + O(q^8)
```

We demonstrate here how modular abelian varieties may be described by a label, using the string "43B". Continuing the previous code, we have:

```
> A := ModularAbelianVariety("43B");
> A eq D[2];
true
```

136.3 Homology

The homology $H_1(A, R)$ of an abelian variety A with coefficients in a ring R is a free R-module of rank equal to twice the dimension of A. For many purposes, we view abelian varieties as complex tori V/Λ, so there is a canonical isomorphism $\Lambda \cong H_1(A, \mathbf{Z})$. (If the sign of A is ± 1, then the homology command below gives a $\mathbf{Z}$-module of rank $\dim A$.)

136.3.1 Creation

The `Homology` command creates the first homology of a modular abelian variety, which is of type `ModAbVarHomol`. This is the only command for creating homology.

`Homology(A)`

The first integral homology of the modular abelian variety A. (If the sign of A is 1 or -1, then this is $\mathbf{Z}$-module of rank equal to the dimension of A.)

Example H136E44

The homology of the elliptic curve $J_0(14)$ is of dimension 2.

```
> A := JZero(14); A;
Modular abelian variety JZero(14) of dimension 1 and level 2*7 over Q
> Homology(A);
Modular abelian variety homology space of dimension 2
```

If, for efficiency purposes, we work in the $+1$ quotient, then we are only working with half the homology, so the dimension of the homology is 1.

```
> Homology(JZero(14,2 : Sign := +1));
Modular abelian variety homology space of dimension 1
```

136.3.2 Invariants

The only invariant of homology is its dimension. If A is an abelian variety, and H is its first homology, then H has dimension equal to twice the dimension of A. (Except if, for efficiency purposes, we are working with sign $+1$ or -1, in which case the dimension of H is equal to the dimension of A.)

`Dimension(H)`

The dimension of the space H of homology.

Example H136E45

```
> Dimension(Homology(JZero(100)));
14
> Dimension(Homology(JZero(100,2 : Sign := +1)));
7
> Dimension(Homology(JZero(100,2 : Sign := -1)));
7
```

136.3.3 Functors to Categories of Lattices and Vector Spaces

The following commands provide convenient functors from the categories of homology and modular abelian varieties to lattices and vector spaces. Mathematically, these functors are defined using the first homology of the complex manifold underlying the complex points of the abelian variety.

`IntegralHomology(A)`

The lattice underlying the homology of the modular abelian variety A.

`Lattice(H)`

The underlying lattice of the homology space H. This is a free $\mathbf{Z}$-module of rank equal to the dimension of H.

`RationalHomology(A)`

A $\mathbf{Q}$-vector space obtained by tensoring the homology of the modular abelian variety A with $\mathbf{Q}$.

`RealHomology(A)`

A vector space over $\mathbf{R}$ obtained by tensoring the homology of the modular abelian variety A with $\mathbf{R}$.

`RealVectorSpace(H)`

The $\mathbf{R}$-vector space whose dimension is the dimension of the homology space H.

`VectorSpace(H)`

The $\mathbf{Q}$-vector space whose dimension is the dimension of the homology space H.

Example H136E46

The integral, rational and real homology of a modular abelian variety A is $H_1(A, R)$, where $R = \mathbf{Z}, \mathbf{R}, \mathbf{Q}$, respectively. This homology is just a free module over R of dimension twice $\dim(A)$ (unless the sign of A is ± 1, in which case the homology is of dimension $\dim(A)$).

```
> J := JZero(22); J;
Modular abelian variety JZero(22) of dimension 2 and level 2*11 over Q
> IntegralHomology(J);
Standard Lattice of rank 4 and degree 4
> RationalHomology(J);
Full Vector space of degree 4 over Rational Field
> RealHomology(J);
Full Vector space of degree 4 over Real Field
> J := JZero(22, 2, +1); J;
Modular abelian variety JZero(22) of dimension 2 and level 2*11 over Q
with sign 1
> RationalHomology(J);
Full Vector space of degree 2 over Rational Field
```

Example H136E47

The following code demonstrates the above commands applied to the abelian surface attached to the space of cusp forms of level 37 with quadratic character.

```
> G<eps> := DirichletGroup(37);
> Order(eps);
2
> A := ModularAbelianVariety(eps); A;
Modular abelian variety of dimension 2 and level 37 over Q
> Decomposition(A);
[
    Modular abelian variety image(37A[18]) of dimension 2, level
    37 and conductor 37^2 over Q
]
> IntegralHomology(A);
Standard Lattice of rank 4 and degree 4
> H := Homology(A); H;
Modular abelian variety homology space of dimension 4
> Lattice(H);
Standard Lattice of rank 4 and degree 4
> RationalHomology(A);
Full Vector space of degree 4 over Rational Field
> RealHomology(A);
Full Vector space of degree 4 over Real Field
> RealVectorSpace(H);
Full Vector space of degree 4 over Real Field
> VectorSpace(H);
```

```
Full Vector space of degree 4 over Rational Field
```

Example H136E48

The code below illustrates that the lattice need not just be the free lattice on $\dim(H)$ generators. For efficiency reasons many internal computations only require knowing $H_1(A, \mathbf{Q})$, which is sometimes all that gets computed, and the lattice returned by this command is $H_1(A, \mathbf{Z})$ written with respect to a basis for $H_1(A, \mathbf{Q})$. Thus `Lattice` should be viewed as giving an integral structure of $H_1(A, \mathbf{Q})$.

```
> J := JZero(37);
> A := J(1);
> Lattice(Homology(A));
Lattice of rank 2 and degree 2
Basis:
( 1  1)
( 1 -1)
> IntegralHomology(JOne(17));
Lattice of rank 10 and degree 10
Basis:
( 0  0  1  0  0  0  1  0  0  0)
( 0  0  1  0  0  0  0  0 -1  0)
( 0  0  0  1  0  0  0  1  0  0)
( 0  0  0  1  0  0  0 -1  0  0)
( 0  0  0  0  1  0  0  0 -1  0)
( 0  0  0  0  0  1  0  0  0  1)
( 0  0  0  0  0  1  0  0  0 -1)
( 0  0  0  0  0  0  1  0 -1  0)
( 0  2  1  0  0  0  0  0  0  0)
( 2 -1  0  0  0  0  1 -1  0  1)
```

136.3.4 Modular Structure

If H is the homology of an abelian variety that is attached to a space of modular symbols, then H remembers that space of modular symbols. The following two functions decide if H is attached to modular symbols, and if so provide the corresponding spaces of modular symbols. The reason that `ModularSymbols` returns a sequence instead of a single modular symbols space is that arbitrary finite products of abelian varieties are allowed, so corresponding direct sums of modular symbols spaces must be allowed, which are represented as sequences because MAGMA currently doesn't have a facility for taking direct sums of modular symbols spaces.

`IsAttachedToModularSymbols(H)`

Return `true` if the homology space H is presented as being attached to a sequence of spaces of modular symbols.

`ModularSymbols(H)`

If the space of homology H is attached to a sequence of spaces of modular symbols, then this is that sequence. Otherwise an error occurs.

Example H136E49

In this example, we create the product $J_0(23) \times J_0(11)$, and find that its homology is attached modular symbols, since both $J_0(23)$ and $J_0(11)$ are attached to modular symbols. We then display the corresponding spaces of modular symbols.

```
> A := JZero(23) * JZero(11);
> H := Homology(A); H;
Modular abelian variety homology space of dimension 6
> IsAttachedToModularSymbols(H);
true
> ModularSymbols(H);
[
    Modular symbols space for Gamma_0(23) of weight 2 and
    dimension 4 over Rational Field,
    Modular symbols space for Gamma_0(11) of weight 2 and
    dimension 2 over Rational Field
]
```

136.3.5 Additional Examples

Example H136E50

The following example illustrates each of the above intrinsics for the homology of $J_0(23)$.

```
> A := JZero(23); A;
Modular abelian variety JZero(23) of dimension 2 and level 23 over Q
> H := Homology(A); H;
Modular abelian variety homology space of dimension 4
```

Notice that homology has its own type:

```
> Type(H);
ModAbVarHomol
```

In this case, the homology is attached to a space of modular symbols, since $J_0(23)$.

```
> IsAttachedToModularSymbols(H);
true
> ModularSymbols(H);
[
    Modular symbols space for Gamma_0(23) of weight 2 and
    dimension 4 over Rational Field
]
> Dimension(H);
```

```
4
> Lattice(H);
Standard Lattice of rank 4 and degree 4
> RealVectorSpace(H);
Full Vector space of degree 4 over Real Field
> VectorSpace(H);
Full Vector space of degree 4 over Rational Field
```

We can also obtain various homologies of A more directly.

```
> IntegralHomology(A);
Standard Lattice of rank 4 and degree 4
> RationalHomology(A);
Full Vector space of degree 4 over Rational Field
> RealHomology(A);
Full Vector space of degree 4 over Real Field
```

Example H136E51

The following example illustrates that a factor of $J_0(37)$ is not presented as something associated to a space of modular symbols. Also notice that the two lattices are different.

```
> J := JZero(37);
> D := Decomposition(J);
> H1 := Homology(D[1]);  H2 := Homology(D[2]);
> IsAttachedToModularSymbols(H1);
false
> Lattice(H1);
Lattice of rank 2 and degree 2
Basis:
( 1  1)
( 1 -1)
> Lattice(H2);
Standard Lattice of rank 2 and degree 2
```

136.4 Homomorphisms

136.4.1 Creation

The commands below create the multiplication by n map on an abelian variety, for any n. Other ways to create homomorphisms are described in other sections of this chapter. These include computing Hecke operators, Atkin-Lehner operators, and computing the full endomorphism and homomorphism rings, and choosing elements in them.

`IdentityMap(A)`

The identity homomorphism from the modular abelian variety A to A.

`ZeroMap(A)`

The zero homomorphism from the modular abelian variety A to A.

`nIsogeny(A, n)`

The multiplication by n isogeny on the modular abelian variety A, where n is a rational number or an integer.

Example H136E52

```
> A := JZero(23);
> IdentityMap(A);
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
> ZeroMap(A);
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
> nIsogeny(A,3);
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[3 0 0 0]
[0 3 0 0]
[0 0 3 0]
[0 0 0 3]
> nIsogeny(A,1/3);
Homomorphism from JZero(23) to JZero(23) (up to isogeny) on integral
homology by:
[1/3   0   0   0]
[  0 1/3   0   0]
[  0   0 1/3   0]
[  0   0   0 1/3]
```

136.4.2 Restriction, Evaluation, and Other Manipulations

Suppose A is an abelian variety, B is an abelian subvariety of A, and ϕ is a homomorphism from A. Then the `Restriction` command computes the restriction of ϕ to B. If, moreover, ϕ is an endomorphism of A (i.e., also has codomain A) and $\phi(B)$ is contained in B, then `RestrictEndomorphism` computes the endomorphism of B induced by ϕ.

If f is a polynomial over the integers or rational numbers and ϕ is an endomorphism of an abelian variety, then the `Evaluate` command computes the endomorphism $f(\phi)$ (if f has denominators, then $f(\phi)$ might only be a morphism on abelian varieties up to isogeny). Together with the `Kernel` command, `Evaluate` is useful for cutting out subvarieties of an abelian variety.

The `SurjectivePart`, `DivideOutIntegers`, and `UniversalPropertyOfCokernel` commands can also all be useful in various contexts for constructing homomorphisms.

`Restriction(phi, B)`

The restriction of the map of abelian varieties ϕ to a morphism from the abelian variety B to the codomain of ϕ, if this obviously makes sense.

`RestrictEndomorphism(phi, B)`

The restriction of the map of abelian varieties ϕ to an endomorphism of the abelian variety B, if this obviously makes sense. If B is not left invariant by ϕ, an error may occur.

`RestrictEndomorphism(phi, i)`

Suppose ϕ is an endomorphism of an abelian variety A and $i : B \to A$ is an injective morphism of abelian varieties such that $i(B)$ is invariant under ϕ. This intrinsic computes the endomorphism ψ of B induced by ϕ. If $i(B)$ is not left invariant by ϕ, an error may occur.

`RestrictionToImage(phi, i)`

Suppose $i : A \to D$ and $\phi : D \to B$ are morphisms of abelian varieties. This intrinsic computes the restriction of ϕ to the image of i. The resulting map is an endomorphism of the image of i.

`Evaluate(f, phi)`

The endomorphism $f(\phi)$ of A, where f is a univariate polynomial and ϕ is an endomorphism of a modular abelian variety A.

`DivideOutIntegers(phi)`

If $\phi : A \to B$ is a homomorphism of abelian varieties, find the largest integer n such that $\psi = (1/n) * \phi$ is also a homomorphism from A to B and return ψ and n.

`SurjectivePart(phi)`

Let $\phi : A \to B$ be a homomorphism. This intrinsic returns the *surjective* homomorphism $\pi : A \to \phi(A)$ induced by ϕ.

`UniversalPropertyOfCokernel(pi, f)`

Uses the universal property of the cokernel to find the unique morphism with a certain property. More precisely, suppose $\pi : B \to C$ is the cokernel of a morphism, so π is surjective with kernel K. Suppose $f : B \to D$ is a morphism whose kernel contains K. By definition of cokernel, there exists a unique morphism $\psi : C \to D$ such that $\pi * \psi = f$. This intrinsic returns ψ. If we only have that the identity component of $\ker(\pi)$ is contained in `ker(f)`, then ψ will only be a homomorphism in the category of abelian varieties up to isogeny, i.e., it will have a nontrivial denominator.

Example H136E53

We use the `Evaluate` and `Kernel` commands to cut out a 2-dimensional abelian subvariety of $J_0(65)$.

```
> J := JZero(65);
> R<x> := PolynomialRing(RationalField());
> T2 := HeckeOperator(J,2);
> Factorization(CharacteristicPolynomial(T2));
[
    <x + 1, 2>,
    <x^2 - 3, 2>,
    <x^2 + 2*x - 1, 2>
]
> phi := Evaluate(x^2-3,T2);
> _,A := Kernel(phi);
> A;
Modular abelian variety of dimension 2 and level 5*13 over Q
```

Example H136E54

This example illustrates the universal property of the cokernel. We decompose $J = J_0(65)$ as a product of simples A, B, and C, of dimensions 1, 2, and 2, respectively. Then we let π be a homomorphism from J onto $B + C$, and f a homomorphism from J onto B. The universal property supplies a morphism ψ from $B + C$ to B such that $\pi * \psi = f$ (i.e., $f(x) = \psi(\pi(x))$ for all x).

```
> J := JZero(65);
> A,B,C := Explode(Decomposition(J));
> pi := NaturalMap(J,B+C);
> IsSurjective(pi);
true
> f := NaturalMap(J,B);
```

```
> psi := UniversalPropertyOfCokernel(pi,f); psi;
Homomorphism from modular abelian variety of dimension 4 to 65B
(up to isogeny) on integral homology by:
 (not printing 8x4 matrix)
> Matrix(psi);
[ 1/2    0    0 -1/2]
[ 1/2    0 -1/2  1/2]
[   0 -1/2  1/2    0]
[   0    0 -1/2    1]
[ 1/2 -1/2 -1/2  1/2]
[ 1/2    0 -1/2  1/2]
[   0  1/2    0    0]
[   0    0  1/2    0]
> pi*psi eq f;    // apply pi then psi is the same as applying f.
true
> Denominator(psi);
2
```

Since ψ has a denominator of 2, it is only a morphism in the category of abelian varieties up to isogeny. This is because only the connected component of the kernel of π is contained in the kernel of f.

```
> G := Kernel(pi); G;
Finitely generated subgroup of abelian variety with invariants
[ 2, 2, 2, 2, 2, 2 ]
> H, K := Kernel(f);
> H;
{ 0 }: finitely generated subgroup of abelian variety with
invariants []
> G subset K;
false
```

Example H136E55

Next we illustrate dividing out by integers, by dividing the 10 out of $10T_3$ acting on $J_0(23)$. Note that this division occurs in the full endomorphism ring, not just the Hecke algebra.

```
> phi := 10*HeckeOperator(JZero(23),3); phi;
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[-10 -20  20   0]
[  0 -30  20 -20]
[ 20 -40  30 -20]
[ 20 -20   0  10]
> DivideOutIntegers(phi);
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[-1 -2  2  0]
[ 0 -3  2 -2]
[ 2 -4  3 -2]
[ 2 -2  0  1]
```

```
10
```

Example H136E56

Next we illustrate the restriction commands using factors of $J_0(65)$.

```
> J := JZero(65);
> A := Decomposition(J)[2]; A;
Modular abelian variety 65B of dimension 2, level 5*13 and
conductor 5^2*13^2 over Q
> T := HeckeOperator(J,3);
> Factorization(CharacteristicPolynomial(T));
[
    <x + 2, 2>,
    <x^2 - 2*x - 2, 2>,
    <x^2 - 2, 2>
]
> Restriction(T,A);
Homomorphism from modular abelian variety of dimension 2 to
JZero(65) given on integral homology by:
[-1  0  0  1  0 -1  0  0  0  0]
[-1  3 -1  1 -3 -1  1  2 -1 -2]
[-1  1 -1  3 -1 -1 -1  0  1 -2]
[-1  0  0  3  0 -1 -2  0  2 -2]
> TonA := RestrictEndomorphism(T,A); TonA;
Homomorphism from 65B to 65B given on integral homology by:
[-1  0  0  1]
[-1  3 -1  1]
[-1  1 -1  3]
[-1  0  0  3]
> Factorization(CharacteristicPolynomial(TonA));
[
    <x^2 - 2*x - 2, 2>
]
```

Finally, we illustrate the `SurjectivePart` command on the non-surjective morphism $T_3 + 2$ of $J_0(65)$.

```
> phi := T+2;
> IsSurjective(phi);
false
> pi := SurjectivePart(phi);
> IsSurjective(pi);
true
> Codomain(pi);
Modular abelian variety of dimension 4 and level 5*13 over Q
```

136.4.3 Kernels

The category of abelian varieties is not an abelian category because kernels need not exist in this category. More precisely, if ϕ is a homomorphism $A \to B$ of abelian varieties, then the kernel of ϕ is usually not an abelian variety because it is rarely connected. Instead, the kernel fits into an exact sequence

$$0 \to C \to \ker(\phi) \to G \to 0,$$

where C is an abelian variety and G is a finite group, both defined over the same field as ϕ.

The `ConnectedKernel` command returns C.

The `ComponentGroupOfKernel` command returns G as a subgroup of A/C.

The `Kernel` command returns a finite subgroup of A that maps to G, the abelian variety C, and a map from C into A.

Note that if $C \neq 0$ then the finite group that the `Kernel` command returns is *not canonical*, since it is just some finite group that maps onto G via quotienting out by C. For the canonical component group, use the `ComponentGroupOfKernel` command, which gives a group defined over the same base field as ϕ, whose main drawback is that the component group is not contained in A.

`ComponentGroupOfKernel(phi)`

Component group of $\ker(\phi)$ where ϕ is a homomorphism of abelian varieties.

`ConnectedKernel(phi)`

The connected component C of $\ker(\phi)$ and a morphism from C to the domain of ϕ where ϕ is a homomorphism of abelian varieties.

`Kernel(phi)`

Let ϕ be a homomorphism of abelian varieties. This intrinsic returns a finite subgroup G of A (the domain of ϕ) as a subgroup of an abelian variety, an abelian variety C such that $\ker(\phi)$ equals $f(C) + G$, and an injective map f from C to A. If $C = 0$, then G has the same field of definition as ϕ; otherwise, G is only known to be defined over the algebraic closure.

Example H136E57

The kernel of $T_2 - 3$ on $J_0(65)$ is an extension of an abelian surface by a finite group isomorphic to $(\mathbf{Z}/2\mathbf{Z})^4$.

```
> J := JZero(65);
> T := HeckeOperator(J,2);
> phi := T^2-3;
> ConnectedKernel(phi);
Modular abelian variety of dimension 2 and level 5*13 over Q
Homomorphism from modular abelian variety of dimension 2 to
JZero(65) given on integral homology by:
```

```
[ 1  0  0  0  0  1 -1  0  1 -1]
[ 0  1  0  0 -1  0  0  1  0 -1]
[ 0  0  1  0  0  0 -1  1  1 -1]
[ 0  0  0  1  0  0 -1  0  1 -1]
> Kernel(phi);
Finitely generated subgroup of abelian variety with
invariants [ 2, 2, 2, 2 ]
Modular abelian variety of dimension 2 and level 5*13 over Q
Homomorphism from modular abelian variety of dimension 2 to
JZero(65) given on integral homology by:
[ 1  0  0  0  0  1 -1  0  1 -1]
[ 0  1  0  0 -1  0  0  1  0 -1]
[ 0  0  1  0  0  0 -1  1  1 -1]
[ 0  0  0  1  0  0 -1  0  1 -1]
> G := ComponentGroupOfKernel(phi); G;
Finitely generated subgroup of abelian variety with
invariants [ 2, 2, 2, 2 ]
> FieldOfDefinition(G);
Rational Field
```

Though G is defined over $\mathbf{Q}$, the ambient variety of G is the quotient of $J_0(65)$ by the two dimensional connected component of the kernel of $T_2 - 3$.

```
> AmbientVariety(G);
Modular abelian variety of dimension 3 and level 5*13 over Q
```

On the other hand, the group returned by the `Kernel` command is a subgroup of $J_0(65)$.

```
> H, C := Kernel(phi);
> H;
Finitely generated subgroup of abelian variety with invariants
[ 2, 2, 2, 2 ]
> FieldOfDefinition(H);
Algebraically closed field with no variables
> AmbientVariety(H);
Modular abelian variety JZero(65) of dimension 5 and level 5*13
over Q
```

136.4.4 Images

These commands compute the image of a modular abelian variety or a finite group under a homomorphism ϕ of modular abelian varieties. Using `@@`, one can also compute a group lifting a given group. Note that this lift is not canonical unless ϕ has finite kernel, in which case `G@@phi` is the full inverse image of G.

`A @ phi`

`phi(A)`

The image of the abelian variety A under the map ϕ of abelian varieties, if this makes sense, i.e., if A is the domain of ϕ, or A has dimension 0, or one of the embeddings of A has codomain equal to the domain of ϕ.

`G @ phi`

`phi(G)`

The image of the subgroup G of an abelian variety under the map ϕ of abelian varieties, if this makes sense. If A is the ambient variety of G, then this makes sense if A is the domain of ϕ, or A has dimension 0, or one of the embeddings of A has codomain equal to the domain of ϕ.

`Image(phi)`

The image C of the morphism ϕ of abelian varieties, which is a modular abelian subvariety contained in the codomain of ϕ, a morphism from C to the codomain of ϕ, and a surjective morphism from the domain of ϕ to C.

`G @@ phi`

A finite group whose image under ϕ, a morphism of abelian varieties, is equal to the subgroup G of a modular abelian variety, if possible. If ϕ has finite kernel, then this is the exact inverse image of G under ϕ. If not, then this is a group generated by a choice of torsion inverse image for each generator of G, which may not be canonical.

Example H136E58

The image of $J_0(37)$ under T_2 is an elliptic curve.

```
> J := JZero(37);
> phi := HeckeOperator(J,2);
> phi(J);
Modular abelian variety of dimension 1 and level 37 over Q
> Image(phi);
Modular abelian variety of dimension 1 and level 37 over Q
Homomorphism from modular abelian variety of dimension 1 to
JZero(37) given on integral homology by:
[ 1 -1  1  0]
[ 1 -1 -1  1]
Homomorphism from JZero(37) to modular abelian variety of dimension
1 given on integral homology by:
[ 0 -1]
[ 1  0]
[-1  1]
```

```
[ 0  0]
```

The Hecke operator T_2 maps the 2-torsion subgroup of $J_0(37)$ maps onto the 2-torsion subgroup of the image of T_2.

```
> G := nTorsionSubgroup(J,2); G;
Finitely generated subgroup of abelian variety with invariants
[ 2, 2, 2, 2 ]
> phi(G);
Finitely generated subgroup of abelian variety with invariants
[ 2, 2 ]
```

The homomorphism $T_2 - 1$ is surjective, and the inverse image of the 2-torsion is a subgroup isomorphic to $(\mathbf{Z}/2\mathbf{Z})^2 \times (\mathbf{Z}/6\mathbf{Z})^2$.

```
> IsSurjective(phi-1);
true
> psi := phi-1;
> H := G@@(psi); H;
Finitely generated subgroup of abelian variety with invariants
[ 2, 2, 6, 6 ]
> psi(H);
Finitely generated subgroup of abelian variety with invariants
[ 2, 2, 2, 2 ]
> H := G@@psi; H;
Finitely generated subgroup of abelian variety with invariants
[ 2, 2, 6, 6 ]
> psi(H);
Finitely generated subgroup of abelian variety with invariants
[ 2, 2, 2, 2 ]
```

136.4.5 Cokernels

`Cokernel(phi)`

The cokernel of the morphism ϕ of abelian varieties and a morphism from the codomain of ϕ to the cokernel.

Example H136E59

We compute a 2-dimensional quotient of the 3-dimensional abelian variety $J_0(33)$ using the Hecke operator T_2 and the `Cokernel` command.

```
> J := JZero(33);
> T := HeckeOperator(J,2);
> Factorization(CharacteristicPolynomial(T));
[
    <x - 1, 2>,
    <x + 2, 4>
```

```
]
> phi := T + 2;
> A, pi := Cokernel(phi);
> A;
Modular abelian variety of dimension 2 and level 3*11 over Q
> pi;
Homomorphism from JZero(33) to modular abelian variety of dimension
2 (not printing 6x4 matrix)
```

136.4.6 Matrix Structure

Homomorphisms are stored and worked with internally using the linear maps they induce on homology. The commands below provide access to those matrices.

`Matrix(phi)`

The matrix on the chosen basis of rational homology that defines the morphism ϕ of abelian varieties.

`Eltseq(phi)`

The `Eltseq` of the underlying matrix that defines the morphism ϕ of abelian varieties. This is a sequence of integers or rational numbers.

`Ncols(phi)`

The number of columns of the matrix which defines the morphism ϕ of abelian varieties. This is also the dimension of the homology of the codomain of ϕ.

`Nrows(phi)`

The number of rows of the matrix that defines the morphism ϕ of abelian varieties. This is also the dimension of the homology of the domain of ϕ.

`Rows(phi)`

The sequence rows of the matrix that defines the morphism ϕ of abelian varieties.

`IntegralMatrix(phi)`

The matrix which defines the morphism ϕ of abelian varieties, written with respect to integral homology.

`IntegralMatrixOverQ(phi)`

The matrix which defines the morphism ϕ of abelian varieties, written with respect to integral homology.

`RealMatrix(phi)`

The matrix which defines the morphism ϕ of abelian varieties, written with respect to real homology.

Example H136E60

The following example demonstrates each of the commands for the Hecke operator T_2 on $J_0(23)$.

```
> phi := HeckeOperator(JZero(23),2); phi;
Homomorphism T2 from JZero(23) to JZero(23) given on integral homology by:
[ 0  1 -1  0]
[ 0  1 -1  1]
[-1  2 -2  1]
[-1  1  0 -1]
> Eltseq(phi);
[ 0, 1, -1, 0, 0, 1, -1, 1, -1, 2, -2, 1, -1, 1, 0, -1 ]
> IntegralMatrix(phi);
[ 0  1 -1  0]
[ 0  1 -1  1]
[-1  2 -2  1]
[-1  1  0 -1]
> Parent($1);
Full RMatrixSpace of 4 by 4 matrices over Integer Ring
> IntegralMatrixOverQ(phi);
[ 0  1 -1  0]
[ 0  1 -1  1]
[-1  2 -2  1]
[-1  1  0 -1]
> Parent($1);
Full Matrix Algebra of degree 4 over Rational Field
> Matrix(phi);
[ 0  1 -1  0]
[ 0  1 -1  1]
[-1  2 -2  1]
[-1  1  0 -1]
> Ncols(phi);
4
> Nrows(phi);
4
> RealMatrix(phi);
[   0    1   -1    0]
[   0    1   -1    1]
[  -1  2/1 -2/1    1]
[  -1    1    0   -1]
> Parent($1);
Full KMatrixSpace of 4 by 4 matrices over Real Field
> Rows(phi);
[
    ( 0  1 -1  0),
    ( 0  1 -1  1),
    (-1  2 -2  1),
    (-1  1  0 -1)
```

```
]
```

136.4.7 Arithmetic

MAGMA supports many standard arithmetic operations with homomorphisms of abelian varieties, including composition, addition, subtraction, and exponentiation.

`Inverse(phi)`

The inverse of ϕ and an integer d such that $d*\phi^{-1}$ is a morphism of abelian varieties. More precisely, if ϕ is an isogeny, then the inverse of ϕ is a morphism in the category of abelian varieties up to isogeny. This intrinsic returns such a morphism or the actual inverse of ϕ if ϕ has degree 1.

`phi * psi`

The composition of the homomorphisms ϕ and ψ of abelian varieties.

`a * phi`

The product of the rational number or integer a and the homomorphism ϕ of abelian varieties. The result might only be a homomorphism, up to isogeny.

`phi * psi`

The product of the matrix that defines the morphism ϕ of abelian varieties and the matrix ψ.

`psi * phi`

The product of the matrix ψ and the matrix that defines the morphism ϕ of abelian varieties.

`phi ^ n`

The n-fold composition ϕ^n of the endomorphism ϕ of an abelian variety with itself. If $n = -1$, then this is the inverse of ϕ, in which case ϕ must be an isogeny or an error occurs.

`phi + psi`

The sum of the homomorphisms ϕ and ψ

`n + phi`

The sum of the morphism multiplication by the rational number or integer n and the endomorphism ϕ of an abelian variety.

`phi + n`

The sum of the endomorphism ϕ of an abelian variety and the endomorphism multiplication-by-n where n is an integer.

`phi + psi`

The sum of the matrix that defines the morphism ϕ of abelian varieties and the matrix ψ.

`psi + phi`

The sum of the matrix ψ and the matrix which defines the morphism ϕ of abelian varieties.

`phi - psi`

The difference between the homomorphisms ϕ and ψ of abelian varieties.

`n - phi`

The difference between the morphism multiplication-by-n, where n is a rational number or an integer, and the endomorphism ϕ of an abelian variety.

`phi - n`

The difference between the endomorphism ϕ of an abelian variety and the endomorphism multiplication-by-n where n is a rational number or an integer.

`phi - psi`

The difference between the matrix that defines the morphism ϕ of abelian varieties and the matrix ψ.

`psi - phi`

The difference between the matrix ψ and the matrix which defines the morphism ϕ of abelian varieties.

Example H136E61

We illustrate several arithmetic operations use the Hecke operators T_2 and T_3 on $J_0(23)$.

```
> J := JZero(23);
> phi := HeckeOperator(J,2);
> psi := HeckeOperator(J,3);
> Matrix(phi)*Matrix(psi);
[-2  1 -1  0]
[ 0 -1 -1  1]
[-1  2 -4  1]
[-1  1  0 -3]
> phi*psi;
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[-2  1 -1  0]
[ 0 -1 -1  1]
[-1  2 -4  1]
[-1  1  0 -3]
> Inverse(phi);
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
```

```
[ 1  1 -1  0]
[ 0  2 -1  1]
[-1  2 -1  1]
[-1  1  0  0]
1
> 1/3*phi;
Homomorphism from JZero(23) to JZero(23) (up to isogeny) on integral
homology by:
[   0  1/3 -1/3    0]
[   0  1/3 -1/3  1/3]
[-1/3  2/3 -2/3  1/3]
[-1/3  1/3    0 -1/3]
> 2+phi;
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[ 2  1 -1  0]
[ 0  3 -1  1]
[-1  2  0  1]
[-1  1  0  1]
> phi+psi;
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[-1 -1  1  0]
[ 0 -2  1 -1]
[ 1 -2  1 -1]
[ 1 -1  0  0]
> phi^4;
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[ 2 -3  3  0]
[ 0 -1  3 -3]
[ 3 -6  8 -3]
[ 3 -3  0  5]
1
> phi^(-4);
Homomorphism from JZero(23) to JZero(23) given on integral homology by:
[ 5  3 -3  0]
[ 0  8 -3  3]
[-3  6 -1  3]
[-3  3  0  2]
1
```

136.4.8 Polynomials

The polynomial commands compute the characteristic and minimal polynomials of endomorphisms of modular abelian varieties. Each command requires that the input homomorphism be a genuine homomorphism (not just a morphism up to isogeny). The same computation could be done by first extracting the matrix of the endomorphism and using the analogous command on the matrix. However, in some special cases there may be special techniques which can be used to do the computation more quickly, so there can be some advantage to using the commands below.

`CharacteristicPolynomial(phi)`

The characteristic polynomial of the endomorphism ϕ of an abelian variety. We can sometimes use extra information about ϕ (e.g., how it was constructed, Deligne bounds) to gain an answer faster.

`FactoredCharacteristicPolynomial(phi)`

The factorization of the characteristic polynomial of the endomorphism ϕ of an abelian variety. We can sometimes use extra information about ϕ (e.g., how it was constructed, Deligne bounds), to gain an answer faster. These factored polynomials are cached.

`MinimalPolynomial(phi)`

The minimal polynomial of the endomorphism ϕ of an abelian variety. We can sometimes use extra information about ϕ (e.g., how it was constructed, Deligne bounds), to gain an answer faster.

Example H136E62

This example illustrates each of the polynomial commands for the Hecke operator T_2 on $J_0(66)$.

```
> J := JZero(66);
> phi := HeckeOperator(J,2);
> R<x> := PolynomialRing(RationalField());
> CharacteristicPolynomial(phi);
x^18 + 4*x^17 + 8*x^16 + 8*x^15 + 6*x^14 - 20*x^12 - 56*x^11 -
    55*x^10 - 4*x^9 + 36*x^8 + 48*x^7 + 104*x^6 + 128*x^5 -
    32*x^4 - 192*x^3 - 112*x^2 + 64*x + 64
> CharacteristicPolynomial(Matrix(phi));
x^18 + 4*x^17 + 8*x^16 + 8*x^15 + 6*x^14 - 20*x^12 - 56*x^11 -
    55*x^10 - 4*x^9 + 36*x^8 + 48*x^7 + 104*x^6 + 128*x^5 -
    32*x^4 - 192*x^3 - 112*x^2 + 64*x + 64
> FactoredCharacteristicPolynomial(phi);
[
    <x - 1, 4>,
    <x + 1, 2>,
    <x^2 - x + 2, 2>,
    <x^2 + 2*x + 2, 4>
```

```
]
> MinimalPolynomial(phi);
x^6 + x^5 + x^4 + x^3 + 2*x^2 - 2*x - 4
> Factorization(MinimalPolynomial(phi));
[
    <x - 1, 1>,
    <x + 1, 1>,
    <x^2 - x + 2, 1>,
    <x^2 + 2*x + 2, 1>
]
```

136.4.9 Invariants

It is possible to retrieve the domain and codomain of homomorphisms between abelian varieties. The degrees of such homomorphisms can be computed. Denominators of homomorphisms can be determined and cleared. A field over which a homomorphism is defined is available. The rank, nullity and trace of a homomorphism can also be computed.

`Domain(phi)`

The domain of the morphism ϕ of abelian varieties.

`Codomain(phi)`

The codomain of the morphism ϕ of abelian varieties.

`Degree(phi)`

The degree of the morphism ϕ of abelian varieties, i.e. the cardinality of the kernel of ϕ. If the kernel of ϕ is not finite, then the degree is defined to be 0.

`Denominator(phi)`

Given a morphism ϕ of abelian varieties return the smallest positive integer n such that $n * \phi$ is a homomorphism. This is also the denominator of the matrix that defines ϕ.

`ClearDenominator(phi)`

Return the morphism $n * \phi$ where ϕ is a morphism of abelian varieties and n is positive and the smallest such that $n * \phi$ is a genuine homomorphism.

`FieldOfDefinition(phi)`

A field of definition of the morphism ϕ of abelian varieties. This is only some field over which ϕ is defined, it is not guaranteed to be minimal.

`Nullity(phi)`

The dimension of the kernel of the morphism ϕ of abelian varieties.

`Rank(phi)`

The dimension of the kernel of the morphism ϕ of abelian varieties.

`Trace(phi)`

The trace of any matrix which represents the action of the morphism ϕ of abelian varieties on integral homology. For example, the trace of multiplication by n on A is $2n\dim(A)$.

Example H136E63

```
> phi := NaturalMap(JZero(11),JZero(33));
> Codomain(phi);
Modular abelian variety JZero(33) of dimension 3 and level 3*11 over Q
> Domain(phi);
Modular abelian variety JZero(11) of dimension 1 and level 11 over Q
> Degree(phi);
1
> Denominator(1/5*phi);
5
> FieldOfDefinition(phi);
Rational Field
> Nullity(phi);
0
> Rank(phi);
1
> Trace(HeckeOperator(JZero(33),2));
-6
> Trace(nIsogeny(JZero(33),5));
30
```

136.4.10 Predicates

A range of predicates are provided allowing a morphism to be checked for whether it is an actual morphism or not, whether its kernel is finite, and whether it is injective or surjective or an isogeny. There are also commands for testing equality of homomorphisms and inclusion in a list.

`IsMorphism(phi)`

Return `true` if and only if the map ϕ between abelian varieties is a morphism in the category of abelian varieties (not just in the category of abelian varieties up to isogeny).

`OnlyUpToIsogeny(phi)`

Return `true` if the map ϕ between abelian varieties is not a homomorphism, but $n*\phi$ is a homomorphism for some positive integer n, i.e., ϕ is only a homomorphism in the category of abelian varieties up to isogeny.

`HasFiniteKernel(phi)`

Return `true` if the kernel of the homomorphism ϕ of abelian varieties is finite.

`IsInjective(phi)`

Return `true` if the map ϕ between abelian varieties is an injective homomorphism.

`IsSurjective(phi)`

Return `true` if the homomorphism ϕ between abelian varieties is surjective.

`IsEndomorphism(phi)`

Return `true` if the morphism ϕ between abelian varieties is an endomorphism, i.e., the domain and codomain of ϕ are equal.

`IsInteger(phi)`

Return `true` if the morphism ϕ between abelian varieties is multiplication by n for some integer n. If so, returns that n as well, otherwise, returns `false`.

`IsIsogeny(phi)`

Return `true` if the morphism ϕ between abelian varieties is a surjective homomorphism with finite kernel. Note that this definition differs from the one in Silverman's books on elliptic curves, agrees with the one in Milne's articles, and is an equivalence relation.

`IsIsomorphism(phi)`

Return `true` if the morphism ϕ between abelian varieties is an isomorphism of abelian varieties.

`IsOptimal(phi)`

Return `true` if the morphism ϕ between abelian varieties is an optimal quotient map, i.e., ϕ is surjective and has connected kernel.

`IsHeckeOperator(phi)`

Returns `true` if the morphism ϕ between abelian varieties was computed using the `HeckeOperator` command, and the argument n passed to the `HeckeOperator` command when ϕ was created, otherwise `false`.

`IsZero(phi)`

Return `true` if the morphism ϕ of abelian varieties is the zero morphism.

`phi eq psi`

Return `true` if the two homomorphisms of abelian varieties ϕ and ψ are equal.

`n eq phi`

`phi eq n`

Return `true` if the morphism ϕ between abelian varieties is equal to multiplication by the integer n.

`phi in X`

Return `true` if the morphism ϕ between abelian varieties is one of the homomorphisms in the list X of homomorphisms.

Example H136E64

```
> phi := HeckeOperator(JZero(65),2)-1;
> HasFiniteKernel(phi);
true
> IsEndomorphism(phi);
true
> IsHeckeOperator(phi);
false 0
> IsInjective(phi);
false
> IsInteger(phi);
false
> IsIsogeny(phi);
true
> IsIsomorphism(phi);
false
> IsMorphism(phi);
true
> IsMorphism(1/2*phi);
false
> IsSurjective(phi);
true
> IsZero(phi);
false
> OnlyUpToIsogeny(phi);
false
> 2 eq phi;
false
> phi eq 2;
false
> 2 eq nIsogeny(JZero(65),2);
true
> phi eq nIsogeny(JZero(65),2);
```

```
false
> phi in [* phi, nIsogeny(JZero(65),2) *];
true
> IsIsomorphism(NaturalMap(JZero(11),JOne(11)));
false
> IsIsomorphism(NaturalMap(JZero(11)^2,JZero(22)));
false
> IsIsomorphism(NaturalMap(JZero(11),JZero(11)));
true
```

136.5 Endomorphism Algebras and Hom Spaces

136.5.1 Creation

Let A and B be modular abelian varieties. Then one can create the finite-rank free abelian group of homomorphisms from A to B or the vector space of homomorphisms in the category of abelian varieties up to isogeny. It is also possible to create the endomorphism algebra of an abelian variety. Such creations do non-trivial computations until further information is requested of the structures, such as rank or basis. In particular, such creations do not compute $\mathrm{Hom}(A, B)$.

`Hom(A, B)`

`Hom(A, B, oQ)`

Return the group of homomorphisms from the abelian variety A to the abelian variety B. If the argument oQ is given and is `true`, return the vector space generated by such homomorphisms.

`End(A)`

`End(A, oQ)`

The endomorphism ring of the abelian variety A. If oQ is given and is `true`, return the endomorphism algebra.

`BaseExtend(H, R)`

The space $H \otimes R$, where H is a group of homomorphisms of a modular abelian variety and R is the rational numbers or integers. When $R = \mathbf{Q}$, this is the space of homomorphisms in the category of abelian varieties up to isogeny.

`HeckeAlgebra(A)`

The Hecke algebra associated to the abelian variety A, which is a commutative subring of $\mathrm{End}(A)$ generated by Hecke operators. For an abelian variety attached to modular symbols, this is the algebra induced by Hecke operators acting on modular symbols (homology). For a general abelian variety, let $\pi : J \to A$ and $e : A \to J$ be the modular parameterization and embedding of A. Then this is the ring of endomorphisms obtained by pulling back the Hecke algebra on J to A using π and e, i.e., it is $e * T * \pi$, where T is the Hecke ring of J. For example, since $J_1(N)$ is represented as a quotient of $J_0(N)$, the Hecke algebra is not what you might expect but may differ by some finite index.

Example H136E65

```
> A := JZero(11); B := JZero(33);
> Hom(A,B);
Group of homomorphisms from JZero(11) to JZero(33)
> Hom(A,B,true);
Group of homomorphisms from JZero(11) to JZero(33) in the category of
abelian varieties up to isogeny
> End(A);
Group of homomorphisms from JZero(11) to JZero(11)
> End(A,true);
Group of homomorphisms from JZero(11) to JZero(11) in the category of
abelian varieties up to isogeny
> BaseExtend(Hom(A,B),RationalField());
_Q: Group of homomorphisms from JZero(11) to JZero(33) in the category
of abelian varieties up to isogeny
> HeckeAlgebra(A);
HeckeAlg(JZero(11)): Group of homomorphisms from JZero(11) to JZero(11)
```

136.5.2 Subgroups and Subrings

Subgroups of $\mathrm{Hom}(A, B)$ can also be formed as well as subrings of $\mathrm{End}(A, B)$. MAGMA provides the computation of saturations of subgroups of $\mathrm{Hom}(A, B)$ where A and B are abelian varieties.

`Subgroup(X)`

The group of homomorphisms from the abelian variety A to the abelian variety B generated by the homomorphisms of abelian varieties in the sequence X.

`Subgroup(X, oQ : parameters)`

`IsBasis` BoolElt *Default :* `false`

The group of homomorphisms generated by homomorphisms of abelian varieties in the sequence X. If the parameter `IsBasis` is `true`, then it is assumed that the elements of X are a basis for their span. If the argument oQ is `true`, return the vector space generated by such homomorphisms.

`Subring(X)`

`Subring(X, oQ)`

The ring of endomorphisms of a modular abelian variety generated by the endomorphisms in the sequence X. It does not need to contain unity. If the argument oQ is given and is `true`, return the algebra generated by such endomorphisms.

`Subring(phi)`

The ring of endomorphisms generated by the endomorphism ϕ of an abelian variety. It does not need to contain unity.

`Saturation(H)`

The saturation of the group H of homomorphisms of abelian varieties in all homomorphisms. Suppose A and B are abelian varieties and H is a subgroup of $\mathrm{Hom}(A, B)$. Then $\mathrm{Hom}(A, B)$ is a free $\mathbf{Z}$-module, and the saturation of H in $\mathrm{Hom}(A, B)$ is a group H' that contains H with finite index such that the quotient of `Hom(A,B)` by H' is torsion free.

`RingGeneratedBy(H)`

The ring of endomorphisms generated by the endomorphisms in the group H of homomorphisms between abelian varieties.

Example H136E66

In this example we use the `Saturation` command to find the integers N up to 60 such that the Hecke algebra of $J_0(N)$ is not saturated in the full ring of endomorphisms.

```
> function ind(N)
>    H := HeckeAlgebra(JZero(N));
>    return Index(Saturation(H),H);
> end function;
> for N in [2..60] do
>    i := ind(N);
>    if i gt 1 then N, i; end if;
> end for;
44 2
46 2
54 3
56 2
```

```
60 2
```

Note that multiplicity one fails at 3 for $J_0(54)$. It might be interesting to find a precise relationship between failure of multiplicity one and the index of the Hecke algebra T in its saturation.

Example H136E67

This example illustrates constructing a subgroup.

```
> J := JZero(33);
> E := End(J); E;
Group of homomorphisms from JZero(33) to JZero(33)
> H := Subgroup([E.1, E.3]); H;
Group of homomorphisms from JZero(33) to JZero(33)
> Rank(H);
2
```

Example H136E68

We compute the subring generated by T_2 on $J_0(100)$.

```
> T2 := HeckeOperator(JZero(100),2);
> R := Subring(T2); R;
Group of homomorphisms from JZero(100) to JZero(100)
> Rank(R);
3
```

Example H136E69

The Hecke operator T_2 and the main Atkin-Lehner involution together generate a commutative ring of rank 10 over $\mathbf{Z}$.

```
> J := JZero(100);
> T2 := HeckeOperator(J,2);W := AtkinLehnerOperator(J,100);
> R := Subring([End(J) | T2,W]);
> Dimension(R);
10
> Dimension(End(J));
13
> IsRing(R);
true
> IsCommutative(R);
false
```

136.5.3 Pullback and Pushforward of Hom Spaces

A homomorphism of abelian varieties induces a map from one space of homomorphisms into another. The three commands below compute the image of such maps.

`Pullback(H, phi)`

Given a space of homomorphisms H in $\mathrm{Hom}(A, B)$ and a morphism $\phi : B \to C$, compute the image of H in $\mathrm{Hom}(A, C)$ via the map that sends f to $f * \phi$.

`Pullback(phi, H)`

Given a space of homomorphisms H in $\mathrm{Hom}(A, B)$ and a morphism $\phi : C \to A$, compute the image of H in $\mathrm{Hom}(C, B)$ via the map that sends f to $\phi * f$.

`Pullback(phi, H, psi)`

Suppose H is a space of homomorphisms $A \to B$ and $\phi : C \to A$ and $\psi : B \to D$. Then this intrinsic computes and returns the ring of homomorphisms of A of the form $\phi * f * \psi$, where f is in H.

Example H136E70

```
> H := Hom(JZero(11),JZero(22));
> phi := NaturalMap(JZero(22),JZero(33));
> psi := NaturalMap(JZero(33),JZero(11));
> Pullback(H,phi);
Group of homomorphisms from JZero(11) to JZero(33)
> Pullback(psi,H);
Group of homomorphisms from JZero(33) to JZero(22)
> Pullback(psi,H,phi);
Group of homomorphisms from JZero(33) to JZero(33)
```

136.5.4 Arithmetic

The + and `meet` command compute the sum and intersection of two subgroups of $\mathrm{Hom}(A, B)$.

`H1 + H2`

The subgroup generated by H_1 and H_2, where H_1 and H_2 are assumed to both be subgroups of $\mathrm{Hom}(A, B)$, for abelian varieties A and B.

`H1 meet H2`

The intersection of H_1 and H_2, where H_1 and H_2 are assumed to both be subgroups of $\mathrm{Hom}(A, B)$, for abelian varieties A and B.

Example H136E71

```
> J := JZero(33);
> E := End(J);
> H1 := HeckeAlgebra(J); H1;
HeckeAlg(JZero(33)): Group of homomorphisms from JZero(33) to JZero(33)
> H2 := Subgroup([E.1,E.2]); H2;
Group of homomorphisms from JZero(33) to JZero(33)
> Dimension(E);
5
> Dimension(H1);
3
> Dimension(H2);
2
> Dimension(H1 meet H2);
1
> Dimension(H1 + H2);
4
```

136.5.5 Quotients

If H_1 and H_2 are subspaces of $\mathrm{Hom}(A, B)$ then the index of H_1 in H_2 can be computed. The quotient H_2/H_1 can also be taken.

`Index(H2, H1)`

The index of H_1 in H_2, where H_1 and H_2 are both subgroups of $\mathrm{Hom}(A, B)$, for abelian varieties A and B. If H_1 is contained in H_2, this is just the cardinality of H_2/H_1, or 0 if this cardinality is infinite. If H_1 is not contained in H_2, then the index is by definition the generalized lattice index $[H_1 + H_2 : H_1]/[H_1 + H_2 : H_2]$, assuming the denominator is nonzero, i.e., that H_2 has finite index in $H_1 + H_2$ (an error occurs if H_2 does not have finite index in $H_1 + H_2$).

`Quotient(H2, H1)`

`H2 / H1`

The abelian group quotient H_2/H_1, a map from H_2 to this quotient, and a lifting map from this quotient to H_2, where H_1 and H_2 are subgroups of $\mathrm{Hom}(A, B)$ and A and B are abelian varieties.

Example H136E72

We define the subgroup of $\mathrm{End}(J_0(54))$ generated by T_1, T_2, T_3, and T_4, find that it has infinite index in the full Hecke algebra, and compute the quotient, which is $\mathbf{Z}$.

```
> J := JZero(54);
> T := HeckeAlgebra(J);
> Dimension(T);
4
> S := Subgroup([HeckeOperator(J,n) : n in [1..4]]);
> Dimension(S);
3
> Index(T,S);
0
> Quotient(T,S);
Abelian Group isomorphic to Z
Defined on 1 generator (free)
Mapping from: HomModAbVar: T to Abelian Group isomorphic to Z
Defined on 1 generator (free) given by a rule [no inverse]
Mapping from: Abelian Group isomorphic to Z
Defined on 1 generator (free) to HomModAbVar: T given by a rule
[no inverse]
> G := T/S; G;
Abelian Group isomorphic to Z
Defined on 1 generator (free)
```

We compute the subgroup generated by T_3, T_4, T_5, and T_{10}, and find that it has index 6 in its saturation.

```
> S := Subgroup([HeckeOperator(J,n) : n in [3,4,5,10]]);
> Sat := Saturation(S);
> Index(Sat,S);
6
> Index(S,Sat);
1/6
> Invariants(Sat/S);
[ 6 ]
```

136.5.6 Invariants

Domain and codomain of the homomorphisms in a space of homomorphisms of abelian varieties can both be retrieved from the space as well as a field which all homomorphisms in a space are defined over.

A possibly time consuming computation is that of the discriminant of a space of homomorphisms. Discriminants of Hecke algebras are particularly interesting to compute because they are closely related to congruences between eigenforms.

`Domain(H)`

The domain of the homomorphisms in the group H of homomorphisms between modular abelian varieties.

`Codomain(H)`

The codomain of the homomorphisms in the group H of homomorphisms between modular abelian varieties.

`FieldOfDefinition(H)`

A field over which all homomorphisms in the group H of homomorphisms of abelian varieties are defined. It is not guaranteed to be minimal.

`Discriminant(H)`

The discriminant of the space H of homomorphisms of abelian varieties with respect to the trace pairing matrix. This is the trace of endomorphisms acting on homology, not left multiplication on themselves, so, e.g., the discriminant of the Hecke algebra will be 2^d times as big as it would be otherwise (if the sign is 0), where d is the dimension of A. If H is over $\mathbf{Q}$, this function returns the discriminant of the lattice of elements in H that are homomorphisms.

Example H136E73

```
> H := Hom(JZero(11),JZero(33));
> Domain(H);
Modular abelian variety JZero(11) of dimension 1 and level 11 over Q
> Codomain(H);
Modular abelian variety JZero(33) of dimension 3 and level 3*11 over
Q
> FieldOfDefinition(H);
Rational Field
> A := BaseExtend(JZero(11),ComplexField());
> H := End(A);
> FieldOfDefinition(H);
Complex Field
```

Though $H = \mathbf{Z}$, so the discriminant of the abstract ring H is 1, the discriminant of the trace pairing of H acting on homology is 2:

```
> Discriminant(H);
2
[2]
```

The prime $p = 389$ is the only known example where p divides the discriminant of the Hecke algebra of $J_0(p)$.

```
> T := HeckeAlgebra(JZero(389,2 : Sign := +1));
> d := Discriminant(T);
> J := JZero(389,2,+1);
```

```
> T := HeckeAlgebra(J);
> d := Discriminant(T);
> d mod 389;
0
> Factorization(d);
[ <2, 53>, <3, 4>, <5, 6>, <31, 2>, <37, 1>, <389, 1>, <3881, 1>,
<215517113148241, 1>, <477439237737571441, 1> ]
```

All the "action" at 389 comes from the 20-dimensional simple factor.

```
> A := J(5); A;
Modular abelian variety 389E of dimension 20, level 389 and
conductor 389^20 over Q with sign 1
> Factorization(Discriminant(HeckeAlgebra(A)));
[ <2, 98>, <5, 41>, <389, 1>, <215517113148241, 1>,
<477439237737571441, 1> ]
```

136.5.7 Structural Invariants

The following commands provide access to a basis and generators for spaces of homomorphisms.

`Basis(H)`

`Generators(H)`

A basis for the space H of homomorphisms of abelian varieties.

`Dimension(H)`

`Rank(H)`

The rank of the space H of homomorphisms of abelian varieties as a $\mathbf{Z}$-module or $\mathbf{Q}$-vector space.

`Ngens(H)`

The number of generators of the space H of homomorphisms of abelian varieties.

`H . i`

The ith generator of the space H of homomorphisms of abelian varieties.

Example H136E74

The following example illustrates each of the commands for $\mathrm{Hom}(J_0(11), J_0(33))$.

```
> H := Hom(JZero(11),JZero(33));
> Basis(H);
[
    Homomorphism from JZero(11) to JZero(33) given on integral homology
    by:
    [ 0  2 -1 -2  0  1]
    [ 1  0 -1 -1  1  1],
    Homomorphism from JZero(11) to JZero(33) given on integral homology
    by:
    [ 1  0 -2  2 -3  0]
    [ 1 -1  0  1 -2  1]
]
> Dimension(H);
2
> Ngens(H);
2
> Rank(H);
2
> H.1;
Homomorphism from JZero(11) to JZero(33) given on integral homology by:
[ 0  2 -1 -2  0  1]
[ 1  0 -1 -1  1  1]
```

136.5.8 Matrix and Module Structure

The following commands associate lattices, vector spaces, matrix algebras, and matrix spaces to subspaces H of $\mathrm{Hom}(A, B)$.

`Lattice(H)`

`VectorSpace(H)`

A lattice or vector space with basis obtained from the components of the matrices of a basis for the space H of homomorphisms of abelian varieties. This free **Z**-module is constructed from the `Eltseq`s of the all of the basis elements of H.

`MatrixAlgebra(H)`

The matrix algebra generated by the underlying matrices of all elements in the space H of homomorphisms of abelian varieties, acting on homology.

`RMatrixSpace(H)`

The matrix space whose basis is the generators for the space H of homomorphisms of abelian varieties.

`RModuleWithAction(H)`

A module over the ring R generated by the space H of homomorphisms of abelian varieties equipped with the action of H, where H must be a ring of endomorphisms.

`RModuleWithAction(H, p)`

A module over H tensor F_p equipped with the action of H tensor F_p, where H must be a ring of endomorphisms of an abelian variety that has not been tensored with $\mathbf{Q}$.

Example H136E75

We first demonstrate some of the commands with $\mathrm{Hom}(J_0(11), J_0(33))$.

```
> H := Hom(JZero(11),JZero(33));
> Lattice(H);
Lattice of rank 2 and degree 12
Basis:
( 0  2 -1 -2  0  1  1  0 -1 -1  1  1)
( 1  0 -2  2 -3  0  1 -1  0  1 -2  1)
> RMatrixSpace(H);
RMatrixSpace of 2 by 6 matrices and dimension 2 over Integer Ring
> RMatrixSpace(BaseExtend(H,RationalField()));
KMatrixSpace of 2 by 6 matrices and dimension 2 over Rational
Field
> VectorSpace(H);
Vector space of degree 12, dimension 2 over Rational Field
User basis:
( 1  0 -2  2 -3  0  1 -1  0  1 -2  1)
( 0 -2  1  2  0 -1 -1  0  1  1 -1 -1)
```

Next we consider the endomorphism algebra of $J_0(22)$.

```
> H := End(JZero(22));
> MatrixAlgebra(H);
Matrix Algebra of degree 4 with 4 generators over Integer Ring
> RModuleWithAction(H);
RModule(IntegerRing(), 4)
Action:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
[ 0  1  0  1]
[ 0  0  0  0]
[ 1  0  1 -1]
[ 0  1  0  1]
[ 0  1  0 -1]
[ 0  1  0  0]
[-1  2 -1  1]
```

```
[-1  1  0  0]
[ 0  1 -2  1]
[-1  2 -1  0]
[-1  0  1 -1]
[ 0 -1  1 -1]
```

Example H136E76

The following example illustrates that `MatrixAlgebra` computes the algebra generated by H, even if H is not itself an algebra.

```
> H := Subgroup([HeckeOperator(JZero(33),2)]); H;
Group of homomorphisms from JZero(33) to JZero(33)
> A := MatrixAlgebra(H); A;
Matrix Algebra of degree 6 with 1 generator over Integer Ring
> Dimension(A);
2
> Dimension(H);
1
```

136.5.9 Predicates

These commands determine whether a space of homomorphisms is a ring, if it's commutative, if it's a field (and what field), if it was created using the `HeckeAlgebra` command, whether it has been tensored with $\mathbf{Q}$, or whether it is saturated in the full ring of endomorphisms. Equality and inclusion can also be tested.

`IsRing(H)`

Return `true` if the space H of homomorphisms of abelian varieties is a ring. (Note that a ring does not have to contain unity.)

`IsField(H)`

Return `true` if the space H of homomorphisms of abelian varieties is a field, and if so returns that field, a map from the field to H, and a map from H to the field.

`IsCommutative(H)`

Return `true` if and only if the space H of homomorphisms of abelian varieties is a commutative ring.

`IsHeckeAlgebra(H)`

Return `true` if the space H of homomorphisms of abelian varieties was constructed using the `HeckeAlgebra` command.

`IsOverQ(H)`

Return `true` if the space H of homomorphisms of abelian varieties is a $\mathbf{Q}$-vector space instead of just a $\mathbf{Z}$-module, i.e., a space of homomorphisms up to isogeny.

`IsSaturated(H)`

Return `true` if the space H of homomorphisms of abelian varieties is equal to its saturation, i.e., the quotient of the ambient $\mathrm{Hom}(A, B)$ by H is torsion free.

`H1 eq H2`

Return `true` if the spaces H_1 and H_2 of homomorphisms of abelian varieties are equal.

`H1 subset H2`

Return `true` if the spaces H_1 and H_2 of homomorphisms of abelian varieties are both subgroups of a common $\mathrm{Hom}(A, B)$, and in addition H_1 is a subset of H_2.

Example H136E77

We illustrate several of the commands for the endomorphism ring of $J_0(33)$.

```
> H := End(JZero(33));
> IsCommutative(H);
false
> IsField(H);
false 0 0 0
> IsHeckeAlgebra(H);
false
> IsOverQ(H);
false
> IsOverQ(BaseExtend(H,RationalField()));
true
> IsRing(H);
true
> IsSaturated(H);
true
```

Next we compare the endomorphism ring with the Hecke algebra of $J_0(33)$.

```
> T := HeckeAlgebra(JZero(33));
> T eq H;
false
> T subset H;
true
> IsSaturated(T);
true
> IsRing(T);
true
> IsHeckeAlgebra(T);
```

```
true
> IsCommutative(T);
true
> IsField(BaseExtend(T,RationalField()));
false 0 0 0
```

The Hecke algebra of $J_0(33)$ is actually a product of 3 fields, so it is not a field. In contrast, the Hecke algebra of $J_0(23)$ is a field.

```
> T := HeckeAlgebra(JZero(23));
> IsField(T);
false 0 0 0
```

In the following code, the answer you get might be different, since computation of the defining polynomial for the number field involves a randomized algorithm.

```
> IsField(BaseExtend(T,RationalField()));
true Number Field with defining polynomial x^2 + 11*x + 29 over
the Rational Field
Mapping from: Number Field with defining polynomial x^2 + 11*x +
    29 over the Rational Field to HeckeAlg(JZero(23))_Q: Group of
homomorphisms from JZero(23) to JZero(23) in the category of abelian
varieties up to isogeny  given by a rule [no inverse]
Mapping from: HeckeAlg(JZero(23))_Q: Group of homomorphisms from
JZero(23) to JZero(23) in the category of abelian varieties up to
isogeny  to Number Field with defining polynomial x^2 + 11*x + 29
over the Rational Field given by a rule [no inverse]
```

136.5.10 Elements

Elements of spaces of homomorphisms exist as maps between abelian varieties as discussed in Section 136.4.

`H ! x`

Coerce x into the space H of homomorphisms of abelian varieties. For this coercion to be successful x must be a homomorphism between 2 abelian varieties, an integer or a rational number or a matrix coercible into the matrix space of H base extended to the rational numbers.

Example H136E78

```
> H := End(JZero(22));
> H ! Matrix(Integers(), 4, 4,
>              [9, -4, 0, 2, 0, 6, 0, 0, 2, -6, 11, -2, 3, -4, -0, 8]);
Homomorphism from JZero(22) to JZero(22) given on integral homology by:
[ 9 -4  0  2]
[ 0  6  0  0]
[ 2 -6 11 -2]
```

```
[ 3 -4  0  8]
```

136.6 Arithmetic of Abelian Varieties

136.6.1 Direct Sum

One can take arbitrary finite direct sums of modular abelian varieties. We do not write `A+B` for the direct sum, since it is already used for the sum of A and B inside a common ambient abelian variety, and this sum need not be direct, unless $A \cap B = 0$.

`DirectSum(A, B)`

`DirectProduct(A, B)`

`A * B`

The direct sum D of abelian varieties A and B, together with the embedding maps from A into D and B into D, respectively, and the projection maps from D onto A and B, respectively. It is not possible to take the direct sum of abelian varieties with different signs.

`DirectSum(X)`

`DirectProduct(X)`

The direct sum D of the sequence X of modular abelian varieties, together with a list containing the embedding maps from each modular abelian variety of X into D and a list containing the projection maps from D onto each modular abelian variety in X. It is not possible to take the direct sum of abelian varieties with different signs.

`A ^ n`

The direct sum of n copies of the abelian variety A. If $n = 0$, the zero subvariety of A. If n is negative, the $(-n)$-th power of the dual of A.

Example H136E79

Using the product operator we can take the direct product of any two modular abelian varieties, even ones of different weights or levels. We first illustrate taking the product of two abelian subvarieties of $J_0(65)$, then taking the product of one of the subvarieties of $J_0(65)$ with the weight 4 motive $J_1(11, 4)$.

```
> J := JZero(65);
> D := Decomposition(J); D;
[
    Modular abelian variety 65A of dimension 1, level 5*13 and
    conductor 5*13 over Q,
    Modular abelian variety 65B of dimension 2, level 5*13 and
    conductor 5^2*13^2 over Q,
```

```
    Modular abelian variety 65C of dimension 2, level 5*13 and
    conductor 5^2*13^2 over Q
]
> A := D[1];
> B := D[2];
> A*B;
Modular abelian variety 65A x 65B of dimension 3 and level 5*13
over Q
Homomorphism from 65A to 65A x 65B given on integral homology by:
[1 0 0 0 0 0]
[0 1 0 0 0 0]
Homomorphism from 65B to 65A x 65B given on integral homology by:
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
Homomorphism from 65A x 65B to 65A (not printing 6x2 matrix)
Homomorphism from 65A x 65B to 65B (not printing 6x4 matrix)
> M := JZero(11,4);M;
Modular motive JZero(11,4) of dimension 2 and level 11 over Q
> P := A*M; P;
Modular motive 65A x JZero(11,4) of dimension 3 and level 5*11*13 over Q
```

The product also returns inclusions of each factor into the product and projection from the product onto each factor.

```
> C,f,g := A*B;
> f;
[*
Homomorphism from 65A to 65A x 65B given on integral homology by:
[1 0 0 0 0 0]
[0 1 0 0 0 0],
Homomorphism from 65B to 65A x 65B given on integral homology by:
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
*]
```

Here we compare direct sums of abelian varieties to the sum of of abelian varieties in a common ambient abelian variety. Thus if A is as above, then

```
> Dimension(A);
1
> Dimension(A*A);
2
> Dimension(A+A);
```

```
1
```

If you take a direct sum of abelian varieties that are defined over different base rings, then MAGMA will first attempt to express them over a common over-ring.

```
> A := JZero(11);
> B := BaseExtend(JZero(14),CyclotomicField(3));
> C := A*B; C;
Modular abelian variety JZero(11) x JZero(14) of dimension 2 and level
2*7*11 over Q(zeta_3)
```

The above would not work if `CyclotomicField(3)` were replaced by `GF(3)`, since the base ring $\mathbf{Q}$ of A is not contained in `GF(3)`.

136.6.2 Sum in an Ambient Variety

The sum `A+B` is the sum of A and B inside a common ambient abelian variety. This sum need not be direct, unless the intersection of A and B is 0.

`A + B`

The sum of the images of the abelian varieties A and B in a common ambient abelian variety.

`SumOf(X)`

The sum of the modular abelian varieties in the sequence X.

`SumOfImages(phi, psi)`

The sum D of the images of the morphisms ϕ and ψ of abelian varieties in their common codomain, a morphism from D into their common codomain, and a list containing a morphism from the domain of each of ϕ and ψ to D. If the codomains are not the same, then the homomorphisms are replaced by homomorphisms into an appropriate direct sum of codomains.

`SumOfMorphismImages(X)`

The sum D of the images of the morphisms of abelian varieties in the list X in their common codomain, a morphism from D into their common codomain, and a list containing a morphism from the domain of each morphism in X to D. If not all codomains of the elements of X are the same, then the homomorphisms are replaced by homomorphisms into an appropriate direct sum of codomains.

`FindCommonEmbeddings(X)`

Return `true` and a list of embeddings into a common abelian variety, if one can be found using `Embeddings(A)` for all abelian varieties A in the sequence X.

136.6.3 Intersections

Two abelian varieties cannot, by themselves, by intersected without choosing an embedding of both varieties in a common ambient abelian variety. The algorithm for computing an intersection is to compute the kernel of a certain homomorphism.

Intersections are computed in MAGMA by finding a homomorphism whose kernel is isomorphic to the intersection. For example, if $f : A \to C$ and $g : B \to C$ are injective homomorphisms, then the intersection of their images is isomorphic to the kernel of $f - g$.

As mentioned above, kernels of morphisms of abelian varieties are frequently not themselves abelian varieties. Instead a kernel is an extension of an abelian variety by a finite group of components. Likewise, intersections of abelian varieties are often not abelian varieties.

The intersection commands also take a sequence of abelian varieties or list of morphisms in order to facilitate computation of n-fold intersections, for any positive integer n.

`A meet B`

`Intersection(X)`

Given abelian varieties A and B or a sequence X of abelian varieties, compute a finite lift G of the component group of the intersection, the *connected* component of the intersection C, and a map from the abelian variety that contains C to the abelian variety that contains G. The relevant intersection is $C + G$. The elements of X are replaced by their images via their modular embedding map. All the elements of X must be embedded in the same abelian variety.

`IntersectionOfImages(X)`

Given a sequence X of morphisms from abelian varieties into a common abelian variety, compute a finite lift G of the component group of the intersection, the *connected* component C of the intersection, and a map from the abelian variety that contains C to the abelian variety that contains G. The morphisms in X do not have to be injective.

`ComponentGroupOfIntersection(A, B)`

`ComponentGroupOfIntersection(X)`

Given abelian varieties A and B or a sequence X of abelian varieties compute the group of components of the intersection of A and B or the varieties in X. (For more details, see the discussion of kernels in Section 136.4.3).

Example H136E80

The intersection of the three simple newform abelian subvarieties of $J_0(65)$ is a group isomorphic to $\mathbf{Z}/2\mathbf{Z}$.

```
> D := Decomposition(JZero(65));
> G := ComponentGroupOfIntersection(D); G;
Finitely generated subgroup of abelian variety with
invariants [ 2 ]
```

```
> FieldOfDefinition(G);
Rational Field
```

The quotient of D[1] by this subgroup of order 2 is an elliptic curve over $\mathbf{Q}$ isogenous to D[1], but not isomorphic to D[1].

```
> B  := D[1]/G; B;
Modular abelian variety of dimension 1 and level 5*13 over Q
> IsIsomorphic(D[1],B);
false
```

Next we compute some non-finite intersections.

```
> A := D[1] + D[2];
> B := D[1] + D[3];
> A meet B;
Finitely generated subgroup of abelian variety with invariants [ 2, 2, 2 ]
Modular abelian variety of dimension 1 and level 5*13 over Q
Homomorphism from modular abelian variety of dimension 1 to
modular abelian variety of dimension 6 given on integral homology
by:
[ 1 -1  0  0  0  0  1 -1  0  0  0 -1]
[ 0  0  1 -1  1 -1  0  0  1 -1  1  0]
Homomorphism from modular abelian variety of dimension 6 to
JZero(65) (not printing 12x10 matrix)
```

We can also intersect images of morphisms.

```
> f := ModularEmbedding(A);
> g := ModularEmbedding(B);
> _, C := IntersectionOfImages([* f, g *]);
> C eq D[1];
true
```

Example H136E81

The following example illustrates failure of multiplicity one for $J_0(p)$ for a prime number p. This is the first such example known, and it was discovered by Lloyd Kilford using MAGMA [Kil02]. There are two elliptic curve factors A and B inside $J_0(431)$. The eigenforms associated to A and B are congruent modulo 2, but the intersection of A and B is trivial.

```
> J := JZero(431);
> IsPrime(431);
true
> A := Decomposition(J)[1];
> B := Decomposition(J)[2];
> G, C := A meet B;
> G;
{ 0 }: finitely generated subgroup of abelian variety with
invariants []
> C;
```

```
Modular abelian variety ZERO of dimension 0 and level 431 over Q
> Newform(A) - Newform(B);
-2*q^3 + 4*q^5 + 2*q^6 - 4*q^7 + O(q^8)
```

136.6.4 Quotients

If B is an abelian subvariety of A (or some natural image of B lies in A), then the quotient A/B is an abelian variety. Also, the cokernel of a homomorphism of abelian varieties is an abelian variety.

`A / B`

The quotient of the abelian variety A by a natural image B' of the abelian variety B. Here B' is the image of B under the modular embedding composed with the modular parameterization to A.

`Cokernel(phi)`

The cokernel of the morphism ϕ of abelian varieties and a morphism from the codomain of ϕ to the cokernel.

Example H136E82

We compute a 2-dimensional quotient of the 3-dimensional abelian variety $J_0(33)$ using the Hecke operator T_2.

```
> J := JZero(33);
> T := HeckeOperator(J,2);
> Factorization(CharacteristicPolynomial(T));
[
    <x - 1, 2>,
    <x + 2, 4>
]
> C := ConnectedKernel(T-1);
> B,psi := J/C;
> B;
Modular abelian variety of dimension 2 and level 3*11 over Q
```

136.7 Decomposing and Factoring Abelian Varieties

By the Poincare reducibility theorem, every abelian variety is isogenous to a product of simple abelian subvarieties. If A is a modular abelian variety over $\mathbf{Q}$, then A is isogenous to a product of simple abelian varieties A_f attached to newforms. The `Decomposition` and `Factorization` commands compute such decompositions.

136.7.1 Decomposition

`Decomposition(A)`

Given an abelian variety A, return a sequence $[B_i]$ of simple modular abelian varieties, whose product is isogenous to A. Each B_i is equipped with an embedding into A such that the sum of the images of the B_i is equal to A. This embedding is the first element of the output of `Embeddings` on page 4547, given a B_i.

`A(n)`

The nth factor in `Decomposition(A)`, denoted $A(n)$ where A is an abelian variety.

Example H136E83

We decompose $A = J_0(37) \times J_0(22)$, then find the embedding into A of a factor which is isogenous to $J_0(11)$.

```
> A := JZero(37) * JZero(22);
> D := Decomposition(A); D;
[
    Modular abelian variety 37A of dimension 1, level 2*11*37 and
    conductor 37 over Q,
    Modular abelian variety 37B of dimension 1, level 2*11*37 and
    conductor 37 over Q,
    Modular abelian variety N(11,814,1)(11A) of dimension 1,
    level 2*11*37 and conductor 11 over Q,
    Modular abelian variety N(11,814,2)(11A) of dimension 1,
    level 2*11*37 and conductor 11 over Q
]
> B := D[3];
> Embeddings(B);
[*
Homomorphism from N(11,814,1)(11A) to JZero(37) x JZero(22) given on
integral homology by:
[ 0  0  0  0  1  0 -1  3]
[ 0  0  0  0  0  1 -2  3]
*]
```

136.7.2 Factorization

`Factorisation(A)`

`Factorization(A)`

Given an abelian variety A, compute pairwise non-isogenous simple newform abelian varieties A_f whose product, with multiplicities, is isomorphic to A. A list of pairs $\langle B, [\phi, \ldots] \rangle$ is returned, where B is an isogeny simple abelian variety and $[\phi, \ldots]$ is a sequence of maps from B into A (whose length is the "multiplicity"), such that the product of all images of all B is isogenous to A, and the sum of the dimensions of the images of B is the dimension of A. Moreover, the B are pairwise non-isogenous. To obtain a list of the images of the B canonically embedded into A, use `Decomposition(A)`.

Example H136E84

```
> A := JZero(37) * JZero(22);
> Factorization(A);
[*
<Modular abelian variety 37A of dimension 1, level 37 and
conductor 37 over Q, [
    Homomorphism N(37,814,1) from 37A to JZero(37) x JZero(22) given on
    integral homology by:
    [ 1 -1  1  0  0  0  0  0]
    [ 1 -1 -1  1  0  0  0  0]
]>,
<Modular abelian variety 37B of dimension 1, level 37 and
conductor 37 over Q, [
    Homomorphism N(37,814,1) from 37B to JZero(37) x JZero(22) given on
    integral homology by:
    [1 1 1 0 0 0 0 0]
    [0 0 0 1 0 0 0 0]
]>,
<Modular abelian variety 11A of dimension 1, level 11 and
conductor 11 over Q, [
    Homomorphism N(11,814,1) from 11A to JZero(37) x JZero(22) given on
    integral homology by:
    [ 0  0  0  0  0  1 -2  3]
    [ 0  0  0  0  1 -1  1  0],
    Homomorphism N(11,814,2) from 11A to JZero(37) x JZero(22) given on
    integral homology by:
    [ 0  0  0  0 -1  0  2 -2]
    [ 0  0  0  0 -1  2 -1  0]
]>
*]
```

136.7.3 Decomposition with respect to an Endomorphism or a Commutative Ring

The following commands use the elements of a commutative subring of endomorphisms to decompose a modular abelian variety A into a direct sum of abelian subvarieties by taking kernels (which are analogous to generalized eigenspaces).

`DecomposeUsing(R)`

Decompose an abelian variety A using the commutative ring of endomorphisms generated by the space of homomorphisms R of A.

`DecomposeUsing(phi)`

Decompose an abelian variety A using the endomorphism ϕ of A.

Example H136E85

```
> T2 := HeckeOperator(JZero(100),2);
> DecomposeUsing(T2);
[
    Modular abelian variety of dimension 1 and level 2^2*5^2 over Q,
    Modular abelian variety of dimension 5 and level 2^2*5^2 over Q,
    Modular abelian variety of dimension 1 and level 2^2*5^2 over Q
]
> W := AtkinLehnerOperator(JZero(100),100);
> DecomposeUsing(W);
[
    Modular abelian variety of dimension 3 and level 2^2*5^2 over Q,
    Modular abelian variety of dimension 4 and level 2^2*5^2 over Q
]
```

136.7.4 Additional Examples

Example H136E86

We compute a decomposition of $J_0(46)$ as a product of simple abelian subvarieties.

```
> J := JZero(46); J;
Modular abelian variety JZero(46) of dimension 5 and level 2*23 over Q
> Decomposition(J);
[
    Modular abelian variety 46A of dimension 1, level 2*23 and
    conductor 2*23 over Q,
    Modular abelian variety N(23,46,1)(23A) of dimension 2, level
    2*23 and conductor 23^2 over Q,
    Modular abelian variety N(23,46,2)(23A) of dimension 2, level
    2*23 and conductor 23^2 over Q
```

```
]
```

Thus J decomposes as a product $E \times A \times B$, where E is an elliptic curve of conductor 46, and A and B are two isogenous images of $J_0(23)$.

```
> J(1);
Modular abelian variety 46A of dimension 1, level 2*23 and
conductor 2*23 over Q
> Conductor(J(1));
46
> Factorization(Conductor(J(2)));
[ <23, 2> ]
```

The `Factorization` command gives an explicit decomposition with embeddings of each factor into $J_0(46)$.

```
> Factorization(Conductor(J(2)));
[ <23, 2> ]
> Factorization(J);
[*
<Modular abelian variety 46A of dimension 1, level 2*23 and
conductor 2*23 over Q, [
    Homomorphism from 46A to JZero(46) given on integral homology
    by:
    [ 1  0 -2 -1 -1  1  1  1 -2  1]
    [ 0  1 -1 -1  0  0  0  1 -1  0]
]>,
<Modular abelian variety 23A of dimension 2, level 23 and
conductor 23^2 over Q, [
    Homomorphism N(23,46,1) from 23A to JZero(46) given on integral
    homology by:
    [-1  1 -1  1  0 -1 -1  1 -1  2]
    [ 0  0 -1  2 -2 -1  0  0  1  0]
    [ 0  0  0  1 -2  0  0  1  0  0]
    [ 0  1  0 -1  0  0  1  0  0  0],
    Homomorphism N(23,46,2) from 23A to JZero(46) given on integral
    homology by:
    [ 0 -1  0  0  1 -1  0  1  0  0]
    [-1  0  0  0  0 -1  2 -1  1 -1]
    [-1  1 -1  0  0  0  2 -2  2 -2]
    [ 0  0 -1  2 -1  0  0  0  0 -1]
]>
*]
```

136.8 Building blocks

The abelian variety $A_f/\mathbf{Q}$ associated to a newform f is isogenous over $\overline{\mathbf{Q}}$ to a power ${B_f}^r$ of some simple variety B_f, which is called a "building block". This section is concerned with computations related to the field of definition of B_f, and its endomorphism algebra.

The functions in this section take spaces of modular symbols, and *must have sign* +1. For instance, the modular symbols of level N, weight 2 and sign +1 are obtained from `ModularSymbols(N,2,1)`. (These spaces have the same dimension as the corresponding spaces of modular forms.) Furthermore, the space of symbols is expected to be *cuspidal, new and irreducible over* $\mathbf{Q}$ (in other words, the corresponding space of cusp forms is spanned by a single Galois orbit of newforms).

Acknowledgement: This section was contributed by Jordi Quer. For more details of the theory, and some tables, see [Que09].

136.8.1 Background and Notation

Let $f = \sum_{n=1}^{\infty} a_n q^n \in S_2^{new}(N, \varepsilon)$ be a newform of weight 2, level N and Nebentypus ε. Let E be the number field generated by the coefficients a_n, and let F be the field generated by the numbers $\mu_p := a_p^2/\varepsilon(p)$ for primes p not dividing N.

The abelian variety $A_f/\mathbf{Q}$ attached to f (that is, associated to the space spanned by f and its conjugates) is a variety of dimension equal to the degree $[E : \mathbf{Q}]$ with endomorphism algebra $\mathrm{End}_{\mathbf{Q}}(A_f) \otimes \mathbf{Q}$ isomorphic to E.

The variety A_f has complex multiplication if there is a nontrivial character χ (necessarily of order 2) such that $a_p = \chi(p)a_p$ for all p not dividing N, and in this case the variety B_f is an elliptic curve with complex multiplication by the quadratic field fixed by the kernel of χ.

We assume for the rest of this section that A_f has no complex multiplication. Then, the field F is totally real, and E/F is an abelian extension. E is totally real when ε is trivial, and otherwise E is a CM field. The endomorphism algebra $\mathrm{End}(B_f) \otimes \mathbf{Q}$ is a division algebra with centre F. It is either equal to F, and then $\dim(B_f) = [F : \mathbf{Q}]$ and $r = [E : F]$, or otherwise is a quaternion algebra over F, and then $\dim(B_f) = 2[F : \mathbf{Q}]$ and $r = [E : F]/2$.

For every element $s \in G_F$ there exists a unique primitive Dirichlet character χ_s, which only depends on the action of s on the field E, such that $a_p^s = \chi_s(p)a_p$ for all p not dividing N. The characters χ_s are called the *inner twists* of the form f.

Let F_δ be the extension of F generated by the square roots of the elements $\mu_p \in F$ (for p not dividing N). Note that $EF_\delta = E(\sqrt{\varepsilon})$, the field obtained by adjoining square roots of the character values, which is either E or a quadratic extension of E. We may associate to each $s \in \mathrm{Gal}(F_\delta/F)$ a primitive quadratic Dirichlet character ψ_s, defined by $\sqrt{\mu_p}^s = \psi_s(p)\sqrt{\mu_p}$ (for p not dividing N). These are called the *quadratic degree characters* of the form f. They are related with the inner twists of f by the identity $\chi_s(p) = \psi_s(p)\sqrt{\varepsilon(p)}^s/\sqrt{\varepsilon(p)}$.

Let $K_P/\mathbf{Q}$ be the field fixed by the kernel of the homomorphism $\delta : G_{\mathbf{Q}} \to F^*/F^{*2}$ that sends $Frob_p$ to μ_p for all p not dividing N with a_p nonzero. (This is enough to determine the homomorphism, because the primes with $a_p = 0$ have density zero). This homomorphism is returned by `DegreeMap` (the syntax uses the notation below).

The Galois group $Gal(K_P/\mathbf{Q})$ is abelian of exponent 2; choose a basis $\sigma_1, ..., \sigma_r$. Let $\psi_1, ..., \psi_r$ be the basis of $\mathrm{Hom}(Gal(K_P/\mathbf{Q}), \mathbf{Z}/2)$ dual to the basis $\{\sigma_i\}$, and consider the ψ_i's as characters on $G_{\mathbf{Q}}$. Also let t_i be a quadratic discriminant such that $\mathbf{Q}(\sqrt{t_i}) \subset K_P$ is the field fixed by the kernel of ψ_i.

Denote by γ_ε the element of $Br(\mathbf{Q})[2]$ given by the two-cocycle sending σ, τ to $\sqrt{\varepsilon(\sigma)}\sqrt{\varepsilon(\tau)}\sqrt{\varepsilon(\sigma\tau)}^{-1}$. Then the class in $Br(F)[2]$ of the endomorphism algebra $\mathrm{End}^0(B_f)$ is the restriction to F of $\gamma_\varepsilon \prod(t_i, \delta(\sigma_i))$. This is computed by `BrauerClass`.

In general the smallest fields of definition up to isogeny for B_f are quadratic extensions of K_P. It is sometimes possible to "descend" B_f (up to isogeny) to the field K_P. The obstruction to this descent lies in $Br(K_P)[2]$, and is given by the restriction of γ_ε to K_P. This is computed by `ObstructionDescentBuildingBlock`.

`BoundedFSubspace(epsilon, k, degrees)`

A sequence containing the irreducible subspaces of the modular symbols of weight k and Nebentypus character ϵ corresponding to non-CM newforms for which the degree $[F : \mathbf{Q}]$ of the centre of the endomorphism algebra (of the associated abelian variety) is in the sequence *degrees*.

`HasCM(M : parameters)`

`IsCM(M : parameters)`

`Proof` BoolElt *Default :* `false`

Return `true` if and only if the modular abelian variety attached to the given space of modular symbols has complex multiplication. When the level is larger than 100, an unproved bound is used in the computation, unless `Proof` is set to `true`.

`InnerTwists(A : parameters)`

`InnerTwists(M : parameters)`

`Proof` BoolElt *Default :* `false`

The inner twists of the newform $f = \sum a_n q^n$ corresponding to the given space of modular symbols, or to the given modular abelian variety. This should be irreducible over $\mathbf{Q}$, and f should not have CM, and only spaces of modular symbols with sign $+1$ are accepted.

The inner twists are Dirichlet characters satisfying $\chi_s(p) = a_p^s/a_p$ (for primes p not dividing the level), where s is in the absolute Galois group of $\mathbf{Q}$.

Warning: when the level is larger than 100, a non-rigorous bound is used in the computation, unless `Proof` is set to `true`. Even in that case, the returned twists are only checked to be inner twists up to precision 10^{-5}.

`DegreeMap(M : parameters)`

`Proof` BoolElt *Default :* `false`

The homomorphism $\delta : G_{\mathbf{Q}} \to F^*/F^{*2}$ (as defined in the introduction), attached to the given space M of modular symbols (which should be new, irreducible over $\mathbf{Q}$, and have sign $+1$). The second object returned is F. The first object returned is a sequence of tuples $< t_i, f_i >$, where $t_i \in \mathbf{Q}$ are quadratic discriminants and $f_i \in F$. The t_i determine a basis σ_i of $\mathrm{Gal}(K_P/\mathbf{Q})$ as in the introduction, and δ is the map sending σ_i to f_i.

`BrauerClass(M)`

Given a space of modular symbols M corresponding to a newform f, this function computes the Brauer class of the endomorphism algebra of the associated abelian variety A_f (or motive M_f). The endomorphism algebra is either a number field F (in which case an empty sequence is returned), or a quaternion algebra over a number field F. The corresponding class in the Brauer group $Br(F)[2]$ is specified by returning the sequence of places of F that ramify in the quaternion algebra.

The given space M should be irreducible over $\mathbf{Q}$, and have sign $+1$, and f should not have CM.

`ObstructionDescentBuildingBlock(M)`

Given a space of modular symbols M corresponding to a newform f, this function computes the obstruction to "descending" the building block B_f to the field K_P (for definitions, see the introduction). The obstruction to the existence of a building block over K_P isogenous to B_f is an element of the Brauer group $Br(K_P)[2]$. (More precisely, for any field L there exists such a building block over L if and only if L is an extension of K_P over which this Brauer element splits.) The Brauer class is specified by returning the sequence of places of K_P where the class is not locally trivial.

The given space M should be irreducible over $\mathbf{Q}$, and have sign $+1$, and f should not have CM.

Example H136E87

The lowest level where a nontrivial obstruction occurs is 28, with a character of order 6. We find that the space of modular symbols (with sign 1) for this character has dimension 2 over the field of character values.

```
> Chi28 := FullDirichletGroup(28);
> chi := Chi28.1*Chi28.2;
> Chi28;
Group of Dirichlet characters of modulus 28
   over Cyclotomic Field of order 6 and degree 2
> Order(Chi28), Order(chi);
12 6
>  M28chi := CuspidalSubspace( ModularSymbols(chi, 2, 1));
> M28chi;
```

```
Modular symbols space of level 28, weight 2, character $.1*$.2, and dimension
2 over Cyclotomic Field of order 6 and degree 2
> qEigenform(M28chi);
q + (1/3*(-zeta_6 - 1)*a - zeta_6)*q^2 + a*q^3 + 1/3*(4*zeta_6 - 2)*a*q^4 +
    (zeta_6 - 2)*q^5 + (-zeta_6*a + (2*zeta_6 - 1))*q^6 + 1/3*(-zeta_6 - 4)*a*q^7
    + O(q^8)
> Parent(Coefficients(qEigenform(M28chi))[2]);
Univariate Quotient Polynomial Algebra in a
   over Cyclotomic Field of order 6 and degree 2
   with modulus a^2 + 3*zeta_6
```

So the q-eigenform is defined over a quadratic extension of $\mathbf{Q}(\zeta_6)$, In fact, this extension is $\mathbf{Q}(\zeta_6, i)$. Since the space `M28chi` has dimension 2 over $\mathbf{Q}(\zeta_6)$, it is irreducible over $\mathbf{Q}(\zeta_6)$. The corresponding abelian variety over $\mathbf{Q}$ therefore has dimension 4.

```
> A := ModularAbelianVariety(M28chi);
> A;
Modular abelian variety of dimension 4 and level 2^2*7 over Q with sign 1
> delta, F := DegreeMap(M28chi);
> F;
Rational Field
```

This means that A is isogenous to B^2 for some abelian variety B over $\overline{\mathbf{Q}}$ of dimension 2, and that the endomorphism algebra of B is a quaternion algebra over $F = \mathbf{Q}$.

```
> BrauerClass(M28chi);
[ 2, 3 ]
```

This means the endomorphism algebra of B is the quaternion algebra over $\mathbf{Q}$ ramified only at 2 and 3. Now we determine the possible fields of definition of B.

```
> delta;  // This came from DegreeMap, above.
[ <-7, 3> ]
```

In particular, this tells us that $K_P = \mathbf{Q}(\sqrt{-7})$.

```
> ObstructionDescentBuildingBlock(M28chi);
[ Place at Prime Ideal
Two element generators:
    [2, 0]
    [0, 1],
Place at Prime Ideal
Two element generators:
    [2, 0]
    [3, 1] ]
> Universe($1);  // What are these places elements of?
Set of Places of Number Field with defining polynomial x^2 + 7
over the Rational Field
```

The obstruction is given as a list of places of $\mathbf{Q}(\sqrt{-7})$. Recall that B can be defined over any extension of K_P for which the obstruction is trivial. In this case, any extension of $\mathbf{Q}(\sqrt{-7})$ which splits the quaternion algebra over $\mathbf{Q}(\sqrt{-7})$ ramified at the two primes above 2.

136.9 Orthogonal Complements

136.9.1 Complements

The following two commands find a complement of an abelian subvariety of an abelian variety. Existence of a complement is guaranteed by the Poincare reducibility theorem (if we were just working with n-dimensional complex tori, then there need not be complements of subtori). MAGMA computes a complement using the module-theoretic structure of the ambient variety.

`Complement(A : parameters)`

`IntPairing` BOOLELT *Default :* `false`

The complement of the image of the abelian variety A under the first embedding of A (the first map in the sequence returned by `Embeddings` on page 4547). If the parameter `IntPairing` is set to `true`, the intersection pairing is used to compute the homology complement.

`ComplementOfImage(phi : parameters)`

`IntPairing` BOOLELT *Default :* `false`

Suppose $\phi : A \to B$ is a morphism of abelian varieties. By the Poincare reducibility theorem, there is an abelian variety C such that $\phi(A) + C = B$ and the intersection of $\phi(A)$ with C is finite. This intrinsic returns a choice of C and an embedding of C into B. If the parameter `IntPairing` is set to `true`, the intersection pairing is used to compute the homology complement.

Example H136E88

We compute a decomposition of $J_0(33)$ as a product of simples, then find a decomposition of the complement of one of the factors.

```
> J := JZero(33);
> D := Decomposition(J); D;
[
    Modular abelian variety 33A of dimension 1, level 3*11 and
    conductor 3*11 over Q,
    Modular abelian variety N(11,33,1)(11A) of dimension 1, level
    3*11 and conductor 11 over Q,
    Modular abelian variety N(11,33,3)(11A) of dimension 1, level
    3*11 and conductor 11 over Q
]
> A := D[3];
> B := Complement(A);
> B;
Modular abelian variety of dimension 2 and level 3*11 over Q
> Decomposition(B);
[
    Modular abelian variety image(33A) of dimension 1, level 3*11
```

```
    and conductor 3*11 over Q,
    Modular abelian variety image(11A) of dimension 1, level 3*11
    and conductor 11 over Q
]
```

Here we compute a somewhat random map from $J_0(11)$ to $J_0(33)$, and compute the complement of the image.

```
> phi := 2*NaturalMap(JZero(11),JZero(33),1) - 3*NaturalMap(JZero(11),
>                         JZero(33),3);
> phi;
Homomorphism from JZero(11) to JZero(33) given on integral homology by:
[ 2  6 -7 -2 -6  3]
[ 5 -2 -3 -1 -1  5]
> C,pi := ComplementOfImage(phi);
> C;
Modular abelian variety of dimension 2 and level 3*11 over Q
> Decomposition(C);
[
    Modular abelian variety image(33A) of dimension 1, level 3*11
    and conductor 3*11 over Q,
    Modular abelian variety image(11A) of dimension 1, level 3*11
    and conductor 11 over Q
]
```

136.9.2 Dual Abelian Variety

It is possible to compute an abelian variety which is dual to a given variety in many of the cases that are of interest. Let A be an abelian variety and suppose the modular map $A \to J$ is injective, where J is attached to a space of modular symbols and J is isomorphic to its dual (e.g., $J = J_0(N)$). To compute the dual of A, we find a complement B of A in J whose homology is orthogonal to the homology of A with respect to the intersection pairing. This can frequently be accomplished (e.g., when A is attached to a newform) without using the intersection pairing by finding a complement B such that the rational homology of B as a module over the Hecke algebra has no simple factors in common with that of A. Then J/B is isomorphic to the dual of A.

`IsDualComputable(A)`

Return `true` if the dual of the abelian variety A can be computed, and the dual. Otherwise, return `false` and a message.

`Dual(A)`

The dual abelian variety of the abelian variety A. The modular map to a modular symbols abelian variety must be injective.

`ModularPolarization(A)`

The polarization on the abelian variety A induced by pullback of the theta divisor.

Example H136E89

We compute the dual of a 2-dimensional newform abelian variety of level 43, and note that it is isomorphic to itself.

```
> J := JZero(43);
> A := Decomposition(J)[2]; A;
Modular abelian variety 43B of dimension 2, level 43 and
conductor 43^2 over Q
> Adual := Dual(A); Adual;
Modular abelian variety of dimension 2 and level 43 over Q
> IsIsomorphic(A,Adual);
true Homomorphism from 43B to modular abelian variety of
dimension 2 given on integral homology by:
[-1  1 -1  1]
[-1  0  1  0]
[-1  0  0  0]
[ 0  0 -1  1]
```

Next we compute the dual of a 2-dimensional newform abelian variety of level 69, and find that it is not isomorphic to itself.

```
> A := Decomposition(JZero(69))[2]; A;
Modular abelian variety 69B of dimension 2, level 3*23 and
conductor 3^2*23^2 over Q
> Adual := Dual(A); Adual;
Modular abelian variety of dimension 2 and level 3*23 over Q
> IsIsomorphic(A,Adual);
false
```

One can show that the natural map from A to its dual is a polarization of degree 484.

```
> phi := NaturalMap(A,Adual);
> phi;
Homomorphism N(1) from 69B to modular abelian variety of
dimension 2 given on integral homology by:
[  3   1   5  -7]
[ -1   5  -1   1]
[ -6   4  -1   7]
[ 11  -7   6 -12]
> Degree(phi);
484
> Factorization(484);
[ <2, 2>, <11, 2> ]
1
```

136.9.3 Intersection Pairing

These commands compute the matrix of the intersection pairing on homology with respect to the fixed basis for rational or integral homology. If A is not a modular symbols abelian variety (such as $J_0(N)$), then the intersection pairing on homology computed below *is the one obtained by pulling back from the pairing on the codomain of the modular embedding* of A. This may not be what you expect, but is easy to compute in great generality.

Computation of intersection pairings is currently only implemented for weight 2.

`IntersectionPairing(H)`

The intersection pairing matrix on the basis of the homology H of an abelian variety.

`IntersectionPairing(A)`

The intersection pairing matrix on the basis for the rational homology of the abelian variety A, pulled back using the modular embedding.

`IntersectionPairingIntegral(A)`

The intersection pairing matrix on the basis for the integral homology of the abelian variety A, pulled back using the modular embedding.

Example H136E90

The intersection pairing on $J_0(11)$ is very simple.

```
> J := JZero(11);
> IntersectionPairing(J);
[ 0 -1]
[ 1  0]
> IntersectionPairingIntegral(J);
[ 0 -1]
[ 1  0]
```

The intersection pairing associated to $J_0(33)$ is more interesting. Note that the representing matrix is skew symmetric and has determinant 1.

```
> I := IntersectionPairingIntegral(JZero(33)); I;
[ 0  1  0  1  0  1]
[-1  0  0  1  0  1]
[ 0  0  0  1  0  1]
[-1 -1 -1  0  0  1]
[ 0  0  0  0  0  1]
[-1 -1 -1 -1 -1  0]
> Determinant(I);
1
```

The intersection pairing on **33A** is surprising, because it is pulled back from the intersection pairing on $J_0(33)$. Thus instead of having determinant 1, it has determinant 9.

```
> A := ModularAbelianVariety("33A");  A;
Modular abelian variety 33A of dimension 1 and level 3*11 over Q
```

```
> I := IntersectionPairingIntegral(A); I;
[ 0  3]
[-3  0]
> Determinant(I);
9
```

136.9.4 Projections

Suppose ϕ is a homomorphism from A to B. Then the image $\phi(A)$ is an abelian subvariety of B. The commands below compute a map π in the endomorphism algebra of B whose image is $\phi(A)$ and such that $\pi^2 = \pi$, i.e., π is projection onto $\phi(A)$. A projection map is not canonical, unless it is required to respect the intersection pairing.

`ProjectionOnto(A` : *parameters*`)`

`ProjectionOntoImage(phi` : *parameters*`)`

`IntPairing` BoolElt *Default :* `false`

Given a morphism $\phi : A \to B$ return a projection onto $\phi(A)$ as an element of $E \otimes \mathbf{Q}$, where E is the endomorphism ring of B. If the optional parameter `IntPairing` is set to `true`, then the projection is also required to respect the intersection pairing, which uniquely determines π.

Example H136E91

```
> pi := ProjectionOnto(ModularAbelianVariety("33A"));  pi;
Homomorphism pi from JZero(33) to JZero(33) (up to isogeny) on
integral homology by:
 (not printing 6x6 matrix)
> Matrix(pi);
[ 2/3  1/3 -1/3  1/3    0 -2/3]
[ 1/3  2/3 -2/3  2/3    0 -1/3]
[ 1/3  1/3 -1/3  1/3    0 -1/3]
[   0  2/3 -2/3  2/3    0    0]
[   0  1/3 -1/3  1/3    0    0]
[-1/3  1/3 -1/3  1/3    0  1/3]
> pi^2 eq pi;
true
> Rank(pi);
1
```

Example H136E92

```
> phi := NaturalMap(JZero(11),JZero(44));
> pi := ProjectionOntoImage(phi); pi;
Homomorphism pi from JZero(44) to JZero(44) (up to isogeny) on
integral homology by:
 (not printing 8x8 matrix)
> A := Image(5*pi); A;
Modular abelian variety of dimension 1 and level 2^2*11 over Q
> IsIsomorphic(JZero(11),A);
true Homomorphism from JZero(11) to modular abelian variety of
dimension 1 given on integral homology by:
[ 0 -1]
[-1  1]
```

136.9.5 Left and Right Inverses

Left and right inverses of homomorphisms in the category of abelian varieties up to isogeny can be computed. A left or right inverse times a minimal integer can be computed instead so that the result is a homomorphism.

MAGMA computes a right inverse of a finite-degree homomorphism ϕ by finding the projection map onto the complement of the image of ϕ, and composing with a inverse from the image. To find a left inverse of a surjective homomorphism $\phi : A \to B$, MAGMA computes the complement C of the kernel of ϕ and inverts ϕ restricted to C. This complement C is an abelian subvariety of A that maps isomorphically onto B.

`LeftInverse(phi : parameters)`

`IntPairing` BOOLELT *Default :* `false`

Given a surjective homomorphism $\phi : A \to B$ of abelian varieties return a homomorphism $\psi : B \to A$ of minimal degree in the category of abelian varieties up to isogeny such that $\psi * \phi$ is the identity map on B and an integer d such that $d * \psi$ is a homomorphism.

If the parameter `IntPairing` is `true` then the intersection pairing is used to compute the homology complement.

`LeftInverseMorphism(phi : parameters)`

`IntPairing` BOOLELT *Default :* `false`

Given a surjective homomorphism $\phi : A \to B$ of abelian varieties, return a homomorphism $\psi : B \to A$ of minimal degree such that $\psi * \phi$ is multiplication by an integer.

If the parameter `IntPairing` is `true` then the intersection pairing is used to compute the homology complement.

`RightInverse(phi : parameters)`

`IntPairing` BoolElt *Default :* `false`

Given a homomorphism $\phi : A \to B$ of abelian varieties with finite kernel, return a map $\psi : B \to A$ in the category of abelian varieties up to isogeny such that $\phi * \psi : A \to A$ is the identity map.

If the parameter `IntPairing` is `true` then the intersection pairing is used to compute the homology complement.

`RightInverseMorphism(phi : parameters)`

`IntPairing` BoolElt *Default :* `false`

Given a homomorphism $\phi : A \to B$ of abelian varieties with finite kernel return a minimal-degree homomorphism $\psi : B \to A$ such that $\phi * \psi : A \to A$ is multiplication by an integer.

If the parameter `IntPairing` is `true` then the intersection pairing is used to compute the homology complement.

Example H136E93

First we compute the difference ϕ of the two natural degeneracy maps $J_0(11) \to J_0(33)$, which has as kernel a group of order 5 (called the Shimura subgroup in this case).

```
> J11 := JZero(11);  J33 := JZero(33);
> d1 := NaturalMap(J11,J33,1);
> d3 := NaturalMap(J11,J33,3);
> phi := d1-d3;
> Degree(phi);
5
```

A right inverse of ϕ is a homomorphism up to isogeny from $J_0(33)$ to $J_0(11)$.

```
> RightInverse(phi);
Homomorphism from JZero(33) to JZero(11) (up to isogeny) on integral
homology by:
 (not printing 6x2 matrix)
15
> RightInverseMorphism(phi);
Homomorphism from JZero(33) to JZero(11) (not printing 6x2 matrix)
> phi*RightInverseMorphism(phi);
Homomorphism from JZero(11) to JZero(11) given on integral homology by:
[15  0]
[ 0 15]
```

Finally we find a left inverse of a map from $J_0(33)$ to $J_0(11)$.

```
> psi := NaturalMap(J33,J11,1) - NaturalMap(J33,J11,3);
> IsSurjective(psi);
true
> LeftInverse(psi);
```

```
Homomorphism from JZero(11) to JZero(33) (up to isogeny) on integral
homology by:
[ 1/5    0 -2/5  2/5 -3/5    0]
[ 1/5 -1/5    0  1/5 -2/5  1/5]
5
> LeftInverseMorphism(psi);
Homomorphism from JZero(11) to JZero(33) given on integral homology by:
[ 1  0 -2  2 -3  0]
[ 1 -1  0  1 -2  1]
> LeftInverseMorphism(psi)*psi;
Homomorphism from JZero(11) to JZero(11) given on integral homology by:
[5 0]
[0 5]
```

136.9.6 Congruence Computations

The congruence modulus and the modular degree are each an integer which measures "congruences" between an abelian variety and other abelian varieties. These two quantities are related because if a prime divides the modular degree, then it divides the congruence modulus, though the converse need not be true (see the example below).

`CongruenceModulus(A)`

Given $A = A_f$, an abelian variety attached to a newform f, return the congruence modulus of the newform f, taken in the space $S_2(N, \epsilon)$, where ϵ is the character of the newform, which is an integer that measures congruences between f and nonconjugate forms in the Peterson complement of f. More precisely, if $f \in S_k(N, \varepsilon)$, which is the direct sum of the spaces of modulus forms with character a Galois conjugate of ε, then we define the congruence modulus of f to be the order of the group $S_k(N, \varepsilon; \mathbf{Z})/(W + W^{\perp})$, where W is the intersection of $S_k(N, \varepsilon; \mathbf{Z})$ with the span of the Galois conjugates of f.

`ModularDegree(A)`

The modular degree of the abelian variety A. This is the square root of the degree of the map from A to its dual A' induced by virtue of A being modular. In some cases where no algorithm is implemented for computing A', a message is printed and the square of the degree of the composition of the modular embedding with the modular parameterization is computed. When any weight of a factor of A is bigger than 2 the square root is not taken.

Example H136E94

The modular degree and congruence modulus of one of the two elliptic curves of conductor 54 are interesting because they are not equal. This is the smallest level of an elliptic curve where these two invariants differ. (For more details, see [AS04].)

```
> J := JZero(54);
> A,B := Explode(Decomposition(NewSubvariety(J)));
> ModularDegree(A);
6
> CongruenceModulus(A);
6
> ModularDegree(B);
2
> CongruenceModulus(B);
6
```

The modular degree and congruence modulus are 4 for a certain abelian surface A of level 65. We also compute the kernel of the modular map and see that it is $A[2]$.

```
> J := JZero(65);
> A := J(2); A;
Modular abelian variety 65B of dimension 2, level 5*13 and
conductor 5^2*13^2 over Q
> CongruenceModulus(A);
4
> ModularDegree(A);
4
> phi := NaturalMap(A,Dual(A));
> Invariants(Kernel(phi));
[ 2, 2, 2, 2 ]
```

136.10 New and Old Subvarieties and Natural Maps

136.10.1 Natural Maps

Suppose M and N are positive integers and M divides N. There are natural maps in both directions between $J_0(N)$ and $J_0(M)$ (and likewise for J_1, etc.), for each divisor of $t = N/M$, which correspond to maps of the form $f(q) \mapsto f(q^t)$ and their duals. Since any modular abelian variety A in MAGMA is equipped with a map $A \to J_e$ and $J_p \to A$, where J_e and J_p are attached to modular symbols, the problem of defining natural maps between A and B is reduced to defining natural maps between modular abelian varieties attached to modular symbols.

`NaturalMap(A, B, d)`

Given abelian varieties A and B and an integer d return the natural map from A to B induced, in a potentially complicated way, from the map $f(q) \mapsto f(q^d)$ on modular forms. In situations where the modular forms associated to A and B have nothing to do with each other, then we define this map to be the zero map.

`NaturalMap(A, B)`

The natural map from the abelian variety A to the abelian variety B induced by the identity on modular forms, or the zero map if there is none.

`NaturalMaps(A, B)`

Given abelian varieties A and B return a sequence of the natural maps from A to B for all divisors d of the level of A over the level B, or the level of B over the level A.

Example H136E95

```
> A := JZero(11)*JZero(22);
> B := JZero(11)*JZero(33);
> phi := NaturalMap(A,B);
> phi;
Homomorphism N(1) from JZero(11) x JZero(22) to JZero(11) x JZero(33) (not
printing 6x8 matrix)
> Nullity(phi);
1
> f := NaturalMap(A,B,3); f;
Homomorphism N(3) from JZero(11) x JZero(22) to JZero(11) x JZero(33) (not
printing 6x8 matrix)
> Nullity(f);
2
> NaturalMaps(JZero(11),JZero(33));
[
    Homomorphism N(1) from JZero(11) to JZero(33) given on integral
    homology by:
    [ 1  0 -2  2 -3  0]
    [ 1 -1  0  1 -2  1],
    Homomorphism N(3) from JZero(11) to JZero(33) given on integral
    homology by:
    [ 0 -2  1  2  0 -1]
    [-1  0  1  1 -1 -1]
]
```

If we take a product of several copies of $J_0(11)$ and of several copies of $J_0(22)$, the `NaturalMaps` command still only returns 2 natural maps, one for each divisor of the quotient of the levels.

```
> A := JZero(11)^2;
> B := JZero(22)^3;
```

```
> NaturalMaps(A,B);
[
    Homomorphism N(1) from JZero(11) x JZero(11) to JZero(22) x JZero(22) x
    JZero(22) given on integral homology by:
    [ 0  1 -2  3  0  1 -2  3  0  1 -2  3]
    [ 1 -1  1  0  1 -1  1  0  1 -1  1  0]
    [ 0  1 -2  3  0  1 -2  3  0  1 -2  3]
    [ 1 -1  1  0  1 -1  1  0  1 -1  1  0],
    Homomorphism N(2) from JZero(11) x JZero(11) to JZero(22) x JZero(22) x
    JZero(22) given on integral homology by:
    [-1  0  2 -2 -1  0  2 -2 -1  0  2 -2]
    [-1  2 -1  0 -1  2 -1  0 -1  2 -1  0]
    [-1  0  2 -2 -1  0  2 -2 -1  0  2 -2]
    [-1  2 -1  0 -1  2 -1  0 -1  2 -1  0]
]
```

136.10.2 New Subvarieties and Quotients

These commands compute the new and r-new subvarieties and quotients of an abelian variety A of level N. The *r-new subvariety* of A is the intersection of the kernels of all natural maps from A to modular abelian varieties of level N/r. The *new subvariety* is the intersection of the r-new subvarieties over all prime divisors r of N. The *r-new quotient* of A is the quotient of A by the sum of all images in A under all natural maps of abelian varieties of level N/r.

`NewSubvariety(A, r)`

The r-new subvariety of the abelian variety A.

`NewSubvariety(A)`

The new subvariety of the abelian variety A.

`NewQuotient(A, r)`

The r-new quotient of the abelian variety A.

`NewQuotient(A)`

The new quotient of the abelian variety A.

Example H136E96

```
> J := JZero(33);
> Dimension(J);
3
> Dimension(NewSubvariety(J,3));
1
> Dimension(NewSubvariety(J));
1
> Dimension(NewSubvariety(J,11));
3
> Dimension(NewQuotient(J));
1
> Dimension(OldSubvariety(J));
2
> Dimension(OldSubvariety(J,3));
2
```

136.10.3 Old Subvarieties and Quotients

These commands compute the old and r-old subvarieties and quotients of an abelian variety A of level N. The *r-old subvariety* of A is the sum of the images of all natural maps from modular abelian varieties of level N/r to A. The *old subvariety* is the sum of the r-old subvarieties as r varies over the divisors of N. The r-old quotient of A is the quotient of A by its r-new subvariety.

`OldSubvariety(A, r)`

The r-old subvariety of the abelian variety A.

`OldSubvariety(A)`

The old subvariety of the abelian variety A.

`OldQuotient(A, r)`

The r-old quotient of the abelian variety A.

`OldQuotient(A)`

The old quotient of the abelian variety A.

Example H136E97

We compute the old subvariety and old quotient of $J_0(100)$, both of which have dimension 6.

```
> J := JZero(100); J;
Modular abelian variety JZero(100) of dimension 7 and level 2^2*5^2
over Q
> J_old := OldSubvariety(J); J_old;
Modular abelian variety JZero(100)_old of dimension 6 and level
2^2*5^2 over Q
> phi := Embeddings(J_old)[1];
> Codomain(phi);
Modular abelian variety JZero(100) of dimension 7 and level 2^2*5^2
over Q
> Jold := OldQuotient(J); Jold;
Modular abelian variety JZero(100)^old of dimension 6 and level
2^2*5^2 over Q
```

The new subvariety and new quotient of $J_0(100)$ intersect in a finite subgroup isomorphic to $\mathbf{Z}/12\mathbf{Z} \times \mathbf{Z}/12\mathbf{Z}$.

```
> J_new := NewSubvariety(J); J_new;
Modular abelian variety JZero(100)_new of dimension 1 and level
2^2*5^2 over Q
> G, A := J_new meet J_old; G;
Finitely generated subgroup of abelian variety with invariants
[ 12, 12 ]
> Dimension(A);
0
```

136.11 Elements of Modular Abelian Varieties

We represent torsion points on modular abelian varieties as follows. Suppose A is an abelian variety defined over the complex numbers $\mathbf{C}$. Then $A(\mathbf{C})$ is canonically isomorphic to $H_1(A, \mathbf{R})/H_1(A, \mathbf{Z})$, and the torsion subgroup of $A(\mathbf{C})$ is isomorphic to $H_1(A, \mathbf{Q})/H_1(A, \mathbf{Z})$. We represent a torsion element of $A(\mathbf{C})$ by giving a representative element of $H_1(A, \mathbf{Q})$. The functions below provide basic arithmetic operations with such elements, application of homomorphisms, and conversion functions.

Sometimes it is useful to consider elements of $H_1(A, \mathbf{R})$, given by floating point vectors (i.e., over `RealField()`). These represent certain points of infinite order, but without further information we do not know exactly what point they represent, or even whether such a point is 0.

Elements can only be created independently of other elements by coercion, see Section 136.2.14.

136.11.1 Arithmetic

The following commands describe the basic arithmetic operations available for elements of modular abelian varieties. Operations include addition, subtraction, and multiplication by an integer, rational number, or real number.

`a * x`

Product of the integer, rational or real number a by the element x of a modular abelian variety.

`x * a`

Product of the element x of a modular abelian variety by the integer, rational or real number a.

`x + y`

The sum of elements x and y of a modular abelian variety.

`x - y`

The difference of elements x minus y of a modular abelian variety.

Example H136E98

In this example, we construct $J_0(23)$, and consider the finite subgroup $\ker(T_3 - 5)$, which has order 400. We then do various arithmetic operations with some of its elements.

```
> A := JZero(23);
> t3 := HeckeOperator(A,3);
> Factorization(CharacteristicPolynomial(t3));
[
    <x^2 - 5, 2>
]
> G := Kernel(t3-5);
> #G;
400
> Generators(G);
[
    Element of abelian variety defined by [1/10 0 1/10 1/5] modulo homology,
    Element of abelian variety defined by [0 0 0 -5/2] modulo homology,
    Element of abelian variety defined by [1/10 -1/10 0 -1/5] modulo homology,
    Element of abelian variety defined by [1 -3/2 2 1] modulo homology
]
> x := G.1;
> 1.5*x;
Element of abelian variety defined by [0.14999999999999999999999999998 0.E-28
0.14999999999999999999999999998
0.29999999999999999999999999996] modulo homology
> (3/2)*x;
Element of abelian variety defined by [3/20 0 3/20 3/10] modulo homology
> 10*x;
```

```
0
> x*1.5;
Element of abelian variety defined by [0.14999999999999999999999999998 0.E-28
0.14999999999999999999999999998
0.29999999999999999999999999996] modulo homology
> 1.5*x eq x*1.5;
true
> x*(3/2);
Element of abelian variety defined by [3/20 0 3/20 3/10] modulo homology
> x*5;
Element of abelian variety defined by [1/2 0 1/2 1] modulo homology
> G.1 + G.2;
Element of abelian variety defined by [1/10 0 1/10 -23/10] modulo homology
> G.1 - G.2;
Element of abelian variety defined by [1/10 0 1/10 27/10] modulo homology
```

136.11.2 Invariants

These commands compute information about the order of an element, the degree of the homology of the parent variety, and a field that the point is defined over.

`Order(x)`

Given an element of a modular abelian variety x, return the order of x, if x is known exactly. Otherwise an error occurs.

`ApproximateOrder(x)`

Given a point x on a modular abelian variety return the exact order of x, if x is known exactly as a torsion point, and if not the order of an approximation of x by a torsion point, obtained using continued fractions.

`Degree(x)`

The dimension of the homology of the parent of x, where x is an element of a modular abelian variety.

`FieldOfDefinition(x)`

A field that x is defined over, which need not be minimal, where x is an element of a modular abelian variety.

Example H136E99

We compute a 2-torsion point on the elliptic curve $J_0(11)$, compute some approximate orders, and compute the degree.

```
> A := JZero(11);
> G := Kernel(nIsogeny(A,2));
> G;
Finitely generated subgroup of abelian variety with invariants
[ 2, 2 ]
> x := G.1;
> ApproximateOrder(Sqrt(2)*x);
175568277047523
> ApproximateOrder(1.000000000000001*x);
2
> Degree(x);
2
```

Notice that `FieldOfDefinition(x)` is valid, but far from optimal. It would be better to return the number field generated by the 2-torsion point.

```
> FieldOfDefinition(x);
Algebraically closed field with no variables
> FieldOfDefinition(0*x);
Rational Field
> Order(x);
2
```

136.11.3 Predicates

These are commands for testing equality, inclusion, whether an element is 0, and whether an element is known exactly, (i.e., as an element of $H_1(A, \mathbf{Q})$, or just as an element of $H_1(A, \mathbf{R})$).

`x eq y`

Return `true` if the elements x and y of a modular abelian variety are equal.

`x in X`

Return `true` if the element x of a modular abelian variety is an element of the list X.

`IsExact(x)`

Return `true` if the element x of a modular abelian variety is known exactly, i.e., x is defined by an element of the rational homology.

IsZero(x)

Return `true` if the element x of a modular abelian variety is known exactly and is equal to 0. If x is not known exactly, return `true` if a real homology vector that represents x is "very close" to an element of the integral homology, where very close means that the distance is within $1/10^n$, where n is `M‘point_precision` and M is the parent of x.

Example H136E100

We demonstrate each of these commands using elements of the 2-torsion subgroups of the two elliptic curves of conductor 37.

```
> J := JZero(37);
> A, B := Explode(Decomposition(J));
> A;
Modular abelian variety 37A of dimension 1, level 37 and
conductor 37 over Q
> B;
Modular abelian variety 37B of dimension 1, level 37 and
conductor 37 over Q
> A2 := Kernel(nIsogeny(A,2));
> B2 := Kernel(nIsogeny(B,2));
> x := A2.1;
> y := B2.2;
> x eq y;
false
> x in [* x, y *];
true
> IsZero(x);
false
> IsZero(0*x);
true
> IsExact(1.00000000000000000001*x);
false
> IsExact((2/3)*x);
true
```

For non-exact elements, `IsZero` means "is quite close to 0".

```
> IsZero(0.0001*x);
false
> IsZero(0.00001*x);
true
> IsZero(0.000000000001*x);
true
> A‘point_precision;
10
```

136.11.4 Homomorphisms

There are two notations for applying a homomorphism to an element. One can find an inverse image of an element using the @@ command.

```
x @ phi
```

```
phi(x)
```

The image of the element x of a modular abelian variety under the homomorphism ϕ of abelian varieties.

```
x @@ phi
```

An inverse image of the element x of a modular abelian variety under the homomorphism ϕ of abelian varieties.

Example H136E101

Let $\phi = T_3 - 5$ acting on the abelian surface $J_0(23)$. We apply ϕ to an element of the kernel G of ϕ, and get 0. We also find an element y such that $\phi(y)$ is a certain element of G.

```
> A := JZero(23);
> phi := HeckeOperator(A,3) - 5;
> G := Kernel(phi);
> x := G.1;
> Order(x);
10
> phi(x);
0
> zero := A!0;
> z := zero@@phi; z;
0
> y := x@@phi; y;
Element of abelian variety defined by [-1/20 1/20 -1/20 -1/20] modulo homology
> phi(y) in G;
true
> y@phi eq phi(y);
true
```

136.11.5 Representation of Torsion Points

An exact torsion point representation of an element of a modular abelian variety can be found using continued fractions to find good rational approximations for each coordinate of a representative real homology class. A representative element of the homology can also be retrieved.

The `Eltseq` command gives the sequence of entries of the vector returned by `Element`.

`ApproximateByTorsionPoint(x : parameters)`

`Cutoff` RngIntElt *Default :* 10^3

If the modular abelian variety element x is defined by an element z in the real homology $H_1(A, R)$, find an element of $H_1(A, \mathbf{Q})$ which approximates z, using continued fractions, and return the corresponding point.

`Element(x)`

The vector in homology which represents the element x of a modular abelian variety.

`LatticeCoordinates(x)`

A vector over the rational or real field which represents the element x with respect to the basis for integral homology of the parent abelian variety of x.

`Eltseq(x)`

The `Eltseq` of `LatticeCoordinates(x)` where x is an element of a modular abelian variety.

Example H136E102

This code illustrates each of the commands for a 3-torsion point in $J_0(33)$.

```
> A := JZero(33);
> x := A![1/3,0,0,0,0,0];
> x;
Element of abelian variety defined by [1/3 0 0 0 0 0] modulo homology
> Order(x);
3
> ApproximateByTorsionPoint(1.001*x);
Element of abelian variety defined by [1001/3000 0 0 0 0 0] modulo homology
> Element(x);
(1/3   0   0   0   0   0)
> Eltseq(x);
[ 1/3, 0, 0, 0, 0, 0 ]
> LatticeCoordinates(x);
(1/3   0   0   0   0   0)
```

The `Element` and `LatticeCoordinates` can differ when the integral structure on the homology is complicated. This is common when the weight is bigger than 2.

```
> A := JZero(11,4); A;
```

```
Modular motive JZero(11,4) of dimension 2 and level 11 over Q
> x := A![1/3,0,0,0];
> Element(x);
(  1/8  1/24 -1/24 -1/24)
> Eltseq(x);
[ 1/3, 0, 0, 0 ]
> LatticeCoordinates(x);
(1/3   0   0   0)
> x;
Element of abelian variety defined by [1/3 0 0 0] modulo homology
```

136.12 Subgroups of Modular Abelian Varieties

136.12.1 Creation

Subgroups can be created from a sequence of elements of a modular abelian variety which generate it. It is also possible to create n-torsion subgroups $A[n]$ and to create subgroups as kernels of homomorphisms or by taking an image of the difference of two cusps.

If a subgroup G contains elements that are not known exactly (i.e., they are defined by floating point approximations to real homology elements), a group of torsion points that approximates G can be found.

`Subgroup(X)`

The subgroup of A generated by the nonempty sequence X of elements of modular abelian variety A.

`ZeroSubgroup(A)`

The zero subgroup of the abelian variety A.

`nTorsionSubgroup(A, n)`

The kernel $A[n]$ of the multiplication by n isogeny on the modular abelian variety A.

`nTorsionSubgroup(G, n)`

The kernel $G[n]$ of the multiplication by n homomorphism on the subgroup G of a modular abelian variety.

`ApproximateByTorsionGroup(G : parameters)`

`Cutoff` RngIntElt *Default :* 10^3

The subgroup generated by torsion approximations of a set of generators of the subgroup G of a modular abelian variety.

Example H136E103

First we list the elements of the 2-torsion subgroup of the elliptic curve 100A, then we compute the 0 subgroup.

```
> A := ModularAbelianVariety("100A"); A;
Modular abelian variety 100A of dimension 1 and level 2^2*5^2
over Q
> G := nTorsionSubgroup(A,2); G;
Finitely generated subgroup of abelian variety with invariants [ 2, 2 ]
> Elements(G);
[
    0,
    Element of abelian variety defined by [1/2 0] modulo homology,
    Element of abelian variety defined by [0 1/2] modulo homology,
    Element of abelian variety defined by [1/2 1/2] modulo homology
]
> ZeroSubgroup(A);
{ 0 }: finitely generated subgroup of abelian variety with
invariants []
```

We can also use the `nTorsionSubgroup` command on subgroups.

```
> nTorsionSubgroup(G,2);
Finitely generated subgroup of abelian variety with
invariants [ 2, 2 ]
> nTorsionSubgroup(G,3);
{ 0 }: finitely generated subgroup of abelian variety with
invariants []
```

One of the 2-torsion elements generates a subgroup H of order 2.

```
> G.1;
Element of abelian variety defined by [0 1/2] modulo homology
> H := Subgroup([G.1]); H;
Finitely generated subgroup of abelian variety
> #H;
2
```

To illustrate the approximation command, we consider the subgroup generated by an approximation to one of the 2-torsion elements.

```
> K := Subgroup([1.00001*G.1]);
> L := ApproximateByTorsionGroup(K);
Finitely generated subgroup of abelian variety with
invariants [ 2 ]
> L eq H;
true
```

136.12.2 Elements

These commands enumerate elements of a finite subgroup of a modular abelian variety and allow access to the generators.

`Elements(G)`

A sequence of all elements of the finite subgroup G of a modular abelian variety.

`Generators(G)`

A sequence of generators for the subgroup G of a modular abelian variety. These correspond to generators for the underlying abelian group.

`Ngens(G)`

The number generators of the subgroup G of a modular abelian variety.

`G . i`

The i-th generator of the subgroup G of a modular abelian variety.

Example H136E104

We illustrate each of the commands using the kernel of the Hecke operator T_3 acting on $J_0(67)$.

```
> J := JZero(67);
> T := HeckeOperator(J,3);
> G := Kernel(T);
> #G;
4
> Elements(G);
[
    0,
    Element of abelian variety defined by [0 0 1/2 0 0 -1/2 -1/2 0 1/2 0] modulo
    homology,
    Element of abelian variety defined by [1/2 -1/2 0 0 -1/2 0 0 -1/2 1/2 0]
    modulo homology,
    Element of abelian variety defined by [1/2 -1/2 1/2 0 -1/2 -1/2 -1/2 -1/2 1 0]
    modulo homology
]
> Generators(G);
[
    Element of abelian variety defined by [1/2 -1/2 0 0 -1/2 0 0 -1/2 1/2 0]
    modulo homology,
    Element of abelian variety defined by [0 0 1/2 0 0 -1/2 -1/2 0 1/2 0]
    modulo homology
]
> Ngens(G);
2
> G.1;
Element of abelian variety defined by [1/2 -1/2 0 0 -1/2 0 0 -1/2 1/2 0] modulo
homology
```

```
> G.2;
Element of abelian variety defined by [0 0 1/2 0 0 -1/2 -1/2 0 1/2 0] modulo
homology
```

136.12.3 Arithmetic

Quotients of abelian varieties by finite subgroups can be formed, finite subgroups can be intersected with other finite subgroups or abelian varieties, and the group generated by two subgroups can be computed.

For several of the arithmetic operations below, finite groups or abelian varieties are replaced by their image in a common abelian variety, so the operation makes sense. This common abelian variety is the one returned by `FindCommonEmbeddings` on page 4594. Note that the "embedding" is only guaranteed to be an embedding up to isogeny.

`Quotient(A, G)`

The quotient of the abelian variety A by the finite subgroup G of A, the isogeny $A \to A/G$ and an isogeny $A/G \to A$, such that composition of the two isogenies is multiplication by the exponent of G.

`Quotient(G)`

The quotient A/G, where A is the ambient variety of the subgroup G of an abelian variety, an isogeny from A to A/G with kernel G, and an isogeny from A/G to A such that the composition of the two isogenies is multiplication by the exponent of G.

`A / G`

Given an abelian variety A and a subgroup G of an abelian variety return the quotient A/G, the isogeny $A \to A/G$ with kernel G, and an isogeny $A/G \to A$ where A is not necessarily the ambient variety of G.

`A meet G`
`G meet A`

The intersection of the finite subgroup G of an abelian variety B with the abelian variety A. If A is not equal to B, then G and A are replaced by their image in a common abelian variety.

`G1 + G2`

The sum of the subgroups G_1 and G_2 of abelian varieties A_1 and A_2. If A_1 is not equal to A_2, then G_1 and G_2 are replaced by their image in a common abelian variety.

`G1 meet G2`

The intersection of the finite subgroups G_1 and G_2 of an abelian variety. If their ambient varieties are not equal, G_1 and G_2 are replaced by their image in a common abelian variety.

Example H136E105

We illustrate these commands using the 2-torsion of $J_0(67)$. First we compute the kernel of T_3, which is a 2-torsion group of order 4.

```
> J := JZero(67); J;
Modular abelian variety JZero(67) of dimension 5 and level 67 over Q
> T := HeckeOperator(J,3);
> Factorization(CharacteristicPolynomial(T));
[
    <x + 2, 2>,
    <x^2 - x - 1, 2>,
    <x^2 + 3*x + 1, 2>
]
> G := Kernel(T); #G;
4
```

Next we quotient $J_0(67)$ out by this subgroup of order 4.

```
> A := Quotient(J,G); A;
Modular abelian variety of dimension 5 and level 67 over Q
```

The result is, of course, isogenous to $J_0(67)$. Unfortunately, testing of isomorphism in this generality is not yet implemented.

```
> IsIsogenous(A,J);
true
> Degree(ModularParameterization(A));
4
```

If the `Quotient` command is given only one argument then the variety being quotiented out by is the ambient variety.

```
> B := Quotient(G); B;
Modular abelian variety of dimension 5 and level 67 over Q
> Degree(ModularParameterization(B));
4
```

We can also use the divides notation for quotients.

```
> C := J/G; C;
Modular abelian variety of dimension 5 and level 67 over Q
```

Next we list the 2-torsion subgroups of the simple factors of $J_0(67)$. Interestingly, the sum of the 2-torsion subgroups of these simple factors is much smaller than the full 2-torsion subgroup $J_0(67)[2]$.

```
> D := Decomposition(J); D;
[
    Modular abelian variety 67A of dimension 1, level 67 and
    conductor 67 over Q,
    Modular abelian variety 67B of dimension 2, level 67 and
    conductor 67^2 over Q,
```

```
    Modular abelian variety 67C of dimension 2, level 67 and
    conductor 67^2 over Q
]
> for A in D do print #(A meet G); end for;
4
1
1
> G2 := nTorsionSubgroup(D[2],2);
> G3 := nTorsionSubgroup(D[3],2);
> H := G + G2 + G3;
> #H;
64
> H eq nTorsionSubgroup(J,2);
false
> #nTorsionSubgroup(J,2);
1024
> G2 eq G3;
true
> G meet G2;
{ 0 }: finitely generated subgroup of abelian variety with invariants []
```

136.12.4 Underlying Abelian Group and Lattice

`AbelianGroup(G)`

Let G be a finitely generated subgroup of an abelian variety A. Return an abstract abelian group H which is isomorphic to G along with isomorphisms in both directions.

`Lattice(G)`

Let G be a finite torsion subgroup of its ambient abelian variety A whose elements are all known exactly, i.e. G is generated by elements of $H_1(A, \mathbf{Q})/H_1(A, \mathbf{Z})$. Return the lattice L in the rational homology of A generated by $H_1(A, \mathbf{Z})$ and all x such that G can be viewed as a set of equivalence classes of the form $x + H_1(A, \mathbf{Z})$.

Example H136E106

This examples illustrate these commands for the 3-torsion subgroup of $J_0(11)$.

```
> A := JZero(11);
> G := nTorsionSubgroup(A,3);
> H,f,g := AbelianGroup(G);
> H;
Abelian Group isomorphic to Z/3 + Z/3
Defined on 2 generators
Relations:
    3*H.1 = 0
```

```
    3*H.2 = 0
```

The lattice of G is $1/3$ times the integral homology.

```
> Lattice(G);
Lattice of rank 2 and degree 2
Basis:
[Identity matrix]
Basis Denominator: 3
> L := IntegralHomology(A); L;
Standard Lattice of rank 2 and degree 2
> Lattice(G)/L;
Abelian Group isomorphic to Z/3 + Z/3
Defined on 2 generators
Relations:
    3*$.1 = 0
    3*$.2 = 0
```

136.12.5 Invariants

`AmbientVariety(G)`

Let G be a finitely generated subgroup of an abelian variety. Return the abelian variety whose elements were used to create G.

`Exponent(G)`

Given a finitely generated subgroup G of an abelian variety, return the smallest positive integer e which kills G, i.e. such that $eG = 0$. We assume G is finite.

`Invariants(G)`

Given a finitely generated subgroup of an abelian variety, return the invariants of an abstract abelian group isomorphic to G.

`Order(G)`

`#G`

Given a finitely generated subgroup G of an abelian variety, return the number of elements in G, when G is known to be finite (an error occurs otherwise).

`FieldOfDefinition(G)`

Given a finitely generated subgroup G of an abelian variety return a field over which the group G is defined. This is a field K such that if σ is an automorphism that fixes K, then $\sigma(G) = G$. Note that K *is not guaranteed to be minimal.*

Example H136E107

We illustrate each command using the kernel of T_3 on $J_0(67)$.

```
> A := JZero(67);
> T3 := HeckeOperator(A,3);
> G := Kernel(T3); G;
Finitely generated subgroup of abelian variety with
[ 2, 2 ]
> AmbientVariety(G);
Modular abelian variety JZero(67) of dimension 5 and level 67 over Q
> Exponent(G);
2
> Invariants(G);
[ 2, 2 ]
> Order(G);
4
> #G;
4
```

The field of definition of G is $\mathbf{Q}$, since G is the kernel of a homomorphism defined over $\mathbf{Q}$ (a Hecke operator).

```
> FieldOfDefinition(G);
Rational Field
```

However, the field of definition of the subgroup of G generated by one of the elements of G could take significant extra work to determine. Currently MAGMA simply chooses the easiest answer, which is $\overline{\mathbf{Q}}$.

```
> H := Subgroup([G.1]);
> FieldOfDefinition(H);
Algebraically closed field with no variables
```

136.12.6 Predicates and Comparisons

A subgroup G of an abelian variety is finite exactly when every element of G is known exactly, since then all elements are torsion and G is finitely generated.

Abelian varieties and subgroups thereof can be tested for inclusion and equality. Equality and subset testing is liberal, in that if the ambient varieties containing the two groups are not equal, then MAGMA attempts to find a natural embedding of both subgroups into a common ambient variety, and checks equality or inclusion there.

`IsFinite(G)`

Return `true` if the subgroup G of a modular abelian variety is known to be finite, i.e., generated by torsion elements. If G is not known exactly, i.e., has elements defined by floating point approximations to homology, then return `false`.

`G1 subset G2`

Return `true` if $G1$ is a subset of $G2$, where $G1$ and $G2$ are both subgroups of modular abelian varieties.

`G subset A`

Return `true` if the subgroup G is a subset of the abelian variety A. If A is not the ambient variety of G, then G and A are first mapped to a common ambient variety and compared.

`A subset G`

Return `true` if the abelian variety A is a subset of the finitely generated subgroup G of a modular abelian variety. This is true only if A is a point, i.e., the 0 dimensional abelian variety.

`G1 eq G2`

Return `true` if the subgroups G_1 and G_2 of modular abelian varieties are equal.

Example H136E108

We work with $J_0(389)$, but work in the $+1$ quotient of homology for efficiency. First we let A and B be the first and fifth factors in the decomposition of J, and let G and H be the corresponding 5-torsion subgroups.

```
> J := JZero(389,2,+1);
> D := Decomposition(J);
> A := D[1];
> B := D[5];
> G := nTorsionSubgroup(A,5);
> H := nTorsionSubgroup(B,5);
```

Note that the torsion subgroups aren't as big because we are working in the $+1$ quotient.

```
> #G;
5
> #H;
95367431640625
```

We now demonstrate each of the above commands for A, B, G, and H.

```
> IsFinite(G);
true
> A subset G;
false
> ZeroModularAbelianVariety() subset G;
true
> G subset A;
true
> G subset B;
true
> H subset A;
```

```
false
```

Since the ambient varieties of G and H are A and B, respectively, the following commands implicitly embed G and H into $J_0(389)$ and make comparisons there.

```
> G subset H;
true
> G eq H;
false
> G eq G;
true

> J := JZero(37);
> A, B := Explode(Decomposition(J));
> A2 := Kernel(nIsogeny(A,2));
> B2 := Kernel(nIsogeny(B,2));
> A2 eq B2;
true
> x := A2.1;
> x in B2;   // uses embedding of both into $J_0(37)$.
true
```

136.13 Rational Torsion Subgroups

The following functions are used for computing information about certain torsion points on modular abelian varieties.

136.13.1 Cuspidal Subgroup

For simplicity, assume A is a modular abelian variety and $\pi : J_0(N) \to A$ is the modular parameterization (the case when $J_0(N)$ is replaced by a more general modular abelian variety is similar). The *cuspidal subgroup* of $J_0(N)$ is the finite torsion group generated by all classes of differences of cusps on $X_0(N)$. The *cuspidal subgroup* of $A(\overline{\mathbf{Q}})$ is the image under π of the cuspidal subgroup of $J_0(N)$. The *rational cuspidal subgroup* is the subgroup generated by differences of cusps that are defined over $\mathbf{Q}$. (Computation of the rational points in the cuspidal subgroup has not yet been implemented.) One important use of the rational cuspidal subgroup is that it gives a lower bound on the cardinality (and structure) of the torsion subgroup of $A(\mathbf{Q})$, which is important in computations involving the Birch and Swinnerton-Dyer conjecture.

`CuspidalSubgroup(A)`

The subgroup of the abelian variety A generated by all differences of cusps, where we view A as a quotient of a modular symbols abelian variety. Note that this subgroup need not be defined over $\mathbf{Q}$.

`RationalCuspidalSubgroup(A)`

Return the finite subgroup of the abelian variety A generated by all differences of $\mathbf{Q}$-rational cusps, where we view A in some way as a quotient of a modular symbols abelian variety.

Example H136E109

We compute the cuspidal and rational cuspidal subgroups of $J_0(100)$.

```
> J := JZero(100);
> G := CuspidalSubgroup(J); G;
Finitely generated subgroup of abelian variety with invariants
[ 6, 30, 30, 30, 30 ]
> [Eltseq(x) : x in Generators(G)];
[
    [ 29/30, -2/5, 16/15, 121/30, -2, -61/30, 3/5, 31/15, 1,
    -89/30, -7/2, -3/2, -3, -1 ],
    [ 1, -5/6, 0, -1, -1, 2/3, -3/2, 0, -2, 0, 2, 5/3, 5/6, 0 ],
    [ -2, 17/15, 1/10, -29/15, 89/30, 26/15, 7/10, -2, 2/5, 2,
    -3/10, -7/30, 59/30, -1/2 ],
    [ 29/30, -1, 1/2, 67/15, -2, -3/2, 14/15, 91/30, 1, -3,
    -38/15, -29/30, -91/30, 1/2 ],
    [ 31/30, -31/30, 2/5, 67/15, -29/15, -43/30, 5/6, 3, 1, -3,
    -13/5, -31/30, -91/30, 0 ]
]
> H := RationalCuspidalSubgroup(J); H;
Finitely generated subgroup of abelian variety with invariants
[ 3, 15, 30 ]
```

Next we compute the cuspidal and rational subgroups for the optimal new elliptic curve of conductor 100.

```
> D := Decomposition(J); A := D[1];
> CuspidalSubgroup(A);
Finitely generated subgroup of abelian variety with invariants
[ 2, 2 ]
> Generators(CuspidalSubgroup(A));
[
    Element of abelian variety defined by [0 1/2] modulo homology,
    Element of abelian variety defined by [1/2 0] modulo homology
]
> RationalCuspidalSubgroup(A);
Finitely generated subgroup of abelian variety with invariants []
> TorsionMultiple(A);
2
```

Because the torsion multiple is 2, some of the cuspidal subgroup can not be defined over $\mathbf{Q}$.

136.13.2 Upper and Lower Bounds

Let A be an abelian variety over a number field K. MAGMA can compute upper and lower bounds on the cardinality of the torsion subgroup of $A(K)$ in the form of a multiple and a divisor of this cardinality.

`TorsionLowerBound(A)`

Given an abelian variety A return a divisor of the cardinality of the K-rational torsion subgroup of $A(K)$. Currently, to compute a bound we require that A be the base extension of an abelian variety B over $\mathbf{Q}$, and the lower bound is simply the cardinality of the rational cuspidal subgroup of $B(\mathbf{Q})$.

`TorsionMultiple(A)`

`TorsionMultiple(A, n)`

Given an abelian variety A return a multiple of the cardinality of the K-rational torsion subgroup of A over K. If n is not given it is assumed to be 50.

This multiple is usually fairly sharp, and is computed as follows. For each good prime $p \leq n$ with $[K : Q] + 1 < p$ and such that p does not divide the level of A, MAGMA computes $\#A(k)$, where k varies over residue class fields of K of characteristic p. Since reduction on torsion is injective for such primes, the greatest common divisor of the $\#A(k)$ is a multiple of the order of the torsion subgroup of $A(K)$. MAGMA computes $\#A(k)$ by using Hecke operators to find the characteristic polynomial of Frobenius on a Tate module of A, and uses this characteristic polynomial to deduce $\#A(k)$. For details, see [AS05].

Example H136E110

```
> J := JZero(100);
> TorsionLowerBound(J);
1350
> #RationalCuspidalSubgroup(J);
1350
> TorsionMultiple(J);
16200
> 16200/1350;
12
> J2 := BaseExtend(J,QuadraticField(2));
> TorsionMultiple(J2);
129600
> 129600/16200;
8
```

136.13.3 Torsion Subgroup

`TorsionSubgroup(A)`

Let A be an abelian variety over a field K. Attempt to compute the subgroup of torsion elements in $A(K)$. Return either `false` and a subgroup of the torsion subgroup, or `true` and the exact torsion subgroup of A over the base field.

Example H136E111

```
> TorsionSubgroup(JZero(11));
true Finitely generated subgroup of abelian variety with invariants [ 5 ]
> TorsionSubgroup(JZero(33));
false Finitely generated subgroup of abelian variety with
invariants [ 10, 10 ]
> TorsionSubgroup(BaseExtend(JZero(11),QuadraticField(5)));
true Finitely generated subgroup of abelian variety with
invariants [ 5 ]
> TorsionSubgroup(ChangeRing(JZero(11),GF(5)));
false { 0 }: finitely generated subgroup of abelian variety with
invariants []
> TorsionSubgroup(JZero(100));
false Finitely generated subgroup of abelian variety with
invariants [ 3, 15, 30 ]
> TorsionSubgroup(JZero(125));
true Finitely generated subgroup of abelian variety with
invariants [ 25 ]
```

136.14 Hecke and Atkin-Lehner Operators

136.14.1 Creation

These commands compute endomorphisms induced by the Atkin-Lehner and Hecke operators on modular abelian varieties. The Atkin-Lehner involution W_q is defined for each positive integer q that exactly divides the level (and is divisible by the conductor of any relevant character).

`AtkinLehnerOperator(A, q)`

The Atkin-Lehner operator W_q of index q induced on the abelian variety A by virtue of A being modular. In general W_q need not be a morphism except in the category of abelian varieties up to isogeny so this intrinsic also returns an integer d such that $d * W_q$ is an endomorphism of A, and when W_q doesn't leave A invariant, returns $d = 0$. If the ambient modular symbols space of A contains a space with character of conductor r, then currently an error occurs unless r divides q.

`AtkinLehnerOperator(A)`

The morphism (or morphism tensor Q) on (or from) the abelian variety A induced by the Atkin-Lehner operator.

`HeckeOperator(A, n)`

The Hecke operator T_n of index n induced on the abelian variety A by virtue of its morphism to a modular symbols abelian variety. In general T_n need not be a morphism. Also, if A is contained in e.g., $J_0(N)$, then the T_n on $J_0(N)$ need not even leave A invariant. In that case this command composes T_n with a map back to A to obtain an endomorphism of A. For the exact Hecke operators induced by their action on $J_0(N)$, say, use the `RestrictEndomorphism` command.

Example H136E112

We compute the main Atkin-Lehner operator and the Hecke operator T_2 on $J_0(23)$.

```
> A := JZero(23);
> AtkinLehnerOperator(A,23);
Homomorphism W23 from JZero(23) to JZero(23) given on integral homology by:
[-1  0  0  0]
[ 0 -1  0  0]
[ 0  0 -1  0]
[ 0  0  0 -1]
> HeckeOperator(A,2);
Homomorphism T2 from JZero(23) to JZero(23) given on integral homology by:
[ 0  1 -1  0]
[ 0  1 -1  1]
[-1  2 -2  1]
[-1  1  0 -1]
```

Next we compute w_4 and w_{25} on J_{100}, and note that their product equals w_{100}.

```
> A := JZero(100); A;
Modular abelian variety JZero(100) of dimension 7 and
level 2^2*5^2 over Q
> w4 := AtkinLehnerOperator(A,4);
> Factorization(CharacteristicPolynomial(w4));
[
    <x - 1, 4>,
    <x + 1, 10>
]
> w25 := AtkinLehnerOperator(A,25);
> Factorization(CharacteristicPolynomial(w25));
[
    <x - 1, 8>,
    <x + 1, 6>
]
> w4*w25 eq AtkinLehnerOperator(A);
```

```
true
```

Next we compute W_{25} acting on $J_1(25)$.

```
> A := Js(17);
> B := BaseExtend(A,CyclotomicField(17));
> w := AtkinLehnerOperator(B);
> Factorization(CharacteristicPolynomial(w));
[
    <x - 1, 4>,
    <x + 1, 6>
]
```

Finally we compute Hecke operators on the quotient of a simple factor of $J_0(65)$ by a finite subgroup.

```
> A := Decomposition(JZero(65))[2]; A;
Modular abelian variety 65B of dimension 2, level 5*13 and conductor
5^2*13^2 over Q
> G := nTorsionSubgroup(A,2); G;
Finitely generated subgroup of abelian variety with invariants
[ 2, 2, 2, 2 ]
> H := Subgroup([G.1]); H;
Finitely generated subgroup of abelian variety with invariants [ 2 ]
> B := A/H; B;
Modular abelian variety of dimension 2 and level 5*13 over Qbar
> T2 := HeckeOperator(B,2); T2;
Homomorphism from modular abelian variety of dimension 2 to
modular abelian variety of dimension 2 (up to isogeny) on
integral homology by:
[ -2 1/2   0   0]
[ -2   2   0   0]
[ -2   1  -2   1]
[ -6   1  -1   2]
> FactoredCharacteristicPolynomial(T2);
[
    <x^2 - 3, 2>
]
```

136.14.2 Invariants

Intrinsics are provided which compute characteristic polynomials, factored characteristic polynomials and minimal polynomials of Hecke operators.

`HeckePolynomial(A, n)`

The characteristic polynomial of the Hecke operator T_n acting on the abelian variety A.

`FactoredHeckePolynomial(A, n)`

The factored characteristic polynomial of the Hecke operator T_n acting on the abelian variety A. This can be faster than first computing T_n, then computing the characteristic polynomial, and factoring, because we can take into account information about the decomposition of A, in order to avoid factoring.

`MinimalHeckePolynomial(A, n)`

The minimal polynomial of the Hecke operator T_n acting on the abelian variety A.

Example H136E113

```
> FactoredHeckePolynomial(JZero(65),2);
[
    <x + 1, 2>,
    <x^2 - 3, 2>,
    <x^2 + 2*x - 1, 2>
]
> HeckePolynomial(JZero(65),2);
x^10 + 6*x^9 + 5*x^8 - 32*x^7 - 62*x^6 + 28*x^5 + 130*x^4 +
    48*x^3 - 51*x^2 - 18*x + 9
> MinimalHeckePolynomial(JZero(65),2);
x^5 + 3*x^4 - 2*x^3 - 10*x^2 - 3*x + 3
```

136.15 L-series

136.15.1 Creation

The `LSeries` command creates the L-series $L(A, s)$ associated to a modular abelian variety A over $\mathbf{Q}$ or a cyclotomic field. No actual computation is performed.

`LSeries(A)`

The L-series associated to the abelian variety A.

Example H136E114

```
> A := JZero(23);
> L := LSeries(A);
> L;
L(JZero(23),s): L-series of Modular abelian variety JZero(23) of
dimension 2 and level 23 over Q
> LSeries(ModularAbelianVariety("65B"));
L(65B,s): L-series of Modular abelian variety 65B of dimension 2
and level 5*13 over Q
```

You can create L-series of abelian varieties over cyclotomic fields, but currently no interesting functionality is implemented for them.

```
> LSeries(BaseExtend(JZero(11),CyclotomicField(5)));
L(JZero(11),s): L-series of Modular abelian variety JZero(11) of
dimension 1 and level 11 over Q(zeta_5)
```

136.15.2 Invariants

`CriticalStrip(L)`

Let L be the L-function of some modular abelian variety A. This function returns integers x and y such that the critical strip for L is the set of complex numbers with real part strictly between x and y. If W is the set of weights of newforms that give rise to factors of A, then this command returns 0 and `Max(W)`.

`ModularAbelianVariety(L)`

The abelian variety the L-series L is associated to.

Example H136E115

We define several L-functions of modular abelian varieties and modular motives, and compute their critical strip (which is from 0 to k, where k is the weight).

```
> L := LSeries(JZero(37));
> CriticalStrip(L);
0 2
> L := LSeries(JZero(37,6));
> CriticalStrip(L);
0 6
> J := JOne(11,3);  J;
Modular motive JOne(11,3) of dimension 5 and level 11 over Q
> CriticalStrip(LSeries(J));
0 3
> A_delta := JZero(1,12);
> L := LSeries(A_delta);
> CriticalStrip(L);
```

```
0 12
> ModularAbelianVariety(L);
Modular motive JZero(1,12) of dimension 1 and level 1 over Q
```

136.15.3 Characteristic Polynomials of Frobenius Elements

Let A be a modular abelian variety. The characteristic polynomials of Frobenius elements acting on the ℓ-adic Tate modules of A define the local L-factors of $L(A, s)$.

`FrobeniusPolynomial(A : parameters)`

`Factored` BoolElt *Default :* `false`

The characteristic polynomial of Frobenius on the abelian variety A defined over a finite field. If `Factored` is set to `true`, return a factorization instead of a polynomial.

`FrobeniusPolynomial(A, p : parameters)`

`Factored` BoolElt *Default :* `false`

The characteristic polynomial of Frob_p acting on any ℓ-adic Tate module of the abelian variety A over a number field, where p and ℓ do not divide the level of A. If the base ring has degree bigger than 1, then return a sequence of characteristic polynomials, one for each prime lying over p, sorted by degree. If `Factored` is set to `true`, return a factorization instead of a polynomial.

`FrobeniusPolynomial(A, P)`

The characteristic polynomial of Frobenius at the nonzero prime ideal P of a number field on the modular abelian variety A, where P is assumed to be a prime of good reduction for A, and A is defined over a field that contains the prime P.

Example H136E116

```
> A := JZero(23);
> FrobeniusPolynomial(A,2);
x^4 + x^3 + 3*x^2 + 2*x + 4
> A := JZero(23) * JZero(11,4) * JOne(13);
> FrobeniusPolynomial(A,2);
x^12 + 2*x^11 + 17*x^10 + 40*x^9 + 145*x^8 + 362*x^7 + 798*x^6 +
    1408*x^5 + 2104*x^4 + 2528*x^3 + 2528*x^2 + 1792*x + 1024
> Factorization($1);
[
    <x^4 - 2*x^3 + 14*x^2 - 16*x + 64, 1>,
    <x^4 + x^3 + 3*x^2 + 2*x + 4, 1>,
    <x^4 + 3*x^3 + 5*x^2 + 6*x + 4, 1>
]
> A := BaseExtend(JZero(23),CyclotomicField(22));
> FrobeniusPolynomial(A,2);
```

```
[
    x^4 + 25*x^3 - 327*x^2 + 25600*x + 1048576
]
```

These characteristic polynomials are used in the algorithm to compute the number of points on modular abelian varieties over finite fields.

```
> A := ChangeRing(JZero(23),GF(2^10));
> NumberOfRationalPoints(A);
1073875 1073875
> Factorization($1);
[ <5, 3>, <11, 2>, <71, 1> ]
```

136.15.4 Values at Integers in the Critical Strip

MAGMA allows evaluation of L-series at integers lying within the critical strip.

There exist algorithms for computing $L(A, s)$ for any complex number s, but these are not currently implemented in MAGMA.

`L(s)`

`Evaluate(L, s)`

`Evaluate(L, s, prec)`

The value of L-series L at s, where s must be an integer that lies in the critical strip for L, computed using *prec* terms of the power series or 100 if *prec* is not given. The power series used are the q-expansions of modular forms corresponding to differentials on A. It is not clear, a priori, what the relation is between *prec* and the precision of the real number output by this command. (It is theoretically possible to give bounds, but we have not done this.) In practice, one can increase *prec* and see how the output result changes.

`LRatio(A, s)`

`LRatio(L, s)`

Given an abelian variety A over $\mathbf{Q}$ attached to a newform or an L-series L of an abelian variety and an integer s, return the ratio $L(A, s) * (s-1)!/((2\pi)^{s-1} * \Omega_s)$, where s is a "critical integer", and Ω_s is the integral (Neron) volume of the group of real points on the optimal quotient A' associated to A when s is odd, and the volume of the -1 eigenspace for conjugation when s is even.

`IsZeroAt(L, s)`

Given an L-series L of a modular abelian variety and an integer s in the critical strip for L return `true` is $L(A, s)$ is zero. In contrast to the output of the `Evaluate` command above, the result returned by this command is provably correct.

Example H136E117

First we demonstrate each evaluation command for the L-series of $J_0(23)$.

```
> L := LSeries(JZero(23));
> L(1);
0.24843186659059928468330576929 0 +  0.E-29*i
> Evaluate(L,1);
0.24843186659059928468330576929 0 +  0.E-29*i
> Evaluate(L,1,200);
0.24843186659059968120725033907 4 +  0.E-29*i
> LRatio(L,1);
1/11
> L := LSeries(JZero(23));
> L(1);
0.24843186659059928468330577047 6 +  0.E-29*i
> Evaluate(L,1,200);
0.24843186659059968120725034014 4 +  0.E-29*i
```

Next we compute the L-series of the motive attached to the weight 12 level 1 modular form Δ.

```
> A := JZero(1,12);
> L := LSeries(A);
> Evaluate(L,1);
0.037441281268515541738770315844 3
> L(5);
0.666709188434003643826130221 64
> Evaluate(L,1,200);
0.037441281268515541738770315844 3
> LRatio(L,1);
11340/691
> LRatio(L,2);
24
> LRatio(L,3);
7
```

We compute some ratios for $J_1(N)$ and factors of $J_1(N)$.

```
> LRatio(JOne(13),1);
1/361
> J := JOne(23);
> Evaluate(LSeries(J),1);
0.000000080777697074785775420090700066 +
0.000000053679621277482217773207669332*i
```

It looks kind of like $L(J_1(23), 1)$ is zero. However, this is not the case! We can not compute `LRatio` for $J_1(23)$, since it not attached to a newform. We can, however, compute `LRatio` for each simple factor.

```
> LRatio(J(1),1);
1/11
```

```
> LRatio(J(2),1);
1/1382426761
```

Each simple factor has nonzero `LRatio`, so $L(J,1) \neq 0$.

136.15.5 Leading Coefficient

The L-function $L(A,s)$ has a Taylor expansion about any critical integer. The leading coefficient and order of vanishing of $L(A,s)$ about a critical integer can be computed.

`LeadingCoefficient(L, s, prec)`

Given an L-series L associated to a modular abelian variety A and an integer s in the critical strip for L return the leading coefficient of the Taylor expansion about s and the order of vanishing of L at s. At present, A must have weight 2 and trivial character (so $s = 1$). It does not have to be attached to a newform. The argument *prec* is the number of terms of the power series which are used.

Example H136E118

```
> LeadingCoefficient(LSeries(JZero(37)),1,100);
0.24426406492583898134986778296 5
> LeadingCoefficient(LSeries(JZero(37)(1)) ,1,100);
0.30599977380008529004409407572 5
> J := JZero(3^5);
> LeadingCoefficient(LSeries(J),1,100);
15.140660788463628991688955015326 +  0.E-27*i 4
```

The order of vanishing of 4 for $J_0(3^5)$ comes from an elliptic curve and a 3-dimensional abelian variety that have order of vanishing 1 and 3, respectively.

```
> LeadingCoefficient(LSeries(J(1)),1,100);
1.4192096493382156160031880842 81
> LeadingCoefficient(LSeries(J(5)),1,100);
1.2280519528591420520347698584 45 3
```

Example H136E119

We give a few more examples.

```
> L := LSeries(ModularAbelianVariety("389A",+1));
> LeadingCoefficient(L,1,100);
0.75931650029224679065762600319 2
>
> A := JZero(65)(2); A;
Modular abelian variety 65B of dimension 2, level 5*13 and
conductor 5^2*13^2 over Q
> L := LSeries(A);
> LeadingCoefficient(L,1,100);
```

```
0.91225158869818984109351402175 +  0.E-29*i 0
> A := JZero(65)(3); A;
Modular abelian variety 65C of dimension 2, level 5*13 and
conductor 5^2*13^2 over Q
> L := LSeries(A);
> LeadingCoefficient(L,1,100);
0.45206792176803106991748613500 0 +  0.E-29*i 0
```

136.16 Complex Period Lattice

136.16.1 Period Map

Let A be a modular abelian variety. The period mapping of A is a map from the rational homology of A to a complex vector space.

`PeriodMapping(A, prec)`

The complex period mapping from the rational homology of the abelian variety A to $\mathbf{C}^d$, where $d = \dim A$, computed using *prec* terms of q-expansions.

136.16.2 Period Lattice

`Periods(A, n)`

Given an abelian variety A and an integer n return generators for the complex period lattice of A, computed using n terms of q-expansions. We use the map from A to a modular symbols abelian variety to define the period mapping (so this map must be injective).

136.17 Tamagawa Numbers and Component Groups of Neron Models

136.17.1 Component Groups

Suppose A is a newform modular abelian variety over $\mathbf{Q}$ over level N. For any prime p that exactly divides N, the order of the component group of A over the algebraic closure of $\mathbf{F}_p$ can be computed. The nontrivial algorithm is described in [CS01] and [KS00]. It is an open problem to compute the structure of the component group or the order under more general hypothesis.

`ComponentGroupOrder(A, p)`

The order of the component group of the special fiber of the Neron model of A over the algebraic closure of $\mathbf{F}_p$. The abelian variety A must be attached to a newform.

Example H136E120

```
> J := JZero(65); J;
Modular abelian variety JZero(65) of dimension 5 and level 5*13 over Q
> A := Decomposition(J)[3];
> ComponentGroupOrder(A,13);
1
> ComponentGroupOrder(A,5);
7
```

136.17.2 Tamagawa Numbers

Suppose A is an abelian variety over $\mathbf{Q}$ that is attached to a newform. A divisor and an integer some power of which is a multiple of the Tamagawa number of A at a prime p can be determined. When p^2 divides the level, the reduction is additive, we use the Lenstra-Oort bound from [LO85].

`TamagawaNumber(A, p)`

A divisor of the Tamagawa number of the abelian variety A at the prime p and an integer some power of which is a multiple of the Tamagawa number of A at p. Also return `true` if the divisor of the Tamagawa number is provably equal to the Tamagawa number of A. The abelian variety A must be attached to a newform.

`TamagawaNumber(A)`

Let c be the product of the Tamagawa numbers of A at primes of bad reduction, where A is an abelian variety over $\mathbf{Q}$ attached to a newform. This command returns a divisor of c, an integer some power of which is a multiple of c, and `true` if the divisor is provably equal to c.

Example H136E121

```
> J := JZero(65);
> TamagawaNumber(J(2),5);
2 2 false
> TamagawaNumber(J(2),13);
3 3 true
> TamagawaNumber(J(3),5);
7 7 true
> TamagawaNumber(J(3),13);
2 2 false
>
> J := JZero(5^2*7);
> TamagawaNumber(J(1));
2 30 false
> TamagawaNumber(J(1),5);
```

```
1 30 false
> TamagawaNumber(J(1),7);
2 2 false
```

136.18 Elliptic Curves

136.18.1 Creation

Modular abelian varieties of dimension 1 are elliptic curves. Given a modular abelian variety A over $\mathbf{Q}$ of dimension 1, MAGMA can compute an elliptic curve that is isogenous over $\mathbf{Q}$ to A. Given an elliptic curve E over $\mathbf{Q}$, a modular abelian variety over $\mathbf{Q}$ that is isogenous to E can be constructed.

It would be very desirable to make these commands more precise, and to extend them to work over other fields. For example, modular abelian varieties should (conjecturally) be associated to $\mathbf{Q}$-curves and their restriction of scalars.

`EllipticCurve(A)`

An elliptic curve isogenous to the modular abelian variety A over the rational field, if there is an elliptic curve associated to A. For A of weight greater than 2 use the `EllipticInvariants` command.

`ModularAbelianVariety(E)`

`Sign` RNGINTELT *Default :* 0

A modular abelian variety isogenous to the elliptic curve E. Note that elliptic curves with small coefficients can have quite large conductor, hence computing the massive modular abelian variety that has E as quotient, which is one thing this function does, could take some time.

Example H136E122

We apply the above two commands to the elliptic curve $J_0(49)$.

```
> A := JZero(49);
> E := EllipticCurve(A); E;
Elliptic Curve defined by y^2 + x*y = x^3 - x^2 - 2*x - 1 over
Rational Field
> B := ModularAbelianVariety(E); B;
Modular abelian variety 'Cremona 49A' of dimension 1 and level
7^2 over Q
```

To see how A and B compare, we first test equality and see they are not equal (since they were constructed differently). However, they are isomorphic.

```
> B eq A;
false
> IsIsomorphic(A,B);
```

```
true Homomorphism from JZero(49) to 'Cremona 49A' given on integral
homology by:
[1 0]
[0 1]
> phi := NaturalMap(A,B);
> Degree(phi);
1
> phi;
Homomorphism N(1) from JZero(49) to 'Cremona 49A' given on integral
homology by:
[1 0]
[0 1]
```

Thus B is embedded in A via the identity map.

136.18.2 Invariants

Let A be an abelian variety over $\mathbf{Q}$ of dimension 1. The following two functions use standard iterative algorithms (see Cremona's book) to compute the invariants c_4, c_6, j, and generators of the period lattice of the optimal quotient of $J_0(N)$ associated to A.

`EllipticInvariants(A, n)`

Invariants c_4, c_6, j, and an elliptic curve, of the one dimensional modular abelian variety A, computed using n terms of q-expansion.

`EllipticPeriods(A, n)`

Elliptic periods w_1 and w_2 of the $J_0(N)$-optimal elliptic curve associated to the modular abelian variety A, computed using n terms of the q-expansion. The periods have the property that w_1/w_2 has positive imaginary part.

Example H136E123

```
> A := ModularAbelianVariety("100A");
> c4,c6,j,E := EllipticInvariants(A,100);
> c4;
1600.052318304045803306867849111720 8 +  0.E-25*i
> c6;
-44002.166592330033618811790218678607 +  0.E-24*i
> j;
3276.801127299202275905948170653 93 +  0.E-25*i
> E;
Elliptic Curve defined by y^2 = x^3 +
(-43201.412594209236689285431925551172 +  0.E-24*i)*x +
(2376116.995985821815415836671800373 00 +  0.E-22*i) over Complex
Field
> jInvariant(E);
```

```
3276.80112729920227590594817070563 +  0.E-25*i
> w1,w2 := EllipticPeriods(A,100);
> w1;
1.26308870071276069371281657330245009108 8 +  0.E-38*i
> w2;
 0.E-38 - 1.01702927066995984919787906165005620863321*i
> w1/w2;
 0.E-38 + 1.24193937887422962244668740609486508404 97*i
```

136.19 Bibliography

[**AS04**] A. Agashe and W. A. Stein. The Manin Constant, Congruence Primes, and the Modular Degree. Preprint, URL:http://modular.fas.harvard.edu/papers/manin-agashe/, 2004.

[**AS05**] Amod Agashe and William Stein. Visible evidence for the Birch and Swinnerton-Dyer conjecture for modular abelian varieties of analytic rank zero. *Math. Comp.*, 74(249):455–484 (electronic), 2005. With an appendix by J. Cremona and B. Mazur.

[**Bos00**] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.

[**CS01**] Brian Conrad and William A. Stein. Component groups of purely toric quotients. *Math. Res. Lett.*, 8(5-6):745–766, 2001.

[**Kil02**] L. J. P. Kilford. Some non-Gorenstein Hecke algebras attached to spaces of modular forms. *J. Number Theory*, 97(1):157–164, 2002.

[**KS00**] David R. Kohel and William A. Stein. Component Groups of Quotients of $J_0(N)$. In Bosma [Bos00].

[**LO85**] H. W. Lenstra, Jr. and F. Oort. Abelian varieties having purely additive reduction. *J. Pure Appl. Algebra*, 36(3):281–298, 1985.

[**Que09**] Jordi Quer. Fields of definition of building blocks. *Math. Comp.*, 78(265):537–554, 2009.

137 HILBERT MODULAR FORMS

Chapter 137
HILBERT MODULAR FORMS

137.1 Introduction

The first version of this package was released in V2.15 (December 2008). It has been developed further in each subsequent release, and is still under development. We encourage users to send feedback regarding the package, and desirable features or improvements.

The package contains implementations of two separate algorithms. The primary focus is on parallel weight 2. Higher weight spaces (including non-parallel weight) are also handled; however some features are only available for weight 2, and the main routines are better optimized in weight 2. All levels $\Gamma_0(N)$ are allowed; some spaces with nontrivial character are also handled.

The main purposes of the current functionality are to efficiently compute Hecke operators on these spaces, to decompose spaces into newforms, and to efficiently obtain large numbers of eigenvalues for newforms (at least those having small degree). Some additional features such as Atkin-Lehner operators are also included.

137.1.1 Definitions and Background

Hilbert modular forms are a generalization of classical modular forms where the modular group is replaced by a subgroup of $\mathrm{GL}_2(\mathbf{Z}_F)$ where $\mathbf{Z}_F$ is the ring of integers in a totally real number field. This section gives a brief and partly informal introduction to Hilbert modular forms. The sections about the algorithms also introduce quaternionic modular forms and the Jacquet-Langlands theory in order to use the package!

The books by Freitag [Fre90] and Garrett [Gar90] are good references for standard material on Hilbert modular forms.

Let F be a totally real field of degree n over $\mathbf{Q}$, with ring of integers $\mathbf{Z}_F$. Let $v_1, \ldots, v_n$ be the embeddings of F into $\mathbf{R}$. For $x \in F$, we write x_i as a shorthand for $v_i(x)$. Each embedding v_i induces an embedding $v_i : \mathrm{Mat}_2(\mathrm{F}) \hookrightarrow \mathrm{Mat}_2(\mathbf{R})$. Extending our shorthand to matrices, for $\gamma \in \mathrm{Mat}_2(\mathrm{F})$ we write γ for $v_i(\gamma)$. Let

$$\mathrm{GL}_2^+(F) = \{\gamma \in \mathrm{GL}_2(F) : \mathrm{det}\gamma_\mathrm{i} > 0 \text{ for all i}\}.$$

The group $\mathrm{GL}_2^+(F)$ acts on the cartesian product H^n by the rule

$$\gamma(\tau, \ldots, \tau_n) = (\gamma\tau_1, \ldots, \gamma\tau_n)$$

where as usual $\mathrm{GL}_2^+(\mathbf{R})$ acts on H by linear fractional transformations.

Let $\gamma = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{GL}_2(\mathbf{R})$ and $\tau \in \mathrm{H}$. We define

$$j(\gamma, \tau) = c\tau + d \in \mathbf{C}.$$

For a *weight* $k = (k_1, \ldots, k_n) \in (\mathbf{Z}_{>0})^n$, we define a right *weight-k action* of $\mathrm{GL}_2^+(F)$ on the space of complex-valued functions on H^n by

$$(f|_k\gamma)(\tau) = f(\gamma\tau)\prod_{i=1}^{n}(\det\gamma)^{\mathrm{k_i}/2}\mathrm{j}(\gamma,\tau_\mathrm{i})^{-\mathrm{k_i}}$$

for $f : \mathrm{H}^n \to \mathbf{C}$ and $\gamma \in \mathrm{GL}_2^+(F)$. The center $F^\times$ of $\mathrm{GL}_2^+(F)$ acts trivially on such f. Therefore, the weight-k action descends to an action of $\mathrm{PGL}_2^+(F) = \mathrm{GL}_2^+(F)/F^\times$.

Now let N be a (nonzero) ideal of $\mathbf{Z}_F$. Define

$$\Gamma_0(N) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{GL}_2^+(\mathbf{Z}_F) : c \in N \right\}.$$

A *Hilbert modular cusp form of weight-k and level* $\Gamma_0(N)$ is an analytic function $f : \mathrm{H}^n \to \mathbf{C}$ such that $f|_k\gamma = f$ for all $\gamma \in \Gamma_0(N)$ and such that f vanishes at the cusps of $\Gamma_0(N)$.

When F has narrow class number 1, it is possible to define double coset operators in the same way as for classical modular forms; among these are Hecke operators, Atkin-Lehner operators and degeneracy maps. More precisely, the definitions work for primes in the trivial ideal class. In the general case, the adelic definition of Hilbert modular forms is more convenient for the purpose of defining operators, among other considerations. We refer to the book by Freitag [Fre90].

137.1.2 Algorithms and the Jacquet-Langlands Correspondence

Two separate algorithms are implemented for computing the Hecke action on cuspidal spaces of Hilbert modular forms. Both rely on the *Jacquet-Langlands correspondence*, which states that the Hecke action on a space of Hilbert cusp forms is the same as the Hecke action on a space of automorphic forms on some order in a suitable quaternion algebra. See [Hid06] for definitions of quaternionic automorphic forms and the correspondence.

Let F be a totally real field of degree n over $\mathbf{Q}$. Algorithm I uses a definite quaternion algebra over F (ramified at all n infinite places). Algorithm II uses the Shimura curve associated to a quaternion algebra ramified at exactly $n-1$ infinite places. By default, the algorithm (and the quaternion order to be used) are selected automatically, but can also be specified by the user. The essential requirement is that the quaternion order can be ramified only at primes p for which the space is p-new. The required conditions are stated fully in the descriptions of the commands `HilbertCuspForms` and `NewSubspace`. It is worth noting that a different algorithm and order may be selected for a newspace than for the space containing it. In fact, there is more freedom for a new space; in particular, over $\mathbf{Q}$ neither algorithm can compute full spaces, but can compute new subspaces for nonsquare levels.

Algorithm II is implemented only for parallel weight 2 (weight $[2, 2, \ldots, 2]$). In the current implementation, Algorithm I is more optimized than Algorithm II, and therefore should be preferred in most cases. For spaces over odd degree fields with level not divisible by any small primes, Algorithm II may be preferable.

An exposition of both algorithms is given in [DV12], in addition to the references mentioned in the following two sections.

137.1.3 Algorithm I (Using Definite Quaternion Orders)

This algorithm, described in [Dem07] and [DD08], is an efficient formulation of the Brandt module approach to computing modular forms.

A key advantage of this algorithm is that the most expensive steps come in the precomputation phase (computations depending only on the quaternion algebra). This means that for a given number field, forms of many different levels and weights can be computed based on the same precomputation.

137.1.4 Algorithm II (Using Indefinite Quaternion Orders)

This algorithm, described in [GV11], is based on computing the homology of a Shimura curve (and therefore is closer to the usual modular symbols algorithm over **Q** than Algorithm I). Voight's algorithm ([Voi09]) for the fundamental domain of the action of the Fuchsian group on the upper half plane is applied. The algorithm simultaneously determines a fundamental domain and generators and relations for the fundamental group. From this a description of the homology is obtained, in such a way that the Hecke action on it can then be explicitly calculated.

137.1.5 Categories

The MAGMA category for spaces of Hilbert modular forms is `ModFrmHil`, while elements in these spaces have type `ModFrmHilElt`.

137.1.6 Verbose Output

To see some information printed during computation about what the program is doing, use `SetVerbose("ModFrmHil",n)`, where `n` is 0 (silent, by default), 1 (prints concise information), 2 or 3 (which may display bulky data).

137.2 Creation of Full Cuspidal Spaces

In the current implementation only cusp forms are supported. In future releases it will be possible to compute Eisenstein series, and create the full space of Hilbert modular forms of given weight and level.

```
HilbertCuspForms(F, N, k)
```

```
HilbertCuspForms(F, N)
```

`QuaternionOrder` ALGASSVORD *Default :*

This creates the space of Hilbert modular forms over the field F (a number field or the rationals) on $\Gamma_0(N)$ with weight k. Here the level N should be an ideal in the maximal order of F, and k should be a sequence of $\deg(\mathrm{F}/\mathbf{Q})$ integers, all at least 2 and all of the same parity. If not specified, the weight is taken to be parallel weight 2, ie. [2, 2, ..., 2].

Computations in the space will be done by realising it as a space of automorphic forms on an order in a suitable quaternion algebra (as explained in the introduction).

This choice is made automatically (in most cases), when it is needed, and is hidden from the user; however the user may specify the order to be used by providing the optional argument `QuaternionOrder`. The quaternion algebra may be definite (ramified at all infinite places of F) or indefinite (ramified at all infinite places except one); this determines which of the two algorithms will be used (see the introduction). Indefinite algebras may only be used for spaces of parallel weight 2. The algebra must be unramified at all finite primes (for full cuspidal spaces, although often a different algebra may be used to compute a `NewSubspace`). In the definite case the order must be maximal, while in the indefinite case it must be an Eichler order with discriminant equal to the level N.

Example H137E1

Here we create some spaces of modular forms over $\mathbf{Q}(\sqrt{85})$.

```
> _<x> := PolynomialRing(Rationals());
> F := NumberField(x^2-85);
> level := 1*Integers(F);
> H := HilbertCuspForms(F, level);
> H;
Cuspidal space of Hilbert modular forms over Number Field with defining polynomial
x^2 - 85 over the Rational Field
   Level = Ideal of norm 1 generated by ( [1, 0] )
   Weight = [ 2, 2 ]
```

This returns instantly because no nontrivial computations have been done yet. We can find out which quaternion order will be used internally to do computations.

```
> QO := QuaternionOrder(H);
> A := Algebra(QO);
> A;
Quaternion Algebra with base ring F
> A.1^2, A.2^2, A.3^2;
-1 -1 -1
> IsMaximal(QO);
true
```

So the order is a maximal order in Hamilton's quaternion algebra over F. We now define a space with level equal to one of the split primes dividing 3.

```
> level := Factorization(3*Integers(F))[1][1];
> Norm(level);
3
> weight := [3,5];
> H := HilbertCuspForms(F, level, weight);
```

If we wish, we may tell Magma to use the same quaternion order. (This would avoid some of the time-consuming computations being done twice.)

```
> H := HilbertCuspForms(F, level, weight : QuaternionOrder:=QO);
```

137.3 Basic Properties

`BaseField(M)`

The field on which the space M was defined.

`Weight(M)`

The weight of the space M.

`CentralCharacter(M)`

The central character of the weight representation defining the space M of Hilbert modular forms. This is of significance only for higher weight spaces (not parallel weight 2).

`Level(M)`

The level of the space M.

`DirichletCharacter(M)`

The nebentypus of the space M of Hilbert modular forms.

`IsCuspidal(M)`

This is true if the space M was created as (a subspace of) a space of cusp forms. In the current implementation this is always `true`.

`IsNew(M)`

This is true if the space M was created as (a subspace of) a new space, for instance using `NewSubspace` or `NewformDecomposition`, or if the space is known to satisfy `NewLevel(M) = Level(M)`.

`NewLevel(M)`

This returns the level at which M is known to be new. (See `NewSubspace`.)

`Dimension(M)`

`UseFormula` BoolElt *Default :* `true`

This computes the dimension of the space M of Hilbert modular forms.

The dimension is determined either by using "dimension formulae" or else by explicitly constructing the space. By default, formulae are used when available and when considered cheap to evaluate. The optional argument `UseFormula` can be set `true` or `false` to override the default.

Dimension formulae are implemented for spaces of parallel weight 2. These formulae are sums over certain cyclotomic extensions of the base field of F.

The results by formula are guaranteed only under GRH, since they may involve class numbers that are calculated conditionally. If at some later point, the space is explicitly computed, the dimension is will then be verified unconditionally.

`QuaternionOrder(M)`

This returns the quaternion order that is used internally to compute the space M of Hilbert modular forms.

`IsDefinite(M)`

This indicates which of the two algorithms is used to compute the space M of Hilbert modular forms. It is equivalent to `IsDefinite(Algebra(QuaternionOrder(M)))`. Note that calling this causes a `QuaternionOrder` for M to be chosen (if it has not been set already).

Example H137E2

We continue with the first example above.

```
> _<x> := PolynomialRing(Rationals());
> F := NumberField(x^2-85);
> OF := Integers(F);
> H := HilbertCuspForms(F, 1*OF);
> Norm(Level(H));
1
> Weight(H);
[ 2, 2 ]
> time Dimension(H);
6
Time: 2.580
> IsDefinite(H);
true
```

This indicates that Algorithm I (described in the introduction) was used to compute the dimension.

```
> level := Factorization(3*OF)[1][1];
> H3 :=  HilbertCuspForms(F, level);
> Level(H3);
Prime Ideal of OF
Two element generators:
    [3, 0]
    [2, 2]
> time Dimension(H3);
14
Time: 2.370
> H3 :=  HilbertCuspForms(F, level : QuaternionOrder:=QuaternionOrder(H) );
> time Dimension(H3);
14
Time: 0.270
```

It is much faster when we recycle the same quaternion order used for the space of level 1. This is because the harder computations involved depend only on the quaternion order, not on the level.

137.4 Elements

The current implementation does not provide functionality for manipulating elements in spaces of Hilbert modular forms. Note that, unlike classical modular forms in MAGMA, Hilbert modular forms are allowed to be defined over extensions of the base field of their parent space.

`Parent(f)`

The space of Hilbert modular forms containing f.

`BaseField(f)`

The field over which the Hilbert modular form f is defined. This is either equal to, or an extension of, the base field of `Parent(f)`.

137.5 Operators

Operators on spaces of Hilbert modular forms are returned as matrices with respect to a basis of M. For spaces of parallel weight 2, operators are matrices over $\mathbf{Q}$, and the basis of M is permanently fixed. For all other weights, two finite extensions of $\mathbf{Q}$ may arise:

(i) The field that is used for the raw computations; operators are originally computed as matrices over this field.

(ii) The minimal field F for which there exists a basis of M such that operators are matrices with entries in F. For all parallel weights, this minimal field is $\mathbf{Q}$. We refer to such a basis as a "rational basis" of M.

Changing the basis can be expensive for large spaces, so for some spaces a "rational basis" is not computed by default. In these cases, Hecke operators are returned as matrices over the "raw" field, with respect to a basis which remains fixed until `SetRationalBasis(M)` is invoked by the user. When that occurs, the basis of the space is changed to a "rational basis" which is then permanently fixed; as a result, the operators also change (they are now given as matrices with respect to the new basis).

A space M is guaranteed to have a "rational basis" (which is permanently fixed) in any of the following circumstances:

(i) M has parallel weight,

(ii) M was constructed using `NewSubspace` (unless `RationalBasis` was set to `false`), or was constructed using `NewformDecomposition`,

(iii) `SetRationalBasis(M)` has been invoked.

`HeckeOperator(M, P)`

This returns a matrix representing the Hecke operator T_P on the space M of Hilbert modular forms.

Example H137E3

```
> R<x> := PolynomialRing(IntegerRing());
> F<a> := NumberField(x^2-2);  OF := Integers(F);
> M3 := HilbertCuspForms(F, 3*OF, [2,4]);
> Dimension(M3);
1
> P2 := Factorization(2*OF)[1][1];
> Norm(P2);
2
> HeckeOperator(M3, P2);
[2]
```

Since the space has dimension 1, it consists of a single eigenform, whose eigenvalues can be read from the Hecke matrices. We now consider the space of level 5, which has dimension 3. The Hecke matrices are given with respect to the basis used to compute the space, over the extension K displayed here.

```
> M5 := HilbertCuspForms(F, 5*OF, [2,4] : QuaternionOrder:=QuaternionOrder(M3) );
> Dimension(M5);
3
> T2 := HeckeOperator(M5, P2);
> K<b> := BaseRing(T2);
> K;
Number Field with defining polynomial $.1^2 + 1 over F
> T2;
[       0    8*a*b -2*b - 2]
[-1/2*a*b        2        0]
[   b - 1        0        0]
```

`AtkinLehnerOperator(M, P)`

This returns a matrix representing the Atkin-Lehner operator w_P on the space M of Hilbert modular forms.

`DegeneracyOperator(M, P, Q)`

This computes degeneracy maps in the "downward" direction as maps from M to itself. Here M is a space of Hilbert modular forms of level N, P is a prime dividing N, and Q either equals P or the unit ideal (1). The function returns a matrix representing a map from M to M whose image equals a copy of the space of level N/P.

When $Q = (1)$, this is double coset operator defined by cosets of an element with determinant 1 (which can be seen as a "norm" map); when $Q = P$, it is the double coset operator defined by cosets of an element with determinant Norm(P).

`DeleteHeckePrecomputation(O)`

`DeleteHeckePrecomputation(O, P)`

These procedures delete data obtained during the precomputation phase of the "definite" algorithm. This data is used in the computation of Hecke operators (and other operators) for given primes, and the same data can be re-used for all spaces that are computed with the some quaternion order O. (Often this is the same for spaces of all weights and levels over a particular number field.) Since the data is the most expensive part of the Hecke computation, it is stored by default. However it is very expensive in memory; these procedures allow the user to reclaim this memory when necessary.

137.6 Creation of Subspaces

`NewSubspace(M)`

`NewSubspace(M, I)`

`QuaternionOrder` AlgAssVOrd *Default :*

Given a cuspidal space M of Hilbert modular forms of level N, and an ideal I dividing N, this constructs the subspace of M consisting of forms that are new at the ideal I (or that are new at N, if I is not given). More precisely, this is the complement of the space generated by all images under degeneracy maps of spaces of level N/P for primes P dividing I. In the current implementation, I must be squarefree and coprime to N/I.

The `NewSubspace` of M is not necessarily represented as an explicit subspace of M: doing so is not always possible, or may not be optimal. (These choices are internal: the difference is not visible in the output.) In many cases new subspace is obtained as an explicit subspace by computing degeneracy maps. In other case it is computed independently of M, using a quaternion order which is chosen automatically. When the optional argument `QuaternionOrder` is specified, this will always be used (overriding other considerations). The allowable orders are similar to those for full cuspidal spaces (see `HilbertCuspForms` above), with the difference that here the quaternion algebra is allowed to be ramified at finite primes dividing I. When the algebra is indefinite, the finite primes where it is ramified must be precisely those dividing I.

In the non-parallel weight case, work may be saved by setting the optional argument `RationalBasis` to `false`. This defers the computation of a "rational basis" of the new space (this is explained below). By default, a rational basis is computed for `NewSubspace(M)`, and (equivalently) `NewSubspace(M)` where I equals `Level(M)`. If it is defered, one may later set a "rational basis" for the new space using `SetRationalBasis`.

`SetRationalBasis(M)`

This is a procedure which changes the basis of M. A full explanation is given in the first part of Section 137.5.

If the basis of M is already known to be a "rational basis", then nothing is done; the basis of M is not modified. In particular, this has no effect on spaces of parallel weight 2.

After this has been invoked, the basis of M is never modified again.

Example H137E4

We investigate new forms and old forms with level dividing 3 over $\mathbf{Q}(\sqrt{10})$.

```
> _<x> := PolynomialRing(Rationals());
> F := NumberField(x^2-10);
> OF := Integers(F);
> primes := [tup[1] : tup in Factorization(3*OF)];
> #primes;
2
> M1 := HilbertCuspForms(F, 1*OF);
> Dimension(M1);
2
> M3 := HilbertCuspForms(F, primes[1]);
> Dimension(M3);
4
```

Hence $M3$ must contain only oldforms (since there are two degeneracy maps $M1 \to M3$, whose images must be linearly independent). We confirm this now:

```
> Dimension(NewSubspace(M3));
0
```

Next we consider level (3).

```
> M9 := HilbertCuspForms(F, 3*OF);
> Dimension(M9);
18
> Dimension(NewSubspace(M9, primes[1]));
14
> Dimension(NewSubspace(M9, primes[2]));
14
> Dimension(NewSubspace(M9));
10
```

The dimensions indicate that the four degeneracy maps $M1 \to M9$ have independent images.

Example H137E5

This illustrates that new subspaces may be computed independently, not using the same algorithm as for the full space. We compute forms over the real subfield of $\mathbf{Q}(\zeta_7)$, which has degree 3.

```
> _<x> := PolynomialRing(Rationals());
```

```
> _<zeta7> := CyclotomicField(7);
> F<a> := NumberField(MinimalPolynomial(zeta7 + 1/zeta7));
> F;
Number Field with defining polynomial x^3 + x^2 - 2*x - 1 over the Rational Field
```

We consider forms of level 3 (which generates a prime ideal in F).

```
> M := HilbertCuspForms(F, 3*Integers(F));
> M;
Cuspidal space of Hilbert modular forms over
Number Field with defining polynomial x^3 + x^2 - 2*x - 1 over the Rational Field
   Level = Ideal of norm 27 generated by ( [3, 0, 0] )
   Weight = [ 2, 2, 2 ]
> Mnew := NewSubspace(M);
> Dimension(M);
1
> Dimension(Mnew);
1
```

So in fact M equals its new subspace, and consists of just one newform. However, different algorithms have been chosen to compute them, as indicated below. (Algorithm I is chosen automatically for the new subspace because it is much faster. It cannot be used for the full space, since over an odd degree field there is no quaternion algebra ramified at precisely the infinite places.)

```
> IsDefinite(M);
false
> IsDefinite(Mnew);
true
```

We now compute some eigenvalues of the newform generating the space, using both algorithms. (The computations for M_{new} are entirely independent from those for M.)

```
> primes := PrimesUpTo(20,F);
> [Norm(P) : P in primes];
[ 7, 8, 13, 13, 13 ]
> time for P in primes do HeckeOperator(Mnew,P); end for;
[-5]
[-4]
[1]
[1]
[1]
Time: 0.810
> time HeckeOperator(M, primes[1]);
[-5]
Time: 38.800
```

137.7 Eigenspace Decomposition and Eigenforms

`HeckeEigenvalueBound(M, P)`

Returns a bound on the absolute value of the Hecke eigenvalue at the prime P which must hold for all newforms in the space M of Hilbert modular forms.

`NewformDecomposition(M)`

Given a space M of Hilbert modular forms which was created as a `NewSubspace`, this decomposes M into subspaces that are irreducible modules under the Hecke action.

`NewformsOfDegree1(M)`

This constructs the list of new eigenforms in M that have rational eigenvalues, i.e. corresponding to the 1-dimensional components in the `NewformDecomposition`. The space M is not required to be a new space. The algorithm avoids constructing the new subspace of M, and makes use of bounds on the eigenvalues.

`Eigenform(M)`

This constructs an eigenform contained in the space M of Hilbert modular forms (which should be an irreducible module under the Hecke action, for instance a space obtained using `NewformDecomposition`).

`Eigenforms(M)`

This is a list containing an eigenform from each space in `NewformDecomposition(M)`.

`HeckeEigenvalueField(M)`

Given a space M constructed using `NewformDecomposition`, this returns the number field over which the `Eigenform` of M is defined.

`HeckeEigenvalue(f, P)`

The computes the eigenvalue of the Hecke operator T_P acting on the eigenform f (which should be a Hilbert modular form constructed using `Eigenform`).

Example H137E6

We compute the newforms corresponding to elliptic curves over $\mathbf{Q}(\sqrt{2})$ of conductor 11, and find there is only the one coming from $\mathbf{Q}$.

```
> R<x> := PolynomialRing(IntegerRing());
> F := NumberField(x^2-2);  OF := Integers(F);
> M := HilbertCuspForms(F, 11*OF);
> Dimension(M);
6
> time decomp := NewformDecomposition(NewSubspace(M)); decomp;
Time: 11.130
[*
```

```
    New cuspidal space of Hilbert modular forms of dimension 1 over
    Number Field with defining polynomial x^2 - 2 over the Rational Field
       Level = Ideal of norm 121 generated by ( [11, 0] )
       Weight = [ 2, 2 ],
    New cuspidal space of Hilbert modular forms of dimension 5 over
    Number Field with defining polynomial x^2 - 2 over the Rational Field
       Level = Ideal of norm 121 generated by ( [11, 0] )
       Weight = [ 2, 2 ]
*]
```

We look at the first few eigenvalues of the 1-dimension piece (at split primes).

```
> f := Eigenform(decomp[1]);
> primes := [P : P in PrimesUpTo(40,F) | IsOdd(Norm(P)) and IsPrime(Norm(P))];
> for P in primes do
>   Norm(P),  HeckeEigenvalue(f,P);
> end for;
7 -2
7 -2
17 -2
17 -2
23 -1
23 -1
31 7
31 7
```

Happily, they agree with the eigenvalues of the elliptic cusp form of conductor 11 over $\mathbf{Q}$:

```
> fQ := Newforms(CuspForms(11))[1][1];
> for P in primes do
>   p := Norm(P);
>   p,  Coefficient(fQ, p);
> end for;
7 -2
7 -2
17 -2
17 -2
23 -1
23 -1
31 7
31 7
```

The 5-dimensional piece conjecturally corresponds to an abelian variety over F of dimension 5, which would be absolutely irreducible and have real multiplication by the following field.

```
> K := HeckeEigenvalueField(decomp[2]);
> K;
Number Field with defining polynomial $.1^5 - 8*$.1^3 + 10*$.1 + 4 over F
> IsTotallyReal(K);
true
```

137.8 Further Examples

Example H137E7

```
> R<x> := PolynomialRing(IntegerRing());
> F := NumberField(x^2-15);  OF := Integers(F);
> M := HilbertCuspForms(F, 1*OF, [2,4]);
> M;
Cuspidal space of Hilbert modular forms over Number Field with defining polynomial
x^2 - 15 over the Rational Field
   Level = Ideal of norm 1 generated by ( [1, 0] )
   Weight = [ 2, 4 ]
> Dimension(M);
2
> T2 := HeckeOperator(M, Factorization(2*OF)[1][1] );
> BaseRing(T2);
Number Field with defining polynomial $.1^2 + 1 over F
```

So the basis used to compute M is over this extension of F. We now replace this with a F-rational basis (this is implemented for new spaces, in particular spaces of level 1).

```
> IsNew(M);
true
> SetRationalBasis(M);
> for P in PrimesUpTo(7,F) do
>    Norm(P), HeckeOperator(M,P);
> end for;
2
[  0   1]
[-20   0]
3
[0 0]
[0 0]
5
[  0   4]
[-80   0]
7
[0 0]
[0 0]
7
[0 0]
[0 0]
> #NewformDecomposition(M);
1
```

In general the Hecke matrices would be over F, however for this space there is a basis where they have entries in $\mathbf{Z}$. The program was able to discover this because the basis is chosen by putting one of the matrices in rational canonical form.

Example H137E8

It is possible to use this package to compute classical modular forms (although this will usually be much slower). Here we compute the newform of level 14 three times independently. First we wish to use Algorithm 1, so we choose the quaternion algebra over $\mathbf{Q}$ ramified at 2 and infinity. (Note: Algorithm I is not implemented over `Rationals()`, so we must work over a number field isomorphic to $\mathbf{Q}$ instead!)

```
> QQ := RationalsAsNumberField();
> ZZ := Integers(QQ);
> M := HilbertCuspForms(QQ, 14*ZZ);
> A := QuaternionAlgebra(2*ZZ, InfinitePlaces(QQ) : Optimized);
> M14 := NewSubspace(M : QuaternionOrder:=MaximalOrder(A) );
> Dimension(M14);
1
> f := Eigenform(NewformDecomposition(M14)[1]);
> primes := PrimesUpTo(50);
> time eigenvalues1:= [ <p, HeckeEigenvalue(f,p*ZZ)> : p in primes ];
Time: 0.220
> eigenvalues1;
[ <2, -1>, <3, -2>, <5, 0>, <7, 1>, <11, 0>, <13, -4>, <17, 6>, <19, 2>, <23, 0>,
  <29, -6>, <31, -4>, <37, 2>, <41, 6>, <43, 8>, <47, -12> ]
```

Now we use Algorithm 2, choosing the indefinite quaternion algebra over $\mathbf{Q}$ ramifed at 2 and 7.

```
> Q := Rationals();
> M := HilbertCuspForms(Q, 14);
> A := QuaternionAlgebra(14 : Al:="Smallest" );
> A.1^2, A.2^2;
7, -2
> M14 := NewSubspace(M : QuaternionOrder:=MaximalOrder(A) );
> IsDefinite(M14); // Not definite means Algorithm 2
false
> Dimension(M14);
1
> f := Eigenform(NewformDecomposition(M14)[1]);
> time eigenvalues2 := [ <p, HeckeEigenvalue(f,p)> : p in primes |
>                                                     GCD(p,14) eq 1];
Time: 2.750
> eigenvalues2;
[ <3, -2>, <5, 0>, <11, 0>, <13, -4>, <17, 6>, <19, 2>, <23, 0>,
  <29, -6>, <31, -4>, <37, 2>, <41, 6>, <43, 8>, <47, -12> ]
```

Finally we check both results agree with the standard modular forms package.

```
> M14 := CuspForms(14);
> time eigenvalues := [ <p, HeckeOperator(M14,p)[1,1]> : p in primes ];
Time: 0.160
> assert eigenvalues1 eq eigenvalues;
> assert eigenvalues2 subset eigenvalues;
```

137.9 Bibliography

[**DD08**] L. Dembele and S. Donnelly. Computing Hilbert Modular Forms Over Fields With Nontrivial Class Group. In S. Pauli F. Hess and M.Pohst, editors, *ANTS VIII*, volume 5011 of *LNCS*. Springer-Verlag, 2008.

[**Dem07**] L. Dembélé. Quaternionic Manin symbols, Brandt matrices, and Hilbert modular forms. *Math. Comp.*, 76(258):1039–1057 (electronic), 2007.

[**DV12**] L. Dembélé and J. Voight. Explicit methods for Hilbert modular forms. *to appear, Elliptic curves, Hilbert modular forms and Galois deformations*, 2012.

[**Fre90**] E. Freitag. *Hilbert modular forms*. Springer-Verlag, Berlin, 1990.

[**Gar90**] Paul B. Garrett. *Holomorphic Hilbert modular forms*. The Wadsworth & Brooks/Cole Mathematics Series. Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, CA, 1990.

[**GV11**] M. Greenberg and J. Voight. Computing systems of eigenvalues associated to Hilbert modular forms. *Math. Comp.*, 80:1071–1092, 2011.

[**Hid06**] Haruzo Hida. *Hilbert modular forms and Iwasawa theory*. Oxford Mathematical Monographs. The Clarendon Press Oxford University Press, Oxford, 2006.

[**Voi09**] J. Voight. Computing fundamental domains for cofinite Fuchsian groups. *J. Théor. Nombres Bordeaux*, 21(2):469–491, 2009.

138 MODULAR FORMS OVER IMAGINARY QUADRATIC FIELDS

Chapter 138
MODULAR FORMS OVER IMAGINARY QUADRATIC FIELDS

138.1 Introduction

This package deals with cuspidal spaces of weight 2 modular forms on $\Gamma_0(N)$, over any imaginary quadratic field. In the current version, one can compute Hecke operators for principal ideals on these spaces, and determine the newforms. The package will be developed further in the future.

We will refer to modular forms over imaginary quadratic fields as *Bianchi modular forms*. These are defined similarly to classical modular forms, with the modular group $SL(2, \mathbf{Z})$ is replaced by $SL(2, O_F)$, where O_F is the ring of integers in an imaginary quadratic field F. This group acts on $\mathbf{H_3}$, 3-dimensional hyperbolic space, and Bianchi modular forms (of weight $k \geq 2$) are functions on $\mathbf{H_3}$ that satisfy a natural automorphy relation under this action.

For an ideal N of O_F, the congruence subgroup $\Gamma_0(N)$ of $GL(2, O_F)$ is defined as the subgroup of matrices that are upper triangular modulo N. The space of *Bianchi modular forms on F of level N* is defined as the space of functions satisfying the automorphy relation on the subgroup $\Gamma_0(N)$.

For precise definitions see [EGM98].

There exist several previous implementations of weight 2 Bianchi modular forms, each for specific fields F. These projects are described in [Cre84] and references cited there for Euclidean fields, [Whi90] for fields of class number one, [Byg99] for $\mathbf{Q}(\sqrt{-5})$ and [Lin05] for $\mathbf{Q}(\sqrt{-23})$ and $\mathbf{Q}(\sqrt{-31})$.

138.1.1 Algorithms

A theorem of Franke states that for algebraic group $\mathbf{G}$ defined over $\mathbf{Q}$ and arithmetic subgroup Γ,

$$H^*(\Gamma; E) \simeq H^*(g, K; A(\Gamma, G) \otimes E)$$

where $A(\Gamma, G)$ denotes the space of automorphic forms. Thus we can think of the cohomology $H^*(\Gamma; E)$ as a concrete realization of certain automorphic forms.

Ash, Gunnells, and Lee-Szczarba [LS78, Ash94, Gun00] define a homology complex $S^*(\Gamma)$ known as the *sharbly complex*. A theorem of Borel-Serre [BS73] gives

$$H^{\nu-k}(\Gamma; \mathbf{C}) \simeq H_k(S_*(\Gamma)),$$

where $\nu = \mathrm{vcd}(\Gamma)$ is the *virtual cohomological dimension* of Γ.

There is a natural Hecke action on this complex which agrees with the Hecke action on the automorphic forms.

The space of positive definite binary Hermitian forms over F form an open cone in a real vector space. There is a natural decomposition of this cone into polyhedral cones corresponding to the facets of the Voronoi polyhedron [Gun99, Koe60, Ash77]. These facets are in 1-1 correspondence with perfect forms over F. The polyhedral cones give rise to ideal polytopes in $\mathbf{H_3}$, 3-dimensional hyperbolic space.

The structure of the polytopes allows us to find a finite (modulo Γ) spanning set for the sharbly complex. This is the analogue of unimodular symbols in the classical case. The modular symbol algorithm, for describing the action of Hecke operators, can now be replaced by a 0-sharbly reduction algorithm. An advantage of this is, compared with a straightforward generalization of the usual modular symbols algorithm, is that the number field does not need to be Euclidean.

Given an imaginary quadratic field F, we compute the structure of the Voronoi polyhedron by computing a complete set of $\mathrm{GL}_2(\mathcal{O})$-class representatives of perfect forms. Given a level $n \subseteq \mathcal{O}$ defining the level of the congruence subgroup, the polyhedron is used to compute $H^2(\Gamma_0(n))$. Given a prime ideal $p \subset \mathcal{O}$, the 0-sharbly reduction algorithm is used to compute the action of the Hecke operator T_p on $H^2(\Gamma_0(n))$.

The algorithm described in [Gun99] is an efficient algorithm for computing the Voronoi polyhedron and provides a replacement for the modular symbol algorithm for computing the action of Hecke operators.

One advantage of this algorithm is that the most expensive steps come in the precomputation phase (computations depending only on the imaginary quadratic field). This means that for a given field, forms of many different levels can be computed based on the same precomputation.

The cohomology, regarded as a module for the algebra generated by the Hecke operators T_p, decomposes into an Eisenstein piece and a cuspidal piece. The cuspidal piece is the same as the cuspidal subspace of modular forms (regarding both as Hecke modules). The Eisenstein part is recognisable by looking at Hecke eigenvalues of the eigenforms contained in it. By this means, we strip away the Eisenstein part; the space returned is the cuspidal subspace only.

138.1.2 Categories

Spaces of Bianchi modular forms in MAGMA are objects of type `ModFrmBianchi`. This is functionally the same as the type `ModFrmHil`: functions for Hilbert modular forms (see Chapter 137) can also be applied to Bianchi spaces, for example `NewformDecomposition`.

138.1.3 Verbose Output

To see some information printed during computation about what the program is doing, use `SetVerbose("Bianchi",n)`, where `n` is 0 (silent, by default), 1 (for concise information), 2, 3 or 4 (which may display bulky data).

138.2 Creation

In the current implementation only cusp forms are supported.

`BianchiCuspForms(F, N)`

`VorData` REC *Default :*

This creates the cuspidal subspace of the space of Bianchi modular forms over the field F (an imaginary quadratic field) on $\Gamma_0(N)$ with weight 2. The level N should be an ideal in the maximal order of F.

The optional argument `VorData` may be set equal to `VoronoiData(M)` where `M` is a previously computed space of forms over the same field. This avoids repeating the time-consuming precomputations that depend only on the field.

138.3 Attributes

`BaseField(M)`

The field on which the space M of Bianchi modular forms was defined.

`Level(M)`

The level of the space M.

`Dimension(M)`

The dimension of the space M. Dimension formulas are not available, so the dimension is computed by explicit construction of the space.

`VoronoiData(M)`

This returns a record containing technical data that is computed in the precomputation phase of the algorithm. This depends only on the base field of M, and the data can be reused when computing spaces of different levels over the same field.

Example H138E1

We create spaces of modular forms over $\mathbf{Q}(\sqrt{-14})$ for various levels.

```
> _<x> := PolynomialRing(Rationals());
>  F := NumberField(x^2+14);
>  OF := Integers(F);
>  level := 1*OF;
>  M := BianchiCuspForms(F, level);
>  M;
Cuspidal space of Bianchi modular forms over
    Number Field with defining polynomial x^2 + 14 over the Rational Field
    Level = Ideal of norm 1 generated by ( [1, 0] )
    Weight = 2
>  time Dimension(M);
0
```

```
Time: 4.980
```

We now define a space with level equal to the square of one of the split primes dividing 3.

```
> level := (Factorization(3*OF)[1][1])^2;
> Norm(level);
9
> time M9 := BianchiCuspForms(F, level);
Time: 7.300
> M9;
Cuspidal space of Bianchi modular forms over
    Number Field with defining polynomial z^2 + 14 over the Rational Field
    Level = Ideal of norm 9 generated by ( [9, 0], [5, 2] )
    Weight = 2
```

If we wish, we may tell Magma to use the same Voronoi data (to avoid repeating the time-consuming precomputation):

```
> time M9 := BianchiCuspForms(F, level : VorData := VoronoiData(M) );
Time: 0.100
> M9;
Cuspidal space of Bianchi modular forms over
    Number Field with defining polynomial z^2 + 14 over the Rational Field
    Level = Ideal of norm 9 generated by ( [9, 0], [5, 2] )
    Weight = 2
>  Level(M9);
Ideal
Two element generators:
    [9, 0]
    [5, 2]
>  time Dimension(M9);
1
Time: 24.540
```

138.4 Hecke Operators

The computations are done essentially on the cohomology of $\Gamma\backslash\mathbf{H_3}$, and so the for a non-principal ideal P, the Hecke operator T_P does not act on this space, but sometimes the Hecke action can be deduced from the action of principal Hecke operators. See for instance [Lin05]. Specifically, suppose p is a prime ideal that is prime to the level. There exists an ideal a, prime to p and the level, such that a^2p is principal. Then the composition $T_{a,a}T_p$ acts on the cohomology.

`HeckeOperator(M, P)`

This returns a matrix representing a certain Hecke action T on the space M of Bianchi modular forms, with respect to the fixed basis of M. The ideal P must be principal (but not necessarily prime) or prime and a square in the class group. If P is principal, then T is the Hecke operator T_P. When P is prime and a square in the class group, T is the composition $T_{a,a}T_P$ for a suitably chosen ideal a.

138.5 Newforms

The functions `NewSubspace` and `NewformDecomposition` may be applied to spaces of Bianchi modular forms. (See Chapter 137.)

Example H138E2

We continue the previous example.

```
> _<x> := PolynomialRing(Rationals());
>  F := NumberField(x^2+14);
>  OF := Integers(F);
>  level := (Factorization(3*OF)[1][1])^2;
>  M9 := BianchiCuspForms(F, level);
>  P:=Factorization(23*OF);
>  P[1,1];
Prime Ideal of OF
Two element generators:
    [23, 0]
    [3, 1]
>  HeckeOperator(M9, P[1,1]);
[8]
>  P[2,1];
Prime Ideal of OF
Two element generators:
    [23, 0]
    [20, 1]
>  HeckeOperator(M9, P[2,1]);
[-8]
>  HeckeOperator(M9, 2*OF);
[1]
```

Since this cuspidal space has dimension 1, it consists of a single eigenform, whose eigenvalues can be read from the Hecke matrices:

138.6 Bibliography

[**Ash77**] Avner Ash. Deformation retracts with lowest possible dimension of arithmetic quotients of self-adjoint homogeneous cones. *Math. Ann.*, 225(1):69–76, 1977.

[**Ash94**] Avner Ash. Unstable cohomology of $\mathrm{SL}(n, \mathcal{O})$. *J. Algebra*, 167(2):330–342, 1994.

[**BS73**] A. Borel and J.-P. Serre. Corners and arithmetic groups. *Comment. Math. Helv.*, 48:436–491, 1973. Avec un appendice: Arrondissement des variétés à coins, par A. Douady et L. Hérault.

[**Byg99**] Jeremy Bygott. *Modular forms and modular symbols over imaginary quadratic fields*. PhD thesis, University of Exeter, 1999.

[**Cre84**] J. E. Cremona. Hyperbolic tessellations, modular symbols, and elliptic curves over complex quadratic fields. *Compositio Math.*, 51(3):275–324, 1984.

[**EGM98**] J. Elstrodt, F. Grunewald, and J. Mennicke. *Groups acting on hyperbolic space*. Springer Monographs in Mathematics. Springer-Verlag, Berlin, 1998. Harmonic analysis and number theory.

[**Gun99**] Paul E. Gunnells. Modular symbols for $\mathbf{Q}$-rank one groups and Voronoi reduction. *J. Number Theory*, 75(2):198–219, 1999.

[**Gun00**] Paul E. Gunnells. Computing Hecke eigenvalues below the cohomological dimension. *Experiment. Math.*, 9(3):351–367, 2000.

[**Koe60**] Max Koecher. Beiträge zu einer Reduktionstheorie in Positivitätsbereichen. I. *Math. Ann.*, 141:384–432, 1960.

[**Lin05**] Mark Lingham. *Modular forms and elliptic curves over imaginary quadratic fields*. PhD thesis, University of Nottingham, 2005.
URL:http://etheses.nottingham.ac.uk/138/.

[**LS78**] Ronnie Lee and R. H. Szczarba. On the torsion in $K_4(\mathbf{Z})$ and $K_5(\mathbf{Z})$. *Duke Math. J.*, 45(1):101–129, 1978.

[**Whi90**] Elise Whitley. *Modular symbols and elliptic curves over imaginary quadratic fields*. PhD thesis, University of Exeter, 1990.

139 ADMISSIBLE REPRESENTATIONS OF $\mathrm{GL}_2(\mathbf{Q}_p)$

Chapter 139
ADMISSIBLE REPRESENTATIONS OF $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$

139.1 Introduction

This package enables one to do the following. Starting with a cuspidal newform, one may define the local component at p of the associated automorphic representation, and determine its key properties. Furthermore, via the local Langlands correspondence, there exists a related Galois representation on the absolute Galois group of $\mathbf{Q}_p$. One may compute (the restriction to inertia of) that Galois representation.

The algorithms implemented here are described in [LW].

139.1.1 Motivation

Let F be a local non-archimedean field and let G be a reductive group over F. The representation theory of G is a rich subject in its own right but also has fascinating (and often conjectural) connections with the representation theory of the absolute Galois group of F. This package deals with admissible irreducible representations in the case that G is the group GL_2 and F is the p-adic field $\mathbf{Q}_p$. Such objects correspond canonically to two-dimensional representations of the absolute Galois group of F of a certain sort.

There are applications to the study of local properties of (global) Galois representations arising from modular forms. Namely, if f is a cuspidal newform, then there is associated to f a family of ℓ-adic Galois representations $\rho_f : \mathrm{Gal}(\overline{\mathbf{Q}}/\mathbf{Q}) \to \mathrm{GL}_2(\overline{\mathbf{Q}}_\ell)$. For example, for a prime p different from ℓ, one may determine the restriction of ρ_f to the decomposition group at p.

139.1.2 Definitions

Here we introduce the category of admissible irreducible representations of GL_2 over a non-archimedean field, which for our purposes will be $\mathbf{Q}_p$. The first systematic study of admissible representations is [JL70]. For an accessible introduction, see [BH06].

Let G be the locally compact group $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$. An *admissible* representation of G on a complex vector space V is a homomorphism $\pi : G \to \mathrm{AutV}$ which satisfies the properties:

(i) every vector $v \in V$ is fixed by a compact open subgroup of G, and

(ii) for every compact open subgroup $K \subset G$, V^K is finite-dimensional.

The center of G is $\mathbf{Q}_p^\times$. If π is an irreducible admissible representation of G, then it has a unique *central character* $\varepsilon : \mathbf{Q}_p^\times \to \mathbf{C}^\times$ such that $\pi(g)$ acts as the scalar $\varepsilon(g)$ for all $g \in \mathbf{Q}_p^\times$. The *conductor* of an irreducible admissible representation π is a measure of how small a compact open subgroup $K \subset G$ must be before one sees nonzero K-invariant vectors; see [Cas73]. Consider the filtration $K_0(p^n)$ of subgroups of G, where $K_0(p^n)$ is the

subgroup of matrices $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{GL}_2(\mathbf{Z}_\mathrm{p})$ with $c \equiv 0 \pmod{p^n}$. If π admits a nonzero vector fixed by $K_0(1) = \mathrm{GL}_2(\mathbf{Z}_\mathrm{p})$, then π is called *spherical* or *unramified principal series* and has conductor 1. If π admits a nonzero vector v for which $\pi\left(\begin{pmatrix} a & b \\ c & d \end{pmatrix}\right) v = \varepsilon(a)v$ for all $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \in K_0(p^n)$, and $n \geq 1$ is minimal for this condition, then the conductor of π is p^n. (Note the similarity with the convention used to define the level of a modular form for $\Gamma_0(N)$.) In both cases the vector v so described is unique up to scaling; see [Cas73]. We shall call v a *new vector* for π.

If χ is a character of $\mathbf{Q}_p^\times$ then let $\pi \otimes \chi$ be the representation $g \mapsto \chi(g)\pi(g)$; such a representation is a *twist* of χ. If π has minimal conductor among all its twists $\pi \otimes \chi$, then π is called *minimal*.

Admissible representations are generally infinite-dimensional, but we will nonetheless be able to present them using Magma infrastructure for representations of finite groups and Dirichlet characters.

139.1.3 The Principal Series

We can directly construct a large class of admissible representations of G. Let χ_1 and χ_2 be two characters of $\mathbf{Q}_p^*$. Let $B \subset G$ be the Borel subgroup of upper triangular matrices. Then $\begin{pmatrix} a & b \\ 0 & d \end{pmatrix} \mapsto |a/d|^{-1/2}\chi_1(a)\chi_2(d)$ is a character χ of B. An admissible representation π is a *principal series* representation if it is a composition factor of the induced representation $\pi(\chi_1, \chi_2) := \mathrm{Ind}_\mathrm{B}^\mathrm{G}\chi$. This induced representation is already irreducible unless $\chi_1\chi_2^{-1}$ equals $|.|^{\pm 1}$, in which case it has length two, with one 1-dimensional and one infinite-dimensional composition factor. For instance, $\mathrm{Ind}_\mathrm{B}^\mathrm{G}1$ has a trivial 1-dimensional submodule and an irreducible infinite-dimensional quotient St_G, the *Steinberg* representation. The unramified principal series representations are either 1-dimensional, in which case they factor through the determinant map, or else they take the form $\pi(\chi_1, \chi_2)$, where χ_1 and χ_2 are unramified characters of $\mathbf{Q}_p^\times$ (meaning they are trivial on $\mathbf{Z}_p^\times$).

The central character of $\pi(\chi_1, \chi_2)$ is $\chi_1\chi_2$, and its conductor is the product of the conductors of the χ_i. Note that $\pi(\chi_1, \chi_2)$ is minimal if and only if one of the characters χ_1, χ_2 is unramified. The Steinberg representation St_G has trivial central character and conductor p.

139.1.4 Supercuspidal Representations

For the purposes of this package, an admissible irreducible representation is *supercuspidal* if it does not belong to the principal series. A supercuspidal representation π has conductor p^c, where $c \geq 2$. There is a convenient, if technical, classification of supercuspidal representations of G. Let π be supercuspidal. By [BH06], Ch. 15, there is a representation Ξ of an open and compact-mod-center subgroup $K \subset G$ for which $\pi = \mathrm{Ind}_\mathrm{K}^\mathrm{G}\Xi$. If c is even, then we may take K to be $\mathbf{Q}_p^\times\mathrm{GL}_2(\mathbf{Z}_\mathrm{p})$, and if c is odd, we may take K to be the

normalizer of the Iwahori subgroup $K_0(p) = \begin{pmatrix} \mathbf{Z}_p^\times & \mathbf{Z}_p \\ p\mathbf{Z}_p & \mathbf{Z}_p^\times \end{pmatrix}$ in G. We call the pair (K, Ξ) a *cuspidal inducing datum.*

139.1.5 The Local Langlands Correspondence

The Local Langlands Correspondence is a canonical bijection $\pi \mapsto \sigma(\pi)$ between irreducible admissible representations of G and local 2-dimensional representations of the absolute Galois group of $\mathbf{Q}_p$ of a certain sort. (Note to purists: the proper Galois-theoretic object to study in this scenario is the Weil-Deligne representation, which consists of the datum of a representation of the Weil group, together with a monodromy operator. See [Tat79].) The bijection manifests as an agreement of L- and ε-factors that one constructs for each category. The foundation for the Local Langlands Correspondence for GL_2 over a non-archimedean field was laid in [JL70]; the work was completed in [Kut80] and [Kut84].

We remark that the conductor of π agrees with the Artin conductor of $\sigma(\pi)$, and that π is principal series (resp., Steinberg, supercuspidal) if and only if $\sigma(\pi)$ is a sum of two characters (resp., reducible but not decomposable, irreducible). We also remark that if $p \neq 2$ and σ is an irreducible 2-dimensional Galois representation of $\mathbf{Q}_p$, then σ must be induced from a character χ of a quadratic field extension $E/\mathbf{Q}_p$. Then (E, χ) is called an *admissible pair* (see [BH06], Ch. 18).

139.1.6 Connection with Modular Forms

The classical theory of modular forms has a modern interpretation in terms of *cuspidal automorphic representations.* These are representations Π of the adele group $\mathrm{GL}_2(\mathbf{A_Q})$ which appear in the Hilbert space of square-integrable cuspidal functions $L_0^2(\mathrm{GL}_2(\mathbf{Q})\backslash\mathrm{GL}_2(\mathbf{A_Q}), \varepsilon)$, where ε is a Dirichlet character. Let f be a cuspidal newform for $\Gamma_0(N)$ with Dirichlet character ε. Then there is associated to f a cuspidal automorphic representation Π_f, see [Gel75]. This is a restricted tensor product $\bigotimes_{p \leq \infty} \pi_{f,p}$, where if p is a finite prime, $\pi_{f,p}$ is an admissible representation of $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$. There is also the Galois representation ρ_f attached to f constructed by Deligne. By [Car83] there is a straightforward relationship between $\sigma(\pi_{f,p})$ and the restriction of ρ_f to the decomposition group at p. Therefore to determine the local properties of ρ_f it is enough to compute the local components $\pi_{f,p}$. These are almost always unramified principal series; the only challenge is to compute $\pi_{f,p}$ when p divides N.

139.1.7 Category

In MAGMA, admissible representations are objects of type `RepLoc`.

139.1.8 Verbose Output

To see information about computations in progress, enter `SetVerbose("RepLoc", 1)`.

139.2 Creation of Admissible Representations

One starts with a classical cuspidal eigenform, given as a space of modular symbols.

`LocalComponent(M, p)`

This returns the admissible representation of $\mathrm{GL}(\mathbf{Q}_\mathrm{p})$ associated to the cuspidal eigenform specified by M. Here M must be a space of modular symbols that is cuspidal and contains only a single Galois conjugacy class of newforms. (Such spaces are created using `NewformDecomposition`).

Example H139E1

We create the local component at 11 of the representation associated to the newform of level 11 and weight 2. We specify the newform as a space of modular symbols of level 11, weight 2 and sign +1.

```
> S11 := CuspidalSubspace(ModularSymbols(11, 2, 1));
> newform_spaces := NewformDecomposition(S11);
> newform_spaces;
[
    Modular symbols space for Gamma_0(11) of weight 2 and dimension 1 over Q
]
> Eigenform(newform_spaces[1]);
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
> LocalComponent(newform_spaces[1], 11);
Steinberg Representation of GL(2,Q_11)
```

139.3 Attributes of Admissible Representations

`CentralCharacter(pi)`

The central character of π, where π is an admissible representation on $\mathrm{GL}(\mathbf{Q}_\mathrm{p})$. This is a Dirichlet character of p-power conductor.

`Conductor(pi)`

The conductor of π, written multiplicatively.

`DefiningModularSymbolsSpace(pi)`

The space of modular symbols from which π was created.

`IsMinimal(pi)`

Given a representation π of $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$, returns `true` if the conductor of π cannot be lowered by twisting by a character of $\mathbf{Q}_p^\times$. If π is not minimal, the function also returns a minimal representation π' together with a Dirichlet character χ, such that π is the twist of π' by χ.

This is true iff `IsMinimalTwist(DefiningModularSymbolsSpace(pi))` is true.

Example H139E2

We continue the previous example.

```
> S11 := CuspidalSubspace(ModularSymbols(11, 2, 1));
> E11 := NewformDecomposition(S11)[1];
> E11;
Modular symbols space for Gamma_0(11) of weight 2 and dimension 1 over Q
> pi := LocalComponent(E11, 11);
> pi;
Steinberg Representation of GL(2,Q_11)
> DefiningModularSymbolsSpace(pi) eq E11;
true
> Conductor(pi);
11
> IsTrivial(CentralCharacter(pi));
true
```

139.4 Structure of Admissible Representations

`IsPrincipalSeries(pi)`

This is `true` iff the admissible representation π belongs to the principal series.

`IsSupercuspidal(pi)`

This is `true` iff the admissible representation π is supercuspidal.

`PrincipalSeriesParameters(pi)`

Given a principal series representation π of $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$, this returns two Dirichlet characters of p-power conductor which represent the restriction to $\mathbf{Z}_p^\times \times \mathbf{Z}_p^\times$ of the character of the split torus of $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$ associated to π.

`CuspidalInducingDatum(pi)`

Given a minimal supercuspidal representation π of $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$, this returns a cuspidal inducing datum that gives rise to π.

Recall (from Section 139.1.4) that a cuspidal inducing datum (K, Ξ) consists of a subgroup K of $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$ and a representation Ξ of K that gives rise to π via induction. Importantly, Ξ factors through some finite quotient K/K_1 of K. This function returns such a representation of K/K_1. From this one can deduce the representation on K, and hence π.

139.5 Local Galois Representations

`GaloisRepresentation(pi)`

`WeilRepresentation(pi)`

`Precision` RngIntElt *Default :* 10

Given a minimal representation π of $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$, this returns the representation of the Weil group associated to π under the local Langlands correspondence. (See Section 139.1.5.)

More precisely, the objects returned are G, α, L and ρ. $L/\mathbf{Q}_p$ is a finite Galois extension which the representation factors through, G is an abstract group and α is a bijective map identifying G with $\mathrm{Gal}(\mathrm{L}/\mathbf{Q}_\mathrm{p})$, and ρ is a G-module. In this way ρ describes the Weil representation on $\mathrm{Gal}(\mathrm{L}/\mathbf{Q}_\mathrm{p})$.

(See Chapter 90 for information about group modules in Magma.)

`AdmissiblePair(pi)`

Given an ordinary minimal supercuspidal representation π of $\mathrm{GL}_2(\mathbf{Q}_\mathrm{p})$, this returns the associated admissible pair (E, χ). (See Section 139.1.5.) Two objects are returned: a quadratic field extension $E/\mathbf{Q}_p$, and a map χ which is a character of the unit group of E.

139.6 Examples

Example H139E3

We consider a newform of weight 5 and level 7, whose local representation at 7 is principal series.

```
> S := CuspidalSubspace(ModularSymbols(Gamma1(7), 5, 1));
> newforms := NewformDecomposition(S);
> Eigenform(newforms[1], 15);
q + q^2 - 15*q^4 + 49*q^7 - 31*q^8 + 81*q^9 - 206*q^11 + 49*q^14 + O(q^15)
> pi := LocalComponent(newforms[1], 7);
> pi;
Ramified Principal Series Representation of GL(2,Q_7)
> chi := CentralCharacter(pi);
> Conductor(chi);
7
> parameters := PrincipalSeriesParameters(pi);
```

These are Dirichlet characters on $\mathbf{Z}/7\mathbf{Z}$ (the trivial character and the character of order 2):

```
> Conductor(parameters[1]), Order(parameters[1]);
1 1
> Conductor(parameters[2]), Order(parameters[2]);
```

```
7 2
```

The principal series representation π is the induction up to $\mathrm{GL}_2(\mathbf{Q}_7)$ of a character of the Borel subgroup inflated from a character of the diagonal group $\mathbf{Q}_7^\times \times \mathbf{Q}_7^\times$. The restriction of this character to $\mathbf{Z}_7^\times \times \mathbf{Z}_7^\times$ gives the pair of Dirichlet characters above. We now compute the Galois representation.

```
> G, map, F, rho := WeilRepresentation(pi);
> F;
Totally ramified extension defined by the polynomial
x^6 + 7*x^5 + 21*x^4 + 35*x^3 + 35*x^2 + 21*x + 7
over 7-adic ring mod 7^10
> IsAbelian(G);
true
```

The Weil representation is simply the sum of the two characters above, considered as characters of the Galois group of $\mathbf{Q}_7$ via local class field theory.

```
> IsIrreducible(rho);
false
GModule of dimension 1 over Cyclotomic Field of order 6 and degree 2
GModule of dimension 1 over Cyclotomic Field of order 6 and degree 2
```

Example H139E4

We consider a supercuspidal representation of conductor 121, associated to a newform of weight 2 and level 121.

```
> S := CuspidalSubspace(ModularSymbols(Gamma0(121), 2, 1));
> newforms := NewformDecomposition(S);
> newforms;
[
    Modular symbols space for Gamma_0(121) of weight 2 and dimension 1 over Q,
    Modular symbols space for Gamma_0(121) of weight 2 and dimension 1 over Q,
    Modular symbols space for Gamma_0(121) of weight 2 and dimension 1 over Q,
    Modular symbols space for Gamma_0(121) of weight 2 and dimension 1 over Q,
    Modular symbols space for Gamma_0(121) of weight 2 and dimension 2 over Q
]
> Eigenform(newforms[2], 11);
q + q^2 + 2*q^3 - q^4 + q^5 + 2*q^6 - 2*q^7 - 3*q^8 + q^9 + q^10 + O(q^11)
> pi := LocalComponent(newforms[2], 11);
> pi;
Supercuspidal Representation of GL(2,Q_11)
```

This means the representation of the Weil group associated to pi is irreducible.

```
> Conductor(pi);
121
> W := CuspidalInducingDatum(pi);
> W;
```

```
GModule W of dimension 10 over Rational Field
```

W is a module over a group which is a quotient of $\mathrm{GL}_2(\mathbf{Z}_{11})$, namely $\mathrm{GL}_2(\mathbf{Z}/11\mathbf{Z})$. The representation π is induced from some extension of W to the open subgroup $\mathbf{Q}_{11}^{\times}\mathrm{GL}_2(\mathbf{Z}_{11})$.

```
> Group(W);
MatrixGroup(2, IntegerRing(11)) of order 2^4 * 3 * 5^2 * 11
Generators:
    [2 0]
    [0 1]

    [1 1]
    [0 1]

    [ 0  1]
    [10  0]
> Group(W) eq GL(2, Integers(11));
true
> G, alpha, L, rho := WeilRepresentation(pi);
```

This gives the Weil representation attached to π up to multiplication by an unramified twist. This consists of a permutation group G, an isomorphism `alpha` identifying G with the automorphisms of the local field $L/\mathbf{Q}_p$, and a 2-dimensional G-module `rho`.

```
> G;
Permutation group G acting on a set of cardinality 6
Order = 6 = 2 * 3
    Id(G)
    (1, 2, 4)(3, 6, 5)
    (1, 3)(2, 5)(4, 6)
> bool := IsIsomorphic(G, DihedralGroup(3));
> bool;
true
> L;
Totally ramified extension defined by the polynomial x^3 + 11
 over Unramified extension defined by the polynomial x^2 + 7*x + 2
 over 11-adic field mod 11^10
```

Example H139E5

We consider a supercuspidal representation of conductor 3^3, associated to a newform of weight 4 and level 27.

```
> S := CuspidalSubspace(ModularSymbols(Gamma0(27), 4, 1));
> newforms := NewformDecomposition(S);
> Eigenform(newforms[1], 14);
q + 3*q^2 + q^4 + 15*q^5 - 25*q^7 - 21*q^8 + 45*q^10 - 15*q^11 + 20*q^13 + O(q^14)
> pi:=LocalComponent(newforms[1], 3);
> pi;
Supercuspidal Representation of GL(2,Q_3)
```

```
> W:=CuspidalInducingDatum(pi);
> W;
GModule W of dimension 2 over Rational Field
> Group(W);
MatrixGroup(2, IntegerRing(9)) of order 2^2 * 3^5
Generators:
    [1 1]
    [0 1]

    [2 0]
    [0 1]

    [1 0]
    [0 2]

    [1 0]
    [3 1]
```

These matrices generate (topologically) the Iwahori subgroup of $\mathrm{GL}_2(\mathbf{Z}_3)$ consisting of matrices which are upper-triangular modulo 3. W is an irreducible two-dimensional G-module. The representation π is induced from some extension of W to the normalizer of the Iwahori in $\mathrm{GL}_2(\mathbf{Q}_3)$.

```
> E, chi:=AdmissiblePair(pi);
> E;
Totally ramified extension defined by the polynomial x^2 - 3
 over 3-adic ring mod 3^10
> E.1^2;
3
> chi(1+E.1);
zeta_3
```

Note that `chi` can only be evaluated on units of E, so that `chi(E.1)` would result in an error.

```
> G, alpha, L, rho := WeilRepresentation(pi);
> L;
Totally ramified extension defined by the polynomial x^6 - 12*x^4 - 12*x^2 - 3
 over Unramified extension defined by the polynomial x + 2
 over 3-adic field mod 3^10
```

139.7 Bibliography

[**BH06**] Colin J. Bushnell and Guy Henniart. *The local Langlands conjecture for* GL(2), volume 335 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, Berlin, 2006.

[**Car83**] Henri Carayol. Sur les Représentations ℓ-adiques attachees aux formes modulaires de Hilbert. *C. R. Acad. Sci. Paris.*, 296(15):629–632, 1983.

[**Cas73**] W. Casselman. The restriction of a representation of $\mathrm{GL}_2(\mathrm{k})$ to $\mathrm{GL}_2(\mathrm{O})$. *Mathematischen Annalen*, 206(4), 1973.

[**Gel75**] S. Gelbart. *Automorphic forms on adele groups.* Princeton University Press, 1975.

[**JL70**] H. Jacquet and R. P. Langlands. *Automorphic forms on* GL(2). Lecture Notes in Mathematics, Vol. 114. Springer-Verlag, Berlin, 1970.

[**Kut80**] Philip Kutzko. The Langlands conjecture for Gl_2 of a local field. *Ann. of Math. (2)*, 112(2):381–412, 1980.

[**Kut84**] P. C. Kutzko. The exceptional representations of Gl_2. *Compositio Math.*, 51(1):3–14, 1984.

[**LW**] David Loeffler and Jared Weinstein. On the computation of local components of a newform. preprint.

[**Tat79**] J. Tate. Number Theoretic Background. *Proc. Symp. Pure Math.*, 33, part 2:3–26, 1979.

PART XVIII
TOPOLOGY

140 SIMPLICIAL HOMOLOGY

Chapter 140
SIMPLICIAL HOMOLOGY

140.1 Introduction

This chapter presents the category of finite simplicial complexes.

We define an abstract simplicial complex K to be a subset of the power set of some set V of vertices, with the property that if $S \in K$ and $T \subset S$ then $T \in K$.

For detailed reading on simplicial complexes and their homology, we refer to [Hat02] and [Arm83].

Simplicial complexes may be defined over any `SetEnum`, however, many of the construction methods operate over `SetEnum[RngIntElt]`. The handbook refers to such simplicial complexes as *normalized*.

A simplicial complex carries the category name `SmpCpx`. Constructors and package internal functions guarantee that the closure under subsets relation is kept intact.

140.2 Simplicial Complexes

The module supports creation of simplicial complexes from lists of faces, as well as a few preprogrammed complex types. Furthermore, several standard techniques for modifying and recombining simplicial complexes are available.

`SimplicialComplex(f)`

Constructs an abstract simplicial complex with the faces in the list f. The argument should be a sequence of sets. There is no requirement on the type of the elements in the face sets, however several of the constructions require the sets to have integer entries. See `Normalization` on page 4696 for automated renumbering of the face set elements.

Example H140E1

We give a simplicial complex by listing the facets, or also redundant faces if we want to.

```
> sc := SimplicialComplex([{1,2,3},{1,2,4},{1,3,4},{2,3,4},{1,5},{2,5}]);
```

We can view the facets (faces not included in any other faces) of the complex, or by giving the `:Maximal` output flag, we can view all faces.

```
> sc;
Simplicial complex
[
    { 2, 3, 4 },
    { 1, 2, 3 },
    { 1, 2, 4 },
```

```
    { 2, 5 },
    { 1, 5 },
    { 1, 3, 4 }
]
> sc:Maximal;
Simplicial complex
{
    {},
    { 4 },
    { 2, 3, 4 },
    { 2, 3 },
    { 3, 4 },
    { 1, 3 },
    { 3 },
    { 1 },
    { 1, 4 },
    { 1, 2, 3 },
    { 2 },
    { 1, 2 },
    { 1, 2, 4 },
    { 2, 5 },
    { 1, 5 },
    { 5 },
    { 2, 4 },
    { 1, 3, 4 }
}
```

`SimplicialComplex(G)`

Constructs a 1-dimensional simplicial complex isomorphic to the graph G.

`FlagComplex(G)`

`CliqueComplex(G)`

Constructs a simplicial complex with an n-simplex for each n-clique in the graph G.

`Dimension(X)`

Returns the dimension of the highest dimensional face of the simplicial complex X. Note that there is a difference between degrees and dimensions of faces. A face is said to have degree n if there are n elements in the face as a set, and it is said to have dimension n if there are $n + 1$ elements in the face as a set.

Example H140E2

The recently defined simplicial complex has top dimension 2.

```
> Dimension(sc);
2
```

`Faces(X, d)`

Returns the faces of one degree d of the simplicial complex X. Recall that the degree of a face is the number of elements in the face as a set, and one more than the dimension of the face.

The order of faces returned for each degree is also the correspondence between faces and the basis of the corresponding module in a chain complex constructed from the simplicial complex.

If the complex does not possess any faces of the requested degree, an empty sequence is returned.

Example H140E3

We can list faces of any degree in the defined simplicial complex.

```
> Faces(sc,2);
[
    { 2, 3 },
    { 3, 4 },
    { 1, 3 },
    { 1, 4 },
    { 1, 2 },
    { 2, 5 },
    { 1, 5 },
    { 2, 4 }
]
> Faces(sc,5);
[]
```

`Facets(X)`

Returns the facets of the simplicial complex X. The facets of a simplicial complex are the faces that are not themselves subsets of another face. This is what the normal printing mode for a simplicial complex outputs.

Example H140E4

We can read the facets both by fetching the sequence of facets and by printing the complex as such.

```
> Facets(sc);
[
    { 2, 3, 4 },
    { 1, 2, 3 },
    { 1, 2, 4 },
    { 2, 5 },
    { 1, 5 },
    { 1, 3, 4 }
]
> sc;
Simplicial complex
[
    { 2, 3, 4 },
    { 1, 2, 3 },
    { 1, 2, 4 },
    { 2, 5 },
    { 1, 5 },
    { 1, 3, 4 }
]
```

`Normalization(X)`

`Normalization(~X)`

Relabels the points (elements of the face sets) of the simplicial complex X using $1, 2, 3, \ldots$. This ensures that the simplicial complex is built on subsets of the integers, which is a prerequisite for several functions handling simplicial complexes. We call a simplicial complex on subsets of the integers *normalized* in this handbook.

Note the two calling signatures of this function. The first signature copies the simplicial complex and returns a new complex with the desired modification, whereas the second signature modifies the complex in place.

`Shift(X, n)`

`Shift(~X, n)`

Shifts all integer points in the normalized simplicial complex X by an offset given by n.

Note the two calling signatures of this function. The first signature copies the simplicial complex and returns a new complex with the desired modification, whereas the second signature modifies the complex in place.

Example H140E5

We define a simplicial complex using string labels, normalize it and then shift the labels.

```
> cpx := Boundary(SimplicialComplex([{"a","b","c"}]));
> cpx;
Simplicial complex
[
    { c, a },
    { c, b },
    { b, a }
]
> Normalization(~cpx);
> cpx;
Simplicial complex
[
    { 1, 3 },
    { 2, 3 },
    { 1, 2 }
]
> Shift(~cpx,-2);
> cpx;
Simplicial complex
[
    { -1, 1 },
    { 0, 1 },
    { -1, 0 }
]
```

`Boundary(X)`

Returns the simplicial complex generated by all simplexes that lie in the boundary of the simplicial complex X. This can be used to easily acquire a simplicial complex representing the n-sphere.

Example H140E6

We can determine the boundary of our previously defined simplicial complex, or construct a 4-sphere.

```
> Boundary(sc);
Simplicial complex
[
    { 2, 3 },
    { 3, 4 },
    { 1, 3 },
    { 1, 4 },
    { 1, 2 },
```

```
    { 5 },
    { 2, 4 }
]
> sph4 := Boundary(SimplicialComplex([{1,2,3,4,5,6}]));
> sph4;
Simplicial complex
[
    { 1, 2, 3, 5, 6 },
    { 2, 3, 4, 5, 6 },
    { 1, 3, 4, 5, 6 },
    { 1, 2, 3, 4, 5 },
    { 1, 2, 4, 5, 6 },
    { 1, 2, 3, 4, 6 }
]
```

`S + T`

Returns the topological sum, or disjoint union, of two simplicial complexes. Requires both complexes to be normalized using Normalization on page 4696.

Example H140E7

We can construct a disjoint union of two circles.

```
> sph2 := Boundary(SimplicialComplex([{1,2,3}]));
> sph2 + sph2;
Simplicial complex
[
    { 4, 6 },
    { 2, 3 },
    { 4, 5 },
    { 5, 6 },
    { 1, 3 },
    { 1, 2 }
]
```

`S eq T`

Compares two simplicial complexes. Will not try to find an isomorphism, rather does the comparison by ensuring that the points are in the same universe and then doing a set comparison on the set of faces.

Example H140E8

Shifting a complex, for instance, will break equality, since the labels differ.

```
> sc eq Shift(sc,2);
false
> sc eq Shift(Shift(sc,2),-2);
true
```

Furthermore, isomorphic complexes with different labels will not be equal.

```
> circ1 := Boundary(SimplicialComplex([{1,2,3}]));
> circ2 := Boundary(SimplicialComplex([{"a","b","c"}]));
> circ1 eq circ2;
false
```

`Product(S,T)`

Returns the cartesian product of two simplicial complexes. This will work on any complexes, since the new points will be pairs of points from the component complexes.

Example H140E9

Using the two different circles from the last example, we can now construct a torus.

```
> Product(circ1,circ2);
Simplicial complex
[
    { <3, b>, <1, b>, <3, c> },
    { <1, c>, <3, c>, <1, a> },
    { <2, a>, <3, b>, <3, a> },
    { <3, b>, <1, b>, <1, a> },
    { <1, b>, <2, b>, <2, c> },
    { <2, a>, <3, c>, <3, a> },
    { <1, c>, <1, b>, <3, c> },
    { <3, c>, <2, b>, <2, c> },
    { <1, c>, <1, b>, <2, c> },
    { <3, c>, <3, a>, <1, a> },
    { <2, a>, <3, c>, <2, c> },
    { <3, b>, <3, a>, <1, a> },
    { <2, a>, <1, a>, <2, b> },
    { <2, a>, <1, a>, <2, c> },
    { <1, c>, <1, a>, <2, c> },
    { <3, b>, <3, c>, <2, b> },
    { <1, b>, <1, a>, <2, b> },
    { <2, a>, <3, b>, <2, b> }
```

```
]
```

If we want something less unwieldy – especially after repeated cartesian products – we can always normalize the resulting complexes.

```
> line := SimplicialComplex([{1,2}]);
> square := Product(line,line);
> cube1 := Product(square,line);
> cube2 := Product(line,square);
> cube1 eq cube2;
false
> Normalization(cube1) eq Normalization(cube2);
false
> cube1;
Simplicial complex
[
    { <<1, 2>, 1>, <<1, 2>, 2>, <<1, 1>, 1>, <<2, 2>, 2> },
    { <<2, 1>, 1>, <<2, 2>, 1>, <<1, 1>, 1>, <<2, 2>, 2> },
    { <<1, 2>, 2>, <<1, 1>, 1>, <<1, 1>, 2>, <<2, 2>, 2> },
    { <<1, 2>, 1>, <<2, 2>, 1>, <<1, 1>, 1>, <<2, 2>, 2> },
    { <<1, 1>, 1>, <<1, 1>, 2>, <<2, 1>, 2>, <<2, 2>, 2> },
    { <<2, 1>, 1>, <<1, 1>, 1>, <<2, 1>, 2>, <<2, 2>, 2> }
]
> Normalization(cube1);
Simplicial complex
[
    { 3, 4, 5, 8 },
    { 4, 5, 6, 8 },
    { 1, 3, 4, 5 },
    { 2, 4, 5, 7 },
    { 1, 2, 4, 5 },
    { 4, 5, 6, 7 }
]
```

`Join(S,T)`

`S * T`

Constructs the join of two simplicial complexes. The join of two simplicial complexes is defined as the complex generated by faces on the form $\{x_1, \ldots, x_r, y_1, \ldots, y_s\}$ for $\{x_1, \ldots, x_r\} \in S$ and $\{y_1, \ldots, y_s\} \in T$. Requires both simplicial complexes to be normalized.

Example H140E10

The join of two edges is a 3-simplex.

```
> SimplicialComplex([{1,2}]) * SimplicialComplex([{1,2}]);
Simplicial complex
[
    { 1, 2, 3, 4 }
]
```

`AddSimplex(X, s)`
`AddSimplex(~X, s)`
`AddSimplex(X, s)`
`Addsimplex(~X, s)`

Adds either a single simplex s expressed as a set or a sequence of simplexes to a simplicial complex. The functional versions of the function call return a new complex with the added simplices included, and the procedural change the complex in place.

`Prune(X, f)`
`Prune(~X, f)`

Removes a face f and all faces containing f from a simplicial complex X. If f is not a face of X, then X is returned unchanged.

Note the two calling signatures of this function. The first signature copies the simplicial complex and returns a new complex with the desired modification, whereas the second signature modifies the complex in place.

`Glue(X, e)`
`Glue(~X, e)`

In the simplicial complex X, identify all points identified by pairs in the sequence e.

The sequence e should be a sequence of pairs of elements of the face sets of X. The identification will eliminate the first component of each pair, replacing it with the second component wherever it occurs in X.

Note the two calling signatures of this function. The first signature copies the simplicial complex and returns a new complex with the desired modification, whereas the second signature modifies the complex in place.

Example H140E11

We can, for instance, construct the connected sum of two tori using `Product`, `Prune`, the topological sum and `Glue`.

```
> circ := SimplicialComplex([{1,2},{2,3},{3,1}]);
> torus := Product(circ,circ);
> Normalization(~torus);
> torus;
Simplicial complex
[
    { 2, 3, 6 },
    { 1, 2, 9 },
    { 2, 6, 8 },
    { 4, 7, 8 },
    { 3, 5, 9 },
    { 4, 6, 7 },
    { 1, 8, 9 },
    { 1, 4, 8 },
    { 6, 8, 9 },
    { 2, 7, 8 },
    { 2, 3, 9 },
    { 5, 6, 9 },
    { 1, 3, 5 },
    { 1, 4, 6 },
    { 3, 6, 7 },
    { 1, 5, 6 },
    { 1, 2, 7 },
    { 1, 3, 7 }
]
> oneholetorus := Prune(torus,{1,2,7});
> twoholetorus := Prune(Prune(torus,{1,2,7}),{6,8,9});
> threetori := oneholetorus + twoholetorus + oneholetorus;
> threetorus := Glue(threetori,[<1,10>,<2,11>,<7,16>,<15,19>,<17,20>,<18,25>]);
```

Note that we find the glue data by considering that the maximal point in the original torus had number 9, so for the second added torus, all points will be shifted by 9 and in the third all points will be shifted by 18. Thus, the excised facets $\{1, 2, 7\}$ and $\{6, 8, 9\}$ turn into $\{10, 11, 16\}$ and $\{15, 17, 18\}$, and the excised facet $\{1, 2, 7\}$ in the third torus turns into $\{19, 20, 25\}$.

`BarycentricSubdivision(X)`

Constructs the barycentric subdivision of the simplicial complex X. Abstractly, this is a simplicial complex whose faces are chains $X_1 \subset \ldots \subset X_n$ of faces from X. The new complex has more faces but the same homotopy type as the old complex.

`Skeleton(X, q)`

Returns the q-skeleton of the simplicial complex X. This is the complex consisting of all faces of X of dimension at most q.

`UnderlyingGraph(X)`

Constructs a graph isomorphic, as a graph, to the 1-skeleton of the simplicial complex X.

`Cone(X)`

Constructs a cone over the simplicial complex X. The cone is generated by all faces of the complex, with an additional vertex included into each face. Any cone is acyclic, in other words all homology groups vanish. Requires a normalized simplicial complex.

`Suspension(X)`

Constructs the suspension, or double cone, over the normalized simplicial complex X. The suspension has the added property that all the homology groups occur in it, shifted up by one dimension.

Example H140E12

For computation of the relevant homology, and a demonstration of the stated facts about the cone and suspension operations, please refer to examples in the Section on homology computation.

```
> circ := Boundary(SimplicialComplex([{1,2,3}]));
> Cone(circ);
Simplicial complex
[
    { 1, 3, 4 },
    { 2, 3, 4 },
    { 1, 2, 4 }
]
> Suspension(circ);
Simplicial complex
[
    { 1, 3, 5 },
    { 1, 3, 4 },
    { 1, 2, 5 },
    { 2, 3, 4 },
    { 1, 2, 4 },
    { 2, 3, 5 }
]
```

140.2.1 Standard Topological Objects

`Simplex(n)`

Returns the n-dimensional simplex as a simplicial complex.

`Sphere(n)`

Returns a simplicial complex triangulating the n-sphere.

`KleinBottle()`

Returns a triangulation of the Klein bottle as an abstract simplicial complex.

`Torus()`

Returns a triangulation of a torus as an abstract simplicial complex.

`Cylinder()`

Returns a triangulation of a cylinder as an abstract simplicial complex.

`MoebiusStrip()`

Returns a triangulation of the Moebius strip as an abstract simplicial complex.

`LensSpace(p)`

Constructs the lens space $L(p, 1)$. This has a 1-dimensional homology generator of order p.

`SimplicialProjectivePlane()`

Constructs a triangulation of the projective plane.

Examples for the usage of `LensSpace` and `SimplicialProjectivePlane` will be given in Section 140.3 on homology computation.

140.3 Homology Computation

The code computes exclusively reduced homology of the given simplicial complexes. If you want the non-reduced homology, just add a single free rank to dimension 0 and let it be generated by any single point in the complex.

`Homology(X)`

`Homology(~X)`

`Homology(X,A)`

`Homology(~X, A)`

Calculates the reduced homology of a simplicial complex `X` with coefficients in the ring A. The procedural form of this command caches the results of the calculation in the simplicial complex object. If no ring is given, then the function defaults to integer coefficients.

Example H140E13

The resulting modules are stored in falling dimension, always including the dimension -1 vanishing homology module at the very end.

```
> circ := Boundary(SimplicialComplex([{1,2,3}]));
> Homology(circ,Integers());
[
    Full Quotient RSpace of degree 1 over Integer Ring
    Column moduli:
    [ 0 ],
    Full Quotient RSpace of degree 0 over Integer Ring
    Column moduli:
    [ ],
    Full Quotient RSpace of degree 0 over Integer Ring
    Column moduli:
    [ ]
]
[
    Mapping from: RSpace of degree 3, dimension 1 over Integer Ring to Full
    Quotient RSpace of degree 1 over Integer Ring
    Column moduli:
    [ 0 ],
    Mapping from: RSpace of degree 3, dimension 2 over Integer Ring to Full
    Quotient RSpace of degree 0 over Integer Ring
    Column moduli:
    [ ],
    Mapping from: Full RSpace of degree 1 over Integer Ring to Full Quotient
    RSpace of degree 0 over Integer Ring
    Column moduli:
    [ ]
]
```

```
> lens3 := LensSpace(3);
> Homology(~lens3,Integers());
> Homology(lens3,Integers())[3];
Full Quotient RSpace of degree 1 over Integer Ring
Column moduli:
[ 3 ]
```

`HomologyGroup(X, q)`

`HomologyGroup(X, q, A)`

Calculates and returns the qth homology group of X with coefficients in A. If no ring is given, the function defaults to integer coefficients. If the homology is cached in X, the cached results are returned. This function will not compute the entire homology in order to return one homology group.

`BettiNumber(X,q)`

`BettiNumber(X,q,A)`

Returns the qth Betti number, computed as the free rank of the q-dimensional homology group, with coefficients in A. If no ring is given, then the function will default to integer coefficients.

Note that the Betti number computations compensate for the homology computations being reduced. Thus, `BettiNumber(X,0) eq Rank(HomologyGroup(X,0)) + 1`.

`TorsionCoefficients(X, q)`

`TorsionCoefficients(X, q, A)`

Returns the torsion coefficients of the qth homology group of X with coefficients in A. If no ring is given, then the function will default to integer coefficients.

`EulerCharacteristic(X)`

Computes the Euler characteristic of the complex. If homology is cached, this is used for computation, and else the characteristic is computed using the ranks of the chain groups.

`BoundaryMatrix(X, q, A)`

Returns the qth boundary matrix of the corresponding chain complex to the simplicial complex X with coefficients in A.

`ChainComplex(X, A)`

Constructs a reduced chain complex of free A-modules corresponding to the abstract simplicial complex X.

Note that the produced complex includes one extra rank 1 module on each end, with the zero map leading to it, to simulate the maps to and from the zero module that would end a chain complex constructed from a simplicial complex in ordinary cases.

Example H140E14

```
> ChainComplex(SimplicialComplex([{1}]),Integers());
Chain complex with terms of degree 2 down to -1
Dimensions of terms: 1 1 1 1
> BoundaryMaps(ChainComplex(SimplicialComplex([{1}]),Integers()));
[*
    [0],
    [-1],
    [0]
*]
> ChainComplex(SimplicialComplex([{1,2,3}]),GF(3));
Chain complex with terms of degree 4 down to -1
Dimensions of terms: 1 1 3 3 1 1
> BoundaryMaps(ChainComplex(SimplicialComplex([{1,2,3}]),GF(3)));
[*
    [0],
    [1 2 2],
    [1 0 2]
    [0 1 2]
    [1 2 0],
    [2]
    [2]
    [2],
    [0]
*]
```

`HomologyGenerators(X)`

`HomologyGenerators(X, A)`

`HomologyGenerators(H, M, X)`

Prints generators of the homology groups of the simplicial complex X with coefficients in A together with their order, in order of dimension. The latter calling form expects H, M to be the result from `H,M := ChainComplex(A,smp);`. This function will recalculate homology each time unless the homology is already cached in the simplicial complex using `Homology(A, smp)`.

If no ring is given, the function defaults to integer coefficients.

Example H140E15

This function gives a condensed form of the actual bases of the homology groups, as well as mappings back to an actual chain representative for each homology class.

```
> HomologyGenerators(threetorus,Integers());
*** dimension 2 ***
inf: { 21, 23, 27 } - { 4, 6, 16 } - { 20, 24, 26 } + { 20, 25, 26 } -
  { 14, 19, 25 } + { 12, 14, 25 } + { 3, 6, 11 } - { 10, 14, 19 } +
  { 19, 20, 27 } - { 3, 9, 11 } - { 10, 13, 20 } + { 10, 13, 19 } -
  { 10, 12, 14 } - { 12, 16, 19 } - { 11, 19, 20 } - { 13, 16, 20 } +
  { 21, 24, 25 } + { 24, 26, 27 } - { 20, 21, 24 } - { 19, 22, 26 } -
  { 3, 10, 16 } - { 4, 6, 10 } - { 22, 25, 26 } + { 11, 16, 20 } +
  { 4, 8, 16 } - { 5, 6, 9 } + { 4, 8, 10 } - { 11, 12, 19 } - { 3, 5, 9 } +
  { 11, 12, 25 } + { 8, 9, 10 } + { 20, 21, 27 } + { 3, 6, 16 } +
  { 10, 11, 25 } + { 19, 22, 24 } - { 9, 10, 11 } + { 3, 5, 10 } +
  { 10, 20, 25 } - { 8, 11, 16 } - { 19, 26, 27 } - { 19, 23, 24 } +
  { 10, 12, 16 } - { 19, 21, 23 } - { 6, 8, 9 } + { 13, 16, 19 } +
  { 19, 21, 25 } + { 23, 24, 27 } - { 22, 24, 25 } + { 5, 6, 10 } +
  { 6, 8, 11 }
*** dimension 1 ***
inf: -1*{ 10, 13 } + { 3, 5 } - { 8, 9 } - { 8, 16 } - { 19, 22 } +
  { 20, 26 } + { 9, 11 } - { 3, 16 } + { 5, 10 } + { 22, 26 } + { 11, 20 } +
  2*{ 10, 20 } + { 13, 19 } - 2*{ 10, 11 }
inf: { 3, 9 } - { 3, 5 } - { 9, 11 } - { 5, 10 } + { 10, 11 }
inf: -1*{ 26, 27 } - { 11, 19 } + 2*{ 19, 22 } - { 20, 26 } -
  2*{ 22,  26 } + { 11, 20 } + { 19, 21 } - { 21, 27 }
inf: { 10, 13 } + { 11, 19 } - { 11, 20 } - { 10, 20 } - { 13, 19 }
inf: -1*{ 11, 19 } + { 19, 22 } - { 20, 26 } - { 22, 26 } + { 11, 20 }
inf: { 11, 20 } + { 10, 20 } - { 10, 11 }
```

The six found generators are the generators of each of the contained torus homology groups. Notice that each generator is printed out with a prefix. This gives the order of the generator - so that for instance torsion elements of homology may be recognized. Thus, we see with the projective plane:

```
> HomologyGenerators(SimplicialProjectivePlane(),Integers());
*** dimension 1 ***
  2: { 3, 6 } + { 2, 3 } - { 2, 6 }
```

We can further take this opportunity to verify the claims about `Cone` and `Suspension` with regard to the homology.

```
> HomologyGenerators(Cone(SimplicialProjectivePlane()),Integers());
Complex is acyclic.
> HomologyGenerators(Suspension(SimplicialProjectivePlane()),Integers());
*** dimension 2 ***
  2: { 1, 5, 7 } - { 1, 4, 7 } + { 1, 2, 5 } + { 1, 5, 8 } - { 2, 3, 8 } -
  { 3, 5, 7 } - { 1, 2, 8 } + { 5, 6, 8 } + { 1, 4, 5 } - { 4, 6, 8 } +
  { 3, 4, 7 } + { 2, 3, 5 } - { 4, 5, 6 } - { 3, 4, 8 }
```

140.4 Bibliography

[**Arm83**] Mark Anthony Armstrong. *Basic topology.* Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1983. Corrected reprint of the 1979 original.

[**Hat02**] Allen Hatcher. *Algebraic topology.* Cambridge University Press, Cambridge, 2002.

PART XIX
GEOMETRY

141 FINITE PLANES

Chapter 141
FINITE PLANES

141.1 Introduction

In this chapter we will present the categories of finite projective and affine planes.

The category names for projective and affine planes are `PlaneProj` and `PlaneAff` respectively. Within each of these categories we have what we will call *classical* planes — those which are defined by a vector space of dimension 2 (for affine planes) or 3 (for projective planes).

Some functions documented here apply to all types of planes, others are specific to projective, affine or classical planes. It should be clear which is the case for each entry.

141.1.1 Planes in Magma

A point of a plane is considered to be a special object, and so points are given their own special type, `PlanePt`, in MAGMA. This allows the points of a plane to be defined over any type of MAGMA object, and also improves the efficiency of the code.

A special structure called the *point–set* acts as the parent structure for points. A point is created by coercing an appropriate MAGMA object into the point–set. It is also possible the get the i-th point, or a random point, from the point–set.

Similarly, lines of a plane have a special type `PlaneLn`, and the *line–set* acts as their parent structure. Lines can be created by coercing a suitable object into the line–set, or by asking for the i-th line, or a random line, from the line–set.

141.2 Construction of a Plane

All functions which create a plane return three values:

(i) the plane itself;

(ii) the point–set of the plane;

(iii) the line–set of the plane.

These "sets" ((ii) and (iii)) are used as the parent structures for points and lines respectively, and are explained more fully in the next section.

`FiniteProjectivePlane< v | X : parameters >`

`FiniteProjectivePlane< V | X : parameters >`

`Check` BOOLELT *Default :* `true`

Construct the projective plane P having as point set the indexed set V (or $\{@1, 2, \ldots . v@\}$ if an integer v is given), and as line set $L = \{L_1, L_2, \ldots, L_b\}$ given by the list X. The value of X must be either:

(a) A list of subsets of the set V.

(b) A sequence, set or indexed set of subsets of V.

(c) A list of lines of an existing plane.

(d) A sequence, set or indexed set of lines of an existing plane.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be defined over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the plane P.

(g) A set of codewords of a linear code with length v. The line set of P is taken to be the set of supports of the codewords.

The optional boolean argument `Check` indicates whether or not to check that the given data satisfies the projective plane axioms.

```
FiniteProjectivePlane(W)
FiniteProjectivePlane(F)
FiniteProjectivePlane(q)
```

Given a 3–dimensional vector space W defined over the field $F = \mathbf{F}_q$, construct the classical projective plane defined by the one–dimensional and two–dimensional subspaces of W.

```
FiniteAffinePlane< v | X : parameters >
FiniteAffinePlane< V | X : parameters >
```

`Check` BoolElt *Default :* `true`

Construct the affine plane P having as point set the indexed set V (or $\{@1, 2, \ldots . v@\}$ if an integer v is given), and as line set $L = \{L_1, L_2, \ldots, L_b\}$ given by the list X. The value of X must be either:

(a) A list of subsets of the set V.

(b) A sequence, set or indexed set of subsets of V.

(c) A list of lines of an existing plane.

(d) A sequence, set or indexed set of lines of an existing plane.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be defined over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the plane P.

(g) A set of codewords of a linear code with length v. The line set of P is taken to be the set of supports of the codewords.

The optional boolean argument `Check` indicates whether or not to check that the given data satisfies the affine plane axioms.

`FiniteAffinePlane(W)`

`FiniteAffinePlane(F)`

`FiniteAffinePlane(q)`

Given a 2–dimensional vector space W defined over the field $F = \mathbf{F}_q$, construct the classical affine plane defined by the cosets of the subspaces of W.

Example H141E1

The classical projective plane of order 3 can be constructed by the following statement:

```
> P, V, L := FiniteProjectivePlane(3);
> P;
Projective Plane PG(2, 3)
> V;
Point-set of Projective Plane PG(2, 3)
> L;
Line-set of Projective Plane PG(2, 3)
```

A non-classical affine plane of order 2 can be constructed in the following way:

```
> A := FiniteAffinePlane< 4 | Setseq(Subsets({1, 2, 3, 4}, 2)) >;
> A: Maximal;
Affine Plane of order 2
Points: {@ 1, 2, 3, 4 @}
Lines:
    {1, 3},
    {1, 4},
    {2, 4},
    {2, 3},
    {1, 2},
    {3, 4}
```

To demonstrate the use of the `Check` argument, we recreate the classical projective plane of order 16 with `Check := true` (the default) and `Check := false`.

```
> P, V, L := FiniteProjectivePlane(16);
> time P2 := FiniteProjectivePlane<
>     Points(P) | {Set(l): l in L} : Check := true >;
Time: 10.769
> time P2 := FiniteProjectivePlane<
>     Points(P) | {Set(l): l in L} : Check := false >;
Time: 0.030
```

141.3 The Point-Set and Line-Set of a Plane

141.3.1 Introduction

An affine or projective plane in MAGMA consists of three objects: the plane P itself, the *point–set* V of P, and the *line–set* L of P.

Although called the point–set and line–set, V and L are not actual MAGMA sets. They simply act as the parent structures for the points and lines (respectively) of the plane P, enabling easy creation of these objects via the `!` and `.` operators.

The point–set V belongs to the MAGMA category `PlanePtSet`, and the line–set L to the category `PlaneLnSet`.

In this section, the functions used to create point–sets, line–sets and the points and lines themselves are described.

141.3.2 Creating Point-Sets and Line-Sets

As mentioned above, the point–set and line–set are returned as the second and third arguments of any function which creates a plane. They can also be created via the following two functions.

`PointSet(P)`

Given a plane P, return the point–set V of P.

`LineSet(P)`

Given a plane P, return the line–set L of P.

141.3.3 Using the Point-Set and Line-Set to Create Points and Lines

For efficiency and clarity, the points and lines of a plane are given special types in MAGMA. The category names for points and lines are `PlanePt` and `PlaneLn` respectively. They can be created in the following ways.

`V . i`

Given the point–set V of a plane P and an integer i, return the i-th point of P.

`V ! [a, b, c]`

Given the point–set V of a classical projective plane $P = PG_2(K)$, and elements a, b, c of the finite field K, create the projective point $(a : b : c)$ in the plane P.

`V ! [a, b]`

Given the point–set V of a classical affine plane $P = AG_2(K)$, and elements a, b of the finite field K, create the point (a, b) in the plane P.

`V ! x`

Given the point–set V of a plane P, return the point of P corresponding to the element x, which should be coercible into the underlying point set for P. (In the case of classical planes, x should be coercible to a vector.)

`Representative(V)`
`Rep(V)`

Given the point–set V of a plane P, return a representative point of P.

`Random(V)`

Given the point–set V of a plane P, return a random point of P.

`L . i`

Given the line–set L of a plane P and an integer i, return the i-th line of P.

`L ! [a, b, c]`

Given the line set L of a classical plane P defined over a finite field K, and elements a, b, c of K, create the line $\langle a : b : c \rangle$ (i.e. the line given by the equation $ax+by+cz = 0$ if P is projective, or $ax + by + c = 0$ if P is affine).

`L ! [m, b]`

Given the line set L of a classical affine plane $P = AG_2(K)$, and elements m, b of the finite field K, create the affine line $y = mx + b$ in P.

`L ! S`

Given the line–set L of a plane P and a set or sequence S of collinear points of P, return the line containing the points of S.

`L ! l`

Given the line–set L of a plane P and a line l of a (possibly) different plane (generally a subplane of P), return the line of P corresponding to l.

`Representative(L)`
`Rep(L)`

Given the line–set L of a plane P, return a representative line of P.

`Random(L)`

Given the line–set L of a plane P, return a random line of P.

Example H141E2

The following example shows how points and lines of a plane can be created. First we study a classical projective plane.

```
> P, V, L := FiniteProjectivePlane(5);
> V;
Point-set of Projective Plane PG(2, 5)
> L;
Line-set of Projective Plane PG(2, 5)
```

Create the third point of P:

```
> V.3;
( 0 : 0 : 1 )
```

Create the point $(1 : 2 : 3)$ of P:

```
> V![1, 2, 3];
( 1 : 2 : 3 )
```

Choose a random point of P:

```
> Random(V);
( 1 : 0 : 0 )
> Random(V);
( 0 : 0 : 1 )
```

Create the sixth line of P:

```
> L.6;
< 1 : 1 : 3 >
```

Create the line of P given by the equation $4x + 3y + 2z = 0$:

```
> L![4, 3, 2];
< 1 : 2 : 3 >
```

Create the line of P containing the points $(0 : 0 : 1)$ and $(0 : 1 : 0)$:

```
> L![ V | [0, 0, 1], [0, 1, 0] ];
< 1 : 0 : 0 >
```

Get a representative from the line-set of P, and a random line:

```
> Rep(L);
< 1 : 0 : 0 >
> Random(L);
< 1 : 2 : 4 >
```

Now we look at a non-classical plane.

```
> V := {2, 4, 6, 8};
> A, P, L := FiniteAffinePlane< SetToIndexedSet(V) | Setseq(Subsets(V, 2)) >;
> A: Maximal;
Affine Plane of order 2
```

```
Points: {@ 2, 4, 6, 8 @}
Lines:
    {6, 8},
    {2, 6},
    {2, 8},
    {2, 4},
    {4, 6},
    {4, 8}
> P;
Point-set of Affine Plane of order 2
> L;
Line-set of Affine Plane of order 2
```

Get the third point of A:

```
> P.3;
6
```

Create the point of A given by the integer 4:

```
> P!4;
4
```

Get a representative from the point–set of A:

```
> Rep(P);
2
```

Get the third line of A:

```
> L.3;
{2, 8}
```

Create the line of A containing the integers 2 and 6:

```
> L![2, 6];
{2, 6}
```

Choose a random line from A:

```
> Random(L);
{6, 8}
```

141.3.4 Retrieving the Plane from Points, Lines, Point-Sets and Line-Sets

The `ParentPlane` function allows you to access the plane to which a point, line, point–set or line–set belongs.

`ParentPlane(V)`

The plane P for which V is the point–set.

`ParentPlane(L)`

The plane P for which L is the line–set.

`ParentPlane(p)`

The plane P for which p is a point.

`ParentPlane(l)`

The plane P for which l is a line.

141.4 The Set of Points and Set of Lines

It may sometimes be desirable to have an explicitly enumerated set of the points or lines of a plane, as opposed to the point–set and line–set, which aren't true MAGMA sets. The following functions have been provided for this purpose.

`Points(P)`

An indexed set E whose elements are the points of the plane P. Note that this creates a standard indexed set and not the point–set of P, in contrast to the function `PointSet`.

`Lines(P)`

An indexed set containing the lines of the plane P. In contrast to the function `LineSet`, this function returns the collection of lines of P in the form of a standard indexed set.

141.5 The Defining Points of a Plane

Points of a plane have their own special type in MAGMA (`PlanePt`). However it may sometimes be necessary to return to the objects used to define the points. These are the elements of the indexed set originally used to create the plane, or, in the case of a classical plane, the vector space elements which define the points of the plane.

The functions listed in this section provide the means to do this.

`Support(P)`

An indexed set E which is the underlying point set of the plane P (i.e. the elements of the set have their "real" types; they are no longer a "PlanePt" of P).

`Support(l)`

The set of underlying points contained in the line l of a plane P (i.e. the elements of the set have their "real" types; they are no longer a "PlanePt" of P).

`Support(P, p)`

`Support(p)`

The MAGMA object corresponding to the point p of a plane P.

Example H141E3

The following code shows the difference between the `Points` and `Support` functions.

```
> A := FiniteAffinePlane< {@ 3, 4, 5, 6 @} | {3, 4}, {3, 5}, {3, 6},
>                                            {4, 5}, {4, 6}, {5, 6} >;
> Pts := Points(A);
> Supp := Support(A);
> Pts, Supp;
{@ 3, 4, 5, 6 @}
{@ 3, 4, 5, 6 @}
```

These sets look the same, but the elements have different types:

```
> Universe(Pts);
Point-set of Affine Plane of order 2
> Universe(Supp);
Integer Ring
```

The classical plane case is slightly different. Here the support is a set of vectors.

```
> P, V, L := FiniteProjectivePlane(2);
> Points(P);
{@ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 1 : 0 ),
   ( 0 : 1 : 1 ), ( 1 : 1 : 1 ), ( 1 : 0 : 1 ) @}
> Support(P);
{@
    (1 0 0),
    (0 1 0),
```

```
    (0 0 1),
    (1 1 0),
    (0 1 1),
    (1 1 1),
    (1 0 1)
@}
> Universe(Points(P));
Point-set of Projective Plane PG(2, 2)
> Universe(Support(P));
Full Vector space of degree 3 over GF(2)
> l := Random(L);
> l;
< 0 : 0 : 1 >
> Support(l);
{
    (1 0 0),
    (0 1 0),
    (1 1 0)
}
```

141.6 Subplanes

The `sub` constructor allows subplanes of a projective or affine plane to be created. For classical planes, the `SubfieldSubplane` function is also provided.

`sub< P | L >`

Given a plane P, construct the subplane of P generated by the points specified by L, where L is a list of one or more items of the following types:

(a) A point of P;

(b) A set or sequence of points of P;

(c) A subplane of P;

(d) A set or sequence of subplanes of P.

The set S of points defined by the list L must include a quadrangle if P is a projective plane and three non-collinear points if P is an affine plane. The function returns the smallest subplane of P containing S.

`SubfieldSubplane(P, F)`

The plane obtained from the classical plane P by taking only those points of P which have all coordinates lying in F, where F must be a subfield of `Field(P)`.

Example H141E4

In the plane $PG_2(4)$, the points $(1:0:0), (0:1:0), (0:0:1)$ and $(1:w:1)$, where w is a primitive element of $\mathbf{F}_4$, form a quadrangle. We form the subplane of $PG_2(4)$ generated by this quadrangle.

```
> K<w> := GF(4);
> P, V, L := FiniteProjectivePlane(K);
> S := sub< P | [ V | [1, 0, 0], [0, 1, 0], [0, 0, 1], [1, w, 1] ] >;
> S: Maximal;
Projective Plane of order 2
Points: {@ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : w : 0 ),
( 1 : 0 : 1 ), ( 1 : w : 1 ), ( 0 : 1 : w^2 ) @}
Lines:
    {( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 0 : 1 : w^2 )},
    {( 1 : 0 : 0 ), ( 0 : 0 : 1 ), ( 1 : 0 : 1 )},
    {( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 1 : w : 0 )},
    {( 1 : 0 : 0 ), ( 1 : w : 1 ), ( 0 : 1 : w^2 )},
    {( 0 : 1 : 0 ), ( 1 : 0 : 1 ), ( 1 : w : 1 )},
    {( 0 : 0 : 1 ), ( 1 : w : 0 ), ( 1 : w : 1 )},
    {( 1 : w : 0 ), ( 1 : 0 : 1 ), ( 0 : 1 : w^2 )}
```

We next form the subplane of $AG_2(4)$ over $\mathbf{F}_2$.

```
> A := FiniteAffinePlane(4);
> S := SubfieldSubplane(A, GF(2));
> S: Maximal;
Affine Plane AG(2, 2)
> S subset A;
true
```

141.7 Structures Associated with a Plane

There are a number of structures naturally associated with a plane. This section lists some functions for accessing or creating them.

`VectorSpace(P)`

The vector space underlying the classical plane P.

`Field(P)`

The field over which the classical plane P is defined.

`IncidenceMatrix(P)`

The incidence matrix of the plane P.

`Dual(P)`

The dual of the projective plane P.

Example H141E5

The functions in this section are illustrated by the following example.

```
> A := FiniteAffinePlane(4);
> VectorSpace(A);
Full Vector space of degree 2 over GF(2^2)
> Field(A);
Finite field of size 2^2
>
> P := FiniteProjectivePlane< 7 | {1, 3, 5}, {1, 2, 7}, {1, 4, 6}, {2, 3, 6},
>                              {2, 4, 5}, {3, 4, 7}, {5, 6, 7} >;
> IP := IncidenceMatrix(P);
> IP;
[1 1 1 0 0 0 0]
[0 1 0 1 1 0 0]
[1 0 0 1 0 1 0]
[0 0 1 0 1 1 0]
[1 0 0 0 1 0 1]
[0 0 1 1 0 0 1]
[0 1 0 0 0 1 1]
> D := Dual(P);
> D;
Projective Plane of order 2
> IncidenceMatrix(D) eq Transpose(IP);
true
```

141.8 Numerical Invariants of a Plane

`Order(P)`

The order of the plane P.

`NumberOfPoints(P)`

`#V`

The cardinality v of the point–set V of the plane P.

`NumberOfLines(P)`

`#L`

The cardinality v of the line–set L of the plane P.

`pRank(P)`

The p-rank of the plane P of order p^t.

`pRank(P, p)`

The p-rank of the plane P.

Example H141E6

We demonstrate some of the above functions with the plane $PG_2(8)$.

```
> P := FiniteProjectivePlane(8);
> NumberOfLines(P);
73
> Order(P);
8
> pRank(P);
28
> pRank(P, 2);
28
> pRank(P, 3);
72
> pRank(P, 5);
73
```

141.9 Properties of Planes

`IsDesarguesian(P)`

Returns `true` if and only if the plane P is a desarguesian plane.

`IsSelfDual(P)`

Returns `true` if and only if the projective plane P is self–dual.

141.10 Identity and Isomorphism

Two planes are considered equal if their point sets are equal and they have the same lines.

`P eq Q`

Return `true` if the planes P and Q are equal, otherwise `false`.

`P ne Q`

Return `true` if the planes P and Q are not equal, otherwise `false`.

`IsIsomorphic(P, Q:` *parameters*`)`

`AutomorphismGroups` MonStgElt *Default :* "*None*"

Return `true` if the planes P and Q are isomorphic, otherwise `false`. If the planes are isomorphic, an isomorphism f from P onto Q is also returned. The function first computes none, one, or both of the automorphism groups of the left and right planes. This may assist the isomorphism testing. The parameter `AutomorphismGroups`, with valid string values `"Both"`, `"Left"`, `"Right"`, `"None"`, may be used to specify which of the automorphism groups should be constructed first if not already known. The default is `"None"`.

`P subset Q`

Returns `true` if P is a subplane of Q, otherwise `false`.

141.11 The Connection between Projective and Affine Planes

There exist natural mathematical constructions to form a projective plane from an affine plane and vice versa. The functions in the this section provide a quick and easy way to do this in MAGMA.

`FiniteAffinePlane(P, l)`

The affine plane obtained by removing the line l from the projective plane P, together with the point set and line set of the affine plane, plus the embedding map from the affine plane to P.

`ProjectiveEmbedding(P)`

The projective completion of the affine plane P, together with the point set and line set of the projective plane, plus the embedding map from P to the projective plane.

Example H141E7

We begin with the classical affine plane A of order 3, and take the projective embedding P of A. We then remove a randomly selected line from P, and show that the affine plane produced by this action is isomorphic to the original affine plane A.

```
> A := FiniteAffinePlane(3);
> P := ProjectiveEmbedding(A);
> P;
Projective Plane of order 3
> A2 := FiniteAffinePlane(P, Random(LineSet(P)));
> A2;
Affine Plane of order 3
> iso, map := IsIsomorphic(A, A2);
> is_iso, map := IsIsomorphic(A, A2);
> is_iso;
true
> map;
Mapping from: PlaneAff: A to PlaneAff: A2
```

We demonstrate the use of the embedding map to get the correspondence between the points of the affine and projective planes.

```
> K<w> := GF(4);
> A, AP, AL := FiniteAffinePlane(K);
> P, PP, PL, f := ProjectiveEmbedding(A);
```

Now take a point of the affine plane and map it into the projective.

```
> AP.5;
```

```
( 1, w )
> AP.5 @ f;
5
```

Our point corresponds to `PP.5`, which in the affine plane is the pair `(1, w)`. The map `f` can be applied to any point or line of the affine plane to get the corresponding point or line of the projective plane. Given any point or line of the projective plane, provided that it is not on the adjoined line at infinity, the preimage in the affine plane can be found.
The line at infinity is always the last line in the line set of the projective plane created by `ProjectiveEmbedding`. We will call this line `linf`:

```
> linf := PL.#PL;
> linf;
{17, 18, 19, 20, 21}
> SetSeed(1, 3);
> p := Random(PP);
> p in linf;
false
> p @@ f;
( w, 1 )
> l := Random(PL);
> l eq linf;
false
> l @@ f;
< 1 : 1 : 0 >
> $1 @ f eq l;
true
```

Since neither `p` nor `l` were infinite we could find their preimages under `f`. Of course, when we map a line from `P` to `A` and back, we get the line we started with.
When an embedding is constructed by `FiniteAffinePlane(P, l)`, then `l` is the line at infinity for this embedding.

141.12 Operations on Points and Lines

All the usual equality, membership and subset functions are provided along with a collection of deconstruction functions and others.

141.12.1 Elementary Operations

`p eq q`

Returns `true` if the points p and q are equal, otherwise `false`.

`p ne q`

Return `true` if the points p and q are not equal, otherwise `false`.

`l eq m`

Return `true` if the lines l and m are equal, otherwise `false`.

`l ne m`

Return `true` if the lines l and m are not equal, otherwise `false`.

`p in l`

Return `true` if point p lies on the line l, otherwise `false`.

`p notin l`

Return `true` if point p does not lie on the line l, otherwise `false`.

`S subset l`

Given a subset S of the point set of the plane P and a line l of P, return `true` if the subset S of points lies on the line l, otherwise `false`.

`S notsubset l`

Given a subset S of the point set of the plane P and a line l of P, return `true` if the subset S of points does not lie on the line l, otherwise `false`.

`l meet m`

The unique point common to the lines l and m.

`Representative(l)`

`Rep(l)`

Given a line l of the plane P, return a representative point of P which is incident with l.

`Random(l)`

Given a line l of the plane P, return a random point of P which is incident with l.

141.12.2 Deconstruction Functions

`Index(P, p)`

Given a point p from the point–set V of a plane P, return the index of p, i.e. the integer i such that p is `V.i`.

`Index(P, l)`

Given a line l, return the index of l in the plane P, i.e. the integer i such that l is $L.i$ (where L is the line–set of P).

`p[i]`

The i-th coordinate of the point p, which must be from a classical plane. If p is from a projective plane, then i must satisfy $1 \leq i \leq 3$; if p is from an affine plane, then i must satisfy $1 \leq i \leq 2$.

`l[i]`

The i-th coordinate of the line l, which must be from a classical plane. The integer i must satisfy $1 \leq i \leq 3$. Recall that in a classical plane $\langle a : b : c \rangle$ (where $a, b, c \in K$) represents the line given by the equation $ax + by + cz = 0$ in a projective plane or $ax + by + c = 0$ in an affine plane.

`Coordinates(P, p)`

Given a point $p = (a : b : c)$ from a classical projective plane P (or $p = (a, b)$ from a classical affine plane P), return the sequence $[a, b, c]$ (or $[a, b]$ in the affine case) of coordinates of p.

`Coordinates(P, l)`

Given a line $l = \langle a : b : c \rangle$ from a classical plane P (projective or affine), return the sequence $[a, b, c]$ of coordinates of l.

`ElementToSequence(p)`

`Eltseq(p)`

Given a point $p = (a : b : c)$ from a classical projective plane P (or $p = (a, b)$ from a classical affine plane P), return the sequence $[a, b, c]$ (or $[a, b]$ in the affine case) of coordinates of p.

`ElementToSequence(l)`

`Eltseq(l)`

Given a line $l = \langle a : b : c \rangle$ from a classical plane P (projective or affine), return the sequence $[a, b, c]$ of coordinates of l.

`Set(l)`

The set of points contained in the line l.

Example H141E8

The following example illustrates the use of some of the elementary and deconstruction functions on lines and points discussed in the previous two subsections.

```
> K<w> := GF(4);
> P, V, L := FiniteProjectivePlane(K);
```

Create the line $x + z = 0$:

```
> l := L![1, 0, 1];
> l;
< 1 : 0 : 1 >
```

Look at the points on the line l:

```
> Set(l);
{  ( 0 : 1 : 0 ), ( 1 : w^2 : 1 ), ( 1 : 0 : 1 ),
```

```
   ( 1 : w : 1 ), ( 1 : 1 : 1) }
```

Get the coordinates of the line l:

```
> Coordinates(P, l);
[ 1, 0, 1 ]
> l[1];
1
```

Find the index of the line l in the line–set L of P, and check it:

```
> Index(P, l);
8
> l eq L.8;
true
```

Test if a point is on the line l:

```
> V![1, 0, 1] in l;
true
```

Test a set of points for containment in l:

```
> S := {V.1, V.2};
> S;
{  ( 1 : 0 : 0 ), ( 0 : 1 : 0 ) }
> S subset l;
false
```

Create the line containing the points in S:

```
> l2 := L!S;
> l2;
< 0 : 0 : 1 >
> S subset l2;
true
```

And finally, find the point common to the lines l and $l2$:

```
> p := l meet l2;
> p;
( 0 : 1 : 0 )
> p[3];
0
```

141.12.3 Other Point and Line Functions

`IsCollinear(P, S)`

Return `true` if the set S of points of the plane P are collinear, otherwise `false`. If the points are collinear, the line which they define is also returned.

`IsConcurrent(P, R)`

Return `true` if the set R of lines of the plane P are concurrent, otherwise `false`. If the lines are concurrent, their common point is returned as a second value.

`ContainsQuadrangle(P, S)`

Return `true` if the set S of points of a plane P contains a quadrangle.

`Pencil(P, p)`

The pencil of lines passing through the point p in the plane P.

`Slope(l)`

The slope of the line l of a classical affine plane P.

`IsParallel(P, l, m)`

Return `true` if the line l is parallel to the line m in the affine plane P.

`ParallelClass(P, l)`

The parallel class containing the line l of an affine plane P.

`ParallelClasses(P)`

The partition into parallel classes of the lines of the affine plane P.

Example H141E9

We use the affine plane $AG_2(3)$ to demonstrate some of the above functions.

```
> A, V, L := FiniteAffinePlane(3);
```

Create the line $y = 2x + 1$ in A, and check its slope:

```
> l := L![2, 1];
> l;
< 1 : 1 : 2 >
> Slope(l);
2
```

Find the lines parallel to l:

```
> ParallelClass(l);
{
    < 1 : 1 : 0 >,
    < 1 : 1 : 1 >,
    < 1 : 1 : 2 >
```

```
}
> [Slope(m): m in ParallelClass(l)];
[ 2, 2, 2 ]
```

Get the pencil of lines through a point of l:

```
> p := Rep(l);
> p;
( 1, 0 )
> Pencil(A, p);
{
    < 1 : 0 : 2 >,
    < 1 : 1 : 2 >,
    < 1 : 2 : 2 >,
    < 0 : 1 : 0 >
}
```

141.13 Arcs

A *k–arc* in a projective or affine plane P is a set of k points of P, no three of which are collinear. A k–arc is *complete* if it cannot be extended to a $(k+1)$–arc by the addition of another point. A *tangent* to an arc A is a line which meets A exactly once; a *secant* is a line which meets A exactly twice; and a *passant*, or *external line*, is a line which does not meet A at all.

`kArc(P, k)`

Return a k–arc for the plane P.

`CompleteKArc(P, k)`

Return a complete k–arc for the plane P (if one exists).

`IsArc(P, A)`

Returns `true` if the set of points A is an arc in the plane P, i.e. no three points of A are collinear.

`IsComplete(P, A)`

Returns `true` if the k–arc A is complete in the plane P.

`Conic(P, S)`

Given a set S of five points belonging to a classical projective plane P of order $n > 3$ and being in general position, construct the unique conic that passes through them.

`QuadraticForm(S)`

Given a set S of five points belonging to a classical projective plane of order $n > 3$ that are in general position, return the quadratic form defining the conic containing the five points.

`Tangent(P, A, p)`

Given an arc A in the plane P, and a point p on A, return a tangent to A at p.

`AllTangents(P, A)`

Given an arc A in the plane P, return the set of tangent lines to A.

`AllSecants(P, A)`

Given an arc A in the plane P, return the set of secant lines to A.

`ExternalLines(P, A)`
`AllPassants(P, A)`

Given an arc A in the plane P, return the set of external lines to A.

`Knot(P, C)`

Given a conic C in the projective plane P of even order, return the knot of the conic C, i.e the intersection point of the tangents to C.

`Exterior(P, C)`

Given a conic C in the projective plane P of odd order, return the exterior points of C, i.e. the points of P that lie on two tangents of C.

`Interior(P, C)`

Given a conic C in the projective plane P of odd order, return the interior points of C, i.e. the points of P that do not lie on any tangent of C.

Example H141E10

The following sequence of instructions constructs an oval design from $PG_2(16)$.

```
> P, V, L := FiniteProjectivePlane(16);
> oval := kArc(P, 18);
> pts := Points(P) diff oval;
> lns := ExternalLines(P, oval);
> I := IncidenceStructure< SetToIndexedSet(pts) | [l meet pts : l in lns] >;
> D := Design(Dual(I), 2);
> D;
2-(120, 8, 1) Design with 255 blocks
```

The next example uses various functions discussed so far, and shows the relationship between a plane and its subplanes.

```
> K<w> := GF(9);
```

```
> P, V, L := FiniteProjectivePlane(K);
> c := kArc(P, 5);
> c;
{  ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 2 : w^3 ),
   ( 1 : w : w) }
> C := Conic(P, c);
> C;
{  ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 2 : w^3 ),
   ( 1 : w : w), ( 1 : w^5 : w^6 ), ( 1 : w^7 : 2 ), ( 1 : w^2 : 1 ),
   ( 1 : w^3 : w^5 ), ( 1 : w^6 : w^2 ) }
> #C;
10
> #Interior(P, C);
36
>
> S, SV, SL := SubfieldSubplane(P, GF(3));
> S subset P;
true
> a := kArc(S, 4);
> IsArc(S, a);
true
> IsArc(P, a);
true
> IsComplete(S, a);
true
> IsComplete(P, a);
false
> a;
{ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 2 : 1 ) }
> S2 := sub< P | a >;
> S2;
Projective Plane of order 3
> S2 eq S;
true
> p := Random(a);
> p;
( 1 : 2 : 1 )
> Tangent(S, a, p);
< 1 : 2 : 1 >
> AllTangents(S, a);
{
    < 1 : 2 : 0 >,
    < 0 : 1 : 2 >,
    < 1 : 2 : 1 >,
    < 1 : 0 : 1 >
}
> AllTangents(P, a);
{
```

```
    < 1 : w^5 : w^6 >,
    < 1 : 0 : w >,
    < 1 : w^2 : w >,
    < 1 : 0 : w^6 >,
    < 1 : 0 : 1 >,
    < 1 : w^6 : w^3 >,
    < 1 : w^6 : 0 >,
    < 0 : 1 : w^6 >,
    < 1 : 2 : 0 >,
    < 0 : 1 : 2 >,
    < 1 : 2 : 1 >,
    < 1 : w^3 : w^5 >,
    < 1 : 0 : w^7 >,
    < 1 : 0 : w^2 >,
    < 1 : w : 0 >,
    < 0 : 1 : w >,
    < 1 : 0 : w^5 >,
    < 1 : 0 : w^3 >,
    < 1 : w : w^7 >,
    < 1 : w^3 : 0 >,
    < 0 : 1 : w^3 >,
    < 1 : w^7 : w^2 >,
    < 1 : w^5 : 0 >,
    < 0 : 1 : w^5 >,
    < 1 : w^7 : 0 >,
    < 0 : 1 : w^7 >,
    < 1 : w^2 : 0 >,
    < 0 : 1 : w^2 >
}
```

141.14 Unitals

A unital in the classical projective plane $PG_2(q^2)$ is a set of $q^3 + 1$ points such that every line meeting two of these points meets exactly $q + 1$ of them.

`IsUnital(P, U)`

Given a set of points U belonging to a projective plane P defined over a field of cardinality q^2, return `true` if U is a unital.

`AllTangents(P, U)`

Given a unital set of points U in the projective plane P, return the set of tangents to the points of U.

`UnitalFeet(P, U, p)`

The set of intersections of the unital set of points U with the tangents to U in the plane P which pass through the point p.

Example H141E11

The following code computes the Hermitian unital given by the equation $x^{q+1} + y^{q+1} + z^{q+1} = 0$ in $PG_2(q^2)$ for $q = 3$.

```
> q := 3;
> F<w> := GaloisField(q ^ 2);
> P, V, L := FiniteProjectivePlane(F);
>
> hu := { V | [x,y,z] : x, y, z in F |
>                x^(q+1) + y^(q+1) + z^(q+1) eq 0 and {x, y, z} ne {0} };
>
> IsUnital(P, hu);
true
> UnitalFeet(P, hu, V.1);
{  ( 0 : 1 : w ), ( 0 : 1 : w^3 ), ( 0 : 1 : w^5 ), ( 0 : 1 : w^7 ) }
```

Since this set has more than one element, `V.1` must not be in *hu*:

```
> V.1 in hu;
false
```

For a point in *hu*:

```
> UnitalFeet(P, hu, Rep(hu));
{  ( 1 : 0 : w^7 ) }
```

Now we construct the design given by *hu*.

```
> blks := [blk : lin in L | #blk eq (q+1) where blk is lin meet hu ];
> D := Design< 2, SetToIndexedSet(hu) | blks >;
> D;
2-(28, 4, 1) Design with 63 blocks
```

141.15 The Collineation Group of a Plane

The automorphism group (or collineation group) A of a plane P is always presented as a permutation group G acting on the standard support $\{1, ..v\}$, where v is the number of points of P. The reasons for this include the fact that if the group is represented as acting on a set of objects relating to the plane, printed permutations are often unreadable.

So the collineation group G of P does not act directly on P. Instead, G–sets are used to transfer the action of G to various sets associated with P. The two most important G–sets, corresponding to action of G on the point set and on the line set, are returned by each of the functions provided for constructing the collineation group or some specified subgroup of it.

In some circumstances, rather than viewing collineations as group elements, it is desirable to view them is as mappings of P into itself. Associated with each incidence structure is a mapping structure, Aut(P), which denotes the set of collineations of P. Note that Aut(P) is the *parent* of the collineations of G so that the function `Aut(P)` simply creates a

shell structure rather than the actual collineation group of P. A transfer map is provided to convert a permutation of the collineation group G into a mapping belonging to $Aut(P)$.

141.15.1 The Collineation Group Function

`CollineationGroup(P)`

`AutomorphismGroup(P)`

`PointGroup(P)`

Construct the collineation group G of the plane P. The group G is returned as a permutation group on the standard support $\{1 \dots v\}$, where v is the number of points of P. The function also returns: a G–set Y being the point set of P acted on by G; a G–set W being the line set of P acted on by G; a power structure S; a transfer map t. Given a permutation g from G, one can create a map $f = t(g)$ which represents the automorphism g as a mapping in S from P to P (which maps both point sets and line sets). The G–sets Y and W should be used if one wishes to compute stabilizers or similar such subgroups of G so that the appropriate action is used.

`LineGroup(P)`

Construct the collineation group G of the plane P in its action on the lines of P. The group G is returned as a permutation group on the standard support $\{1 \dots l\}$, where l is the number of lines of P. A power structure S and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the automorphism g as a mapping in S from L to L, where L is the line set of P.

`CollineationGroupStabilizer(P, k)`

A subgroup G of the collineation group of the plane P which stabilizes the first k base points, together with the points G–set, the lines G–set, the power structure A of all automorphisms of P, and the transfer map t from G into A.

`CollineationSubgroup(P)`

A subgroup G of the collineation group of the plane P generated by one element, together with the points G–set, the lines G–set, the power structure A of all automorphisms of P, and the transfer map t from G into A.

141.15.2 General Action of Collineations

The collineation group G of a plane P is given in its action on the standard support. This support may be regarded as the indices of the points of P. The action of G on P is obtained using the G–set mechanism. The two basic G–sets associated with P correspond to the action of G on the set of points V and the set of lines L of P. These two G–sets are given as return values of the function `AutomorphismGroup`. Additional G–sets associated with P may be built using the G–set constructors. Given a G–set Y for G, the action of G on Y may be studied using the permutation group functions that allow a G–set as an argument. In this section, only a few of the available functions are described: see the chapter on permutation groups for a complete list.

The action of the collineation group on the plane may also be obtained using the ^ operator.

`y ^ g`

Let G be a subgroup of the collineation group for the plane P and suppose $g \in G$. Given an element y that is either a point or line of P, return the image of y under g.

`y ^ G`

Let G be a subgroup of the collineation group for the plane P. Given an element y that is either a point or line of P, return the orbit of y under G.

`Image(g, Y, y)`

Let G be a subgroup of the collineation group for the plane P and let Y be a G–set for G. Given an element y belonging either to Y or to a G–set derived from Y, find the image of y under G.

`Orbit(G, Y, y)`

Let G be a subgroup of the collineation group for the plane P and let Y be a G–set for G. Given an element y belonging either to Y or to a G–set derived from Y, construct the orbit of y under G.

`Orbits(G, Y)`

Let G be a subgroup of the collineation group for the plane P and let Y be a G–set for G. This function constructs the orbits of the action of G on Y.

`Stabilizer(G, Y, y)`

Let G be a subgroup of the collineation group for the plane P and let Y be a G–set for G. Given an element y belonging either to Y or to a G–set derived from Y, construct the stabilizer of y in G.

`Action(G, Y)`

Given a subgroup G of the collineation group of the plane P, and a G–set Y for G, construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. The function returns:

(a) The natural homomorphism $\phi : G \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of G).

`ActionImage(G, Y)`

Given a subgroup G of the collineation group of the plane P, and a G–set Y for G, construct the permutation group L giving the action of G on the set Y.

`ActionKernel(G, Y)`

Given a subgroup G of the collineation group of the plane P, and a G–set Y for G, construct the kernel of the action of G on the set Y.

Example H141E12

We illustrate the use of the G–sets returned by the `CollineationGroup` function.

```
> P := FiniteProjectivePlane(3);
> G, Y, W := CollineationGroup(P);
```

Compute the stabilizer of the first point of P:

```
> H := Stabilizer(G, Y, Points(P)[1]);
> H;
Permutation group H acting on a set of cardinality 13
    (3, 9)(5, 7)(6, 11)(8, 12)
    (2, 8, 12)(3, 9)(4, 6, 7, 10, 5, 11)
    (2, 9)(3, 4)(5, 8)(10, 13)
    (2, 8)(3, 9)(4, 5)(6, 10)
    (2, 3)(4, 13)(6, 8)(9, 10)
    (2, 13, 8)(3, 6, 4)(5, 10, 9)
    (2, 3)(4, 9)(7, 12)(10, 13)
> H eq CollineationGroupStabilizer(P, 1);
true
> lines := {m : m in Lines(P) | Points(P)[1] in m};
> l := Random(lines);
> l ^ H;
GSet{
  < 0 : 1 : 0 >,
  < 0 : 0 : 1 >,
  < 0 : 1 : 2 >,
  < 0 : 1 : 1 >
```

```
}
```

Compute the stabilizer of the first line of P:

```
> Stabilizer(G, W, Lines(P)[1]);
Permutation group acting on a set of cardinality 13
    (1, 8, 12, 4, 7, 9)(2, 3, 5)(6, 13)
    (3, 5, 11)(6, 7, 9)(8, 12, 13)
    (4, 10)(5, 11)(6, 7)(8, 12)
    (1, 10, 6, 7)(2, 11)(3, 5)(4, 13, 9, 12)
    (5, 11)(6, 12)(7, 8)(9, 13)
    (1, 10)(3, 5)(6, 7)(12, 13)
    (1, 4, 10)(3, 11)(6, 13, 9, 8, 7, 12)
```

Example H141E13

The following function `Bundle` returns a projective bundle in $PG_2(q)$.

```
> Bundle := function(q)
>     K<w> := GF(q^3);
>     P, V, L := FiniteProjectivePlane(q^3);
>     G, Y := CollineationGroup(P);
>     S := Support(P);  // normalized vectors
>     sig := sub< G |
>                [Index(P, V ! [S[i][1], S[i][2]^q, S[i][3]^q]) : i in [1..#V]]>;
>                        // group of planar collineations of order 3
>     p := V ! [1, w^2, w];
>     T := Orbit(sig, Y, p);
>     e2 := V![0, 1, 0];
>     e3 := V![0, 0, 1];
>     S := Points(SubfieldSubplane(P, GF(q)));
>     c23 := Conic(P, T join {e2, e3}) meet S;
>     e1 := Rep(S diff c23);
>     c12 := (Conic(P, T join {e1, e2}) meet S) diff { e1 };
>     c13 := (Conic(P, T join {e1, e3}) meet S) diff { e1 };
>     bundle := [ Conic(P, T join {e1, e}) meet S : e in c23 ] cat
>               [ Conic(P, T join {v1, v2}) meet S : v2 in c13, v1 in c12 ];
>     return FiniteProjectivePlane< S | bundle >;
>
> end function;
>
> PB := Bundle(3);
> PB;
Projective Plane of order 3
```

Example H141E14

The function `BaerDerivation` below uses a Baer subplane to construct an affine plane.

```
> BaerDerivation := function(q)
> //--------------------------
> // Construct an affine plane by the technique of derivation using
> // Baer subplanes
>
>    Fq2< w > := FiniteField(q^2);
>    V := VectorSpace(Fq2, 3);
>    Plane, Pts, Lns := FiniteProjectivePlane(V);
>    G, Y := CollineationGroup(Plane);
```

Construct a Baer subplane:

```
>    Subplane := SubfieldSubplane(Plane, GF(q));
```

The Baer segment consists of those points of the Baer subplane that lie on the line at infinity. Take the line $x = 0$ as the line at infinity.

```
>    LineInf := Lns![1, 0, 0];
>
>    BaerSeg := Points(Subplane) meet LineInf;
```

We now find the subgroup of the collineation group that fixes the Baer segment. The translates of the Baer subplane under this subgroup will give us those Baer subplanes that contain the set `BaerSeg`. We use the G–set `Y` to specify the action of `G` on the points of `Plane`.

```
>    StabSeg := Stabilizer(G, Y, BaerSeg);
```

Rather than computing the translates of the entire Baer subplane, we compute the translates of `Subplane - BaerSeg` so that we get exactly those sets which become new affine lines.

```
>    BaerLines := Orbit(StabSeg, Y, Points(Subplane) diff BaerSeg);
```

We complete the new plane by taking those lines of $PG(2, q^2)$ which intersect the line at infinity at points other than those in the Baer segment. Upon removing the intersection point with `LineInf`, each such line becomes a line of the new affine plane.

```
>    AffLines := BaerLines join { Set(l) diff LineInf : l in Lns |
>                                 (BaerSeg meet l) eq {} };
>    return FiniteAffinePlane< SetToIndexedSet(&join(AffLines)) | Setseq(AffLines)
>                                          : Check := false >;
> end function; /*BaerDerivation*/
```

141.15.3 Central Collineations

Let p be a point and l a line of a projective plane P. A (p, l)–*central collineation* is a collineation α of P which fixes l pointwise and p linewise. The line l is called the *axis* of α and the point p is called the *centre* of α.

`CentralCollineationGroup(P, p, l)`

The group G of (p, l)–central collineations of a projective plane P. A power structure S and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the permutation g as a mapping in S from P to P. (which maps both point sets and line sets).

`CentralCollineationGroup(P, p)`

The group of central collineations with centre p of a projective plane P. A power structure S and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the permutation g as a mapping in S from P to P. (which maps both point sets and line sets).

`CentralCollineationGroup(P, l)`

The group of central collineations with axis l of a projective plane P. A power structure S and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the permutation g as a mapping in S from P to P. (which maps both point sets and line sets).

`IsCentralCollineation(P, g)`

Returns `true` iff the collineation g of the projective plane P is a central collineation; if `true`, also returns the centre and axis of g. The support of the parent of g must be the point set of P or the standard support $\{1 \dots v\}$, where v is the number of points of P.

Example H141E15

We find a group of central collineations of a plane P, and check that a random element of the group is in fact a central collineation and has the correct axis and centre.

```
> P, V, L := FiniteProjectivePlane< 13 |
>     {1, 2, 3, 4}, {1, 5, 6, 7}, {1, 8, 9, 10},
>     {1, 11, 12, 13}, {2, 5, 8, 11}, {2, 6, 9, 12},
>     {2, 7, 10, 13}, {3, 5, 9, 13}, {3, 6, 10, 11},
>     {3, 7, 8, 12}, {4, 5, 10, 12}, {4, 6, 8, 13},
>     {4, 7, 9, 11} >;
> p := V!3;
> l := L.1;
> G := CentralCollineationGroup(P, p, l);
> G;
Permutation group G acting on a set of cardinality 13
Order = 3
```

```
    (5, 13, 9)(6, 11, 10)(7, 12, 8)
> g := Random(G);
> g;
(5, 9, 13)(6, 10, 11)(7, 8, 12)
> is_cent_coll, centre, axis := IsCentralCollineation(P, g);
> is_cent_coll;
true
> centre eq p;
true
> axis eq l;
true
```

Any line through the centre of a central collineation must be fixed by the collineation.

```
> lines := {m : m in Lines(P) | p in m};
> m := Random(lines);
> m;
{3, 7, 8, 12}
> m ^ G;
GSet{
  {3, 7, 8, 12}
}
```

141.15.4 Transitivity Properties

`IsPointTransitive(P)`

`IsTransitive(P)`

Return `true` iff the collineation group of the plane P acts transitively on the points of P.

`IsLineTransitive(P)`

Return `true` iff the collineation group of the plane P acts transitively on the lines of P.

Example H141E16

We check the transitivity of the collineation group of $AG_2(4)$.

```
> P := FiniteAffinePlane(4);
> IsPointTransitive(P);
true
> IsLineTransitive(P);
true
```

141.16 Translation Planes

The following functions may be used to construct translation planes by derivation. The original code is due to Jenny Key.

`BaerDerivation(q2)`

An affine plane constructed by the technique of derivation with respect to a Baer subplane, where $q2$ is an even power of a prime.

`BaerSubplane(P)`

A Baer subplane of the projective plane P.

`OvalDerivation(q: parameters)`

`HallOval`	BoolElt	*Default :* `false`
`Print`	BoolElt	*Default :* `false`

A translation plane from $\mathrm{PG}(2, q)$, where q is a power of 2, computed by derivation with respect to an oval. By default, the oval chosen is that defined by the points of the conic $y^2 = xz$ together with the nucleus, in $\mathrm{PG}(2, q)$. In the case $q = 16$, a Hall oval in the plane may be requested by assigning the parameter `HallOval` the value `true`. If the parameter `Print` is assigned the value `true`, then some information is printed during the computation.

141.17 Planes and Designs

Projective and affine planes can be viewed as special kinds of designs. The following functions convert between designs and planes.

`Design(P)`

The design corresponding to the points and lines of the plane P.

`FiniteAffinePlane(D)`

The affine plane corresponding to the incidence structure D.

`FiniteProjectivePlane(D)`

The projective plane corresponding to the incidence structure D.

Example H141E17

The development of a Singer difference set provides a design which satisfies the projective plane axioms, and thus can be converted to a projective plane in MAGMA.

```
> sds := SingerDifferenceSet(2, 3);
> sds;
{ 0, 1, 3, 9 }
> sdv := Development(sds);
> sdv;
2-(13, 4, 1) Design with 13 blocks
> spp := FiniteProjectivePlane(sdv);
> spp: Maximal;
Projective Plane of order 3
Points: {@ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 @}
Lines:
    {0, 1, 3, 9},
    {1, 2, 4, 10},
    {2, 3, 5, 11},
    {3, 4, 6, 12},
    {0, 4, 5, 7},
    {1, 5, 6, 8},
    {2, 6, 7, 9},
    {3, 7, 8, 10},
    {4, 8, 9, 11},
    {5, 9, 10, 12},
    {0, 6, 10, 11},
    {1, 7, 11, 12},
    {0, 2, 8, 12}
> Universe(Support(spp));
Residue class ring of integers modulo 13
```

141.18 Planes, Graphs and Codes

MAGMA provides the following functions to create the graphs and codes naturally associated to a projective or affine plane.

`LineGraph(P)`

The line graph of the plane P.

`IncidenceGraph(P)`

The incidence graph of the plane P. This bipartite graph has as vertex set the union of the point set V and line set L of P. A vertex $p \in V$ is adjacent to a vertex $l \in L$ whenever $p \in l$.

`LinearCode(P, K)`

Given a plane P with v points and a finite field K, this function returns the linear code C of length v generated by the characteristic functions of the lines of P considered as vectors of the K-space $K^{(v)}$.

Example H141E18

```
> P, V, L := FiniteAffinePlane(3);
> #V, #L;
9 12
> IncidenceGraph(P);
Graph
Vertex  Neighbours
 1       10 11 12 19 ;
2       16 17 18 19 ;
3       13 14 15 19 ;
4       10 15 17 21 ;
5       12 14 16 21 ;
6       11 13 18 21 ;
7       10 14 18 20 ;
8       11 15 16 20 ;
9       12 13 17 20 ;
10      1 4 7 ;
11      1 6 8 ;
12      1 5 9 ;
13      3 6 9 ;
14      3 5 7 ;
15      3 4 8 ;
16      2 5 8 ;
17      2 4 9 ;
18      2 6 7 ;
19      1 2 3 ;
20      7 8 9 ;
21      4 5 6 ;
 > LinearCode(P, Field(P));
[9, 6, 3] Linear Code over GF(3)
Generator matrix:
[1 0 0 0 0 1 0 1 0]
[0 1 0 0 0 1 0 2 2]
[0 0 1 0 0 1 0 0 1]
[0 0 0 1 0 2 0 1 2]
[0 0 0 0 1 2 0 2 1]
[0 0 0 0 0 0 1 1 1]
```

142 INCIDENCE GEOMETRY

Chapter 142
INCIDENCE GEOMETRY

142.1 Introduction

This chapter presents the functions designed for constructing and computing with incidence geometries and coset geometries.

We recall the basic definitions and notation in the field of Incidence Geometry. We refer to the *Handbook of Incidence Geometry* [Bue95], edited by Francis Buekenhout, or to Antonio Pasini's book *Diagram Geometries* [Pas94], for a more detailed overview of the subject.

Let X and I be two finite sets. Let $t : X \to I$ be a mapping from X onto I. Let $\sim$ be a reflexive and symmetric relation such that $\forall x, y \in X$, $x \sim y$ and $t(x) = t(y) \Rightarrow x = y$. The four-tuple $\Gamma(X, \sim, t, I)$ is what we call an *Incidence Geometry* in MAGMA. Remark that it is not a geometry in the sense of Buekenhout since we do not impose that every flag (i.e. clique of the incidence graph) of Γ must be contained in a chamber (i.e. a clique containing one element of each type). If the latter condition is satisfied, then an incidence geometry is a geometry in the sense of Buekenhout. The set X contains the *elements* of the geometry, while I is called the set of *types*. The function t is called the *type function* and $\sim$ is called the *incidence relation* of Γ. The cardinality of I is the *rank* of Γ.

It is possible to construct incidence geometries from a group and some of its subgroups using an algorithm first introduced by Jacques Tits in 1962 [Tit62]. Let G be a group and let I be a finite set. Let $\{G_i, i \in I\}$ be a set of subgroups of G. Define $X = \{G_i g, g \in G, i \in I\}$ to be the set of elements of Γ. Define the type function as $t : X \to I : G_i g \to i$ and the incidence relation as follows: $G_i g \sim G_j h$ iff $G_i g \cap G_j h \neq \emptyset$. The subgroups $\{G_i, i \in I\}$ are called the *maximal parabolic subgroups*. The subgroup $\cap_{i \in I} G_i$ is called the *Borel subgroup*. Finally, the subgroups $\{\cap_{j \in I \setminus \{i\}} G_j, i \in I\}$ are called the *minimal parabolic subgroups*. These geometries are called *Coset Geometries* in MAGMA to remind the user that they are constructed from a group. Again, a coset geometry is not a geometry in the sense of Buekenhout. If every flag of the coset geometry is contained in a chamber, then it is a Buekenhout geometry. We will see that, using coset geometries, it is easy to build huge incidence geometries by giving very little data.

The category names for the incidence geometries and coset geometries are:

- Incidence Geometry : `IncGeom`
- Coset Geometry : `CosetGeom`

142.2 Construction of Incidence and Coset Geometries

142.2.1 Construction of an Incidence Geometry

`IncidenceGeometry(G)`

Construct the incidence geometry IG having as incidence graph the (labelled) graph G.

If G is an unlabelled graph, then the set of elements of IG is the set of vertices and edges of G, the set of types is $\{@1, 2@\}$ where elements of type 1 are vertices of G and elements of type 2 are edges of G. An element x of type 1 is incident to an element y of type 2 iff the vertex x in G is on the edge y of G.

If G is a labelled graph, i.e. a graph whose vertices have labels, then the set of elements of IG is the set of vertices of G, the set of types is the set of labels of the vertices of G and the incidence graph is G.

The function returns one value: the incidence geometry IG.

Example H142E1

The Petersen graph, considered as a rank two incidence geometry, may be constructed by the following statement:

```
> gr := Graph<10|
>   {1,2},{1,5},{1,6},{2,7},{2,3},{3,8},{3,4},{4,9},{4,5},{5,10},
>   {6,8},{8,10},{10,7},{7,9},{9,6}>;
> ig := IncidenceGeometry(gr);
> ig;
Incidence geometry ig with 2 types and with underlying graph:
Graph
Vertex  Neighbours
1       11 12 13 ;
2       11 14 15 ;
3       14 16 17 ;
4       16 18 19 ;
5       12 18 20 ;
6       13 21 22 ;
7       15 23 24 ;
8       17 21 25 ;
9       19 22 23 ;
10      20 24 25 ;
11      1 2 ;
12      1 5 ;
13      1 6 ;
14      2 3 ;
15      2 7 ;
16      3 4 ;
17      3 8 ;
18      4 5 ;
```

```
19      4 9 ;
20      5 10 ;
21      6 8 ;
22      6 9 ;
23      7 9 ;
24      7 10 ;
25      8 10 ;
```

Example H142E2

The rank three geometry consisting of the vertices, edges and faces of the cube, with symmetric inclusion as incidence might be encoded in the following way. The first 8 vertices of the graph correspond to the vertices of the cube, the next 12 vertices correspond to the edges of the cube and the last 6 are the faces of the cube.

```
> g := Graph<26|
>    {1,9},{1,12},{1,13},{1,21},{1,22},{1,25},
>    {2,9},{2,10},{2,14},{2,21},{2,22},{2,23},
>    {3,10},{3,11},{3,15},{3,21},{3,23},{3,24},
>    {4,11},{4,12},{4,16},{4,21},{4,24},{4,25},
>    {5,13},{5,18},{5,17},{5,22},{5,25},{5,26},
>    {6,14},{6,19},{6,18},{6,23},{6,22},{6,26},
>    {7,15},{7,19},{7,20},{7,24},{7,26},{7,23},
>    {8,17},{8,20},{8,16},{8,26},{8,24},{8,25},
>    {9,21},{9,22},{10,21},{10,23},{11,21},{11,24},
>    {12,21},{12,25},{13,22},{13,25},{14,22},{14,23},
>    {15,23},{15,24},{16,24},{16,25},{17,25},{17,26},
>    {18,22},{18,26},{19,23},{19,26},{20,24},{20,26}>;
> v := VertexSet(g);
> AssignLabels(v,[1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,3,3,3,3,3,3]);
> cube := IncidenceGeometry(g);
> cube;
Incidence geometry cube with 3 types and with underlying graph:
Graph
Vertex  Neighbours
1       9 12 13 21 22 25 ;
2       9 10 14 21 22 23 ;
3       10 11 15 21 23 24 ;
4       11 12 16 21 24 25 ;
5       13 17 18 22 25 26 ;
6       14 18 19 22 23 26 ;
7       15 19 20 23 24 26 ;
8       16 17 20 24 25 26 ;
9       1 2 21 22 ;
10      2 3 21 23 ;
11      3 4 21 24 ;
12      1 4 21 25 ;
13      1 5 22 25 ;
```

```
14      2 6 22 23 ;
15      3 7 23 24 ;
16      4 8 24 25 ;
17      5 8 25 26 ;
18      5 6 22 26 ;
19      6 7 23 26 ;
20      7 8 24 26 ;
21      1 2 3 4 9 10 11 12 ;
22      1 2 5 6 9 13 14 18 ;
23      2 3 6 7 10 14 15 19 ;
24      3 4 7 8 11 15 16 20 ;
25      1 4 5 8 12 13 16 17 ;
26      5 6 7 8 17 18 19 20 ;
```

Example H142E3

The Hoffman-Singleton graph $HoSi$: the following description is taken from the book *Diagram Geometry*, by A. Cohen and F. Buekenhout [BCon].
Start with the set $X_1 = \binom{7}{3}$ of all triples from 7. There are 30 collections of 7 triples that form the lines of a *Fano plane* with point set 7. The group $Sym(7)$ acts transitively on this set of 30 elements, but the alternating group $Alt(7)$ has two orbits, of cardinality 15 each. Select one and call it X_2. Now consider the following graph Ω constructed from the incidence graph $(X_1 \cup X_2, *)$ by additionally joining two elements of X_1 whenever they have empty intersection. The following lines of MAGMA-code implement the Hoffman-Singleton graph as an incidence geometry of rank 2. We will study it more later.

```
> HoffmanSingletonGraph := function()
>   pentagram := Graph< 5 | { {1,3}, {3,5}, {5,2}, {2,4}, {4,1} } >;
>   pentagon := PolygonGraph(5);
>   PP := pentagram join pentagon;
>   HS := &join [ PP : i in [1..5] ];
>   return HS + { { Vertices(HS) | i + j*10, k*10 + 6 + (i+j*k) mod 5 } :
>                                  i in [1..5], j in [0..4], k in [0..4] };
> end function;
> ig := IncidenceGeometry(HoffmanSingletonGraph());
```

Example H142E4

A rank four geometry constructed from the Hoffman-Singleton graph : the following description is taken from the book *Diagram Geometry*, by A. Cohen and F. Buekenhout.
Take X_1 and $X_{1'}$ to be two copies of the vertex set of $HoSi$. The group $Sym(7)$ acts transitively on the set of maximal cocliques of $HoSi$. The subgroup $Alt(7)$ has two orbits of size 15 each. Let X_2 be one of them and $X_{2'}$ be a copy of X_2. Define incidence $\sim$ for $a \in X_1$, $a' \in X_{1'}$, $b \in X_2$ and $b' \in X_{2'}$, by
$a \sim a' \Leftrightarrow \{a, a'\}$ is an edge of $HoSi$
$a \sim b \Leftrightarrow a \notin b$
$a \sim b' \Leftrightarrow a \in b'$
$a' \sim b \Leftrightarrow a' \notin b$

$a' \sim b' \Leftrightarrow a' \notin b'$
$b \sim b' \Leftrightarrow b \cap b' = \emptyset$
The geometry $\Gamma(X_1 \cup X_{1'} \cup X_2 \cup X_{2'}, \sim)$ is the *Neumaier geometry*. The following lines of MAGMA-code implement this geometry as a rank four incidence geometry. We assume that *ig* is the incidence geometry corresponding to the Hoffman-Singleton graph, as described in the previous example.

```
> gr := HoffmanSingletonGraph();
> ig := IncidenceGeometry(gr);
> aut := AutomorphismGroup(ig);
> n := NormalSubgroups(aut);
> X1 := VertexSet(gr);
> g := CompleteGraph(50);
> e := EdgeSet(gr);
> ebis := [];
> for x in EdgeSet(gr) do
>     for i := 1 to 50 do
>         for j := i+1 to 50 do
>             if VertexSet(gr)!i in x then
>                 if VertexSet(gr)!j in x then
>                     Append(~ebis,[i,j]);
>                 end if;
>             end if;
>         end for;
>     end for;
> end for;
> for i := 1 to #ebis do
>     RemoveEdge(~g,ebis[i][1],ebis[i][2]);
> end for;
> c := MaximumClique(g);
> cbis := [];
> for i := 1 to 50 do
>     if (VertexSet(gr)!i in c) then Append(~cbis,i); end if;
> end for;
> c := Set(cbis);
> X2 := Orbit(n[2]`subgroup,c);
> X2 := SetToSequence(X2);
> e := [];
> for i := 1 to 50 do
>     for j := 1 to 50 do
>         if X1!i in Neighbours(X1!j) then Append(~e,{i,j+50}); end if;
>     end for;
> end for;
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if not(i in X2[j]) then Append(~e,{i,j+100});end if;
>     end for;
> end for;
```

```
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if i in  X2[j] then Append(~e,{i,j+150});end if;
>     end for;
> end for;
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if i in  X2[j] then Append(~e,{i+50,j+100});end if;
>     end for;
> end for;
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if not(i in  X2[j]) then Append(~e,{i+50,j+150});end if;
>     end for;
> end for;
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if X2[i] meet X2[j] eq {} then Append(~e,{i+100,j+150});end if;
>     end for;
> end for;
> gr2 := Graph<200|Set(e)>;
> v := VertexSet(gr2);
> labs := [i: j in {1..50}, i in {1..4}];
> AssignLabels(v,labs);
> neumaier := IncidenceGeometry(gr2);
```

142.2.2 Construction of a Coset Geometry

`CosetGeometry(G, S, I)`

Construct the coset geometry CG with set of types I, obtained from the group G and the set S of subgroups of G. The sets S and I must have same cardinality.

If G is a permutation group and S is a set of subgroups of G, the corresponding coset geometry, obtained by using Tits' algorithm (see above) is constructed. If S and I are indexed sets, then the cosets of the subgroup $S[i]$ are elements of type $I[i]$, for all $i \in \{0, \ldots, n-1\}$ where n is the cardinality of S (and I). The function returns one value: the coset geometry CG.

`CosetGeometry(G, S)`

Construct the coset geometry CG obtained from the group G and the set S of subgroups of G.

If G is a permutation group and S is a set of subgroups of G, the corresponding coset geometry, obtained by using Tits' algorithm (see above) is constructed. To every subgroup in S is associated a number from 0 to $n-1$ where n is the cardinality of S. The function returns one value: the coset geometry CG.

Example H142E5

The Petersen graph, considered as a rank two coset geometry, can be implemented in the following way :

```
> g := Sym(5);
> g0 := Stabilizer(g,{1,2});
> g01 := Stabilizer(g,[{1,2},{3,4}]);
> g1 := sub<g|g01,(1,3)(2,4)>;
> cg := CosetGeometry(g,{g0,g1});
> cg;
Coset geometry cg with 2 types
Group:
Symmetric group g acting on a set of cardinality 5
Order = 120 = 2^3 * 3 * 5
    (1, 2, 3, 4, 5)
    (1, 2)
Maximal Parabolic Subgroups:
Permutation group g0 acting on a set of cardinality 5
Order = 12 = 2^2 * 3
    (3, 4)
    (1, 2)
    (4, 5)
Permutation group g1 acting on a set of cardinality 5
Order = 8 = 2^3
    (1, 2)
    (3, 4)
    (1, 3)(2, 4)
Type Set:
{@ 0, 1 @}
```

Example H142E6

The rank three geometry consisting of the vertices, edges and faces of the cube, can be implemented as a coset geometry in the following way, which is much easier than having to give the full incidence graph :

```
> g := sub<Sym(8)|
>     (1,2,3,4)(5,6,7,8), (1,5)(2,6)(3,7)(4,8), (2,4,5)(3,8,6)>;
> g0 := Stabilizer(g,1);
> g1 := Stabilizer(g,{1,2});
> g2 := Stabilizer(g,{1,2,3,4});
> cg := CosetGeometry(g,{g0,g1,g2});
> cg;
Coset geometry cg with 3 types
Group:
Permutation group g acting on a set of cardinality 8
Order = 48 = 2^4 * 3
    (1, 2, 3, 4)(5, 6, 7, 8)
```

```
    (1, 5)(2, 6)(3, 7)(4, 8)
    (2, 4, 5)(3, 8, 6)
Maximal Parabolic Subgroups:
Permutation group acting on a set of cardinality 8
Order = 8 = 2^3
    (1, 2)(3, 4)(5, 6)(7, 8)
    (2, 4)(6, 8)
Permutation group acting on a set of cardinality 8
Order = 6 = 2 * 3
    (2, 4, 5)(3, 8, 6)
    (3, 6)(4, 5)
Permutation group acting on a set of cardinality 8
Order = 4 = 2^2
    (1, 2)(3, 5)(4, 6)(7, 8)
    (1, 2)(3, 4)(5, 6)(7, 8)
Type Set:
{@ 0, 1, 2 @}
```

Example H142E7

The following lines of MAGMA-code construct a coset geometry of rank 6 for the group $Sym(6)$.

```
> g := Sym(6);
> s := [Stabilizer(g,{1..j}) : j in {1..5}];
> cg := CosetGeometry(g,Set(s));
> cg;
Coset geometry cg with 5 types
Group:
Symmetric group g acting on a set of cardinality 6
Order = 720 = 2^4 * 3^2 * 5
    (1, 2, 3, 4, 5, 6)
    (1, 2)
Maximal Parabolic Subgroups:
Permutation group acting on a set of cardinality 6
Order = 48 = 2^4 * 3
    (3, 4)
    (4, 5)
    (5, 6)
    (1, 2)
Permutation group acting on a set of cardinality 6
Order = 48 = 2^4 * 3
    (1, 2)
    (2, 3)
    (3, 4)
    (5, 6)
Permutation group acting on a set of cardinality 6
Order = 36 = 2^2 * 3^2
    (1, 2)
```

```
    (4, 5)
    (5, 6)
    (2, 3)
Permutation group acting on a set of cardinality 6
Order = 120 = 2^3 * 3 * 5
    (2, 3)
    (3, 4)
    (4, 5)
    (5, 6)
Permutation group acting on a set of cardinality 6
Order = 120 = 2^3 * 3 * 5
    (1, 2)
    (2, 3)
    (3, 4)
    (4, 5)
Type Set:
{@ 0, 1, 2, 3, 4 @}
```

142.3 Elementary Invariants

`Points(D)`

`Elements(D)`

Given an incidence geometry D, return the set of elements of D. These elements are the points of the incidence graph of D.

`Types(D)`

Given an incidence geometry D, return the set of types of D.

`Types(C)`

Given a coset geometry C, return the set of types of C.

`Rank(D)`

Given an incidence geometry D, return the rank of D, i.e. the cardinality of the set of types.

`Rank(C)`

Given a coset geometry C, return the rank of C.

`IncidenceGraph(D)`

Given an incidence geometry D, return the incidence graph of D, its vertex set and its edge set.

We remark that this function is not implemented for coset geometries but we may convert a coset geometry into an incidence geometry using `IncidenceGeometry` and then compute its incidence graph.

`Group(C)`

Given a coset geometry C, return the group from which C is constructed.

`MaxParabolics(C)`
`MaximalParabolics(C)`

Given a coset geometry C, return an indexed set containing the maximal parabolics of C.

`MinParabolics(C)`
`MinimalParabolics(C)`

Given a coset geometry C, return an indexed set containing the minimal parabolics of C.

`Borel(C)`
`BorelSubgroup(C)`

Given a coset geometry C, return the *Borel subgroup* of C, i.e. the intersection of all maximal parabolic subgroups of C.

`Kernel(C)`

Given a coset geometry C, return a permutation group which is its kernel, i.e. the subgroup of the Borel subgroup of C that fixes all elements of the geometry C.

`Kernels(C)`

Given a coset geometry C, return a sequence containing the i-kernel K_i of each maximal parabolic subgroup G_i of C. The i-kernel of the subgroup G_i is the subgroup consisting of all the elements of G_i that fix all the elements of the residue of G_i.

`Quotient(C, K)`

Given a coset geometry $C = (G; (G_i)_{i \in I})$ and a permutation group K, return the coset geometry $(G/K; (G_i/K)_{i \in I})$ provided that K is a normal subgroup of G and of all the maximal parabolic subgroups of C.

142.4 Conversion Functions

In this section we describe the functions that are available to convert incidence geometries and coset geometries in other objects.

`IncidenceGeometry(C)`

Construct the incidence geometry IG from the coset geometry C. This is done using Tits' algorithm described in the introduction of this chapter . The function returns one value: the incidence geometry IG.

`CosetGeometry(D)`

Convert the incidence geometry D into a coset geometry.

If D is an incidence geometry that can be converted into a coset geometry, the coset geometry isomorphic to it is constructed in the following way. The group G of the coset geometry CG is the automorphism group of D. MAGMA determines a chamber C of D, that is a clique of the incidence graph of D containing one element of each type. To every element x in C, MAGMA associates a subgroup G_x which is the stabilizer of x in G. The subgroups $(G_x, x \in C)$ are the maximal parabolic subgroups of CG. In order to obtain a coset geometry combinatorially isomorphic to the incidence geometry we started with, the group G must be transitive on every rank two truncation of D. If this condition is satisfied, the function returns a boolean set to the value `true` and the coset geometry CG. Otherwise, the function returns `false`.

`Graph(D)`

If `IsGraph(D)` returns `true`, this function construct the undirected graph corresponding to the incidence geometry D.

`Graph(C)`

If `IsGraph(C)` returns `true`, this function construct the undirected graph corresponding to the coset geometry C.

Example H142E8

Taking back the last example for incidence geometries, we can convert the Neumai er geometry into a coset geometry by typing the following command (`neumaier` is the Neumaier geometry constructed above):

```
> ok,cg := CosetGeometry(neumaier);
> ok;
true
```

This means the conversion has been done successfully. So *cg* is the coset geome try corresponding to `neumaier`.

```
> cg;
Coset geometry cg with 4 types
Group:
```

```
Permutation group acting on a set of cardinality 200
Order = 126000 = 2^4 * 3^2 * 5^3 * 7
Maximal Parabolic Subgroups:
Permutation group acting on a set of cardinality 200
Order = 2520 = 2^3 * 3^2 * 5 * 7
Permutation group acting on a set of cardinality 200
Order = 2520 = 2^3 * 3^2 * 5 * 7
Permutation group acting on a set of cardinality 200
Order = 2520 = 2^3 * 3^2 * 5 * 7
Permutation group acting on a set of cardinality 200
Order = 2520 = 2^3 * 3^2 * 5 * 7
Type Set:
{@ 1, 2, 3, 4 @}
```

142.5 Residues

Let $\Gamma(X, \sim, t, I)$ be an incidence geometry and let F be a flag of Γ (i.e. a clique of the incidence graph of Γ).

We say that an element $x \in F$ is incident to the flag F if and only if x is incident to all elements in F, and we denote it $x \sim F$.

The *residue* Γ_F of the flag F in Γ is the geometry whose set of elements is $\{x \in X : x \sim F\} \backslash F$ and whose set of types is $I \backslash t(F)$, together with the restricted type function and incidence relation.

Let $\Gamma(G; (G_i)_{i \in I})$ be a coset geometry and assume that G acts flag–transitively on Γ. Let F be a flag of Γ. The *residue* of F is the coset geometry $\Gamma_F = \Gamma(\cap_{j \in F} G_j; (G_i \cap (\cap_{j \in F} G_j))_{i \in I \backslash t(F)})$.

`Residue(D, f)`

Given an incidence geometry D and a flag f of D, return the residue of the flag f as an incidence geometry.

`Residue(C, f)`

Given a coset geometry C and a subset f of the set of types of C, return the residue of the flag consisting in the maximal parabolics of C whose type is in f.

142.6 Truncations

Let $\Gamma(X, \sim, t, I)$ be an incidence geometry and let J be a subset of I. Then the *J–truncation* of Γ is the geometry whose set of elements is $t^{-1}(J)$, together with the restricted type function and incidence relation.

Let $J \subseteq I$. The *J–truncation* of the coset geometry $\Gamma(G; (G_i)_{i \in I})$ is the coset geometry $\Gamma(G; (G_j)_{j \in J})$.

`Truncation(D, t)`

Given an incidence geometry D and t a subset of the set of types of D, return the *t–truncation* of D as an incidence geometry.

`Truncation(C, t)`

Given a coset geometry C and t a subset of the set of types of C, return the *t–truncation* of C as a coset geometry.

142.7 Shadows

Let $\Gamma(X, \sim, t, I)$ be an incidence geometry. Let J be a subset of I and let F be a flag of Γ such that $t(F) \cap J = \emptyset$. The *J-shadow* of the flag F, denoted by $\sigma_J(F)$, is the set of flags of type J in the residue of the flag F.

`Shadow(D, I, F)`

Given an incidence geometry D, a subset I of the set of types of D and a flag F of D, return the I-shadow of the flag F as an indexed set of subsets of points of D.

142.8 Shadow Spaces

Let $\Gamma(X, \sim, t, I)$ be an incidence geometry. Let J be a subset of I. The *shadow space* $\Gamma(J)$ of Γ is the incidence structure whose point-set is the set of flags of type J of Γ and whose blocks are the $J-$*shadows* of the flags of Γ.

`ShadowSpace(D,I)`

Given an incidence geometry D and a subset I of the set of types of D, return the shadow space $D(I)$ as an incidence structure.

142.9 Automorphism Group and Correlation Group

These function are currently only available for incidence geometries.

An *automorphism* α of an incidence geometry $\Gamma(X, \sim, t, I)$ is an automorphism of the incidence graph of Γ such that for all $x \in X$, $t(\alpha(x)) = t(x)$. In other words, an automorphism cannot change the type of an element. The *automorphism group* of Γ, denoted $Aut(\Gamma)$, is the group of all automorphisms of Γ.

A *correlation* α of an incidence geometry $\Gamma(X, \sim, t, I)$ is an automorphism of the incidence graph of Γ such that for all $x, y \in X$, $t(x) = t(y) \Rightarrow t(\alpha(x)) = t(\alpha(y))$. The *correlation group* of Γ, denoted $Cor(\Gamma)$, is the group of all correlations of Γ.

It is obvious that $Aut(\Gamma)$ is a subgroup of $Cor(\Gamma)$.

For an incidence geometry Γ, we can compute $Aut(\Gamma)$ and $Cor(\Gamma)$ using the commands described below.

`AutomorphismGroup(D)`

Given an incidence geometry D, return the group of type–preserving automorphisms of D as a permutation group of type `GrpPerm` acting on the set of elements of D.

`CorrelationGroup(D)`

Given an incidence geometry D, return the group of automorphisms of D as a permutation group of type `GrpPerm` acting of the set on elements of D.

142.10 Properties of Incidence Geometries and Coset Geometries

Let us recall definitions of the properties described in this section.

An incidence geometry Γ is *flag–transitive* if for every two flags x, y of the same type of Γ, there exists an element g of $Aut(\Gamma)$ such that $g(x) = y$. We also say that $Aut(\Gamma)$ acts flag–transitively in this case.

Moreover, it is a flag–transitive geometry if it contains at least one chamber.

A coset geometry $\Gamma(G; (G_i)_{i \in I})$ is *flag–transitive* if for every two flags x, y of the same type of Γ, there exists an element g of G such that $g(x) = y$. It is then a flag–transitive geometry since the set $\{(G_i)_{i \in I}\}$ is a chamber of Γ.

`IsFTGeometry(D)`

Given an incidence geometry D, return `true` if and only if the automorphism group of D acts flag–transitively on D and D has at least one chamber.

`IsFTGeometry(C)`

Given a coset geometry C, return `true` if and only if the group of C acts flag–transitively on C.

`IsFirm(X)`

Given either a coset geometry or an incidence geometry X that is flag transitive, return `true` if and only if every flag of X is contained in at least two chambers.

`IsThin(X)`

Given either a coset geometry or an incidence geometry X that is flag transitive, return `true` if and only if every flag of X is contained in exactly two chambers.

`IsThick(X)`

Given either a coset geometry or an incidence geometry X that is flag transitive, return `true` if and only if every flag of the geometry is contained in exactly three chambers.

`IsResiduallyConnected(X)`

`IsRC(X)`

Given either a coset geometry or an incidence geometry X that is flag transitive, return `true` if and only if every residue of rank at least two of X has a connected incidence graph.

`IsGraph(D)`

Given an incidence geometry D, tests if this incidence geometry corresponds to a graph: D must be of rank two and such that for one of the two types, say e, all elements of this type are incident with exactly two elements of the other type. Elements of type e then correspond to edges of an undirected graph and elements of the other type to the vertices of that graph.

`IsGraph(C)`

Given a coset geometry C, tests if this geometry corresponds to a graph: C must be of rank two and one of the two maximal parabolic subgroups, say G_e, must contain the Borel subgroup as a subgroup of index 2. In that case, the cosets of G_e correspond to edges of a graph and the cosets of the other maximal parabolic subgroup correspond to the vertices of this graph.

142.11 Intersection Properties of Coset Geometries

Let $\Gamma(X, *, t, I)$ be an incidence geometry. Given a type $i \in I$, for any flag F of Γ, we define the *i-shadow* $\sigma_i(F)$ as the set of elements of type i incident with F.

We define the intersection property (IP) as it appears in [Bue79].

(IP) for every type i, the intersection of the i-shadows of a variety x and a flag F is empty or it is the i-shadow of a flag incident to x and F. The same holds on the residues.

In earlier works, the second author, together with other persons, among whom are Francis Buekenhout, Michel Dehon and Philippe Cara, imposed a condition called $(IP)_2$. This condition asks that all rank two residues of Γ satisfy (IP).

If Γ is a geometry of rank n, we could define a property $(IP)_k$ in the following way (for $k = 2, \ldots, n$) as suggested by Francis Buekenhout.

$(IP)_k$ for every residue R of rank k of Γ, for every type i in the set of types of R, the intersection of the i-shadows of a variety x and a flag F is empty or it is the i-shadow of a flag incident to x and F.

In [JL04], Pascale Jacobs and Dimitri Leemans have designed good algorithms to test these intersection properties. The Magma implementation is available with the following functions.

`HasIntersectionPropertyN(C,n)`

Given a coset geometry C and a positive integer n, this function determines firstly, whether C satisfies the intersection property of rank n, and secondly, whether C satisfies the weak intersection property of rank n. A boolean value corresponding to each of these cases is returned.

`HasIntersectionProperty(C)`

Given a coset geometry C, this function returns the boolean value `true` if and only if C satisfies the intersection property.

`HasWeakIntersectionProperty(C)`

Given a coset geometry C, this function returns the boolean value `true` if and only if C satisfies the weak intersection property.

142.12 Primitivity Properties on Coset Geometries

`IsPrimitive(C)`

`IsPRI(C)`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is a primitive geometry, i.e. all of its maximal parabolic subgroups are maximal subgroups of its group.

`IsWeaklyPrimitive(C)`

`IsWPRI(C)`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is a weakly primitive geometry, i.e. at least one of its maximal parabolic subgroups is a maximal subgroup of its group.

`IsResiduallyPrimitive(C)`

`IsRPRI(C)`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is a primitive geometry and all of its residues are as well.

`IsResiduallyWealyPrimitive(C)`

`IsRWPRI(C)`

`IsRWP(C)`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is a weakly primitive geometry and all of its residues are as well.

`IsLocallyTwoTransitive(C)`

`Is2T1(C)`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is locally two-transitive, i.e. all of its minimal parabolic subgroups have a two-transitive action on the cosets of the Borel subgroup.

142.13 Diagram of an Incidence Geometry

As Francis Buekenhout defined it in [Bue79], the diagram of a firm, residually connected, flag–transitive geometry Γ is a complete graph K, whose vertices are the elements of the set of types I of Γ, provided with some additional structure which is further described as follows. To each vertex $i \in I$, we attach the order s_i which is $\mid \Gamma_F \mid -1$ where F is any flag of type $I\backslash\{i\}$, and the number n_i of elements of type i of Γ. To every edge $\{i, j\}$ of K, we associate three positive integers d_{ij}, g_{ij} and d_{ji} where g_{ij} (the gonality) is equal to half the girth of the incidence graph of a residue Γ_F of type $\{i, j\}$, and d_{ij} (resp. d_{ji}), the *i–diameter* (resp. *j–diameter*) is the greatest distance from some fixed i–element (resp. j–element) to any other element in Γ_F.

On a picture of the diagram, this structure will often be depicted as follows.

```
i  d_ij  g_ij  d_ji  j
O--------------------O
s_i-1              s_j-1
n_i                  n_j
```

Moreover, when $g_{ij} = d_{ij} = d_{ji} = n \neq 2$, we only write g_{ij} and if $n = 2$, we do not draw the edge $\{i, j\}$.

`Diagram(D)`

`Diagram(C)`

Given a firm, residually connected and flag–transitive incidence geometry D or a coset geometry C, return a complete graph K whose vertices and edges are labelled in the following way. Every vertex i of K is labelled with a sequence of two positive integers, the first one being s_i and the second one being the number of elements of type i in D. Every edge $\{i, j\}$ is labelled with a sequence of three positive integers which are respectively d_{ij}, g_{ij} and d_{ji}.

Observe that both these functions do the same but the second one works much faster than the first one since it uses groups to compute the parameters of the diagram. So when the user has to compute the diagram of an incidence geometry, it is strongly advised that he first converts it into a coset geometry and then computes the diagram of the corresponding coset geometry.

Example H142E9

Back to the Petersen graph : let us now explore a little more the first incidence geometry we constructed. Suppose that *ig* is the rank two incidence geometry corresponding to the Petersen graph. We may test whether it is firm, residually connected and flag–transitive.

```
> IsFirm(ig);
true
> IsRC(ig);
true
> IsFTGeometry(ig);
true
```

Since it satisfies these three properties, we can compute its diagram.

```
> d, v, e := Diagram(ig);
> d;
Graph
Vertex  Neighbours
1       2 ;
2       1 ;
> for x in v do print x, Label(x);end for;
1 [ 1, 10 ]
2 [ 2, 15 ]
> for x in e do print x, Label(x);end for;
{1, 2} [ 5, 5, 6 ]
```

We thus see that the diagram of *ig* can be drawn as follows.

```
1      5  5  6      2
0-------------------0
1                   2
10                 15
```

Example H142E10

Back to the cube: let *ig* be the rank three incidence geometry of the cube as constructed above and compute its diagram.

```
> cube := IncidenceGeometry(g);
> d, v, e := Diagram(cube);
> d;
Graph
Vertex  Neighbours
1       2 3 ;
2       1 3 ;
3       1 2 ;
> for x in v do x, Label(x); end for;
1 [ 1, 8 ]
2 [ 1, 12 ]
3 [ 1, 6 ]
```

```
> for x in e do x, Label(x); end for;
{1, 2} [ 4, 4, 4 ]
{1, 3} [ 2, 2, 2 ]
{2, 3} [ 3, 3, 3 ]
```

So the diagram can be depicted as follows.

```
1         4         2         3         3
0-------------------0-------------------0
1                   1                   1
8                  12                   6
```

Example H142E11

Back to the Hoffman-Singleton graph:

```
> ig := IncidenceGeometry(HoffmanSingletonGraph());
> d, v, e := Diagram(ig); d;
Graph
Vertex  Neighbours
1       2 ;
2       1 ;
> for x in v do print x, Label(x); end for;
1 [ 1, 50 ]
2 [ 6, 175 ]
> for x in e do print x, Label(x); end for;
{1, 2} [ 5, 5, 6 ]
```

So the diagram can be depicted as follows.

```
1      5  5  6      2
0-------------------0
1                   6
50                175
```

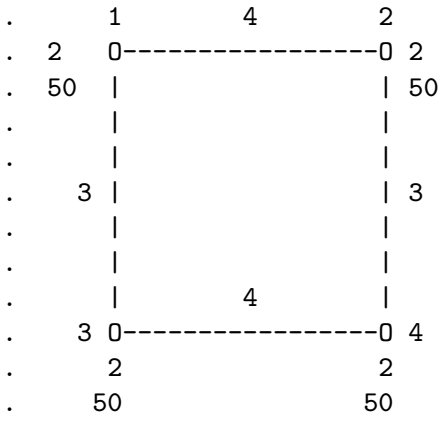
Example H142E12

Back to Neumaier's geometry : let *ig* be the rank four incidence geometry of Neumaier, as described above and compute its diagram.

```
>  d, v, e := Diagram(neumaier);
>  d;
Graph
Vertex  Neighbours
1       2 3 4 ;
2       1 3 4 ;
3       1 2 4 ;
4       1 2 3 ;
>  for x in v do print x, Label(x); end for;
1 [ 2, 50 ]
```

```
2 [ 2, 50 ]
3 [ 2, 50 ]
4 [ 2, 50 ]
>  for x in e do print x, Label(x); end for;
{1, 2} [ 4, 4, 4 ]
{1, 3} [ 2, 2, 2 ]
{1, 4} [ 3, 3, 3 ]
{2, 3} [ 3, 3, 3 ]
{2, 4} [ 2, 2, 2 ]
{3, 4} [ 4, 4, 4 ]
```

So the diagram can be depicted as follows.



142.14 Bibliography

[**BCon**] Francis Buekenhout and Arjeh M. Cohen. *Diagram geometry.* In preparation.

[**Bue79**] Francis Buekenhout. Diagrams for geometries and groups. *J. Combin. Theory Ser. A*, 27(2):121–151, 1979.

[**Bue95**] Francis Buekenhout. *Handbook of incidence geometry.* North-Holland, Amsterdam, 1995.

[**JL04**] Pascale Jacobs and Dimitri Leemans. An algorithmic analysis of the intersection property. *LMS J. Comput. Math.*, 7:284–299 (electronic), 2004.

[**Pas94**] Antonio Pasini. *Diagram geometries.* Oxford Science Publications. The Clarendon Press Oxford University Press, New York, 1994.

[**Tit62**] Jacques Tits. Géométries polyédriques et groupes simples. *Atti 2a Riunione Groupem. Math. Express. Lat. Firenze*, pages 66–88, 1962.

143 CONVEX POLYTOPES AND POLYHEDRA

Chapter 143
CONVEX POLYTOPES AND POLYHEDRA

143.1 Introduction and First Examples

This is a package with tools to build and analyse finite-dimensional, finitely-generated, rational polyhedra. In general a polyhedron is the Minkowski sum of a rational polytope (the compact, convex hull of finitely many rational points) and a cone (the convex hull of finitely many rational half-lines emanating from the origin).

It is important to note that we are discussing only *rational* polyhedra: their vertices lie in in some finite-dimensional rational vector space $L = \mathbf{Q}^n$ with the supporting hyperplanes represented in its dual (loosely speaking, all faces have rational gradients). There is a natural lattice in this setup: the integral sublattice $\mathbf{Z}^n$ of L. This is a lattice only in the sense of being a $\mathbf{Z}$-module; it does not come with a preferred definite quadratic form (or norm), and so, in particular, lengths of vectors are not defined. (We explain below how volumes are nevertheless defined relative to the integral lattice.) A point of L is called 'integral' if it lies in the integral sublattice, but there is no requirement that vertices of polyhedra be integral. Notwithstanding all these remarks, we refer to such ambient spaces L as 'lattices'.

This chapter is organised so that functions applying to compact polyhedra, namely polytopes, are collected together in coherent blocks as much as possible. Section 143.2 lists various ways of constructing polytopes, cones and polyhedra, including Minkowski sum and other arithmetical and set-theoretical operations. Section 117.10.2 presents the basic combinatorics associated to all polyhedra such as recovering their vertices or faces. The next section, Section 117.10.2, is dedicated to polytopes. Most of the functions described here rely on compactness; they include enumeration of points and triangulation. Then Section 117.10.2 details functions that apply to polyhedra more generally, although this includes some functions that apply to cones in particular: computing the integral generators of $C \cap \mathbf{Z}^n$ as a semigroup, for example, where C is a cone in L. Finally, there is section outlining the functions that operate on the ambient lattices. These functions are not usually required, since for the most part the ambient space is a book-keeping device that the package handles automatically, but they arise as domains and codomains of maps. This provides a mechanism for modifying the integral sublattice, which can be used to change the underlying notion of which points count as integral.

Before starting, we outline the capabilities of the package by extended examples. The first illustrates the basic machinery available for the analysis of polytopes—that is, the compact convex hulls of finitely many points.

Example H143E1

A polytope can be constructed as the convex hull of finitely many points; the points can be denoted simply as sequences of integers (or even rational numbers), although they will be cast as

points of an ambient lattice (that is, a $\mathbf{Q}$-vector space marked with a spanning lattice $\mathbf{Z}^n$). For example, we build a polytope P as the convex hull of the four points $(1,0,0)$, $(0,1,0)$, $(1,-3,5)$ and $(-2,2,-5)$.

```
> P := Polytope([ [0,0,0], [1,0,0], [0,1,0], [1,-3,5], [-2,2,-5] ]);
> P;
3-dimensional polytope P with 5 generators:
    ( 0,  0,  0),
    ( 1,  0,  0),
    ( 0,  1,  0),
    ( 1, -3,  5),
    (-2,  2, -5)
```

No analysis of P has been carried out in its construction. In fact, one of its defining 'generators' is not required, as can be seen from the vertices of P (which after this calculation will set as the default printing information).

```
> Vertices(P);
[
    (1, 0, 0),
    (0, 1, 0),
    (1, -3, 5),
    (-2, 2, -5)
]
> P;
3-dimensional polytope P with 4 vertices:
    ( 1,  0,  0),
    ( 0,  1,  0),
    ( 1, -3,  5),
    (-2,  2, -5)
```

One can extract the combinatorial components of P. Here we recover the facets of P, its faces of codimension 1, each of which is regarded as a polytope in its own right.

```
> Facets(P);
[
    2-dimensional polytope with 3 vertices:
        ( 1, -3,  5),
        (-2,  2, -5),
        ( 0,  1,  0),
    2-dimensional polytope with 3 vertices:
        ( 1, -3,  5),
        (-2,  2, -5),
        ( 1,  0,  0),
    2-dimensional polytope with 3 vertices:
        (1, -3,  5),
        (0,  1,  0),
        (1,  0,  0),
    2-dimensional polytope with 3 vertices:
        (-2,  2, -5),
```

```
        ( 0,  1,  0),
        ( 1,  0,  0)
]
```

Computing the (integral) points contained in a polytope (including points on its boundary) is one of the principal operations for polytopes.

```
> Points(P);
[
    (-2, 2, -5),
    (0, 0, 0),
    (0, 1, 0),
    (1, -3, 5),
    (1, 0, 0)
]
```

The interior points or boundary points can be retrieved separately using `InteriorPoints(P)` and `BoundaryPoint(P)`.
One can also compute the volume of a polytope.

```
> Volume(P);
20
```

The volume is the lattice-normalised volume: $\mathrm{Vol}(P) = n! \times \mathrm{vol}(P)$, where $\mathrm{vol}(P)$ is the Euclidean volume of P.
The polar, or dual, to P can be computed.

```
> D := Polar(P);
> D;
3-dimensional polytope D with 4 vertices:
    ( 3,  -1, -7/5),
    (-1,   3,  9/5),
    (-1,  -1,  1/5),
    (-1,  -1, -3/5)
```

The polar of P is again a polytope in this case (simply because the origin lies in the strict interior of P), although its vertices need not be integral.
The Ehrhart series,

$$\mathrm{Ehr}(D) = \Sigma_{n \geq 0} \# \left(nD \cap \mathbf{Z}^n \right) t^n,$$

computes the number of points in multiples of a polytope.

```
> EhrhartCoefficients(D,10);
[ 1, 7, 33, 91, 193, 355, 585, 899, 1309, 1827, 2469 ]
```

The (infinite) Ehrhart series can be recovered as a rational function.

```
> E<t> := EhrhartSeries(D);
> E;
(t^7 + 4*t^6 + 15*t^5 + 12*t^4 + 12*t^3 + 15*t^2 + 4*t + 1)/(t^8 - 3*t^7 + 3*t^6
    - t^5 - t^3 + 3*t^2 - 3*t + 1)
```

It is possible to make maps of the underlying lattices and apply them to polyhedra.

```
> L := Ambient(P);
```

```
> K := Quotient(L![1,2,3]);
> K;
2-dimensional toric lattice K = (Z^3) / <1, 2, 3>
> f := LatticeMap(L, K, Matrix(3,2,[1,0,0,1,0,0]));
> Q := Image(f,P);
> Vertices(Q);
[
    (1, 0),
    (0, 1),
    (1, -3),
    (-2, 2)
]
```

In this case the image polytope $Q = f(P)$ is not a simplex. We can triangulate it.

```
> Triangulation(Q);
{
    2-dimensional polytope with 3 vertices:
        (1, 0),
        (1, -3),
        (0, 1)
    2-dimensional polytope with 3 vertices:
        (1, -3),
        (-2, 2),
        (0, 1)
}
```

Our second example illustrates the machinery available for cones and polyhedra more generally.

Example H143E2

A ray is a rational half-line $\mathbf{Q}^+v$ from the origin passing through a point $v \in \mathbf{Q}^n$. A cone is the convex hull of finitely many rays.

```
> C := Cone([[2,2],[1,1/2]]);
> C;
Cone C with 2 generators:
    ( 2,   2),
    ( 1, 1/2)
```

Although any non-zero point on a ray is enough to define it, often the non-zero integral point nearest the origin is preferred. As with polytopes, there is no analysis of a cone C on its construction, but the minimal generators are displayed once some function has forced their calculation.

```
> Rays(C);
[
    (1, 1),
    (2, 1)
]
```

```
> C;
2-dimensional simplicial cone C with 2 minimal generators:
    (1, 1),
    (2, 1)
```

A polyhedron is the Minkowski sum (simply denoted by a plus sign) of a polytope and a cone.

```
> P := StandardSimplex(2);
> P + C;
2-dimensional polyhedron with 2-dimensional finite part with 3
vertices:
    (1, 0),
    (0, 1),
    (0, 0)
and 2-dimensional infinite part given by a cone with 2 minimal
generators:
    (2, 1),
    (1, 1)
```

The polyhedron $P + C$ is now subject to standard combinatorial analysis, such as computing its facets; some of these may be (compact) polytopes, while others may themselves be polyhedra described as the sum of a compact, or finite, part and a cone, or infinite part.

```
> Facets(P + C);
[
    1-dimensional polytope with 2 vertices:
        (1, 0),
        (0, 0),
    1-dimensional polyhedron with 0-dimensional finite part with one
    vertex:
        (0, 1)
    and 1-dimensional infinite part given by a cone with one minimal
    generator:
        (1, 1),
    1-dimensional polytope with 2 vertices:
        (0, 1),
        (0, 0),
    1-dimensional polyhedron with 0-dimensional finite part with one
    vertex:
        (1, 0)
    and 1-dimensional infinite part given by a cone with one minimal
    generator:
        (2, 1)
]
```

143.2 Polytopes, Cones and Polyhedra

A *(rational) polytope* (in a lattice $L = \mathbf{Q}^n$) is the convex hull of finitely many points in L. (The points do not need to be integral points.)

A *cone* C (in a lattice L) is the convex hull of finitely many rays (or zero). Precisely, a *ray* is a half line emanating from the origin, but, as is common, we treat rays synonymously with the first integral lattice point on the ray outside the origin—thus, for example, an intrinsic that returns a ray will actually return an primitive integral point of the ambient lattice. Rays that are the intersection of a linear hyperplane with C are called *extreme rays* of C—these are the corner edges of C. A cone is *regular* if it is generated (as a semigroup in the $\mathbf{Z}$-module $\mathbf{Z}^n \subset L$) by its extreme rays (in other words, by the integral vectors on its extreme rays).

Combining all concepts so far, a *(rational) polyhedron* is the Minkowski sum of a polytope and a cone—the latter is the *tail cone* and is unique; the smallest possible polytope is unique but seems not to be used in the theory. We also use the expressions *compact part* and *infinite part* to indicate the polytope and the cone used to define a polyhedron.

143.2.1 Polytopes

`Polytope(Q)`

The polytope defined by taking the convex hull of the sequence of points Q, where the points can be specified as sequences of integers of rational numbers (of some fixed length) or as points of a common lattice.

`RandomPolytope(L,n,k)`

`RandomPolytope(d,n,k)`

A polytope in the toric lattice L (or a toric lattice of dimension d) generated by n random integral points with coefficients of modulus at most the non-negative integer k.

`Polar(P)`

The polar dual polyhedron to the polyhedron P, namely

$$P^\star = \{u \in L^\vee \mid u \cdot v \geq -1 \ \forall v \in P\}\,.$$

`CrossPolytope(L)`

`CrossPolytope(d)`

The maximum dimensional cross-polytope in the lattice L, or of dimension d for an integer d.

`StandardSimplex(L)`

`StandardSimplex(d)`

The simplex given by the standard basis and the origin of the lattice L, or the d-dimensional simplex given by the standard basis and the origin for an integer d.

`CyclicPolytope(L,n)`

`CyclicPolytope(d,n)`

The cyclic polytope generated by n points in the lattice L, or by n points in d-dimensional space for an integer d.

143.2.2 Cones

Cones in lattices are of type `TorCon`.

`Cone(A)`

The cone in a lattice generated by a sequence A of its elements (or simply of sequences of integer or rational coefficients).

`Cone(v)`

The cone in a lattice generated by a single element v of that lattice; namely the single ray $\mathbf{Q}^+ v$.

`ConeWithInequalities(B)`

The cone in a lattice L defined by the set B of elements of the dual lattice $L^\vee$ (or simply by a set of sequences of integer or rational coefficients). The cone is the intersection of half-spaces $v \cdot u \geq 0$ as v runs through B:

$$\bigcap_{v \in B} \{u \in L \mid v \cdot u \geq 0\}\,.$$

`FullCone(L)`

`FullCone(n)`

The cone which is the entire lattice L, or the lattice of dimension n for a positive integer n.

`PositiveQuadrant(L)`

`PositiveQuadrant(n)`

The cone in the lattice L generated by the standard basis vectors of L, or that of the lattice of dimension n for a positive integer n.

`ZeroCone(L)`

`ZeroCone(n)`

The cone which consists of only the origin of the lattice L, or of the lattice of dimension n for a positive integer n.

`Dual(C)`

The dual cone to the cone C, namely

$$C^\vee = \{u \in L^\vee \mid v \cdot u \geq 0 \text{ for all } v \in C\}\,.$$

where C lies in the lattice L.

`NormalisedCone(P)`

A cone C such that the polyhedron P is the intersection of C with a hyperplane at height one, together with the embedding of the ambient lattice of P into the ambient lattice of C.

`ConeInSublattice(C)`

The cone of maximal dimension given by the intersection of the cone C with its linear span. Also gives the embedding of the sublattice in the ambient toric lattice.

`ConeQuotientByLinearSubspace(C)`

The strictly convex cone given by the quotient of the cone C by its maximal linear subspace. Also gives the quotient map.

143.2.3 Polyhedra

Polytopes and polyhedra in lattices are of type `TorPol`.

`Polyhedron(C,H,h)`

`Polyhedron(C,H,h)`

The polyhedron constructed as the slice of the cone C by the hyperplane determined by the primitive dual vector H at height given by the integer or rational number h.

`Polyhedron(C)`

`level` RngIntElt *Default :* 1

The polyhedron arising as the intersection of the cone C with the hyperplane at height one (can be changed via 'level').

`HalfspaceToPolyhedron(v,h)`

`HalfspaceToPolyhedron(Q,h)`

The halfspace $\{u|v \cdot u \geq h\}$ as a polyhedron, where v is a point of a toric lattice (or a sequence of integral or rational numbers that are its coefficients) and $h \in \mathbf{Q}$.

`HyperplaneToPolyhedron(v,h)`

`HyperplaneToPolyhedron(Q,h)`

The hyperplane $\{u|v \cdot u = h\}$ as a polyhedron, where v is a point of a toric lattice (or a sequence of integral or rational numbers that are its coefficients) and $h \in \mathbf{Q}$.

`Polyhedron(C,f,v)`

The polyhedron arising as the preimage of the cone C under the affine map $f + v$.

`EmptyPolyhedron(L)`

The empty polyhedron in the toric lattice L.

`ConeToPolyhedron(C)`

The cone C as a polyhedron.

`PolyhedronInSublattice(P)`

The polyhedron of maximal dimension given by the intersection of the toric lattice containing the polyhedron P with an affine sublattice of dimension equal to the dimension of P. Also gives the affine embedding as a lattice embedding and translation.

`FixedSubspaceToPolyhedron(G)`

`FixedSubspaceToPolyhedron(L,G)`

The subspace (realised as a polyhedron) fixed by the action of the matrix group G on the toric lattice L. G should be a subgroup of either $GL(n, \mathbf{Z})$ or of $SL(n, \mathbf{Z})$, where n is the dimension of L.

Example H143E3

We construct some polyhedra by taking slices of the positive quadrant in 3-space.

```
> C := PositiveQuadrant(3);
> C;
3-dimensional simplicial cone C with 3 minimal generators:
    (1, 0, 0),
    (0, 1, 0),
    (0, 0, 1)
```

We slice using dual vectors, so we recover the dual lattice M to the ambient lattice of C. Some slices are compact, some are not.

```
> M := Dual(Ambient(C));
> P := Polyhedron(C, M ! [1,2,3], 1);
> IsPolytope(P);
true
> Q := Polyhedron(C, M ! [1,-2,3], 1 );
> IsPolytope(Q);
false
> Q;
2-dimensional polyhedron Q with 1-dimensional finite part with 2 vertices:
    ( 1,   -1),
    ( 0, -1/3)
and 2-dimensional infinite part given by a cone with 2 minimal generators:
```

```
    ( 2,   -1),
    ( 0,    1)
```

We can change the level, or height, at which the slice is taken.

```
> R := Polyhedron(C, M ! [1,2,3], -5);
> R;
Empty polyhedron
```

143.2.4 Arithmetic Operations on Polyhedra

`C eq D`

Return `true` if and only if the cones C and D are equal, that is, they lie in the same toric lattice and have identical supports.

`P eq Q`

Return `true` if and only if the polyhedra P and Q are equal, that is, they lie in the same toric lattice and have identical supports.

`C meet D`

The intersection of the cones C and D; the cones must lie in the same toric lattice otherwise an error is reported.

`P meet Q`

The intersection of the polyhedra P and Q; the polyhedra must lie in the same toric lattice otherwise an error is reported.

`P subset Q`

Return `true` if and only if the support of the polyhedron P is contained in that of the polyhedron Q; the polyhedra must lie in the same toric lattice otherwise an error is reported.

`C + D`
`P + Q`

The Minkowski sum of the cones C and D or of the polyhedra P and Q.

`P + C`
`C + P`

The polyhedron constructed as the Minkowski sum of the polytope P and the cone C.

`P * Q`

The convex hull of the Cartesian product of the polyhedra P and Q.

`k * P`

The dilation of the polyhedron P by the rational number k.

`-P`

The negation of the polyhedron P.

143.3 Basic Combinatorics of Polytopes and Polyhedra

Cones, polytopes and polyhedra have similar combinatorial features: they are composed of faces of various dimensions that meet in faces of lower dimensions. Some functions apply to all of these geometrical objects, some do not.

A face of a cone C is the intersection of C with an affine hyperplane whose defining equation is non-negative on C. A facet of a cone C is a codimension 1 face of C. Defintions for polytopes and polyhedra are similar.

143.3.1 Vertices and Inequalities

`Vertices(P)`

A sequence containing the vertices of the polyhedron P.

`NumberOfVertices(P)`

The number of vertices of the polyhedron P.

`Rays(C)`

The sequence of generators of the rays of the cone C (returned as primitive lattice points). If C is strictly convex, this is the same as the minimal **R**-generators.

`Ray(C,i)`

The ith ray of the of the cone C (in the order returned by `Rays(C)`).

`LinearSpanEquations(C)`

`LinearSpanEquations(Q)`

A sequence of equations defining the minimal linear subspace containing the cone C (or the sequence of toric lattice points Q).

`LinearSpanGenerators(C)`

`LinearSpanGenerators(Q)`

A sequence of generators of the minimal linear subspace containing the cone C (or the sequence of toric lattice points Q).

`LinearSubspaceGenerators(C)`

A basis of the maximal linear subspace contained in the cone C.

Inequalities(C)

Inequalities(P)

MinimalInequalities(C)

If the cone C or polyhedron P lies in a toric lattice L, then return a finite sequence of vectors in the dual lattice $\check{L}$ which define supporting hyperplanes of C or P. The list is forced to be minimal for `MinimalInequalities`, but might not be otherwise. If the argument is a polyhedron, then an integer k is returned as a second return value, so that the first k inequalities correspond to the facets of P while the remaining cut out the subspace containing P.

Example H143E4

Build a polytope and recover the inequalities that define it: the second return value, 3, says that all three inequalities define facets of P—we see an example below where this is not the case.

```
> P := Polytope([[1,0],[0,1],[-1,-1]]);
> Inequalities(P);
[
    <(2, -1), -1>,
    <(-1, -1), -1>,
    <(-1, 2), -1>
]
3
```

For each inequality use `HalfspaceToPolyhedron(m,h)` to define the corresponding halfspace, and then intersect them all to recover P—and finally recover the vertices we started with.

```
> PP := &meet  [HalfspaceToPolyhedron(H[1],H[2]) : H in Inequalities(P)];
> PP eq P;
true
> PP;
2-dimensional polytope PP with 3 vertices:
    ( 0,  1),
    (-1, -1),
    ( 1,  0)
```

It can happen that a polytope does not span the ambient toric lattice in which it lies and that some of the inequalities are used to cut out the affine subspace that it does span.

```
> Q := Polytope([[1,0,2],[0,1,2],[-1,0,2],[0,-1,2]]);
> Inequalities(Q);
[
    <(1, -1, 0), -1>,
    <(-1, -1, 0), -1>,
    <(-1, 1, 0), -1>,
    <(1, 1, 0), -1>,
    <(0, 0, 1), 2>,
    <(0, 0, -1), -2>
```

```
]
4
```

In this case the final two inequalities are opposites of one another and cut out the affine hyperplane $z = 2$. The second return value 4 indicates that the first 4 inequalities are cutting out facets of the polytope while the remaining inequalities are cutting out the affine hyperplane.

143.3.2 Facets and Faces

`#fVector(C)`
`fVector(P)`

The f-vector of the polyhedron P or cone C.

`#hVector(C)`
`#hVector(P)`

The h-vector of the polyhedron P or cone C.

`Facets(C)`
`Facets(P)`

A sequence containing all facets of the toric cone C or polyhedron P.

`FacetIndices(P)`

A sequence of sets describing the facets of the polytope P. The jth set gives the indices of the vertices of P which define the jth facet of P.

`NumberOfFacets(P)`

The number of facets of the polyhedron P.

`Faces(C)`
`Faces(P)`
`Faces(C,i)`
`Faces(P,i)`

A sequence containing all face cones of the toric cone C or polyhedron P, or only those of dimension i if an integer i is also specified.

`FaceIndices(P,i)`

A sequence of sets describing the i-dimensional faces of the polyhedron P. The jth set gives the indices of the vertices of P which define the jth i-dimensional face.

`Edges(P)`

A sequence containing all the edges of the polyhedron P.

`EdgeIndices(P)`

A sequence of sets describing the edges of the polhedron P. The jth set gives the indices of the vertices of P which define the jth edge of P.

`Graph(P)`

The graph of the face lattice of the polyhedron P. The vertices of the graph are labeled by the dimension of the corresponding face.

`FaceSupportedBy(C,H)`

The face of the toric cone C supported by the toric lattice element H in the dual lattice to the one containing C (so H is a linear form on the ambient lattice of C).

`IsSupportingHyperplane(v,h,P)`

Return `true` if and only if the hyperplane defined by $v \cdot u = h$ is a supporting hyperplane of the polyhedron P, where v is a lattice point of the dual ambient lattice of P and h is a rational number. If so, also gives the sign τ such the hyperplane is a support of P (i.e. τ in $\{-1, 0, +1\}$ such that $\text{Sign}(v \cdot u - h)$ is either 0 or τ for all u in P). If P is contained within the hyperplane, then τ will be 0.

`SupportingCone(P,v)`

The cone C such that $C + v$ is a supporting cone of the polyhedron P, where v is a vertex of P.

143.4 The Combinatorics of Polytopes

143.4.1 Points in Polytopes

Given a polygon, one might want to know how many integral points it contains, and one might want to list them. These are computed by different algorithms (although one can of course list the points and then count them). For point counting we use the methods of Barvinok and Pommersheim ([BP99]), as described in [DLHTY04]. More generally, one might want to know the number of points in integral dilations of a polytope: these numbers are the coefficients of a generating function, the Ehrhart series; this is discussed in a later section.

`Points(P)`

`InteriorPoints(P)`

`BoundaryPoints(P)`

The integral toric lattice points, the strictly interior lattice points, or the boundary lattice points of the polytope P.

`NumberOfPoints(P)`

The number of integral toric lattice points of the polytope P.

143.4.2 Ehrhart Theory of Polytopes

`EhrhartSeries(P)`

The rational generating function of the Ehrhart series for the polytope P.

`EhrhartPolynomial(P)`

A sequence representing the Ehrhart (quasi-)polynomial for the polytope P. That is, a sequence of polynomials $[p_0, \ldots, p_{r-1}]$ of length r, the quasi-period of the Ehrhart polynomial, so that the number of lattice points of the dilation nP is the value of $p_s(k)$ where $n = kr + s$ is the Euclidean division of n by r; in other words, s is the least residue of n modulo r. Note that since MAGMA indexes sequences from 1, we have that $p_i =$ `EhrhartPolynomial(P)[i+1]`.

`EhrhartCoefficients(P,l)`

The first $l + 1$ coefficients of the Ehrhart series for the polytope P (starting with $0P$ up to and including lP).

`EhrhartCoefficient(P,k)`

The number of lattice points in the (non-negative, integral) dilation kP of the polytope P.

143.4.3 Automorphisms of a Polytope

`AutomorphismGroup(P)`

The subgroup of $GL(n, \mathbf{Z})$ (acting on the ambient lattice) which leaves the polyhedron P unchanged.

Example H143E5

```
> P:=CrossPolytope(2);
> P;
2-dimensional polytope P with 4 vertices:
    ( 1,  0),
    ( 0,  1),
    (-1,  0),
    ( 0, -1)
> AutomorphismGroup(P);
MatrixGroup(2, Integer Ring)
Generators:
    [0 1]
    [1 0]
    [ 0  1]
    [-1  0]
```

143.4.4 Operations on Polytopes

`Triangulation(P)`

A sequence of polytopes that make a triangulation of the polytope P.

`TriangulationOfBoundary(P)`

A sequence of polytopes that make a triangulation of the boundary of the polytope P.

143.5 Cones and Polyhedra

143.5.1 Generators of Cones

`ZGenerators(C)`

`level` RngIntElt *Default : infinity*

If C lies in a toric lattice L, then return a finite sequence of elements of L that generate C as a monoid. If specified, the parameter `level` restricts the search for generators to that given level; this assumes that C is graded.

`RGenerators(C)`

`MinimalRGenerators(C)`

If C lies in a toric lattice L, then return a finite sequence of elements of L that generate C over $\mathbf{Q}_+$. The list is forced to be minimal for `MinimalRGenerators`, but might not be otherwise.

`Points(C,H,h)`

The set of integral lattice points in the cone C contained in the hyperplane given by the section determined by the dual lattice element H at height given by the rational number h. The intersection of H with C is required to be compact.

Example H143E6

Here we find the generators of a cone in the sublattice spanned by its defining lattice points. First build some points in a toric lattice and make the cone they generate.

```
> L := ToricLattice(4);
> B := [ L | [1,2,3,-3], [-1,0,1,0], [1,2,1,-2], [2,0,0,-1], [0,0,2,-1] ];
> L1,f := Sublattice(B);
> Dimension(L1);
3
> C := Cone(B);
> Points(Polytope(B));
{
    (1, 2, 1, -2),
    (2, 0, 0, -1),
```

```
    (-1, 0, 1, 0),
    (1, 2, 3, -3),
    (1, 0, 1, -1),
    (0, 0, 2, -1),
    (0, 1, 1, -1),
    (1, 1, 2, -2)
}
> ZGenerators(C);
[
    (-1, 0, 1, 0),
    (0, 1, 1, -1),
    (1, 2, 1, -2),
    (2, 0, 0, -1)
]
```

Using the embedding map f, pull the cone back to the lattice span of C.

```
> C1 := C @@ f;
```

In the smaller lattice, this cone is nonsingular.

```
> IsNonsingular(C1);
true
```

We can find out which points generate the cone in the smaller lattice and compare them with the generators of the original cone.

```
> B1 := ZGenerators(C1);
> B1;
[
    (-1, 0, 1),
    (1, 1, 0),
    (2, 0, -1)
]
> [ Index(B,Image(f,b)) : b in B1 ];
[ 2, 3, 4 ]
```

143.5.2 Properties of Polyhedra

`CompactPart(P)`

The polytope defined by taking the convex hull of the vertices of the polyhedron P. This is the smallest polytope which can occur as a factor in an expression of P as the Minkowski sum of a polytope and a cone.

`IntegralPart(P)`

The polyhedron defined by taking the convex hull of the integral points contained in the polyhedron P.

`InfinitePart(P)`

The tail cone of the polyhedron P.

`IsEmpty(P)`

Return `true` if and only if the polyhedron P is empty.

Example H143E7

One can build a polytope as the convex hull of a finite sequence of points—these may be elements of a toric lattice or simply sequences of integer or rational coefficients.

```
> P := Polytope([[-4,2,1],[0,3,4],[3,1,-3],[3,0,0],[2,-1,1]]);
```

The polar polyhedron comprises dual vectors that evaluate to at least -1 on P. It is compact if and only if the origin lies in the interior of P.

```
> D := Polar(P);
> D;
3-dimensional polyhedron D with 3-dimensional finite part with 5 vertices:
    ( 1/2,      1,    -1),
    (-1/3,    1/2,   1/6),
    (-1/3,   1/21,  -2/7),
    (-1/3,  -3/13, -1/13),
    (1/67, -37/67, 11/67)
and 3-dimensional infinite part given by a cone with 3 minimal generators:
    (   1,      2,     0),
    (   2,      9,     5),
    (   7,      9,    10)
```

Since D is infinite, there is no intrinsic to list its integral points, but one can determine whether it contains integral points at all.

```
> HasIntegralPoint(D);
true
```

Computing the span of all the integral points of D is possible.

```
> IntegralPart(D);
3-dimensional polyhedron with 3-dimensional finite part with 6 vertices:
    (0,  0,  0),
    (0,  1,  0),
    (0,  2,  1),
    (1,  1,  1),
    (1,  2, -1),
    (2,  2,  3)
and 3-dimensional infinite part given by a cone with 3 minimal generators:
    (1,  2,  0),
    (2,  9,  5),
    (7,  9, 10)
```

The compact and infinite parts of D can be recovered.

```
> cD := CompactPart(D);
```

```
> cD;
3-dimensional polytope cD with 5 vertices:
    ( 1/2,      1,    -1),
    (-1/3,    1/2,   1/6),
    (-1/3,   1/21,  -2/7),
    (-1/3,  -3/13, -1/13),
    (1/67, -37/67, 11/67)
```

Polytopes are special cases of polyhedra (those whose infinite tail cone is the zero cone, or equivalently those that are compact), and the distinction can be determined.

```
> IsPolytope(cD);
true
> IsPolytope(D);
false
```

Example H143E8

```
> C:=Cone([[0,1,0],[0,1,1],[1,1,2],[1,1,4]]);
> P:=Polyhedron(C);
> P;
2-dimensional polyhedron P with 1-dimensional finite part with 2
vertices:
    (1, 2),
    (1, 4)
and 2-dimensional infinite part given by a cone with 2 minimal
generators:
    (1, 0),
    (1, 1)

> CC:=Cone([[1,0],[1,1]]);
> QQ:=Polytope([[1,2],[1,4]]);
> PP:=CC + QQ;
> PP;
2-dimensional polyhedron PP with 1-dimensional finite part with 2
vertices:
    (1, 2),
    (1, 4)
and 2-dimensional infinite part given by a cone with 2 minimal
generators:
    (1, 0),
    (1, 1)
```

But there's a potential catch.

```
> PP eq P;
false
> Ambient(PP);
2-dimensional toric lattice Z^2
```

```
> Ambient(P);
2-dimensional toric lattice ker <1, 0, 0>
> P:=ChangeAmbient(P,Ambient(PP));
> PP eq P;
true
```

The ambients are different simply because of the method of construction of the two polyhedra, but they can be forced into the same space.

`IsMaximalDimension(C)`
`IsMaximalDimension(P)`

Return `true` if and only if the cone C or polyhedron P has dimension equal to that of its ambient lattice.

`IsStrictlyConvex(C)`

Return `true` if and only if the cone C is strictly convex; that is, if there exists a hyperplane H such that C is contained on one side of H and C meets H in single point 0.

`IsSimplicial(C)`
`IsSimplicial(P)`

Return `true` if and only if the cone C or the polyhedron P is simplicial.

`IsSimplex(P)`

Return `true` if and only if the polyhedron P is a simplex.

`IsSimple(P)`

Return `true` if and only if the polyhedron P is simple.

`IsAffineLinear(P)`

Return `true` if and only if the polyhedron P is an affine linear space.

`IsZero(C)`

Return `true` if and only if the cone C is supported at the origin of its ambient toric lattice.

143.5.3 Attributes of Polyhedra

`Dimension(C)`
`Dimension(P)`

The dimension of the toric cone C or polyhedron P.

`Index(C)`

The index of the sublattice generated by the minimal $\mathbf{R}$-generators of the cone C in its linear span.

`IsShellable(P)`

Return `true` if and only if the integral polytope P is shellable; i.e. if there exists a polytope S such that each lattice point $u \in P$ lies on the boundary of $v + i * (S - v)$ for some $0 \leq i \leq k$, where v is the barycentre of the vertices of P. If `true`, also returns S, v, and k.

143.5.4 Combinatorics of Polyhedral Complexes

`Ambient(C)`
`Ambient(P)`

The ambient toric lattice of the toric cone C or polyhedron P.

`ChangeAmbient(C,L)`
`ChangeAmbient(P,L)`

Make a cone or polyhedron identical to the toric cone C or polyhedron P but lying in the toric lattice L; the identification of the existing ambient toric lattice with L is simply by identification of the two standard bases and it requires the dimensions of the two spaces to be equal.

143.6 Toric Lattices

One often begins to study toric geometry by considering a 'lattice' $L = \mathbf{Z}^n$ and discussing polygons or cones in its overlying rational (or real) vector space $L_{\mathbf{Q}} = L \otimes \mathbf{Q}$ whose vertices lie on L; these are examples of so-called 'lattice polytopes'. Anybody who has given a seminar on toric geometry will know the constant frustration of distinguishing between L and $L_{\mathbf{Q}}$ (or worse, a bit later in the story, the duals of these). In MAGMA, we bind these two spaces together in a single object, a *toric lattice*. These spaces lie underneath all the combinatorics. It is not often necessary to be explicit about them, since the package handles them invisibly in the background, but they are useful for the usual parent and data typing purposes: when one creates an empty sequence intended to house elements of a toric lattice, it should be properly typed from the outset, for instance. Toric lattices created as duals record that relationship.

Toric lattices are of type `TorLat` and their elements are of type `TorLatElt`.

143.6.1 Toric Lattices

A toric lattice is a finite-dimensional rational vector space with a distinguished free **Z**-module L that spans it: it is the pair $L \otimes \mathbf{Q} \supset L$ where $L \cong \mathbf{Z}^n$. We usually refer to the toric lattice as L, and although the feeling of using L is that it is simply the given **Z**-module, the covering vector space allows fluent working with non-integral points of L.

`ToricLattice(n)`

Create an n-dimensional toric lattice $\mathbf{Q}^n \supset \mathbf{Z}^n$. The integer n must be non-negative.

`ScalarLattice()`

The unique one-dimensional toric lattice of scalars $\mathbf{Q} \supset \mathbf{Z}$.

Example H143E9

A properly typed sequence of points of a toric lattice can be summed with the usual convention.

```
> L := ToricLattice(3);
> nopoints := [ L | ];
> &+ nopoints;
(0, 0, 0)
```

A toric lattice is really a vector space marked with a finitely-generated **Z**-module that spans it. In particular, its points may well have rational coefficients, although those with integral coefficients are distinguished.

```
> somepoints := [ L | [1/2,2/3,3/4], [1,2,3] ];
> [ IsIntegral(v) : v in somepoints ];
[ false, true ]
```

`Dual(L)`

The dual lattice of the toric lattice L.

Example H143E10

The dual of a toric lattice is also a toric lattice: integrality of dual points is clear.

```
> L := ToricLattice(2);
> L;
2-dimensional toric lattice L = Z^2
> M := Dual(L);
> M;
2-dimensional toric lattice M = (Z^2)^*
```

MAGMA's default printing for toric lattices tries to help keep track of which lattice is which—this becomes more useful when working with toric varieties. The relationship between L and M is preserved, and double-dual returns L.

```
> M eq L;
false
```

```
> L eq Dual(M);
true
```

`L + M`

`DirectSum(L,M)`

`DirectSum(Q)`

The direct sum of the two toric lattices L and M. The following natural maps are also returned: the embedding of L into the sum, the embedding of M, the projection of the sum onto L, and the projection onto M.

The argument can also be a sequence Q of toric lattices. In this case, there are three return values: the direct sum of all lattices in Q, a sequence of inclusion maps of lattices in Q into the sum, and a sequence of projection maps from the sum onto each of the elements of Q in turn.

`L ^ n`

The direct sum of the toric lattice L with itself n times. Also return are a sequence of injections of L as factors into the sum and a sequence of projections of the sum onto its factors. This is identical to `DirectSum([L,L,...,L])` where the sequence has length n.

`Dimension(L)`

The dimension of the toric lattice L.

143.6.2 Points of Toric Lattices

Points of L are interpreted to mean points of $L \otimes \mathbf{Q}$. If the coefficients of a point are in fact integral, then this is recognised: so it is possible to tell whether a point is in fact a point of L, not just the rational span.

The usual rational vector space arithmetic operates on these lattice points.

`L ! [a,b,...]`

`LatticeVector(L,Q)`

`LatticeVector(Q)`

The point $(a, b, \ldots)$ as an element of the toric lattice L, where $Q = [a, b, \ldots]$ is a sequence of rational numbers or integers. (If the toric lattice L is omitted, then it will be created.)

`L . i`

`Basis(L,i)`

The ith standard basis element of the toric lattice L.

`Basis(L)`

The standard basis element of the toric lattice L as a sequence of points of L.

`Form(L,Q)`

The point $(a, b, \ldots)$ as an element of the dual of the toric lattice L, where $Q = [a, b, \ldots]$ is a sequence of rational numbers or integers.

`Zero(L)`

The zero vector in the toric lattice L.

`P + Q`
`P - Q`
`n * P`
`P / n`

The sum and difference of toric lattice points P and Q (or points of the dual), the rational multiple nP and quotient P/n, where $n \in \mathbf{Q}$.

`P eq Q`

Return `true` if and only if the toric lattice points P and Q are the same point of the same lattice.

`AreProportional(P,Q)`

Return `true` if and only if the toric lattice points P and Q are rational multiples of one another; the factor Q/P is returned in that case.

`P / Q`

The rational factor P/Q in the case that the toric lattice points P and Q are rational multiples of one another.

Example H143E11

This simple example builds some points in a toric lattice and performs arithmetic on them.

```
> L := ToricLattice(3);
> a := L ! [1,2,3];
> a;
(1, 2, 3)
> L eq Parent(a);
true
> b := L ! [1/2,1,3/2];
> a + b;
(3/2, 3, 9/2)
> a eq b;
false
> a eq 2*b;
true
```

```
> b/a;
1/2
```

`v in L`

Return `true` if and only if the toric lattice point v lies in the toric lattice L.

`IsZero(v)`

Return `true` if and only if the toric lattice point v is the zero vector.

`IsIntegral(v)`

Return `true` if and only if the coefficients of the toric lattice point v are integral.

`IsPrimitive(v)`

Return `true` if and only if the toric lattice point v is a primitive integral vector; that is, v is not divisible as an integral vector in its ambient toric lattice.

`PrimitiveLatticeVector(v)`

The first toric lattice point on the ray spanned by v.

Example H143E12

A point of a sublattice may be primitive in the sublattice even though it is not primitive in the bigger lattice: treating sublattices as the images of embeddings makes this point transparent. First build a sublattice K of L together with the embedding map.

```
> L := ToricLattice(2);
> K,emb := Sublattice([L | [2,0],[0,2]]);
```

Construct a point in L that is clearly not primitive.

```
> vL := L ! [2,2];
> IsPrimitive(vL);
false
```

Pulling this point back to K shows that it is obviously primitive in K.

```
> vK := vL @@ emb;
> vK;
(1, 1)
> IsPrimitive(vK);
true
```

But notice that coercion of the point of L into K is not the same thing: it simply works with the coefficients of the point and gives an answer that is not what is wanted in this case.

```
> K ! vL;
(2, 2)
> IsPrimitive(K ! vL);
false
```

143.6.3 Operations on Toric Lattices

`L eq K`

Return `true` if and only if the toric lattices L and K are the same object in MAGMA (and not merely isomorphic).

`Sublattice(Q)`

A toric lattice L_1 isomorphic to the sublattice of a toric lattice L generated by the sequence Q of elements of L together with an embedding map of L_1 in L. The MAGMA subobject constructor `sub< L | Q >` can also be used, although the sequence Q must be a sequence of elements of L and cannot be interpreted more broadly.

`ToricLattice(Q)`

The toric sublattice of $\mathbf{Q}^n \supset \mathbf{Z}^n$ (regarded as a toric lattice) generated by the sequences of integers of length n of which the sequence Q comprises.

`Quotient(C)`
`Quotient(Q)`
`Quotient(v)`

A toric lattice isomorphic to the toric lattice that is the quotient of a toric lattice L by the linear span of the cone C, or the elements $v \in L$ that comprise the sequence Q or the single primitive vector $v \in L$. The projection map of L to the quotient is also returned.

`AddVectorToLattice(v)`
`AddVectorToLattice(Q)`

A toric lattice L_1 isomorphic to the toric lattice generated by a toric lattice L together with the vector $v \in L$ (or by a sequence of vectors of L). The inclusion map of L in L_1 is also returned.

`IsSublattice(L)`

Return `true` if and only if the toric lattice L was constructed as a sublattice of another toric lattice.

`IsSuperlattice(L)`

Return `true` if and only if the toric lattice L was constructed as a superlattice of another toric lattice (by the addition of a rational vector).

`IsDirectSum(L)`

Return `true` if and only if the toric lattice L was constructed as the direct sum of other toric lattices.

`IsQuotient(L)`

Return `true` if and only if the toric lattice L was constructed as the quotient of another toric lattice.

`Sublattice(L)`

If the toric lattice L was constructed as the extension of a toric lattice K by the addition of a vector, then return K.

`Superlattice(L)`

If the toric lattice L was constructed as a sublattice of a toric lattice K, then return K.

`Summands(L)`

If the toric lattice L was constructed as a direct sum, return a sequence of the toric lattice summands used (and parallel sequences of their embedding maps in L and the projections from L to them).

Example H143E13

Construct a sublattice of a toric lattice L and compute its natural basis in the original coordinates of L; the user does not have control over the choice of coordinates and inclusion map.

```
> L := ToricLattice(2);
> L1,phi1 := Sublattice([L| [1,2],[2,1]]);
> L1;
2-dimensional toric lattice L1 = sub(Z^2)
> phi1(L1.1), phi1(L1.2);
(1, 2)
(0, 3)
```

A similar calculation for a quotient map.

```
> L2,phi2 := Quotient(L ! [3/2,2]);
> L2;
1-dimensional toric lattice L2 = (Z^2) / <3, 4>
> phi2(L.1), phi2(L.2);
(-4)
(3)
```

And again for an extension of L.

```
> L3,phi3 := AddVectorToLattice(L ! [1/5,2/5]);
> L3;
2-dimensional toric lattice L3 = (Z^2) + <1/5, 2/5>
> phi3(L1.1), phi3(L1.2);
(1, 0)
(2, 5)
```

143.6.4 Maps of Toric Lattices

Maps between toric lattices are of type `TorLatMap`. They can be constructed using the `hom< L -> K | M >` constructor, where M is the matrix of the desired linear map with respect to the standard bases of L and K. The usual evaluation operations `@` and `@@` can be applied, although there are image and preimage intrinsics too. Basic arithmetic of maps (sum, difference and scalar multiple, each taken pointwise) is available.

`ZeroMap(L,K)`

The zero map between toric lattices L and K.

`IdentityMap(L)`

The identity map on the toric lattice L.

`hom< L -> K | M >`

`LatticeMap(L,K,M)`

The map between toric lattices L and K determined by the matrix M (with respect to the standard bases of L and M).

`LatticeMap(L,Q)`

The toric lattice map from toric lattice L to the toric lattice M determined by the sequence Q of toric lattice points, where M is the toric lattice containing the points of Q.

`DefiningMatrix(f)`

The defining matrix of the lattice map f.

`Image(f,C)`

`Image(f,P)`

`Image(f,v)`

The image of the cone C or polyhedron P or lattice element v under the toric lattice map f; the type of the object is preserved under the map.

`Preimage(f,C)`

`Preimage(f,P)`

`Preimage(f,v)`

The preimage of the cone C or polyhedron P or lattice element v by the toric lattice map f; the type of the object is preserved.

`KernelEmbedding(f)`

`KernelEmbedding(v)`

The inclusion of the kernel of the toric lattice map f, regarded as a distinct toric lattice, into the domain of f; or the same for the dual subspace annihilated by lattice element v.

`KernelBasis(f)`

`KernelBasis(v)`

A basis for the kernel of the toric lattice map f or the dual of the sublattice annihilated by the lattice element v.

`ImageBasis(f)`

A basis for the image of the toric lattice map f.

`IsCokernelTorsionFree(f)`

Return `true` if and only if the cokernel of the toric lattice map f is torsion free.

143.7 Bibliography

[**BP99**] Alexander Barvinok and James E. Pommersheim. An algorithmic theory of lattice points in polyhedra. In *New perspectives in algebraic combinatorics (Berkeley, CA, 1996–97)*, volume 38 of *Math. Sci. Res. Inst. Publ.*, pages 91–147. Cambridge Univ. Press, Cambridge, 1999.

[**DLHTY04**] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symbolic Comput.*, 38(4):1273–1302, 2004.

PART XX
COMBINATORICS

144 ENUMERATIVE COMBINATORICS

Chapter 144
ENUMERATIVE COMBINATORICS

144.1 Introduction

This chapter presents some of the tools provided by MAGMA for enumerative combinatorics.

144.2 Combinatorial Functions

`Factorial(n)`

The factorial $n!$ for non-negative small integer n.

`NumberOfPermutations(n, k)`

The number of permutations of n distinct objects taken k at a time.

`Binomial(n, r)`

The binomial coefficient $\binom{n}{r}$.

`Multinomial(n, [r`$_1$`, ... r`$_n$`])`

Given a sequence $Q = [r_1, \ldots, r_k]$ of positive integers such that $n = r_1 + ... + r_k$, return the multinomial coefficient $\binom{n}{r_1,\ldots,r_k}$.

`Fibonacci(n)`

Given an integer n, this function returns the n-th Fibonacci number F_n, which can be defined via the recursion $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for all integers n. Note that n is allowed to be negative, and that $F_{-n} = (-1)^{n+1}F_n$.

`Catalan(n)`

Given a small non-negative integer n, this function returns the n-th Catalan number C_n, defined via the recursion $C_0 = 1$, $C_{n+1} = C_n \cdot (4n + 2)/(n + 2)$.

`Lucas(n)`

Given an integer n, this function returns the n-th Lucas number L_n, which can be defined via the recursion $L_0 = 2$, $L_1 = 1$, $L_n = L_{n-1} + L_{n-2}$ for all integers n. Note that n is allowed to be negative, and that $L_{-n} = (-1)^n L_n$.

`GeneralizedFibonacciNumber(g0, g1, n)`

The nth member of the generalized Fibonacci sequence defined by $G_0 = g_0$, $G_1 = g_1$, $G_n = G_{n-1} + G_{n-2}$ for all integers n. Note that n is allowed to be negative. The Fibonacci and Lucas numbers are special cases where $(g_0, g_1) = (0, 1)$ or $(2, 1)$ respectively.

`StirlingFirst(n, k)`

The Stirling number of the first type, $[{n \atop k}]$, where n and k are non-negative integers.

`StirlingSecond(n, k)`

The Stirling number of the second type, $\{{n \atop k}\}$, where n and k are non-negative integers.

`Bell(n)`

The nth Bell number, giving the number of partitions of a set of size n. (Not to be confused with `NumberOfPartitions(n)`, which gives the number of partitions of the *integer* n.) This is equal to the sum of `StirlingSecond(n,k)` for k between 0 and n (inclusive).

`EulerianNumber(n, r)`

The number $E(n, r)$ of permutations p of $\{1, \ldots, n\}$ having exactly r ascents (i.e., places where $p_i < p_{i+1}$)

`HarmonicNumber(n)`

The nth harmonic number $H_n = \Sigma_{i=1}^{n} \frac{1}{i}$.

`BernoulliNumber(n)`

Returns the nth Bernoulli number B_n as a rational number.

`BernoulliApproximation(n)`

Returns a real approximation to the nth Bernoulli number B_n.

`BernoulliPolynomial(n)`

The nth Bernoulli polynomial $B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k}$ where B_n is the nth Bernoulli number.

144.3 Subsets of a Finite Set

`Subsets(S)`

The set of all subsets of the set S.

`Subsets(S, k)`

The set of subsets of the set S of size k. If k is larger than the cardinality of S then the result will be empty.

`Multisets(S, k)`

The set of multisets consisting of k not necessarily distinct elements of the set S.

`Subsequences(S, k)`

The set of sequences of length k with elements from the set S.

`Permutations(S)`

The set of permutations (stored as sequences) of the elements of the set S.

`Permutations(S, k)`

The set of permutations (stored as sequences) of each of the subsets of the set S of cardinality k.

Example H144E1

The use of Subsets is illustrated in the construction of the Petersen graph as the third Odd Graph. The nth Odd Graph has its vertices in correspondence with the $(n-1)$-element subsets of $\{1 \ldots 2n-1\}$, and an edge between two vertices if and only if their corresponding sets have empty intersection.

```
> V := Subsets( {1 .. 2*n-1}, n-1) where n is 3;
> V;
{
    { 1, 5 },
    { 2, 5 },
    { 1, 3 },
    { 1, 4 },
    { 2, 4 },
    { 3, 5 },
    { 2, 3 },
    { 1, 2 },
    { 3, 4 },
    { 4, 5 }
}
> E := { {u, v} : u,v in V | IsDisjoint(u, v) };
> Petersen := Graph< V | E >;
```

145 PARTITIONS, WORDS AND YOUNG TABLEAUX

Chapter 145
PARTITIONS, WORDS AND YOUNG TABLEAUX

145.1 Introduction

This chapter presents some of the tools provided by MAGMA for partitions, words and Young tableaux.

145.2 Partitions

A *partition* of a positive integer n is a decreasing sequence $[n_1, n_2, \ldots, n_k]$ of positive integers such that $\sum n_i = n$. The n_i are called the *parts* of the partition.

All partition functions in MAGMA operate only on small, positive integers.

`NumberOfPartitions(n)`

Given a positive integer n, return the total number of partitions of n.

`Partitions(n)`

Given a positive integer n, return the sequence of all partitions of n.

`Partitions(n, k)`

Given positive integers n and k, return the sequence of all the partitions of n into k parts.

`RestrictedPartitions(n, M)`

Given a positive integer n and a set of positive integers M, return the sequence of all partitions of n, where the parts are restricted to being elements of the set M.

`RestrictedPartitions(n, k, M)`

Given positive integers n and k, and a set of positive integers M, return the sequence of all partitions of n into k parts, where the parts are restricted to being elements of the set M.

`IsPartition(S)`

A sequence S is considered to be a partition if it consists of weakly decreasing positive integers. A sequence is allowed to have trailing zeros, and the empty sequence is accepted as a partition (of zero).

`RandomPartition(n)`

Returns a weakly decreasing sequence of positive integers which is random partition of the positive integer n.

`Weight(P)`

Given a sequence of positive integers P which is a partition, return a positive integer which is the sum of it's parts.

`IndexOfPartition(P)`

Given a sequence of positive integers P which is a partition, return its lexicographical order among partitions of the same weight.

Lexicographical ordering of partitions is such that for partitions P_1 and P_2, then $P_1 > P_2$ implies that P_1 is greater in the first part which differs from P_2. The first index is zero.

Example H145E1

The conjugacy classes of the symmetric group on n elements correspond to the partitions of n. The function `PartnToElt` below converts a partition to an element of the corresponding conjugacy class.

```
> PartitionToElt := function(G, p)
>     x := [];
>     s := 0;
>     for d in p do
>         x cat:= Rotate([s+1 .. s+d], -1);
>         s +:= d;
>     end for;
>     return G!x;
> end function;
>
> ConjClasses := function(n)
>     G := Sym(n);
>     return [ PartitionToElt(G, p) : p in Partitions(n) ];
> end function;
>
> ConjClasses(5);
[
    (1, 2, 3, 4, 5),
    (1, 2, 3, 4),
    (1, 2, 3)(4, 5),
    (1, 2, 3),
    (1, 2)(3, 4),
    (1, 2),
    Id($)
]
> Classes(Sym(5));
```

```
Conjugacy Classes
-----------------
[1]     Order 1       Length 1
        Rep Id($)

[2]     Order 2       Length 10
        Rep (1, 2)

[3]     Order 2       Length 15
        Rep (1, 2)(3, 4)

[4]     Order 3       Length 20
        Rep (1, 2, 3)

[5]     Order 4       Length 30
        Rep (1, 2, 3, 4)

[6]     Order 5       Length 24
        Rep (1, 2, 3, 4, 5)

[7]     Order 6       Length 20
        Rep (1, 2, 3)(4, 5)
```

Example H145E2

The number of ways of changing money into five, ten, twenty and fifty cent coins can be calculated using `RestrictedPartitions`. There is also a well known solution using generating functions, which we use as a check.

```
> coins := {5, 10, 20, 50};
> T := [#RestrictedPartitions(n, coins) : n in [0 .. 100 by 5]];
> T;
[ 1, 1, 2, 2, 4, 4, 6, 6, 9, 9, 13, 13, 18, 18, 24, 24, 31, 31, 39, 39, 49 ]
> F<t> := PowerSeriesRing(RationalField(), 101);
> &*[1/(1-t^i) : i in coins];
1 + t^5 + 2*t^10 + 2*t^15 + 4*t^20 + 4*t^25 + 6*t^30 + 6*t^35 + 9*t^40 + 9*t^45
    + 13*t^50 + 13*t^55 + 18*t^60 + 18*t^65 + 24*t^70 + 24*t^75 + 31*t^80 +
    31*t^85 + 39*t^90 + 39*t^95 + 49*t^100 + O(t^101)
```

145.3 Words

In this and following sections, words are considered to be elements of some *ordered* monoid, that is a free monoid whose generators have an explicit ordering. The most important example is the monoid of words generated by the positive integers, though MAGMA also allows the creation of finitely generated ordered monoids. The words in this section are developed in connection with the theory of Young tableaux (see section 145.4), which are defined over some ordered set of labels (generators of an ordered monoid in MAGMA).

The importance of the ordering on the generators is in the definition of an equivalence known as *Knuth* equivalence. Knuth equivalence is defined by the two relations

$$yxz \sim yxz \qquad x < y \leq z$$
$$xzy \sim zxy \qquad x \leq y < z$$

The *plactic* monoid is the quotient of an ordered monoid with respect to Knuth equivalence. This plactic monoid is in fact isomorphic to the monoid of tableaux over the same generators. Elements of the plactic monoid are represented by a canonical representative from the Knuth equivalence class of the ordered monoid. This canonical representative is in fact the *row word* of the corresponding tableau (see section 145.4).

145.3.1 Ordered Monoids

An ordered monoid is a free monoid with an ordered basis. MAGMA has two different types of ordered monoids. The first is the monoid of words over the positive integers, which retain their natural ordering. The second are finitely generated monoids, whose generators may be assigned names.

`OrderedMonoid(n)`

Given a positive integer n, return the monoid with n ordered generators.

`OrderedIntegerMonoid()`

Return the ordered monoid of words over the positive integers.

`Id(O)`
`O ! 1`

Given an ordered monoid O, return its identity element, i.e., the null word.

`O . i`

Given an ordered monoid O and a positive integer i, return the i–th generator of O.

`O ! [w1, ..., wn]`

Given an ordered monoid O, and a sequence of elements from O, return the word $w_1 \cdots w_n$.

`O ! [i1, ..., in]`

Given the ordered monoid O over the positive integers, and a sequence of integers, return the word $i_1 \cdots i_n$.

`w1 eq w2`

Given two words w_1 and w_2 from the same ordered monoid, return `true` if they are equal.

`w1 * w2`

Given two words w_1 and w_2 from the same ordered monoid, return their product under word concatenation.

`IsKnuthEquivalent(w1, w2)`

Two words w_1 and w_2 from the same monoid are Knuth equivalent if they can be transformed into one another using elementary Knuth transformations, (defined in the introduction to this section).

`w[i]`

Given a word w from an ordered monoid, expressed as a product of generators, return the i–th generator in the product.

`ElementToSequence(w)`

`Eltseq(w)`

Given a word w from the ordered monoid of positive integers, return w as a sequence of integers.

`Length(w)`

`#w`

Given a word w from an ordered monoid, return its length.

`Content(w)`

Given a word w from an ordered monoid, return a sequence of non-negative integers denoting its content. The content of a word is a sequence where the i–th position denotes the number of occurrences of the i–th generator in the word.

`IsReverseLatticeWord(w)`

A word w from an ordered monoid is said to be a reverse lattice word (or Yamanouchi word) if for any $n > 0$, the last n letters of w have a content which is a partition.

`MaximalIncreasingSequence(w)`

Given a word w from an ordered monoid, return a weakly increasing subsubsequence of w of maximal length. This sequence is not necessarily unique.

`MaximalIncreasingSequences(w, k)`

Given a word w from an ordered monoid and some positive integer k, return a sequence of k distinct increasing subsequences of the word w, such that the maximal number of entries from w is used. Empty sequences are returned if all entries from w are used. These sequences are not necessarily unique.

Example H145E3

We create the ordered monoid over the positive integers and look at a few elements.

```
> O := OrderedIntegerMonoid();
> O;
The monoid of words over the positive integers
>
> w1 := O ! [3,5,2,8];
> w1;
3 5 2 8
> w2 := O ! [6,7,2,4];
> w2;
6 7 2 4
>
> w2[3];
2
> Eltseq(w1);
[ 3, 5, 2, 8 ]
>
> w1*w2;
3 5 2 8 6 7 2 4
```

Example H145E4

We create a finitely generated ordered monoid and create an element.

```
> O<a,b,c,d> := OrderedMonoid(4);
> O;
The monoid of words over 4 generators: a b c d
>
> w := a*b*a*d*c;
> w;
a b a d c
```

Example H145E5

We take a word of integers and look at weakly increasing subsequences of maximal length.

```
> O := OrderedIntegerMonoid();
> w := O ! [1,3,4,2,3,4,1,2,2,3,3,2];
>
```

```
> MaximalIncreasingSequence(w);
1 1 2 2 2 3
> MaximalIncreasingSequences(w,2);
[ 1 1 2 2 2 3 , 2 3 3  ]
> MaximalIncreasingSequences(w,3);
[ 1 1 2 2 2 3 , 2 3 3 , 3 4 4  ]
> MaximalIncreasingSequences(w,4);
[ 1 1 2 2 2 3 , 2 3 3 , 3 4 4 , Id(0) ]
```

145.3.2 Plactic Monoids

The *plactic monoid* of an ordered monoid is the quotient defined by Knuth equivalence. Elements of a plactic monoid are equivalence classes of words from the original ordered monoid, and are represented by a canonical representative.

`PlacticMonoid(O)`

Given an ordered monoid O, return the plactic monoid obtained by factoring O by Knuth equivalence.

`PlacticIntegerMonoid()`

Return the plactic monoid obtained by factoring the ordered monoid over the positive integers by Knuth equivalence.

`OrderedMonoid(P)`

Given a plactic monoid P, return the ordered monoid on which P is based.

`Id(P)`
`P ! 1`

Given a plactic monoid P, return its identity element which is the null word.

`P . i`

Given a plactic monoid P and a positive integer i, return the i–th generator of P.

`P ! [u`$_1$`, ..., u`$_n$`]`

Given a plactic monoid P, and a sequence of elements from P, return the product $u_1 \cdots u_n$.

`P ! [i`$_1$`, ..., i`$_n$`]`

Given the plactic monoid P over the positive integers, and a sequence of integers, return the element of P corresponding to the Knuth equivalence class of $i_1 \cdots i_n$.

`P ! w`

Given a plactic monoid P, and a word w from the ordered monoid that its based on, return the element of P corresponding to the Knuth equivalence class of w.

`P ! [w1, ..., wn]`

Given a plactic monoid P, and a sequence of elements from the ordered monoid that its based on, return the element of P corresponding to the Knuth equivalence class of $w_1 \cdots w_n$.

`P ! t`

Given a plactic monoid P and a tableau t which are both associated with the same ordered monoid, return the element of P which is uniquely associated to t.

`u1 eq u2`

Given two elements u_1 and u_2 from the same plactic monoid, return `true` if they are equal.

`u1 * u2`

Given two elements u_1 and u_2 belonging to the same plactic monoid, return their product which is inherited from word concatenation in the ordered monoid.

`Length(u)`

`#u`

Given a element u from a plactic monoid, return its length. The length of a word is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

`Content(u)`

Given a word u from a plactic monoid, return a sequence of non-negative integers denoting its content. The content of a word is a sequence where the i–th position denotes the number of occurrences of the i–th generator in the word. The content of a word is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

Example H145E6

We create both the ordered monoid and plactic monoid over the integers and look at several elements.

```
> O := OrderedIntegerMonoid();
> P := PlacticIntegerMonoid();
>
> w1 := O ! [2,7,4,8,1,5,9];
> w1;
2 7 4 8 1 5 9
> P!w1;
7 2 8 1 4 5 9
```

First we look at a word that is Knuth equivalent to w_1,

```
> w2 := O ! [7,2,1,8,4,5,9];
> w2;
```

```
7 2 1 8 4 5 9
>
> IsKnuthEquivalent(w1,w2);
true
> (P!w1) eq (P!w2);
true
```

and then one that is not Knuth equivalent.

```
> w3 := O ! [7,1,5,8,2,9,4];
> w3;
>
7 1 5 8 2 9 4
> IsKnuthEquivalent(w1,w3);
false
> (P!w1) eq (P!w3);
false
> P!w3;
7 5 8 1 2 4 9
```

Example H145E7

We create a finitely generated ordered monoid, its associated plactic monoid, and look at some properties which are invariant under Knuth equivalence.

```
> O<a,b,c,d,e> := OrderedMonoid(5);
> P := PlacticMonoid(O);
> P;
The plactic monoid of words over 5 generators: a b c d e
>
> w := b*c*e*e*a*d*a*d;
> w;
b c e e a d a d
> Length(w);
8
> Content(w);
[ 2, 1, 1, 2, 2 ]
>
> u := P!w;
> u;
a a b c d d e e
> Length(u);
8
> Content(u);
[ 2, 1, 1, 2, 2 ]
```

145.4 Tableaux

The tableaux module in MAGMA is loosely based on the computational algebra system SYMMETRICA developed by Adalbert Kerber and Axel Kohnert [KKL92]. The main reference for the theory of this section is "Young Tableaux" by William Fulton [Ful97].

A Young diagram, or Ferrers diagram, is a collection of boxes, or cells, arranged in left–justified rows, with a weakly decreasing number of cells in each row. Listing the number of cells in each row gives a partition (its shape) of n, where n is the total number of cells in the diagram. Conversely, each partition corresponds to a unique Young diagram. A numbering of a Young diagram is an assignment of a positive integer to each box. More generally a numbering can be done using any ordered set of labels, in MAGMA this means the generators of some ordered monoid, (see section 145.3).

A Young tableau, or simply tableau, is a numbering of a Young diagram which has (i) weakly increasing entries across each row, and (ii) strictly increasing entries down each column. These tableaux form a monoid with respect to a multiplication which may be defined in either of two ways, Schensted's "bumping" row insertion, or Schützenberger's "sliding" *jeu de taquin*, (see [Ful97] for full details).

A tableau monoid is constructed from an ordered monoid (see section 145.3), the generators of the ordered monoid comprising the tableau labels. A tableau monoid and a plactic monoid derived from the same ordered monoid are in fact isomorphic to one another.

Flipping a diagram over its main diagonal gives the conjugate diagram, its shape being the `ConjugatePartition` of the original shape.

A skew diagram or skew shape is the diagram obtained by deleting a smaller Young diagram from inside a larger one. A skew tableau is a numbering on a skew diagram obeying the same restrictions on its entries.

145.4.1 Tableau Monoids

Let O be an ordered monoid, elements of O are expressed as products of its generators. The generators of O can be used as the set of labels for a family of tableaux M, called a *tableau monoid*.

A tableau monoid has only one defining characteristic, the ordered monoid which specifies its labels.

The most important tableau monoid is the one defined over the set of all positive integers.

`TableauMonoid(O)`

`TableauMonoid(P)`

Given an ordered monoid O or a plactic monoid P, (in which case let O be the ordered monoid on which P is based), then return the tableau monoid of tableaux whose labels are the generators of O.

`TableauIntegerMonoid()`

Return the tableau monoid over the positive integers.

`OrderedMonoid(M)`

Given a tableau monoid M, return the ordered monoid from which M gets its labels.

Example H145E8

We create the tableau monoid over the positive integers. We take a word in the associated ordered monoid and look at its image in the tableau monoid.

```
> M := TableauIntegerMonoid();
> M;
Monoid of Tableaux labelled by the generators of:
The monoid of words over the positive integers
>
> w := OrderedMonoid(M) ! [3,6,2,8,3,9,1];
> w;
3 6 2 8 3 9 1
> M ! w;
Tableau of shape: 4 2 1
1 3 8 9
2 6
3
```

Example H145E9

We create a tableau monoid over a finite set of labels, and look at the image of a word.

```
> O<x,y,z> := OrderedMonoid(3);
> M := TableauMonoid(O);
> M;
Monoid of Tableaux labelled by the generators of:
The monoid of words over 3 generators: x y z
>
> M ! (z*y*x*z*y*y*x*y*x*z);
Tableau of shape: 5 3 2
x x x y z
y y y
z z
```

145.4.2 Creation of Tableaux

Tableaux are created from the words of some ordered monoid. For tableaux over the positive integers, the rows can be input directly as sequences. Words can be used to exactly specify the rows of a tableau, or an entire word can be mapped to a tableau using row insertion.

`Tableau(Q)`

Given a sequence of sequences of non-negative integers Q, return the tableau with the elements of Q as its rows. Zeroes will indicate skew entries. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`Tableau(Q)`

Given a sequence Q of words from some ordered monoid, return the (non-skew) tableau with the words of Q as its rows. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`Tableau(S, Q)`

Given a sequence of nonnegative integers S which is a partition, and a sequence of sequences of positive integers Q. Create a skew tableau with skew (inner) shape given by the partition S, and the non-skew part of each row being given by the entries of Q. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`Tableau(S, Q)`

Given a sequence of nonnegative integers S which is a partition, and a sequence Q of words from some ordered monoid. Create a skew tableau with skew (inner) shape given by the partition S, and the non-skew part of each row being given by the words of Q. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`WordToTableau(w)`
`WordToTableau(u)`

Given a word w from an ordered monoid, return the tableau obtained by row inserting the entries of w (from the left) into the empty tableau.

If given an element u of a plactic monoid, any word of its Knuth equivalence class is used. This map is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

It is a surjective homomorphism from the ordered monoid to the tableau monoid, and an isomorphism from the plactic monoid to the tableau monoid.

`M ! w`

Given a tableau monoid M and a word w from its associated ordered monoid, return the tableau corresponding to w through row insertion.

`M ! u`

Given a tableau monoid M and an element u from a plactic monoid associated with the same ordered monoid as M, return the tableau corresponding to u through row insertion.

`M ! [i1, ..., in]`

Given the tableau monoid M over the positive integers, and a sequence of positive integers, return the tableau corresponding to the word $i_1 * \cdots * i_n$.

`M ! Q`

Given the tableau monoid M over the positive integers, and a sequence of sequences of non-negative integers Q, return the tableau with the elements of Q as its rows. Zeroes will indicate skew entries. The resulting tableau must have weakly increasing rows and strictly increasing columns.

`M ! Q`

Given a tableau monoid M, and a sequence Q of words from its associated ordered monoid, return the (non-skew) tableau with the words of Q as its rows. The resulting tableau must have weakly increasing rows and strictly increasing columns.

Example H145E10

We create a tableau over the positive integers in two ways. First we input its rows explicitly, then we use a single word which specifies the tableau. We also create a skew tableau by using zero entries in a sequence of sequences.

```
> O := OrderedIntegerMonoid();
>
> T := Tableau( [ [2,5,5,6], [5,7,9,9], [6,9] ]);
> T;
Tableau of shape: 4 4 2
2 5 5 6
5 7 9 9
6 9
> WordToTableau( O ! [6,9,5,7,9,9,2,5,5,6] );
Tableau of shape: 4 4 2
2 5 5 6
5 7 9 9
6 9
> Tableau([ [0,0,0,3,4,5], [0,1,2,4], [1,6] ]);
Skew tableau of shape: 6 4 2 / 3 1
0 0 0 3 4 5
0 1 2 4
1 6
```

Example H145E11

We create a skew tableau in a tableau monoid with finitely many labels.

```
> O<a,b,c,d> := OrderedMonoid(4);
> T := Tableau( [3,2] , [ a*a*b*d, b*d*d, a*b]);
> T;
Skew tableau of shape: 7 5 2 / 3 2
0 0 0 a a b d
0 0 b d d
a b
```

Example H145E12

We create the ordered, plactic and tableau monoids each over the positive integers, and examine the natural maps between them.

```
> O := OrderedIntegerMonoid();
> P := PlacticIntegerMonoid();
> M := TableauIntegerMonoid();
> w1 := O ! [4,6,2,6,9,6,2,2,1,7];
> w2 := O ! [9,4,2,1,6,2,6,2,6,7];
> w1 eq w2;
false
> IsKnuthEquivalent(w1,w2);
true
```

Knuth equivalence implies equality for the image in both the plactic and tableau monoids.

```
> (P!w1) eq (P!w2);
true
> P!w1;
9 4 2 6 6 1 2 2 6 7
>
> (M!w1) eq (M!w2);
true
> M!w1;
Tableau of shape: 5 3 1 1
1 2 2 6 7
2 6 6
4
9
```

The plactic and tableau monoids are isomorphic. We confirm that the natural image of an equivalence class from the plactic monoid is the same as the natural images of its member words.

```
> M!w1 eq M ! (P!w1);
true
```

145.4.3 Enumeration of Tableaux

Multiple tableaux may be created corresponding to specified parameters.

`StandardTableaux(P)`

`StandardTableaux(M, P)`

Returns the set of all standard tableau from the tableau monoid M of shape P, which is a partition.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain enough labels to fill the shape P, (i.e. more than `Weight(`P`)`).

`StandardTableauxOfWeight(n)`

`StandardTableauxOfWeight(M, n)`

Returns the set of all standard tableau from the tableau monoid M and of weight n, which is a positive integer.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain at least n labels.

`TableauxOfShape(S, m)`

`TableauxOfShape(M, S, m)`

Given a tableau monoid M, a sequence of positive integers S which is a partition, and a positive integer m, then return the set of all tableau of shape S which use only the first m labels of the tableau monoid M.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain at least m labels.

`TableauxOnShapeWithContent(S, C)`

`TableauxOnShapeWithContent(M, S, C)`

Given a tableau monoid M, a sequence of positive integers S which is a partition, and a sequence of non–negative integers C, then return the set of all tableau from M having shape S and *content* C (see section 145.4.6 for definition of content). If C prescribes fewer cells than would completely fill the shape S, then the tableau within the given shape is returned.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must have at least as many labels as C has entries.

`TableauxWithContent(C)`

`TableauxWithContent(M, C)`

Given a tableau monoid M, and a sequence of non-negative integers C, return the set of all tableau from M with *content* C (see section 145.4.6 for definition of content).

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must have at least as many labels as C has entries.

Example H145E13

We create all tableaux of a given shape, then check that the number of tableaux created agrees with the combinatorical result.

```
> P := [ 3, 2];
> S := StandardTableaux( P );
> S;
{ Tableau of shape: 3 2
 1 2 3
 4 5, Tableau of shape: 3 2
 1 3 4
 2 5, Tableau of shape: 3 2
 1 3 5
 2 4, Tableau of shape: 3 2
 1 2 5
 3 4, Tableau of shape: 3 2
 1 2 4
 3 5 }
>
> #S eq NumberOfStandardTableaux( P );
true
```

Example H145E14

We create all tableau over the ordered generators a, b, c, d which have shape $[4, 2]$ and exactly two a's, one b, two c's and one d. There is in fact four of them.

```
> O<a,b,c,d> := OrderedMonoid(4);
> M := TableauMonoid(O);
> TableauxOnShapeWithContent( M, [4,2], [2,1,2,1]);
{ Tableau of shape: 4 2
a a c c
b d , Tableau of shape: 4 2
a a b c
c d , Tableau of shape: 4 2
a a b d
c c , Tableau of shape: 4 2
a a c d
b c  }
```

145.4.4 Random Tableaux

The functions in this section are based on ideas from [GNW84].

`RandomHookWalk(P, i, j)`

`RandomHookWalk(t, i, j)`

A random hook walk is an essential tool for the creation of random tableau. The *hook* of a cell on a Young diagram is the cells to the right and below its position. The Young diagram is specified by either the sequence of positive integers P which is a partition, or the tableau t.

Given positive integers i and j such that (i, j) lies on the shape, the walk proceeds as follows. Starting from the specified (i, j)-th cell, move to another cell chosen at random from its hook. Repeat this process until a outside corner of the Young diagram is reached, and return the two integers representing the coordinates of this final cell.

`RandomTableau(S)`

`RandomTableau(M, S)`

Given a tableau monoid M, and a sequence of positive integers S which is a partition, return a random standard tableau from M of shape S.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain enough labels to fill the shape S, (i.e. more than `Weight(`S`)`).

`RandomTableau(n)`

`RandomTableau(M, n)`

Given a tableau monoid M, and a positive integer n, return a (not completely) random standard tableau of weight n. The probability of any specific tableau of shape S being returned is $N_S/n!$, where N_S is the number of standard tableaux of shape S. So tableaux on more populous shapes are more likely to occur.

If a tableau monoid M is not specified then the monoid over the positive integers is used. If M is specified then it must contain at least n labels.

Example H145E15

We numerically test the random creation of tableau of a given shape, by calculating the average percentage difference in the number of each tableau created. We find it to be less than one percent.

```
> P := [4,3,2];
> Runs := 10000;
>
> S := Setseq( StandardTableaux( P ) );
> Count := [0: i in [1..#S]];
>
> for k in [1..Runs] do
>     T := RandomTableau( P );
```

```
>     i := Index(S,T);
>     Count[i] +:= 1;
> end for;
>
> Average :=  Runs/#S;
> Diff := [RealField(2) | Abs( (Count[i] - Average)/Average )
>                       : i in [1..#Count]];
>
> AvgDiff := (&+ Diff)/#Diff;
> AvgDiff;
0.006
```

145.4.5 Basic Access Functions

`Shape(t)`

`OuterShape(t)`

The (outer) shape of a tableau t is the sequence of positive integers denoting the (decreasing) row lengths of its Young Diagram, which results in a partition.

`SkewShape(t)`

`InnerShape(t)`

The Young diagram of a skew tableau t has a smaller Young diagram inside of it deleted. The skew, or inner, shape of a skew tableau is the sequence of positive integers denoting the (decreasing) row lengths of this deleted diagram, which results in a partition.

A non-skew tableau is a special case of a skew tableau, whose skew shape is simply the null sequence.

Trailing zeroes are added to the inner shape of t to give it the same length as the outer shape of t.

`PartitionCovers(P1, P2)`

Given two sequences of integers, P_1 and P_2, which are partitions, return `true` if the Young diagram of P_1 covers the Young diagram of P_2.

`ConjugatePartition(P)`

Given a sequence of integers P which is a partition, return the partition corresponding to the *conjugate* Young diagram of P (see section 145.4 for a full description).

`Weight(t)`

Given a tableau t, return the positive integer which is the number of (non–skew) cells in its Young Diagram.

`SkewWeight(t)`

Given a tableau t, return the positive integer which is the number of skew cells in its Young Diagram.

`NumberOfRows(t)`

Given a tableau t, return a positive integer which is its number of rows.

`NumberOfSkewRows(t)`

Given a tableau t, return a positive integer which is its number of skewed rows.

`Row(t, i)`

Given a tableau t, return the non-skewed entries of the i–th row as a word of an ordered monoid.

`Rows(t)`

Given a tableau t, return a sequence of words from an ordered monoid which are the non-skewed entries of its rows.

`Column(t, j)`

Given a tableau t, return the non-skewed entries of the j–th column as a word of an ordered monoid.

`Columns(t)`

Given a tableau t, return a sequence of words from an ordered monoid which are the non-skewed entries of its columns.

`RowSkewLength(t, i)`

Given a tableau t, return the length of the skewed portion of the i–th row (zero if no skewed portion).

`ColumnSkewLength(t, j)`

Given a tableau t, return the length of the skewed portion of the j–th column (zero if no skewed portion).

`FirstIndexOfRow(t, i)`

Given a tableau t and a positive integer i, return the index of the first non–skew entry of the i–th row of t. If the row has no non–skew entries then the index will be one greater than the length of the row, i.e., out of bounds.

`LastIndexOfRow(t, i)`

`RowLength(t, i)`

Given a tableau t and a positive integer i, return the index of the last entry of the i–th row of t, which is the length of the i–th row.

`FirstIndexOfColumn(t, j)`

Given a tableau t and a positive integer j, return the index of the first non–skew entry of the j–th column of t. If the column has no non–skew entries then the index will be one greater than the length of the column, i.e., out of bounds.

`LastIndexOfColumn(t, j)`

`ColumnLength(t, j)`

Given a tableau t and a positive integer j, return the index of the last entry in the j–th column of t, which is the length of the j–th column.

Example H145E16

Given a skew tableau, we access a number of its structural properties.

```
> T := Tableau( [3,3,2] , [ [4, 6 ] ,[5, 9], [1, 8] ]);
> T;
Skew tableau of shape: 5 5 4 / 3 3 2
0 0 0 4 6
0 0 0 5 9
0 0 1 8
> Shape(T);
[ 5, 5, 4 ]
> Weight(T);
6
> SkewWeight(T);
8
> NumberOfRows(T);
3
> NumberOfSkewRows(T);
3
> RowSkewLength(T, 2);
3
> Row(T, 2);
5 9
> ColumnSkewLength(T, 3);
2
> Column(T, 3);
1
```

145.4.6 Properties

`HookLength(t, i, j)`

`HookLength(P, i, j)`

The *hook* of a cell on a Young diagram comprises the cells to the right and below its position. The Young diagram is specified by either the sequence of positive integers P which is a partition or the tableau t. The number of cells contained in a hook is its *length*.

For positive integers i and j such that (i, j) is the co-ordinates of a cell lying on the specified Young diagram, a positive integer is returned which is the length of the hook of that cell.

`Content(t)`

Given a tableau t, return a sequence of non-negative integers which is its *content*.

The content of t is such that its i–th value denotes the number of times the i–th label occurs in t.

`Word(t)`

`RowWord(t)`

Given a tableau t, its row word is formed by reading its entries from left to right, bottom to top. The result is a word of an ordered monoid.

`ColumnWord(t)`

Given a tableau t, its column word is formed by reading its labels from bottom to top, left to right. The result is a word of an ordered monoid.

`IsStandard(t)`

Given a tableau t of weight n, then t is standard if and only if its entries are exactly the first n labels of its parent monoid. So t is standard if and only if it has a content of the form $[1, 1, \ldots, 1]$.

`IsSkew(t)`

Returns `true` if the tableau t is a skew tableau.

`IsLittlewoodRichardson(t)`

Given a tableau t, then it is a Littlewood–Richardson tableau if the content of t forms a reverse lattice word, (see section 145.3 for a definition of a reverse lattice word).

Example H145E17

We create a tableau which is not standard. Examining its content shows a simple to way to transform it into a standard tableau.

```
> O := OrderedIntegerMonoid();
> T := WordToTableau( O ! [8,1,7,3,6,2,5,9] );
> T;
Tableau of shape: 4 2 1 1
 1 2 5 9
 3 6
 7
 8
> Content(T);
[ 1, 1, 1, 0, 1, 1, 1, 1, 1 ]
> IsStandard(T);
false
>
> RowInsert(~T, 4);
> T;
Tableau of shape: 4 2 1 1 1
 1 2 4 9
 3 5
 6
 7
 8
> Content(T);
[ 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
> IsStandard(T);
true
```

Example H145E18

Given a tableau, check that the row and column words are Knuth equivalent.

```
> T := Tableau( [2, 2], [ [1,2,3], [4,6,6], [1,5], [2,6] ]);
> T;
Skew tableau of shape: 5 5 2 2 / 2 2
0 0 1 2 3
0 0 4 6 6
1 5
2 6
>
> RW := RowWord(T);
> RW;
2 6 1 5 4 6 6 1 2 3
> CW := ColumnWord(T);
> CW;
2 1 6 5 4 1 6 2 6 3
```

```
> IsKnuthEquivalent(RW, CW);
true
```

145.4.7 Operations

Several operations exist to manipulate tableaux and allow them to interact.

`t1 eq t2`

Given two tableaux t_1 and t_2, return `true` if they are equal.

`t1 * t2`

Given two tableaux t_1 and t_2, return another tableau which is their product. The product of two tableaux can be defined using either row insertion or *jeu de taquin*.

`DiagonalSum(t1, t2)`

The `DiagonalSum` of two tableau $t1$ and $t2$ is formed by first building a rectangle of empty cells, with the same number of columns as $t1$ and the same number of rows as $t2$. Then $t1$ is attached below the rectangle and $t2$ attached to right of it.

`Conjugate(t)`

Given a tableau t with strictly increasing rows, the rows and columns of t are transposed to form a new tableau on the conjugate Young diagram.

`JeuDeTaquin(~t, i, j)`

`JeuDeTaquin(t, i, j)`

Given a skew tableau t, and the co-ordinates (i, j) which must be a skew cell that is an *inside corner* (a corner on the skew shape of t). The skewed (i, j)–th cell is removed using *jeu de taquin*.

`JeuDeTaquin(~t)`

`JeuDeTaquin(t)`

`Rectify(~t)`

`Rectify(t)`

Given a tableau t, remove all skewed cells successively using *jeu de taquin*.

`InverseJeuDeTaquin(~t, i, j)`

`InverseJeuDeTaquin(t, i, j)`

Given a tableau t, and co-ordinates (i, j) which must lie on the outside of t resulting in a new and valid shape. A new skew tableau with skew degree one higher than t is created. This is achieved by adding a new skew cell in the (i, j)–th position, then performing *jeu de taquin* in reverse, which moves the new entry to create a valid skew tableau.

`RowInsert(~t, w)`

`RowInsert(t, w)`

`RowInsert(~t, u)`

`RowInsert(t, u)`

Given a tableau t and a word w from the same ordered monoid that t is associated to, insert w into t using Schensted's row insertion algorithm.

If given an element u of a plactic monoid, any word of its Knuth equivalence class is used. This map is invariant under Knuth equivalence and so is well defined for elements of the plactic monoid.

`RowInsert(~t, x)`

`RowInsert(t, x)`

Given a tableau t over the positive integers, and a positive integer x, insert x into t using the Schensted's row insertion algorithm.

`InverseRowInsert(~t, i, j)`

`InverseRowInsert(t, i, j)`

Given a tableau t and positive integers i and j such that (i, j) corresponds to an outside corner cell of the tableau t, then the row insertion algorithm is applied in reverse to remove the specified entry. In the functional form, both the generated tableau and the value of the removed entry as a word of an ordered monoid are returned.

Example H145E19

The order with which the skew entries of a skew tableau are removed using jeu de taquin is irrelevant to the final tableau produced. We give an example of this.

```
> T := Tableau([2,1], [ [2], [3,4] ,[3] ]);
> T1 := T;
> T;
Skew tableau of shape: 3 3 1 / 2 1
 0 0 2
 0 3 4
 3
>
> JeuDeTaquin(~T, 1, 2); T;
Skew tableau of shape: 3 2 1 / 1 1
 0 2 4
 0 3
 3
> JeuDeTaquin(~T, 2, 1); T;
Skew tableau of shape: 3 2 / 1
 0 2 4
 3 3
```

```
> JeuDeTaquin(~T, 1, 1); T;
Tableau of shape: 3 1
 2 3 4
 3
>
> JeuDeTaquin(~T1, 2, 1); T1;
Skew tableau of shape: 3 3 / 2
 0 0 2
 3 3 4
> JeuDeTaquin(~T1, 1, 2); T1;
Skew tableau of shape: 3 2 / 1
 0 2 4
 3 3
> JeuDeTaquin(~T1, 1, 1); T1;
Tableau of shape: 3 1
 2 3 4
 3
>
> T eq T1;
true
```

Example H145E20

We manually compare the two different methods of multiplying tableau.

```
> O<a,b,c,d,e,f,g,h> := OrderedMonoid(8);
> T1 := Tableau( [ a*c*f, d*g]);
> T2 := Tableau( [ b*b*e*f, d*g*h]);
```

First using row insert,

```
> w := Word(T2);
> w;
d g h b b e f
> Res1 := RowInsert(T1, w);
> Res1;
Tableau of shape: 5 4 2 1
a b b e f
c d g h
d f
g
```

and then using JeuDeTaquin.

```
> Res2 := DiagonalSum(T1, T2);
> Res2;
Skew tableau of shape: 7 6 3 2 / 3 3
0 0 0 b b e f
0 0 0 d g h
a c f
```

```
d g
> Rectify(~Res2);
> Res2;
Tableau of shape: 5 4 2 1
a b b e f
c d g h
d f
g
```

Now we check our results.

```
> Res1 eq Res2;
true
> Res1 eq (T1*T2);
true
```

145.4.8 The Robinson-Schensted-Knuth Correspondence

While the map from the plactic monoid to tableaux is an isomorphism, many different words from the original ordered monoid map to the same tableau. However a bijective map exists between an ordered monoid and *pairs* of tableau of the same shape, the second of which being a standard tableau. This is known as the *Robinson correspondence*.

By removing the restriction that the second tableau be standard, this correspondence is extended further to a bijective map with all matrices with non-negative integers. This is known as the Robinson-Schensted-Knuth (RSK) correspondence.

This latter correspondence can also be made to all lexicographically ordered pairs of words (both having the same length).

`LexicographicalOrdering(~w1, ~w2)`

`LexicographicalOrdering(w1, w2)`

Given two words w_1 and w_2 of the same length, place them in increasing lexicographical order (the functional version returns two new words). Lexicographical ordering on the pairs $(w_1[i], w_2[i])$ is determined primarily by the comparison $w_2[i] < w_2[j]$, but if $w_2[i] = w_2[j]$ then the comparison $w_1[i] < w_1[j]$ is taken into account.

`IsLexicographicallyOrdered(w1, w2)`

Given two words w_1 and w_2, return `true` if the pairs $(w_1[i], w_2[i])$ are ordered according to the ordering described in `LexicographicalOrdering`.

`RSKCorrespondence(w)`

Given a word w, use the bijective map described by the Robinson-Schensted-Knuth correspondence to return two tableau t_1 and t_2. The first tableau, t_1, will be labelled by the entries of w, while t_2 is a standard tableau over the positive integers.

The map described by the correspondence is as follows. Map the word w to t_1 by row inserting its entries into the empty tableau, (as is done in `WordToTableau`). As t_1 is being generated, the insertion tableau t_2 is built simultaneously. As $w[k]$ is inserted into t_1, an additional cell is added to its shape. A new cell is added to t_2 in exactly the same place, and it is labelled with the integer k.

`InverseRSKCorrespondenceSingleWord(t1, t2)`

Given a pair of tableaux t_1, t_2, of the same shape, where t_2 is over the positive integers, return the preimage of t_1 and t_2 using the bijective map described by the Robinson-Schensted-Knuth correspondence.

The result is returned as a single word of the ordered monoid of t_1, as described in the single word version of `RSKCorrespondence`.

`RSKCorrespondence(w1, w2)`

Given two lexicographically ordered words w_1 and w_2 of the same length, where w_2 must be over the positive integers, use the bijective map described by the Robinson-Schensted-Knuth correspondence to return two tableau t_1 and t_2. The first tableau, t_1, will be labelled by the entries of w_1, while t_2 will be labelled by the entries of w_2.

The map described by the correspondence is as follows. Map the word w_1 to t_1 by row inserting its entries into the empty tableau, (as is done in `WordToTableau`). As t_1 is being generated, the insertion tableau t_2 is built simultaneously. As $w[k]$ is inserted into t_1, an additional cell is added to its shape. A new cell is added to t_2 in exactly the same place, and it is labelled with the integer $w_2[k]$.

`InverseRSKCorrespondenceDoubleWord(t1, t2)`

Given a pair of tableaux t_1, t_2, of the same shape, where t_2 is over the positive integers, return the preimage of t_1 and t_2 using the bijective map described by the Robinson-Schensted-Knuth correspondence.

The result is returned as words of the ordered monoids of t_1 and t_2 respectively, as described in the double word version of `RSKCorrespondence`.

`RSKCorrespondence(M)`

Given a matrix M of non-negative integers, use the bijective map described by the Robinson-Schensted-Knuth correspondence to return two tableau t_1 and t_2 over the positive integers.

The map described by the correspondence is as follows. Initialise two words over the positive integers, w_1 and w_2, as null words. For the (i, j)–th entry of M, say m_{ij}, append i to w_1 m_{ij} times, and append j to w_2 m_{ij} times. These two words are then used in the correspondence described in the above version of `RSKCorrespondence`.

`InverseRSKCorrespondenceMatrix(t1, t2)`

Given a pair of tableaux t_1, t_2, of the same shape, both over the positive integers, return the preimage of t_1 and t_2 using the bijective map described by the Robinson-Schensted-Knuth correspondence.

The result is a matrix of non-negative integers, as described in the matrix version of `RSKCorrespondence`.

Example H145E21

We take two words from the ordered monoid over the positive integers. Although they correspond to the same tableau under the natural mapping into the tableau monoid, we show they have a different image under the RSK correspondence. That is, they have different *insertion* tableaux.

```
> O := OrderedIntegerMonoid();
> M := TableauMonoid(O);
> a := O ! [4,7,2,8,4,9,2];
> a;
4 7 2 8 4 9 2
> b := O ! [7,4,2,4,2,8,9];
> b;
7 4 2 4 2 8 9
```

Now we will see that a and b have the same image in the tableau monoid, but that they are distinguished under the RSK correspondence.

```
> (M!a) eq (M!b);
true
> Ta1, Ta2 := RSKCorrespondence(a);
> Tb1, Tb2 := RSKCorrespondence(b);
> Ta1 eq Tb1;
true
> Ta2 eq Tb2;
false
> Ta2;
Tableau of shape: 4 2 1
1 2 4 6
3 5
7
> Tb2;
Tableau of shape: 4 2 1
1 4 6 7
2 5
3
```

To check that the map is indeed bijective we find a and b as the pre-images of the created tableaux.

```
> InverseRSKCorrespondenceSingleWord(Ta1,Ta2 );
4 7 2 8 4 9 2
> InverseRSKCorrespondenceSingleWord(Tb1,Tb2 );
```

```
7 4 2 4 2 8 9
```

Example H145E22

We apply the RSK correspondence to two arbitrary words, the first from a finitely generated ordered monoid. The second word must always be over the positive integers. The first step is to put them into lexicographical order.

```
> O1<a,b,c,d,e,f> := OrderedMonoid(6);
> O2 := OrderedIntegerMonoid();
>
> w1 := e*a*e*b*f*d*a;
> w2 := O2 ! [3,8,2,8,3,3,6];
> LexicographicalOrdering(~w1, ~w2);
> w1, w2;
e d e f a a b  2 3 3 3 6 8 8
```

Now we use the correspondence,

```
> T1, T2 := RSKCorrespondence(w1, w2);
> T1;
Tableau of shape: 3 3 1
a a b
d e f
e
> T2;
Tableau of shape: 3 3 1
2 3 3
3 8 8
6
```

and check that the inverse is correct.

```
> InverseRSKCorrespondenceDoubleWord(T1, T2);
e d e f a a b  2 3 3 3 6 8 8
```

Example H145E23

We give an example of the bijective map provided by the RSK correspondence between matrices of non-negative integers and pairs of tableaux of the same shape.

```
> M := Matrix(2,3,[0,0,2,3,1,2]);
> M;
[0 0 2]
[3 1 2]
> T1, T2 := RSKCorrespondence(M);
> T1;
Tableau of shape: 6 2
1 1 1 2 3 3
3 3
```

```
> T2;
Tableau of shape: 6 2
1 1 2 2 2 2
2 2
> InverseRSKCorrespondenceMatrix(T1, T2);
[0 0 2]
[3 1 2]
```

Looking now at preimage of the tableaux in the double word format, the (i, j) co-ordinates of entries of M can clearly be seen with the correct multiplicities.

```
> wj, wi := InverseRSKCorrespondenceDoubleWord(T1, T2);
> wi;
1 1 2 2 2 2 2 2
> wj;
3 3 1 1 1 2 3 3
```

Example H145E24

We illustrate the remarkable property that a permutation p (in image notation) corresponds to the tableaux pair $(T1, T2)$ if and only if the inverse permutation p^{-1} corresponds to the tableaux pair $(T2, T1)$.

```
> O := OrderedIntegerMonoid();
> n := Random([1..100]);
> G := SymmetricGroup(n);
> p := Random(G);
>
> T1, T2 := RSKCorrespondence( O ! Eltseq(p) );
>
> p1 := InverseRSKCorrespondenceSingleWord( T2, T1);
> p1 := G ! Eltseq(p1);
>
> p1 eq p^-1;
true
```

145.4.9 Counting Tableaux

`NumberOfStandardTableaux(P)`

Given a sequence of positive integers P which is a partition, return the number of standard tableaux of shape P. This is the same as the number of Knuth equivalent words corresponding to each standard tableaux of shape P.

`NumberOfStandardTableauxOnWeight(n)`

Given a positive integer n, return the number of standard tableaux having weight n.

`NumberOfTableauxOnAlphabet(P, m)`

Given a sequence of positive integers P which is a partition, return the number of tableau of shape P with entries from $[1, \ldots, m]$.

`KostkaNumber(S, C)`

Given a sequence of positive integers S forming a partition, and a sequence of non-negative integers C, return the number of tableau having shape S and content C. If C prescribes fewer cells than would completely fill the shape S, then the number of tableau within the given shape is returned.

Example H145E25

There is a correspondence between the number of standard tableaux on a given shape, and the number of words associated with each of those tableaux. We manually count the number of words corresponding to a specific standard tableau, and show that this is equal to the number of standard tableaux on that shape.

```
> O := OrderedIntegerMonoid();
> T := Tableau( [ [ 1, 2, 3, 4, 7],
>                 [ 5, 8],
>                 [ 6]    ] );
> T;
Tableau of shape: 5 2 1
 1 2 3 4 7
 5 8
 6
>
> n := Weight(T);
> G := SymmetricGroup(n);
> S := [1..n];
>
> Count := 0;
> for p in G do
>     T1 := WordToTableau( O ! (S^p) );
>     if T1 eq T then
>         Count +:= 1;
>     end if;
> end for;
>
> Count;
64
> NumberOfStandardTableaux( Shape(T) );
64
```

Example H145E26

If a tableaux is constructed of the shape $[1, 1, 1, \ldots, 1]$ then the entries must be strictly increasing, hence non-repeating. So for such a tableau of length n on an alphabet of m symbols, each subset of n symbols will correspond to one single distinct tableau. So the number of tableau will be $\binom{m}{n}$.

```
> m := Random(1,50);
> n := Random(1,m);
>
> Shape := [ 1 : i in [1..n] ];
> Binomial(m,n) eq NumberOfTableauxOnAlphabet(Shape, m);
true
```

145.5 Bibliography

[**Ful97**] William Fulton. *Young Tableaux*. Cambridge University Press, Cambridge, 1997.

[**GNW84**] Curtis Greene, Albert Nijenhuis, and Herbert S. Wilf. Another probabilistic method in the theory of Young tableaux. *J. Combin. Theory Ser. A*, 37(2):127–135, 1984.

[**KKL92**] Adalbert Kerber, Axel Kohnert, and Alain Lascoux. SYMMETRICA, an object oriented computer-algebra system for the symmetric group. *J. Symbolic Comp.*, 14(2-3):195–203, 1992.
URL:http://www.mathe2.uni-bayreuth.de/axel/symneu_engl.html.

146 SYMMETRIC FUNCTIONS

Chapter 146
SYMMETRIC FUNCTIONS

146.1 Introduction

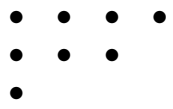
A *symmetric function* is a polynomial which is invariant under permutations of its indeterminates. The symmetric functions over a commutative ring with unity, with an arbitrary number of indeterminates, form an algebra, denoted by Λ.

The symmetric functions of fixed degree n form a submodule of Λ denoted by Λ^n. When analysing and computing with elements of Λ, there are 5 important bases to consider. These are the bases consisting of the *Schur*, *Homogeneous*, *Power Sum*, *Elementary* and *Monomial* symmetric functions. The size of any basis of Λ^n is equal to the number of partitions of weight n, and each basis has its elements indexed by those partitions.

A *partition* is a weakly decreasing sequence of positive integers. So, for example, $[5, 3, 1]$ is a partition of weight 9, and $[3, 1]$ is a partition of weight 4. For more information on partitions see Section 145.2.

MAGMA allows computations with symmetric functions in any of the 5 bases, and any symmetric function can be expressed in any of the different bases. The main reference on the theory of symmetric functions used here is the book of Macdonald [Mac95].

In order to understand the different symmetric function bases, it is necessary to have an understanding of *Young Tableaux*. Firstly, for a given partition let us visualise its *shape* making a diagram. We arrange "boxes" in left-justified rows corresponding to the entries of the partition. So for example, the partition $[4, 3, 1]$ of weight 8 gives



This is known as a *Ferrers diagram*.

Now, to create a Young tableau, we must fill these boxes with integer entries, but subject to some restrictions. We require that each row of the tableau must be weakly increasing, and each column must be strictly increasing. So, for example, a possible tableau of shape $[4, 3, 1]$ would be

$$T \quad = \quad \begin{array}{cccc} 1 & 3 & 3 & 4 \\ 2 & 4 & 5 & \\ 4 & & & \end{array}$$

A tableau can be mapped onto a monomial by treating each entry in the tableau as an indeterminate. So, the above tableau will correspond to the monomial

$$T \mapsto x^T = x_1 x_2 x_3^2 x_4^3 x_5$$

With this connection between tableaux and monomials we are now able to define the bases of an algebra of symmetric functions.

An important point to realise is that when dealing with a symmetric function, the number of indeterminates of the corresponding polynomial is neither fixed nor important. In fact a symmetric function has an image as a symmetric polynomial in an arbitrary number of indeterminates, even an infinite number. MAGMA is only equipped to deal with polynomial rings with finite rank.

Since the partitions of weight n prescribe the symmetric function basis elements of degree n which generate Λ^n, (the symmetric polynomials in n indeterminates), then the number of indeterminates being used will normally be equal to the degree of the symmetric functions considered.

- Schur Functions

  Given a partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$ of weight n, the Schur function s_λ is given by $s_\lambda = \sum_T x^T$ where the sum is taken over all Young tableaux with shape λ, whose entries are all less than or equal to m, (the number of required indeterminates).

- Elementary Functions

  For a positive integer k, define the k-th elementary symmetric function, e_k, as the Schur function of $[1^k]$. Then, for any partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$, we define the elementary symmetric function indexed by λ to be $e_\lambda = \prod_i e_{\lambda_i}$.

- Homogeneous Functions

  For a positive integer k, define the k-th homogeneous symmetric function, h_k, as the Schur function of $[k]$. Then, for any partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$, we define the homogeneous symmetric function indexed by λ to be $h_\lambda = \prod_i h_{\lambda_i}$.

- Monomial Functions

  For a partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$, let x^λ be given by $x_1^{\lambda_1} x_2^{\lambda_2} \cdots x_l^{\lambda_l}$. The monomial symmetric function m_λ is the sum over the orbits of x^λ under the action of the symmetric group. Where the number of indeterminates exceeds the length of the partition, λ can be considered to have trailing zeros.

- Power Sums

  For a positive integer k, define the k-th power sum symmetric function, p_k, as the monomial function of $[k]$. Then, for any partition $\lambda = [\lambda_1, \lambda_2, \ldots, \lambda_l]$, we define the power sum symmetric function indexed by λ to be $p_\lambda = \prod_i p_{\lambda_i}$.

For each of these classes of functions, the set of all functions indexed by partitions of weight n forms a basis for Λ^n. Taking functions indexed by partitions of *all* weights gives a basis of Λ.

It is possible to define an inner product on Λ by requiring that the bases m_λ and h_λ be dual to each other:

$$\langle m_\lambda, h_{\lambda'} \rangle = \delta_{\lambda,\lambda'}$$

With respect to this inner product, the Schur functions are an orthonormal basis of Λ^n and Λ and the power sum basis is an orthogonal basis.

Example H146E1

We give an example illustrating the correspondence between symmetric functions written with respect to a symmetric function basis and symmetric polynomials. While the degree of the sym-

metric function fixes the degree of the resulting polynomial, the number of indeterminates is arbitrary.

```
> Q := Rationals();
> S := SFASchur(Rationals());
> e := S.[2,1];
> P2<[x]> := PolynomialRing(Q, 2);
> P2 ! e;
x[1]^2*x[2] + x[1]*x[2]^2
> P3<[y]> := PolynomialRing(Q, 3);
> P3 ! e;
y[1]^2*y[2] + y[1]^2*y[3] + y[1]*y[2]^2 + 2*y[1]*y[2]*y[3] + y[1]*y[3]^2 +
    y[2]^2*y[3] + y[2]*y[3]^2
> P4<[z]> := PolynomialRing(Q, 4);
> P4 ! e;
z[1]^2*z[2] + z[1]^2*z[3] + z[1]^2*z[4] + z[1]*z[2]^2 + 2*z[1]*z[2]*z[3] +
    2*z[1]*z[2]*z[4] + z[1]*z[3]^2 + 2*z[1]*z[3]*z[4] + z[1]*z[4]^2 +
    z[2]^2*z[3] + z[2]^2*z[4] + z[2]*z[3]^2 + 2*z[2]*z[3]*z[4] +
    z[2]*z[4]^2 + z[3]^2*z[4] + z[3]*z[4]^2
>
> P5<[w]> := PolynomialRing(Q, 5);
> P5 ! e;
w[1]^2*w[2] + w[1]^2*w[3] + w[1]^2*w[4] + w[1]^2*w[5] + w[1]*w[2]^2 +
    2*w[1]*w[2]*w[3] + 2*w[1]*w[2]*w[4] + 2*w[1]*w[2]*w[5] + w[1]*w[3]^2 +
    2*w[1]*w[3]*w[4] + 2*w[1]*w[3]*w[5] + w[1]*w[4]^2 + 2*w[1]*w[4]*w[5] +
    w[1]*w[5]^2 + w[2]^2*w[3] + w[2]^2*w[4] + w[2]^2*w[5] + w[2]*w[3]^2 +
    2*w[2]*w[3]*w[4] + 2*w[2]*w[3]*w[5] + w[2]*w[4]^2 + 2*w[2]*w[4]*w[5] +
    w[2]*w[5]^2 + w[3]^2*w[4] + w[3]^2*w[5] + w[3]*w[4]^2 +
    2*w[3]*w[4]*w[5] + w[3]*w[5]^2 + w[4]^2*w[5] + w[4]*w[5]^2
```

146.2 Creation

146.2.1 Creation of Symmetric Function Algebras

Symmetric functions are symmetric polynomials over some coefficient ring, hence the algebra of symmetric functions is defined by specifying this ring. There are five standard bases that can be used to express symmetric functions. In MAGMA there are separate constructions for creating an algebra using each distinct basis.

`SymmetricFunctionAlgebra(R)`

`SFA(R)`

`Basis` MonStgElt *Default :* "*Schur*"

Given a ring R, return the algebra of symmetric functions over R. By default this algebra will use the basis of `"Schur"` functions, though the parameter `Basis` allows the user to specify that the basis be one of the `"Homogeneous"`, `"PowerSum"`, `"Elementary"` or `"Monomial"` functions.

`SymmetricFunctionAlgebraSchur(R)`

`SFASchur(R)`

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Schur* symmetric functions, which are indexed by partitions.

`SymmetricFunctionAlgebraHomogeneous`

`SFAHomogeneous(R)`

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Homogeneous* symmetric functions, which are indexed by partitions.

`SymmetricFunctionAlgebraPower(R)`

`SFAPower(R)`

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Power Sum* symmetric functions, which are indexed by partitions.

`SymmetricFunctionAlgebraElementary(R)`

`SFAElementary(R)`

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Elementary* symmetric functions, which are indexed by partitions.

`SymmetricFunctionAlgebraMonomial(R)`

`SFAMonomial(R)`

Given a commutative ring with unity R, create the algebra of symmetric functions (polynomials) with coefficients from R. Its elements are expressed in terms of the basis of *Monomial* symmetric functions, which are indexed by partitions.

Example H146E2

A symmetric polynomial can be expressed in terms of any of the 5 standard bases.

```
> R := Rationals();
> S := SymmetricFunctionAlgebraSchur(R);
Symmetric Algebra over Rational Field, Schur symmetric functions as basis
> E := SymmetricFunctionAlgebra(R : Basis := "Elementary");
Symmetric Algebra over Rational Field, Elementary symmetric functions as basis
> f1 := S.[2,1];
> f1;
S.[2,1]
>
> f2 := E ! f1;
> f2;
E.[2,1] - E.[3]
> f1 eq f2;
true
>
> P<x,y,z> := PolynomialRing(R, 3);
> P ! f1;
x^2*y + x^2*z + x*y^2 + 2*x*y*z + x*z^2 + y^2*z + y*z^2
> P ! f2;
x^2*y + x^2*z + x*y^2 + 2*x*y*z + x*z^2 + y^2*z + y*z^2
```

146.2.2 Creation of Symmetric Functions

For each of the 5 different standard basis for symmetric functions, basis elements are indexed via partitions (weakly decreasing positive sequences). The *weight* of a partition is the sum of its entries, and gives the degree of the element.

For example, $[3, 1]$ is a partition of weight 4, hence a symmetric function basis element corresponding to $[3, 1]$ will be a symmetric polynomial of degree 4.

General symmetric functions are linear combinations of basis elements and can be created by taking such linear combinations or via coercion from either another basis or directly from a polynomial.

`A . P`

Given a partition P, which is a weakly decreasing positive sequence of integers, return the basis element of the algebra of symmetric functions A corresponding to P.

`A . i`

Given a positive integer i, return the basis element of the algebra of symmetric functions A corresponding to the partition $[i]$.

Example H146E3

We can create elements via linear combinations of basis elements.

```
> R := Rationals();
> M := SFAMonomial(R);
> M.[3,1];
M.[3,1]
>  M.4;
M.[4]
> 3 * M.[3,1] - 1/2 * M.4;
3*M.[3,1] - 1/2*M.[4]
```

`IsCoercible(A, f)`

`A ! f`

Given a multivariate polynomial f which is symmetric in all of its indeterminates (and hence is a symmetric function), return f as an element of the algebra of symmetric functions A. This element returned will be expressed in terms of the symmetric function basis of A.

Example H146E4

Symmetric polynomials are symmetric functions. These can be coerced into algebras of symmetric functions and hence expressed in terms of the relevant basis. Polynomials which are not symmetric cannot be coerced.

```
> R := Rationals();
> M := SFAMonomial(R);
> P<[x]> := PolynomialRing(R, 3);
> f := -3*x[1]^3 + x[1]^2*x[2] + x[1]^2*x[3] + x[1]*x[2]^2 + x[1]*x[3]^2 -
>          3*x[2]^3 + x[2]^2*x[3] + x[2]*x[3]^2 - 3*x[3]^3;
> M!f;
M.[2,1] - 3*M.[3]
>  M ! (x[1] + x[2]*x[3]);
>> M ! (x[1] + x[2]*x[3]);
     ^
Runtime error in '!': Polynomial is not symmetric
```

`A ! r`

Create the scalar element r in the symmetric function algebra A.

Example H146E5

```
> R := Rationals();
> P := SFAPower(R);
> m := P!3;
> m;
3
> Parent(m);
Symmetric Algebra over Rational Field, Power sum symmetric functions as basis
> m + P.[3,2];
3 + P.[3,2]
```

`A ! m`

Given a symmetric function algebra A and a symmetric function m (possibly with a different basis), return the element m expressed in terms of the basis of A.

Example H146E6

A symmetric function can easily be expressed in terms of any of the symmetric function bases, or as a polynomial.

```
> R := Rationals();
> S := SFASchur(R);
> H := SFAHomogeneous(R);
> P := SFAPower(R);
> E := SFAElementary(R);
> M := SFAMonomial(R);
>
> m := S.[3,1];
> S!m;
S.[3,1]
> H!m;
H.[3,1] - H.[4]
> P!m;
1/8*P.[1,1,1,1] + 1/4*P.[2,1,1] - 1/8*P.[2,2] - 1/4*P.[4]
> E!m;
E.[2,1,1] - E.[3,1] - E.[2,2] + E.[4]
> M!m;
3*M.[1,1,1,1] + 2*M.[2,1,1] + M.[3,1] + M.[2,2]
>
> PP<[x]> := PolynomialRing(R, 4);
> PP ! m;
x[1]^3*x[2] + x[1]^3*x[3] + x[1]^3*x[4] + x[1]^2*x[2]^2 + 2*x[1]^2*x[2]*x[3] +
    2*x[1]^2*x[2]*x[4] + x[1]^2*x[3]^2 + 2*x[1]^2*x[3]*x[4] + x[1]^2*x[4]^2 +
    x[1]*x[2]^3 + 2*x[1]*x[2]^2*x[3] + 2*x[1]*x[2]^2*x[4] + 2*x[1]*x[2]*x[3]^2 +
    3*x[1]*x[2]*x[3]*x[4] + 2*x[1]*x[2]*x[4]^2 + x[1]*x[3]^3 +
    2*x[1]*x[3]^2*x[4] + 2*x[1]*x[3]*x[4]^2 + x[1]*x[4]^3 + x[2]^3*x[3] +
```

```
x[2]^3*x[4] + x[2]^2*x[3]^2 + 2*x[2]^2*x[3]*x[4] + x[2]^2*x[4]^2 +
x[2]*x[3]^3 + 2*x[2]*x[3]^2*x[4] + 2*x[2]*x[3]*x[4]^2 + x[2]*x[4]^3 +
x[3]^3*x[4] + x[3]^2*x[4]^2 + x[3]*x[4]^3
```

Example H146E7

We show that the k-th homogeneous symmetric function is defined as the sum over all monomial symmetric functions indexed by partitions of weight k.

```
> R := Rationals();
> M := SFA(R : Basis := "Monomial");
> H := SFA(R : Basis := "Homogeneous");
>
> H ! (M.[1]);
H.[1]
> H ! (M.[2] + M.[1,1]);
H.[2]
> H ! (M.[3] + M.[2,1] + M.[1,1,1]);
H.[3]
> H ! (M.[4] + M.[3,1] + M.[2,2] + M.[2,1,1] + M.[1,1,1,1]);
H.[4]
>
> k := 5;
> H ! &+ [ M.P : P in Partitions(k)];
H.[5]
> k := 10;
> H ! &+ [ M.P : P in Partitions(k)];
H.[10]
```

146.3 Structure Operations

146.3.1 Related Structures

The main structure related to an algebra of symmetric functions is its coefficient ring. An algebra of symmetric functions belongs to the MAGMA category `AlgSym`.

`BaseRing(L)`

`CoefficientRing(L)`

Return the coefficient ring of an algebra of symmetric functions L.

`Category(L)` `Parent(L)` `PrimeRing(L)`

146.3.2 Ring Predicates and Booleans

The usual ring functions returning boolean values are available on algebras of symmetric functions.

`IsCommutative(L)` `IsUnitary(L)` `IsFinite(L)` `IsOrdered(L)`
`IsField(L)` `IsEuclideanDomain(L)` `IsPID(L)` `IsUFD(L)`
`IsDivisionRing(L)` `IsEuclideanRing(L)` `IsDomain(L)`
`IsPrincipalIdealRing(L)`

`L eq M`
`L ne M`

Return `true` if L and M are equal (respectively, not equal). MAGMA considers two algebras to be equal if they are over the same ring.

146.3.3 Predicates on Basis Types

`HasSchurBasis(A)`

Returns `true` if A is an algebra with a Schur basis.

`HasHomogeneousBasis(A)`
`HasElementaryBasis(A)`
`HasPowerSumBasis(A)`
`HasMonomialBasis(A)`

Returns `true` if A is an algebra with a homogeneous, elementary, power sum or monomial basis respectively.

146.4 Element Operations

146.4.1 Parent and Category

The category for elements in algebras of symmetric functions is `AlgSymElt`.

`Parent(f)` `Category(f)`

146.4.2 Print Styles

By default, elements are printed using a lexicographical ordering on the partitions indexing the basis elements. For partitions of the same weight w this ordering is the reverse of that in `Partitions(w)`. More generally, basis elements indexed by partitions with smaller entries print first (see 10.5.2). It is possible to rearrange the linear combination of basis elements so that the basis elements indexed by the longest partition print first and also so that the basis elements indexed by the partition with greatest maximal part (first entry) print first.

`A ‘ PrintStyle`

Retrieve or set the style in which elements of the algebra A will print. The default is the lexicographical ordering described above, `"Lex"`. Other options are `"Length"` and `"MaximalPart"`.

Example H146E8

We demonstrate the setting of the `PrintStyle` and the corresponding printing of elements.

```
> M := SFAMonomial(Rationals());
> M‘PrintStyle;
Lex
> P := Partitions(3);
> P;
[
    [ 3 ],
    [ 2, 1 ],
    [ 1, 1, 1 ]
]
> f := &+[M.p : p in P];
> f;
M.[1,1,1] + M.[2,1] + M.[3]
> M‘PrintStyle := "Length";
> f;
M.[3] + M.[2,1] + M.[1,1,1]
> M‘PrintStyle := "MaximalPart";
> f;
M.[1,1,1] + M.[2,1] + M.[3]
```

146.4.3 Additive Arithmetic Operators

The usual unary and binary ring element operations are available for symmetric functions. It is possible to combine elements of different algebras (so long as the coefficient rings are compatible) in these operations. Where the elements are written with respect to different bases, the result will be written with respect to the basis of the second operand.

`+ a` `- a`

`a + b` `a - b`

`a +:= b` `a -:= b`

146.4.4 Multiplication

Where the elements being multiplied are expressed with respect to different bases, the result will again be expressed with respect to the basis of the second operand. The algorithm used for multiplication is dependent on the bases with respect to which the elements are expressed.

`a * b`

`a *:= b`

The product of the symmetric functions a and b.

If b is expressed with respect to a Schur basis and a is expressed with respect to a power sum, elementary or monomial basis then the algorithm based on Muirs rule [Mui60] is used. If a is expressed with respect to a homogeneous basis the algorithm based on the Pieri rule [Mac95] is used and if a is also expressed with respect to a Schur basis then the method of Schubert polynomials [LS85] is used.

Special algorithms are also used if b is expressed with respect to a monomial basis and a is expressed with respect to a homogeneous, elementary or monomial basis.

When both a and b have a homogeneous, elementary or power sum basis, the multiplication of basis elements involves the merging of the parts of the partitions. This follows from the definition of the basis elements $f_\lambda = \prod_i f_{\lambda_i}$ where f_λ is a homogeneous, elementary or power sum basis element.

Otherwise a is coerced into the parent of b before the elements are multiplied.

The degree of the result is the sum of the degrees of both the operands.

`a ^ k`

Example H146E9

```
> Q := Rationals();
> s := SFASchur(Q);
> m := SFAMonomial(Q);
> m.[3]*s.[2,1];
s.[2,1,1,1,1] + s.[5,1] - s.[2,2,2] - s.[3,3]
```

To illustrate the merging of the partitions :

```
> E := SFAElementary(Q);
> E.4*E.3*E.1;
E.[4,3,1]
```

This is the definition of $e_{[4,3,1]}$.

146.4.5 Plethysm

Plethysm is also referred to as composition of symmetric functions.

`a ~ b`

This operator computes the plethysm or composition of the symmetric functions a and b. The result is given with respect to the basis of the second operand.

The degree of the result is the product of the degrees of operands which may be very large.

Example H146E10

```
> Q := Rationals();
> s := SFASchur(Q);
> m := SFAMonomial(Q);
> m.[3]~s.[2,1];
s.[4,1,1,1,1,1] - s.[3,2,1,1,1,1] - s.[5,1,1,1,1] + s.[2,2,2,1,1,1] +
s.[3,3,1,1,1] + s.[6,1,1,1] - s.[2,2,2,2,1] - s.[3,3,2,1] - s.[6,2,1] +
s.[4,4,1] + s.[3,2,2,2] + s.[5,2,2] - s.[4,3,2] + 2*s.[3,3,3] + s.[6,3] -
s.[5,4]
```

146.4.6 Boolean Operators

`IsHomogeneous(s)`

Returns `true` if the partitions indexing the basis elements present in the symmetric function s are all of the same weight. This implies that each term has the same degree so is the same as the polynomial expansion of s being a homogeneous polynomial.

`IsZero(s)` `IsOne(s)` `IsMinusOne(s)`

`s eq t`

`s ne t`

Return `true` if the symmetric functions s and t are (not) the same.

146.4.7 Accessing Elements

`Coefficient(s, p)`

Given a symmetric function s return the coefficient of the basis element A_p, where A is the parent of s and p is a sequence defining a partition. The coefficient may be zero.

`Support(s)`

Return two parallel sequences of the partitions indexing the basis elements and the coefficients of those basis elements in the symmetric function s, which is a linear combination of basis elements.

`Length(s)`

Given a symmetric function s, return the length of s, i.e., the number of basis elements having non zero coefficients in s, with respect to the current basis.

Example H146E11

We pull apart an element and show that we can put it back together again.

```
> H := SFAHomogeneous(Rationals());
> P := Partitions(4);
> f := &+[Random(1, 5)*H.p : p in P];
> f;
H.[1,1,1,1] + 4*H.[2,1,1] + 4*H.[2,2] + 5*H.[3,1] + 5*H.[4]
> s, e := Support(f);
> s, e;
[
    [ 1, 1, 1, 1 ],
    [ 2, 1, 1 ],
    [ 2, 2 ],
    [ 3, 1 ],
    [ 4 ]
]
[ 1, 4, 4, 5, 5 ]
> f eq &+[e[i]*H.s[i] : i in [1 .. Length(f)]];
true
```

`Degree(s)`

Given a symmetric function s, return the degree of s, i.e., the maximal degree of the basis elements having non zero coefficients in s, which is the maximal weight of the partitions indexing those basis elements.

146.4.8 Multivariate Polynomials

A symmetric function may be seen as a polynomial in any number of variables.

`P ! s`

Return the polynomial expansion of the symmetric function s in the polynomial ring P.

Example H146E12

```
> S := SFASchur(GF(7));
> s := S.[3,1];
```

Now we compute the multivariate polynomial we get by restricting to 5 variables.

```
> G<e1, e2, e3, e4, e5> := PolynomialRing(GF(7), 5);
> p := G!s;
> p;
e1^3*e2 + e1^3*e3 + e1^3*e4 + e1^3*e5 + e1^2*e2^2 + 2*e1^2*e2*e3 + 2*e1^2*e2*e4
    + 2*e1^2*e2*e5 + e1^2*e3^2 + 2*e1^2*e3*e4 + 2*e1^2*e3*e5 + e1^2*e4^2 +
    2*e1^2*e4*e5 + e1^2*e5^2 + e1*e2^3 + 2*e1*e2^2*e3 + 2*e1*e2^2*e4 +
    2*e1*e2^2*e5 + 2*e1*e2*e3^2 + 3*e1*e2*e3*e4 + 3*e1*e2*e3*e5 + 2*e1*e2*e4^2 +
    3*e1*e2*e4*e5 + 2*e1*e2*e5^2 + e1*e3^3 + 2*e1*e3^2*e4 + 2*e1*e3^2*e5 +
    2*e1*e3*e4^2 + 3*e1*e3*e4*e5 + 2*e1*e3*e5^2 + e1*e4^3 + 2*e1*e4^2*e5 +
    2*e1*e4*e5^2 + e1*e5^3 + e2^3*e3 + e2^3*e4 + e2^3*e5 + e2^2*e3^2 +
    2*e2^2*e3*e4 + 2*e2^2*e3*e5 + e2^2*e4^2 + 2*e2^2*e4*e5 + e2^2*e5^2 + e2*e3^3
    + 2*e2*e3^2*e4 + 2*e2*e3^2*e5 + 2*e2*e3*e4^2 + 3*e2*e3*e4*e5 + 2*e2*e3*e5^2
    + e2*e4^3 + 2*e2*e4^2*e5 + 2*e2*e4*e5^2 + e2*e5^3 + e3^3*e4 + e3^3*e5 +
    e3^2*e4^2 + 2*e3^2*e4*e5 + e3^2*e5^2 + e3*e4^3 + 2*e3*e4^2*e5 + 2*e3*e4*e5^2
    + e3*e5^3 + e4^3*e5 + e4^2*e5^2 + e4*e5^3
```

To check the polynomial is actually symmetric, we can use the MAGMA intrinsic `IsSymmetric`, which also computes an expansion as a sum of elementary symmetric polynomials.

```
> IsSymmetric(p,G);
true e1^2*e2 + 6*e1*e3 + 6*e2^2 + e4
```

Which is identical to the result of

```
> E := SFAElementary(GF(7));
> E!s;
E.[2,1,1] + 6*E.[2,2] + 6*E.[3,1] + E.[4]
```

Example H146E13

These conversions may be used to change the alphabet of a symmetric function. For example, if we substitute the variable x_i by $x_i + 1$, the result is again a symmetric function, however we use polynomials to do the evaluation.

```
> S := SFASchur(Rationals());
> R<a, b, c, d, e> := PolynomialRing(Rationals(), 5);
```

```
> p := Polynomial(S.[3,2], R);
> q := Evaluate(p, [a+1, b+1, c+1, d+1, e+1]);
> x, y := IsCoercible(S, q);
> y;
175 + 175*S.[1] + 70*S.[1,1] + 70*S.[2] + 35*S.[2,1] + 7*S.[2,2] + 10*S.[3] +
5*S.[3,1] + S.[3,2]
```

146.4.9 Frobenius Homomorphism

There is an automorphism which maps the elementary symmetric function to the homogeneous symmetric function.

`Frobenius(s)`

Given any symmetric function s, compute the image of s under the Frobenius automorphism. The image will have the same parent as s. When the power sum symmetric functions are involved, it may be necessary to work with a coefficient ring which allows division by an integer.

Example H146E14

It is known that the Frobenius automorphism on the Schur functions acts just by conjugating the indexing partitions.

```
> S := SFASchur(Integers());
> E := SFAElementary(Integers());
> h := S!E.[3,3,3];
> h;
S.[1,1,1,1,1,1,1,1,1] + 2*S.[2,1,1,1,1,1,1,1] + S.[3,1,1,1,1,1,1] +
3*S.[2,2,1,1,1,1,1] + 2*S.[3,2,1,1,1,1] + 4*S.[2,2,2,1,1,1] + S.[3,3,1,1,1] +
3*S.[3,2,2,1,1] + 2*S.[2,2,2,2,1] + 2*S.[3,3,2,1] + S.[3,2,2,2] + S.[3,3,3]
```

Now apply the Frobenius automorphism.

```
> f:=Frobenius(h);
> f;
S.[3,3,3] + 2*S.[4,3,2] + S.[4,4,1] + S.[5,2,2] + 3*S.[5,3,1] + 2*S.[5,4] +
2*S.[6,2,1] + 4*S.[6,3] + S.[7,1,1] + 3*S.[7,2] + 2*S.[8,1] + S.[9]
```

To check whether the coefficient of the basis element indexed by a partition in one element is the same as the coefficient of the basis element indexed by the conjugate partition in the other :

```
> p:=Partitions(f);
> for pp in p do
>    if Coefficient(h, ConjugatePartition(pp)) ne Coefficient(f, pp) then
>       print pp;
>    end if;
> end for;
```

146.4.10 Inner Product

An inner product on Λ is defined by

$$\langle m_\lambda, h_{\lambda'} \rangle = \delta_{\lambda,\lambda'}.$$

This definition ensures that the bases m_λ and h_λ are dual to each other. This is the inner product used by MAGMA.

`InnerProduct(a,b)`

Computes the inner product of the symmetric functions a and b.

Example H146E15

The inner product of a elementary symmetric function indexed by a partition and the homogeneous symmetric function indexed by the conjugate partition is 1. (This allows the computation of irreducible representations of the symmetric group.) To check this with MAGMA, we do the following:

```
> E := SFAElementary(Rationals());
> H := SFAHomogeneous(Rationals());
> p:=RandomPartition(45);
> pc:=ConjugatePartition(p);
> InnerProduct(E.p,H.pc);
1
```

As it should be the result is 1.

146.4.11 Combinatorial Objects

The Schur function is the generating function of standard tableaux. Therefore, it is possible to get the corresponding tableaux.

`Tableaux(sf, m)`

Given a Schur function sf over the integers with positive coefficients, return the multiset of the tableaux, with maximal entry m, for which sf is the generating function.

146.4.12 Symmetric Group Character

A Schur function indexed by a single partition, i.e. a basis element, corresponds to an irreducible character of the symmetric group.

`SymmetricCharacter(sf)`

Given an element sf of an algebra of symmetric functions return a linear combination of irreducible characters of the symmetric group, whose coefficients are the coefficients of sf with respect to the Schur function basis.

Example H146E16

We look at a result in the representation theory of the symmetric group. There is exactly one irreducible character contained in both the induced character from the identity character of a Young subgroup indexed by the partition I and the induced character from the alternating character of a Young subgroup indexed by the partition J, conjugate to I. As the induced character of the identity character of a Young subgroup corresponds to the homogeneous symmetric function h_I and the induced character of the alternating character corresponds to the elementary symmetric function e_J, we can verify this using the following routine:

```
> H := SFAHomogeneous(Rationals());
> E := SFAElementary(Rationals());
> p := Partitions(7);
> for I in p do
>     a := SymmetricCharacter(H.I);
>     J := ConjugatePartition(I);
>     b := SymmetricCharacter(E.J);
>     i := InnerProduct(a,b);
>     if i ne 1 then print i; end if;
> end for;
```

And there should be no output.

146.4.13 Restrictions

It is possible to form symmetric functions whose support is a subset of the support of a given symmetric function, subject to some restrictions.

`RestrictDegree(a, n)`

`Exact` BoolElt *Default :* `true`

Return the symmetric function which is a linear combination of those basis elements of the symmetric function a with degree n. This is the restriction of a into the submodule Λ^n. If `Exact` is `false` then the basis elements included will have degree $\leq n$. This is the restriction of a into $\bigcup_{k \leq n} \Lambda^k$.

`RestrictPartitionLength(a, n)`

`Exact` BoolElt *Default :* `true`

Return the symmetric function which is a linear combination of those basis elements of the symmetric function a whose indexing partitions are of length n. If `Exact` is `false`, then the indexing partitions will be of length $\leq n$.

`RestrictParts(a, n)`

`Exact` BoolElt *Default :* `true`

Return the symmetric function which is a linear combination of those basis elements of the symmetric function a whose indexing partitions have maximal part n. If `Exact` is `false`, then the indexing partitions will have maximal part $\leq n$.

146.5 Transition Matrices

One area of interest in the theory of symmetric functions is the study of the change of bases between the five different bases. The matrices of change of base between the s_λ, h_λ, m_λ and the e_λ are all integer matrices. Only when changing from one of these four bases to the p_λ, is the matrix over the rationals. For a discussion of their computation and interactions see [Mac95, pages 54–58].

In MAGMA there are routines available to obtain all these matrices. These routines are described below. Some interactions of these matrices are verified in examples.

146.5.1 Transition Matrices from Schur Basis

`SchurToMonomialMatrix(n)`

Computes the matrix for the expansion of a Schur function indexed by a partition of weight n as a sum of monomial symmetric functions. This matrix is also known as the table of *Kostka* numbers. These are the numbers of tableaux of each shape and content. The content of a tableau prescribes its entries, so for example a tableau with content $[2, 1, 4, 1]$ has two 1's, one 2, four 3's and one 4. The entries of the matrix are non negative integers. The matrix is upper triangular.

Example H146E17

Compute the base change matrix from the Schur functions to the monomial symmetric functions for degree 5. The entries in this matrix are what are known as the *Kostka* numbers. They count the number of young tableaux on a given shape Look on the order of the labelling partitions, and check whether the entry 3 in the upper right corner is right by generating the corresponding tableaux.

```
> M := SchurToMonomialMatrix(5);
> M;
[1 1 1 1 1 1 1]
[0 1 1 2 2 3 4]
[0 0 1 1 2 3 5]
[0 0 0 1 1 3 6]
[0 0 0 0 1 2 5]
[0 0 0 0 0 1 4]
[0 0 0 0 0 0 1]
> Parts := Partitions(5);
> Parts;
[
    [ 5 ],
    [ 4, 1 ],
    [ 3, 2 ],
    [ 3, 1, 1 ],
    [ 2, 2, 1 ],
    [ 2, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1 ]
]
```

```
> #TableauxOnShapeWithContent(Parts[2], Parts[6]);
3
> M[2,6];
3
```

`SchurToHomogeneousMatrix(n)`

Computes the matrix for the expansion of a Schur function indexed by a partition of weight n as a sum of homogeneous symmetric functions. The entries of the matrix are positive and negative integers. The matrix is lower triangular. It is the transpose of the matrix returned by `MonomialToSchurMatrix(n)`.

`SchurToPowerSumMatrix(n)`

Computes the matrix for the expansion of a Schur function indexed by a partition of weight n as a sum of power sum symmetric functions. The entries of the matrix are rationals.

`SchurToElementaryMatrix(n)`

Computes the matrix for the expansion of a Schur function indexed by a partition of weight n as a sum of elementary symmetric functions. The entries of the matrix are positive and negative integers. The matrix is upper left triangular.

Example H146E18

We verify by hand that the action of base change matrix is the same as coercion.

```
> S := SFASchur(Rationals());
> P := SFAPower(Rationals());
> NumberOfPartitions(4);
5
> m := SchurToPowerSumMatrix(4);
> Partitions(4);
[
    [ 4 ],
    [ 3, 1 ],
    [ 2, 2 ],
    [ 2, 1, 1 ],
    [ 1, 1, 1, 1 ]
]
> s := S.[3, 1] + 5*S.[1, 1, 1, 1] - S.[4];
> s;
5*S.[1,1,1,1] + S.[3,1] - S.[4]
> p, c := Support(s);
> c;
[ 5, 1, -1 ]
> p;
[
```

```
    [ 1, 1, 1, 1 ],
    [ 3, 1 ],
    [ 4 ]
]
> cm := Matrix(Rationals(), 1, 5, [-1, 1, 0, 0, 5]);
> cm*m;
[-7/4  4/3  3/8 -5/4 7/24]
> P!s;
7/24*P.[1,1,1,1] - 5/4*P.[2,1,1] + 3/8*P.[2,2] + 4/3*P.[3,1] - 7/4*P.[4]
```

The coefficients of the coerced element are the reverse of the matrix product, consistent with the partition in the coerced element being in reverse order to those in `Partitions(4)`.

146.5.2 Transition Matrices from Monomial Basis

`MonomialToSchurMatrix(n)`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight n as a sum of Schur symmetric functions. The entries of the matrix are positive and negative integers. The matrix is upper triangular. It is the transpose of the matrix returned by `SchurToHomogeneousMatrix(n)`.

`MonomialToHomogeneousMatrix(n)`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight n as a sum of homogeneous symmetric functions. The entries are positive and negative integers.

`MonomialToPowerSumMatrix(n)`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight n as a sum of power sum symmetric functions. The entries are rationals. The matrix is lower triangular.

`MonomialToElementaryMatrix(n)`

Computes the matrix for the expansion of a monomial symmetric function indexed by a partition of weight n as a sum of elementary symmetric functions. The entries of the matrix are positive and negative integers. The matrix is upper left triangular.

146.5.3 Transition Matrices from Homogeneous Basis

`HomogeneousToSchurMatrix(n)`

Computes the matrix for the expansion of a homogeneous symmetric function indexed by a partition of weight n as a sum of Schur symmetric functions. The entries of the matrix are positive integers. The matrix is lower triangular.

Example H146E19

It is known that the matrix computed by `HomogeneousToSchurMatrix` is the transpose of the matrix computed by `SchurToMonomialMatrix`.

```
> SchurToMonomialMatrix(7) eq Transpose(HomogeneousToSchurMatrix(7));
true
```

`HomogeneousToMonomialMatrix(n)`

Computes the matrix M for the expansion of a homogeneous symmetric function indexed by a partition of weight n as a sum of monomial symmetric functions. The entries of the matrix are positive integers. The matrix has no zero entries.

The coefficient $M_{\mu,\lambda}$ in the expansion $h_\lambda = \sum_\mu M_{\mu,\lambda} m_\mu$ is the number of non negative integer matrices with row sums λ_i and column sums μ_j, see [Mac95, page 57].

Example H146E20

The matrix converting from homogeneous basis to monomial basis is symmetric.

```
> IsSymmetric(HomogeneousToMonomialMatrix(7));
true
```

`HomogeneousToPowerSumMatrix(n)`

Computes the matrix for the expansion of a homogeneous symmetric function indexed by a partition of weight n as a sum of power sum symmetric functions. The entries of the matrix are positive rationals. The matrix is upper triangular.

`HomogeneousToElementaryMatrix(n)`

Computes the matrix for the expansion of a homogeneous symmetric function indexed by a partition of weight n as a sum of elementary symmetric functions. The entries of the matrix are integers. The matrix is upper triangular.

Example H146E21

It is known that the matrix compute by `HomogeneousToElementaryMatrix` is the same as the matrix computed by `ElementaryToHomogeneousMatrix`.

```
> HomogeneousToElementaryMatrix(7) eq ElementaryToHomogeneousMatrix(7);
true
```

146.5.4 Transition Matrices from Power Sum Basis

`PowerSumToSchurMatrix(n)`

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight n as a sum of Schur symmetric functions. The entries of the matrix are positive and negative integers. This matrix is the character table of the symmetric group.

Example H146E22

The matrix returned by `PowerSumToSchurMatrix` is compared to the character table of the appropriate symmetric group.

```
> PowerSumToSchurMatrix(5);
[ 1 -1  0  1  0 -1  1]
[ 1  0 -1  0  1  0 -1]
[ 1 -1  1  0 -1  1 -1]
[ 1  1 -1  0 -1  1  1]
[ 1  0  1 -2  1  0  1]
[ 1  2  1  0 -1 -2 -1]
[ 1  4  5  6  5  4  1]
> CharacterTable(Sym(5));


Character Table
---------------


-----------------------------
Class |   1  2  3  4  5  6  7
Size  |   1 10 15 20 30 24 20
Order |   1  2  2  3  4  5  6
-----------------------------
p  =  2   1  1  1  4  3  6  4
p  =  3   1  2  3  1  5  6  2
p  =  5   1  2  3  4  5  1  7
-----------------------------
X.1   +   1  1  1  1  1  1  1
X.2   +   1 -1  1  1 -1  1 -1
```

```
X.3   +   4  2  0  1  0 -1 -1
X.4   +   4 -2  0  1  0 -1  1
X.5   +   5  1  1 -1 -1  0  1
X.6   +   5 -1  1 -1  1  0 -1
X.7   +   6  0 -2  0  0  1  0
```

In the character table the first row is the unity character, which corresponds to the first column of the transition matrix. The second row of the character table is the alternating character which corresponds to the last column of the transition matrix. The first column of the character table contains the dimensions of the irreducible characters, this is the last row of the transition matrix.

`PowerSumToMonomialMatrix(n)`

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight n as a sum of monomial symmetric functions. The entries of the matrix are positive integers. The matrix is lower triangular.

`PowerSumToHomogeneousMatrix(n)`

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight n as a sum of homogeneous symmetric functions. The entries of the matrix are integers. The matrix is upper triangular.

`PowerSumToElementaryMatrix(n)`

Computes the matrix for the expansion of a power sum symmetric function indexed by a partition of weight n as a sum of elementary symmetric functions. The entries of the matrix are integers. The matrix is upper triangular.

146.5.5 Transition Matrices from Elementary Basis

`ElementaryToSchurMatrix(n)`

Computes the matrix for the expansion of an elementary symmetric function indexed by a partition of weight n as a sum of Schur symmetric functions. The entries of the matrix are positive integers.

`ElementaryToMonomialMatrix(n)`

Computes the matrix M for the expansion of an elementary symmetric function indexed by a partition of weight n as a sum of monomial symmetric functions. The entries of the matrix are positive integers.

The coefficient $M_{\mu,\lambda}$ in the expansion $e_\lambda = \sum_\mu M_{\mu,\lambda} m_\mu$ is the number of 0-1 integer matrices with row sum λ_i and column sum μ_j, see [Mac95, page 57].

Example H146E23

The matrix converting from elementary basis to monomial basis is symmetric.

```
> IsSymmetric(ElementaryToMonomialMatrix(7));
true
```

`ElementaryToHomogeneousMatrix(n)`

Computes the matrix for the expansion of a elementary symmetric function indexed by a partition of weight n as a sum of homogeneous symmetric functions.

`ElementaryToPowerSumMatrix(n)`

Computes the matrix for the expansion of a elementary symmetric function indexed by a partition of weight n as a sum of power sum symmetric functions.

146.6 Bibliography

[**LS85**] Alain Lascoux and Marcel-Paul Schützenberger. Schubert polynomials and the Littlewood-Richardson rule. *Lett. Math. Phys.*, 10(2-3):111–124, 1985.

[**Mac95**] I. G. Macdonald. *Symmetric functions and Hall polynomials.* The Clarendon Press Oxford University Press, New York, second edition, 1995. With contributions by A. Zelevinsky, Oxford Science Publications.

[**Mui60**] Thomas Muir. *A treatise on the theory of determinants.* Dover Publications Inc., New York, 1960.

147 INCIDENCE STRUCTURES AND DESIGNS

Chapter 147
INCIDENCE STRUCTURES AND DESIGNS

147.1 Introduction

Since there is some variation between authors of the terminology employed in design theory, we begin with some definitions. An *incidence structure* is a triple $D = (P, B, I)$, where:

(a) P is a set, the elements of which are called *points*;

(b) B is a set, the elements of which are called *blocks*;

(c) I is an incidence relation between P and B, so that $I \subset P \times B$. The elements of I are called *flags*.

Usually, blocks will be subsets of P, so that instead of writing $(p, b) \in I$, we write $p \in b$. In general, repeated blocks are allowed so that different blocks may correspond to the same subset of P. If D has no repeated blocks, then we say that D is *simple*.

An incidence structure D is said to be *uniform* with *blocksize* k if D has at least one block and all blocks contain exactly k points. A uniform incidence structure is called *trivial* if each k-subset of the point set appears as a block (at least once).

Let $t \geq 0$ be an integer. Then an incidence structure D is said to be *t–balanced* if there exists an integer $\lambda \geq 1$ such that each t–subset of the point set is contained in exactly λ blocks of D.

A *near–linear space* is an incidence structure in which every block contains at least two points and any two points lie in at most one block. A *linear space* is a near–linear space in which any two points lie in *exactly* one block. It is usual, when discussing near–linear spaces, to use the term *line* in place of the term *block*.

Let v, k, t and λ be integers with $v \geq k \geq t \geq 0$ and $\lambda \geq 1$. A *t–design* with v points and *blocksize* k is an incidence structure $D = (P, B, I)$ where:

(a) The cardinality of P is v;

(b) D is uniform with blocksize k;

(c) D is simple;

(d) For each t–subset T of P there are exactly λ blocks of B incident with all the points of T (so D is t–balanced).

Such a design is usually referred to as a t–(v, k, λ) design. The parameter λ is called the *index* of the design. If b denotes the cardinality of B, a t–design with $v = b$ and $t \geq 2$ is called a *symmetric design*. A t–design with $\lambda = 1$ is called a *Steiner design*. A design which is trivial is also called a *complete design*. Note that a design D must contain at least one block (i.e $b > 0$).

The category names for the different families of incidence structures are as follows:

- Incidence structure : `Inc`
- Near-linear space : `IncNsp`
- Linear space : `IncLsp`
- t–design : `Dsgn`

147.2 Construction of Incidence Structures and Designs

`IncidenceStructure< v | X >`

`IncidenceStructure< P | X >`

Construct the incidence structure D having point set $P = \{@p_1, p_2, \ldots, p_v@\}$ (where $p_i = i$ for each i if the first form of the constructor is used), and block set $B = \{B_1, B_2, \ldots, B_b\}$ given by X. The value of X must be either:

(a) A list of subsets of the set P.

(b) A sequence, set or indexed set of subsets of P.

(c) A list of blocks of an existing incidence structure.

(d) A sequence, set or indexed set of blocks of an existing incidence structure.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be defined over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the incidence structure D.

(g) A set of codewords of a linear code with length v. The block set of D is taken to be the set of supports of the codewords.

The incidence structure D produced by this constructor will have P as its point set and B as its set of blocks. The function returns three values:

(i) The incidence structure D;

(ii) The point-set P for D; and

(iii) The block-set B for D.

`NearLinearSpace< v | X : parameters >`

`NearLinearSpace< P | X : parameters >`

`Check` BoolElt *Default :* `true`

Construct the near–linear space S on the set of points $P = \{@p_1, p_2, \ldots, p_v@\}$ (where $p_i = i$ for each i if the first form of the constructor is used), with lines $L = \{L_1, L_2, \ldots, L_b\}$ given by X. The value of X must be either:

(a) A list of subsets of the set P.

(b) A sequence, set or indexed set of subsets of P.

(c) A list of blocks of an existing incidence structure.

(d) A sequence, set or indexed set of blocks of an existing incidence structure.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the near–linear space S.

(g) A set of codewords of a linear code with length v. The line set of S is taken to be the set of supports of the codewords.

The set of lines L defined by X must satisfy two properties:

(a) Each line of L must contain at least two points;

(b) Any two points of P may lie on at most one line.
The optional boolean argument `Check` indicates whether or not to check that these two properties are satisfied.

The near–linear space S produced by this constructor will have P as its point set and L as its set of lines. The function returns three values:

(i) The near–linear space S;

(ii) The point-set P for S; and

(iii) The line-set L for S.

`LinearSpace< v | X :` *parameters* `>`

`LinearSpace< P | X :` *parameters* `>`

`Check` BoolElt *Default :* `true`

Construct the linear space S on the set of points $P = \{@p_1, p_2, \ldots, p_v@\}$ (where $p_i = i$ for each i if the first form of the constructor is used), with lines $L = \{L_1, L_2, \ldots, L_b\}$ given by X. The value of X must be either:

(a) A list of subsets of the set P.

(b) A sequence, set or indexed set of subsets of P.

(c) A list of blocks of an existing incidence structure.

(d) A sequence, set or indexed set of blocks of an existing incidence structure.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be defined over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the linear space S.

(g) A set of codewords of a linear code with length v. The line set of S is taken to be the set of supports of the codewords.

The set of lines L defined by X must satisfy two properties:

(a) Each line of L must contain at least two points;

(b) Any two points of P must lie on precisely one line.
The optional boolean argument `Check` indicates whether or not to check that these two properties are satisfied.

The linear space S produced by this constructor will have P as its point set and L as its set of lines. The function returns three values:

(i) The linear space S;

(ii) The point-set P for S; and

(iii) The line-set L for S.

```
Design< t, v | X : parameters >
```

```
Design< t, P | X : parameters >
```

`Check`	BoolElt	*Default :* `true`
`Al`	MonStg	*Default :* "*NoOrbits*"

Construct the t–design D on the set of points $P = \{@p_1, p_2, \ldots, p_v@\}$ (where $p_i = i$ for each i if the first form of the constructor is used), with blocks $B = \{B_1, B_2, \ldots, B_b\}$ given by X. The value of X must be either:

(a) A list of subsets of the set P.

(b) A sequence, set or indexed set of subsets of P.

(c) A list of blocks of an existing incidence structure.

(d) A sequence, set or indexed set of blocks of an existing incidence structure.

(e) A combination of the above.

(f) A $v \times b$ $(0, 1)$-matrix A, where A may be over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the t–(v, k, λ) design D.

(g) A set of codewords of a linear code with length v. The block set of D is taken to be the set of supports of the codewords.

The set of blocks B defined by X must satisfy three properties:

(a) Each block must contain the same number, k say, of points.

(b) Every t–subset of P must lie in the same number, λ say (where $\lambda > 0$), of blocks.

(c) There must be no repeated blocks.

The optional boolean argument `Check` indicates whether or not to check that these three properties are satisfied. The optional parameter `Al` can be used to specify the algorithm used for balance testing, see the introduction to Section 147.8 for a full description of its usage.

The t–(v, k, λ) design D produced by this constructor will have P as its point set and B as its set of blocks. The function returns three values:

(i) The design D;

(ii) The point-set P for D; and

(iii) The block-set B for D.

Example H147E1

The Fano plane, considered as a 2-$(7, 3, 1)$ design, may be constructed by the following statement:

```
> F := Design< 2, 7 | {1,2,3}, {1,4,5}, {1,6,7}, {2,4,7},
>                                {2,5,6}, {3,5,7}, {3,4,6} >;
> F: Maximal;
2-(7, 3, 1) Design with 7 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7 @}
Blocks:
    {1, 2, 3},
    {1, 4, 5},
    {1, 6, 7},
    {2, 4, 7},
    {2, 5, 6},
    {3, 5, 7},
    {3, 4, 6}
```

General incidence structures are allowed repeated blocks, as in the following:

```
> S := IncidenceStructure< {@ 4, 5, 7, 9 @} | [{4, 5, 7}, {7, 9},
>                                              {5, 7, 9} , {7, 9}] >;
> S: Maximal;
Incidence Structure on 4 points with 4 blocks
Points: {@ 4, 5, 7, 9 @}
Blocks:
    {4, 5, 7},
    {7, 9},
    {5, 7, 9},
    {7, 9}
```

We now construct a linear space by giving its incidence matrix:

```
> R := RMatrixSpace(Integers(), 5, 6);
> I := R![ 1, 0, 1, 1, 0, 0,
>          1, 0, 0, 0, 1, 1,
>          1, 1, 0, 0, 0, 0,
>          0, 1, 1, 0, 1, 0,
>          0, 1, 0, 1, 0, 1];
> L := LinearSpace< 5 | I >;
> L: Maximal;
Linear Space on 5 points with 6 lines
Points: {@ 1, 2, 3, 4, 5 @}
Lines:
    {1, 2, 3},
    {3, 4, 5},
    {1, 4},
    {1, 5},
    {2, 4},
```

```
    {2, 5}
```

Finally, we use the minimum weight codewords of the unextended binary Golay code to construct a 4-(23, 7, 1) design. Since we know that this is indeed a 4-design, we set the checking parameter to `false`.

```
> C := GolayCode(GF(2), false);
> C;
[23, 12, 7] Unextended Golay Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 1]
[0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 1]
[0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1]
[0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 1 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 1]
> minwt := MinimumWeight(C);
> minwt;
7
> wds := Words(C, minwt);
> D := Design< 4, Length(C) | wds : Check := false >;
> D;
4-(23, 7, 1) Design with 253 blocks
```

147.3 The Point-Set and Block-Set of an Incidence Structure

147.3.1 Introduction

An incidence structure created by Magma consists of three objects: the *point-set* P, the *block-set* B and the incidence structure D itself.

Although called the point-set and block-set, P and B are not actual MAGMA sets. They simply act as the parent structures for the points and blocks (respectively) of the incidence structure D, enabling easy creation of these objects via the `!` and `.` operators.

The point-set P belongs to the MAGMA category `IncPtSet`, and the block-set B to the category `IncBlkSet`.

In this section, the functions used to create point-sets, block-sets and the points and blocks themselves are described.

147.3.2 Creating Point-Sets and Block-Sets

As mentioned above, the point-set and block-set are returned as the second and third arguments of any function which creates an incidence structure. They can also be created via the following two functions.

`PointSet(D)`

Given an incidence structure D, return the point-set P of D.

`BlockSet(D)`

Given an incidence structure D, return the block-set B of D.

147.3.3 Creating Points and Blocks

For efficiency and clarity, the points and blocks of an incidence structure are given special types in MAGMA. The category names for points and blocks are `IncPt` and `IncBlk`, respectively. They can be created in the following ways.

`Point(D, i)`

The i-th point of the incidence structure D.

`P . i`

Given the point-set P of an incidence structure D and an integer i, return the i-th point of D.

`Representative(P)`
`Rep(P)`

Given the point-set P of an incidence structure D, return a representative point of D.

`Random(P)`

Given the point-set P of an incidence structure D, return a random point of D.

`P ! x`

Given the point-set P of an incidence structure D, return the point of D corresponding to the element x of the indexed set used to create D.

`Block(D, i)`

The i-th block of the incidence structure D.

`B . i`

Given the block-set B of an incidence structure D and an integer i, return the i-th block of D.

`Representative(B)`

`Rep(B)`

Given the block-set B of an incidence structure D, return a representative block of D.

`Random(B)`

Given the block-set B of an incidence structure D, return a random block of D.

`B ! S`

Given the block-set B of an incidence structure D, and a set S, tries to coerce S into B.

`Representative(b)`

`Rep(b)`

Given a block b of an incidence structure D, return a representative point of D which is incident with b.

`Random(b)`

Given a block b of an incidence structure D, return a random point incident with b.

Example H147E2

The following example shows how points and blocks of an incidence structure can be created.

```
> V := {@ 2, 4, 6, 8, 10 @};
> D, P, B := IncidenceStructure< V | {2, 4, 6}, {2, 8, 10}, {4, 6, 8} >;
> D;
Incidence Structure on 5 points with 3 blocks
> P;
Point-set of Incidence Structure on 5 points with 3 blocks
> B;
Block-set of Incidence Structure on 5 points with 3 blocks
> B.2;
{2, 8, 10}
> P.4;
8
> P!4;
4
> P.5 eq Point(D, 5);
true
> b := Random(B);
> b;
{2, 4, 6}
> Parent(b);
Block-set of Incidence Structure on 5 points with 3 blocks
> p := Rep(b);
> p;
```

```
2
> Parent(p);
Point-set of Incidence Structure on 5 points with 3 blocks
> B!{2, 8, 10};
{2, 8, 10}
```

147.4 General Design Constructions

Each of these functions returns three values:

(i) The incidence structure D;

(ii) The point-set P of D;

(iii) The block-set B of D.

147.4.1 The Construction of Related Structures

All operations defined for incidence structures apply also to near–linear spaces, linear spaces and designs.

`Complement(D)`

The complement of the incidence structure D.

`Dual(D)`

The dual of the incidence structure D.

`Contraction(D, p)`

Given an incidence structure $D = (P, B)$, and a point $p \in P$, form the incidence structure

$$E = (P - \{p\}, \{b - \{p\} : b \in B | p \in b\}).$$

Thus, E is constructed from D by deleting p and retaining only those blocks incident with it.

`Contraction(D, b)`

Given an incidence structure $D = (P, B)$, and a block $b \in B$, form the incidence structure

$$E = (b, \{b \cap c : c \in B | c \neq b\}).$$

Thus, E has point set b and its blocks are the non–empty intersections of b with the blocks of D other than b itself.

`Residual(D, b)`

Given an incidence structure $D = (P, B)$, and a block $b \in B$, form the incidence structure

$$E = (P - b, B - \{b\}).$$

Thus, E has point set $P - b$ and its blocks are the non–empty intersections of $P - b$ with the blocks of D.

`Residual(D, p)`

Given an incidence structure $D = (P, B)$, and a point $p \in P$, form the incidence structure

$$E = (P - \{p\}, \{x : x \in B | p \notin x\}).$$

Thus, E has point set $P - \{p\}$ and its blocks are the blocks of D which do not contain p.

`Simplify(D)`

Simplify the incidence structure D; i.e., remove repeated blocks from D.

`Sum(Q)`

Given a sequence $Q = [D_1, \ldots, D_l]$ of incidence structures, each of which is defined over the same set P of points, form the incidence structure obtained by taking the union of the block sets of $D_1, \ldots, D_l$. Thus, if $D_i = (P, B_i)$ then $D = (P, B_1 \cup \ldots \cup B_l)$.

`Union(D, E)`

The union of incidence structures D and E. That is, if $D = (P, B)$ and $E = (Q, C)$, then return $U = (P \cup Q, B \cup C)$. The point sets P and Q must be disjoint.

`Restriction(D, S)`

The restriction of the (near–)linear space D to the set of points S.

Example H147E3

We illustrate some of the above functions with an example.

```
> K := Design< 3, 8 | {1,3,7,8}, {1,2,4,8}, {2,3,5,8}, {3,4,6,8}, {4,5,7,8},
> {1,5,6,8}, {2,6,7,8}, {1,2,3,6}, {1,2,5,7}, {1,3,4,5}, {1,4,6,7}, {2,3,4,7},
> {2,4,5,6}, {3,5,6,7} >;
> CK := Contraction(K, Point(K, 8));
> RK := Residual(K, Block(K, 1));
> K: Maximal;
3-(8, 4, 1) Design with 14 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8 @}
Blocks:
    {1, 3, 7, 8},
    {1, 2, 4, 8},
```

```
    {2, 3, 5, 8},
    {3, 4, 6, 8},
    {4, 5, 7, 8},
    {1, 5, 6, 8},
    {2, 6, 7, 8},
    {1, 2, 3, 6},
    {1, 2, 5, 7},
    {1, 3, 4, 5},
    {1, 4, 6, 7},
    {2, 3, 4, 7},
    {2, 4, 5, 6},
    {3, 5, 6, 7}
> CK: Maximal;
2-(7, 3, 1) Design with 7 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7 @}
Blocks:
    {1, 3, 7},
    {1, 2, 4},
    {2, 3, 5},
    {3, 4, 6},
    {4, 5, 7},
    {1, 5, 6},
    {2, 6, 7}
> RK: Maximal;
Incidence Structure on 4 points with 13 blocks
Points: {@ 2, 4, 5, 6 @}
Blocks:
    {2, 4},
    {2, 5},
    {4, 6},
    {4, 5},
    {5, 6},
    {2, 6},
    {2, 6},
    {2, 5},
    {4, 5},
    {4, 6},
    {2, 4},
    {2, 4, 5, 6},
    {5, 6}
> RKS := Simplify(RK);
> RKS: Maximal;
Incidence Structure on 4 points with 7 blocks
Points: {@ 2, 4, 5, 6 @}
Blocks:
    {2, 4},
    {2, 5},
    {4, 6},
```

```
{4, 5},
{5, 6},
{2, 6},
{2, 4, 5, 6}
```

147.4.2 The Witt Designs

The 5–(12, 6, 1) and 5–(24, 8, 1) designs constructed by Witt, also known as the small and large Mathieu designs, respectively, can be constructed in Magma with the following function.

`WittDesign(n)`

The Witt 5–design on n points, where $n = 12$ or 24.

Example H147E4

We construct the Witt 5–(24, 8, 1) design and take its contraction at a point. This contraction is in fact isomorphic to the design constructed above from the unextended binary Golay code.

```
> D, P, B := WittDesign(24);
> D;
5-(24, 8, 1) Design with 759 blocks
> p := P.1;
> Cp := Contraction(D, p);
> Cp;
4-(23, 7, 1) Design with 253 blocks
```

147.4.3 Difference Sets and their Development

Let G be a group of order v and let k and λ be positive integers such that $1 < k < v$. A (v, k, λ) *difference set* for G is a set D of k group elements such that the set

$$\{gh^{-1} : g, h \in D | g \neq h\}$$

contains every non–identity element of G exactly λ times.

`DifferenceSet(p, t)`

The difference set of type given by t (which must be one of `"Q"`, `"H6"`, `"T"`, `"B"`, `"B0"`, `"O"`, `"O0"`, or `"W4"`) corresponding to the prime p. The types have the same interpretation as given by Marshall Hall in [Hal86], pp. 141–142.

`SingerDifferenceSet(n, q)`

The Singer difference set corresponding to a hyperplane of $\mathrm{PG}(n, q)$.

`IsDifferenceSet(B)`

Returns `true` iff B is a difference set over an integer residue class ring or a finite group (with an iterator). If `true`, the value of the parameter λ (i.e., the number of times each non–identity group/ring element appears as a "difference" of elements of B) is also returned.

`Development(B)`

Let B be a subset of a magma A which is a difference set relative to A, where A is either the ring Z/mZ, a finite abelian group or an arbitrary finite group (with an iterator). This function constructs the symmetric design having point set A and whose blocks consist of the sets obtained by translating B by each element of A in turn.

`Development(T)`

Let $T = \{B_1, \ldots, B_l\}$ be a difference family consisting of subsets of a magma A which is either the ring $\mathbf{Z}/m\mathbf{Z}$, a finite abelian group or an arbitrary finite group (with an iterator). This function constructs the incidence structure with point set A and whose i-th block is the set $\{B_1 \cup \ldots \cup B_l\}$ translated by the i-th element of A.

Example H147E5

The set $\{1, 3, 4, 5, 9\}$, where the elements are residues modulo 11, forms an $(11, 5, 2)$ difference set. We develop this set and construct a 2-$(11, 5, 2)$ design.

```
> Z11 := IntegerRing(11);
> B := { Z11 | 1, 3, 4, 5, 9};
> IsDifferenceSet(B);
true 2
> D := Development(B);
> D: Maximal;
2-(11, 5, 2) Design with 11 blocks
Points: {@ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 @}
Blocks:
    {1, 3, 4, 5, 9},
    {2, 4, 5, 6, 10},
    {0, 3, 5, 6, 7},
    {1, 4, 6, 7, 8},
    {2, 5, 7, 8, 9},
    {3, 6, 8, 9, 10},
    {0, 4, 7, 9, 10},
    {0, 1, 5, 8, 10},
    {0, 1, 2, 6, 9},
    {1, 2, 3, 7, 10},
```

```
    {0, 2, 3, 4, 8}
```

We now construct the twin primes (type "T") difference set modulo 323 (= 17 × 19), and its development.

```
> B := DifferenceSet(17, "T");
> D := Development(B);
> D;
2-(323, 161, 80) Design with 323 blocks
```

147.5 Elementary Invariants of an Incidence Structure

All operations defined for incidence structures apply also to near–linear spaces, linear spaces and designs.

`NumberOfPoints(D)`
`#P`

The cardinality v of the point set P of the incidence structure D.

`Points(D)`

An indexed set E whose elements are the points of the incidence structure D. Note that this creates a standard set and not the point-set of D, in contrast to the function `PointSet`.

`Support(D)`

An indexed set E which is the underlying point set of the incidence structure D (i.e., the elements of the set have their "real" types; they are no longer from the category IncPt).

`PointDegrees(D)`

A sequence whose i-th term gives the number of blocks containing the i-th point of the design D.

`NumberOfBlocks(D)`
`#B`

The number of blocks b of the incidence structure D with block-set B.

`Blocks(D)`

An indexed set containing the blocks of the incidence structure D. In contrast to the function `BlockSet`, this function returns the collection of blocks of D in the form of a standard set.

`BlockDegrees(D)`

`BlockSizes(D)`

A sequence whose i-th term gives the number of points in the i-th block of the incidence structure D.

`Covalence(D, S)`

Given a subset S of the point set of an incidence structure D, return the number of blocks of D that contain S.

`IncidenceMatrix(D)`

The incidence matrix of the incidence structure D.

`pRank(D, p)`

The p-rank of the incidence structure D.

147.6 Elementary Invariants of a Design

The following functions can be applied only to designs.

`Parameters(D)`

The parameters t–(v, b, r, k, λ) of the design D returned as a record.

`ReplicationNumber(D)`

The number of blocks r containing any point of the t–(v, k, λ) design D, where $t > 0$.

`BlockDegree(D)`

`BlockSize(D)`

The number of points in a block of the design D.

`Covalence(D, s)`

Given a t–(v, k, λ) design D and an integer s such that $0 \leq s \leq t$, return the value of λ_s; i.e., the number of blocks that contain an arbitrary s-subset of the points of D.

`Order(D)`

The order of the t–(v, k, λ) design D. This is defined only for designs with $t \geq 2$.

`IntersectionNumber(D, i, j)`

The block intersection number λ_i^j; i.e., the number of blocks of the design D containing an i-set and disjoint from a j-set. The arguments i and j must satisfy $i + j \leq t$.

`PascalTriangle(D)`

The "Pascal triangle" of the design D, returned as a sequence; the i-th element of the sequence is a sequence representing the i-th row of the triangle. That is, the i-th element of the sequence is

$$[\lambda_0^{i-1}, \lambda_1^{i-2}, \ldots, \lambda_{i-1}^0].$$

If D is a Steiner t–design, then the triangle returned has $k+1$ rows (where k is the size of a block of D); otherwise the triangle has $t+1$ rows.

Example H147E6

We illustrate some of the functions of the previous two sections with an example.

```
> F := Design< 2, 7 | {1,2,4}, {1,3,7}, {2,3,5}, {1,5,6}, {3,4,6}, {4,5,7},
>   {2,6,7} >;
> G := IncidenceStructure< 7 | Blocks(F), {1, 3, 7}, {1, 2, 4},
>   {3, 4, 5}, {2, 3, 6}, {2, 5, 7}, {1, 5, 6}, {4, 6, 7} >;
> F;
2-(7, 3, 1) Design with 7 blocks
> G;
Incidence Structure on 7 points with 14 blocks
> Points(G);
{@ 1, 2, 3, 4, 5, 6, 7 @}
> Blocks(F);
{@
    {1, 2, 4},
    {1, 3, 7},
    {2, 3, 5},
    {1, 5, 6},
    {3, 4, 6},
    {4, 5, 7},
    {2, 6, 7}
@}
> IncidenceMatrix(F);
[1 1 0 1 0 0 0]
[1 0 1 0 0 0 1]
[0 1 1 0 1 0 0]
[1 0 0 0 1 1 0]
[0 0 1 1 0 1 0]
[0 0 0 1 1 0 1]
[0 1 0 0 0 1 1]
> P := Points(F);
> P, Universe(P);
{@ 1, 2, 3, 4, 5, 6, 7 @}
Point-set of 2-(7, 3, 1) Design with 7 blocks
> S := Support(F);
> S, Universe(S);
```

```
{@ 1, 2, 3, 4, 5, 6, 7 @}
Integer Ring
> Covalence(G, {1, 2});
2
> Order(F);
2
> PascalTriangle(F);
         7
       4   3
     2   2   1
   0   2   0   1
```

147.7 Operations on Points and Blocks

In incidence structures, particularly simple ones, blocks are basically sets. For this reason, the elementary set operations such as `join, meet` and `subset` have been made to work on blocks. However, blocks are not true MAGMA enumerated sets, and so the functions `Set` and `Support` below have been provided to convert a block to an enumerated set of points for other uses.

`p in B`

Returns `true` if point p lies in block B, otherwise `false`.

`p notin B`

Returns `true` if point p does not lie in block B, otherwise `false`.

`S subset B`

Given a subset S of the point set of the incidence structure D and a block B of D, return `true` if the subset S of points lies in B, otherwise `false`.

`S notsubset B`

Given a subset S of the point set of the incidence structure D and a block B of D, return `true` if the subset S of points does not lie in B, otherwise `false`.

`PointDegree(D, p)`

The number of blocks of the incidence structure D that contain the point p.

`BlockDegree(D, B)`

`BlockSize(D, B)`

`#B`

The number of points contained in the block B of the incidence structure D.

`Set(B)`

The set of points contained in the block B.

`Support(B)`

The set of underlying points contained in the block B (i.e., the elements of the set have their "real" types; they are no longer from the category IncPt).

`IsBlock(D, S)`

Returns `true` iff the set (or block) S represents a block of the incidence structure D. If `true`, also returns one such block.

`Line(D, p, q)`

`Block(D, p, q)`

A block of the incidence structure D containing the points p and q (if one exists). In linear spaces, such a block exists and is unique (assuming p and q are different).

`ConnectionNumber(D, p, B)`

The connection number $c(p, B)$; i.e., the number of blocks joining p to B in the incidence structure D.

Example H147E7

The following examples uses some of the functions of the previous section.

```
> D, P, B := Design< 2, 7 | {3, 5, 6, 7}, {2, 4, 5, 6}, {1, 4, 6, 7},
>   {2, 3, 4, 7}, {1, 2, 5, 7}, {1, 2, 3, 6}, {1, 3, 4, 5} >;
> D: Maximal;
2-(7, 4, 2) Design with 7 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7 @}
Blocks:
    {3, 5, 6, 7},
    {2, 4, 5, 6},
    {1, 4, 6, 7},
    {2, 3, 4, 7},
    {1, 2, 5, 7},
    {1, 2, 3, 6},
    {1, 3, 4, 5}
> P.1 in B.1;
false
> P.1 in B.3;
true
> {P| 1, 2} subset B.5;
true
> Block(D, P.1, P.2);
{1, 2, 5, 7}
> b := B.4;
```

```
> b;
{2, 3, 4, 7}
> b meet {2, 8};
{ 2 }
> S := Set(b);
> S, Universe(S);
{ 2, 3, 4, 7 }
Point-set of 2-(7, 4, 2) Design with 7 blocks
> Supp := Support(b);
> Supp, Universe(Supp);
{ 2, 3, 4, 7 }
Integer Ring
```

147.8 Elementary Properties of Incidence Structures and Designs

Testing the t-balance: A general Note on Parameter `Al`

A parameter `Al` is provided to allow users to specify the algorithm to be used for balance testing. The default value is `"NoOrbits"`, which applies a "brute force" test. One alternative is `"Orbits"`, which uses the orbits of t-sets under the automorphism group of the incidence structure under consideration. This is much faster than `"NoOrbits"` for some cases, but slower for others.

The other alternative for checking if an object is t-balanced is `"FastBalanceTest"`: It is also a "brute force" test but whose implementation allows for a dramatic improvement in efficiency, especially for larger t ($t \geq 4$). This implementation has a drawback however as it may necessitate more memory space than is available, thus resulting in an out-of-memory error. It is for this reason that `"FastBalanceTest"` is not the default setting for `Al`, but as a general rule we strongly recommend its usage as we surmise that execution should successfully complete in most cases.

`IsSimple(D)`

Returns `true` if and only if the incidence structure D has no repeated blocks.

`IsTrivial(D)`

Returns `true` if and only if the incidence structure D is a trivial incidence structure.

`IsSelfDual(D)`

Returns `true` if and only if the incidence structure D is self-dual, that is, if D is isomorphic to its dual.

`IsUniform(D)`

Returns `true` if and only if the incidence structure D is uniform; that is, each block contains the same number of points. If `true`, also returns the blocksize.

`IsNearLinearSpace(D)`

Returns `true` if and only if the incidence structure D is a near–linear space.

`IsLinearSpace(D)`

Returns `true` if and only if the incidence structure D is a linear space.

`IsDesign(D, t:` *parameters*`)`

`Al` MonStgElt *Default :* "*NoOrbits*"

Returns `true` if and only if the incidence structure D is a t–design. If `true`, then the number of blocks of D containing a general t-set is also returned. The optional parameter `Al` can be used to specify the algorithm used for balance testing, see the introduction to Section 147.8 for a full description of its usage.

`IsBalanced(D, t:` *parameters*`)`

`Al` MonStgElt *Default :* "*NoOrbits*"

Returns `true` if and only if the incidence structure D is balanced (with respect to t). If `true`, then the number of blocks of D containing a general t-set is also returned. The optional parameter `Al` can be used to specify the algorithm used for balance testing, see the introduction to Section 147.8 for a full description of its usage.

`IsComplete(D)`

Return `true` if and only if D is the complete design.

`IsSymmetric(D)`

Return `true` if and only if the design D is symmetric.

`IsSteiner(D, t)`

`Al` MonStg *Default :* "*NoOrbits*"

Return `true` if and only if the design D is a Steiner t–design. The optional parameter `Al` can be used to specify the algorithm used for balance testing, see the introduction to Section 147.8 for a full description of its usage.

`IsPointRegular(D)`

Return `true` if and only if the (near–)linear space D is point regular. If `true`, the point regularity is also returned.

`IsLineRegular(D)`

Return `true` if and only if the (near–)linear space D is line regular. If `true`, the line regularity is also returned.

147.9 Resolutions, Parallelisms and Parallel Classes

Let D be an incidence structure with v points. A *resolution* of D is a partition of the blocks of D into classes C_i such that each class is a 1-design with v points and index λ. The positive integer λ is called the *index* of the resolution. A resolution with $\lambda = 1$ is called a *parallelism*. In this case the classes C_i are called *parallel classes*.

All the functions which deal with resolutions apply to general incidence structures. The functions `HasParallelism`, `AllParallelisms`, `HasParallelClass`, `IsParallelClass` and `AllParallelClasses` require that the incidence structure be uniform. The function `IsParallelism` however applies to a general incidence structure.

`HasResolution(D)`

Return `true` if and only if the incidence structure D has a resolution. If `true`, one resolution is returned as the second value of the function and the index of the resolution is returned as the third value.

`HasResolution(D, λ)`

Return `true` if and only if the incidence structure D has a resolution with index λ. If `true`, one such resolution is returned as the second value of the function.

`AllResolutions(D)`

Returns all resolutions of the incidence structure D.

`AllResolutions(D, λ)`

Returns all resolutions of the incidence structure D having index λ. When the problem is to find all parallelisms ($\lambda = 1$) in a uniform design D, it is best to use the function `AllParallelisms` described below.

`IsResolution(D, P)`

Returns `true` if and only if the set P of blocks (or sets) is a resolution of the incidence structure D. If `true`, also returns the index of the resolution as the second value of the function.

`HasParallelism(D: parameters)`

`Al` MonStg *Default :* "*Backtrack*"

Returns `true` if and only if the uniform incidence structure D has a parallelism. If `true`, a parallelism is returned as the second value of the function. The parameter `Al` allows the user to specify the algorithm used.

`Al := "Backtrack"`: A backtrack search is employed. This is in general very efficient when the design D is parallelizable, irrespective of the number of blocks. In particular, it compares very favourably with the algorithm `Clique`, described below. The later algorithm is recommended when the design appears to have no parallelism. Backtrack is the default.

`Al := "Clique"`: This algorithm proceeds in two stages. Assume that D has v points and b blocks of size k. Define the graph G_1 of D to be the graph whose vertices are the blocks of D and where two blocks are adjacent if and only if they

are disjoint. The first stage of the Clique algorithm involves constructing the graph G_1 and finding all cliques of size v/k in G_1. Any such clique is a parallel class of D. The second stage involved building a graph G_2 of D where the vertices are the parallel classes of D and two parallel classes are adjacent if and only if they are disjoint. The second stage of the Clique algorithm involves searching for cliques of size $b/(v/k)$ in G_2. If such a clique exists then it yields a parallelism of D. (This algorithm was communicated by Vladimir Tonchev).

The clique algorithm is recommended when the design is suspected of having no parallelism. As an example, in the case of a non–parallelizable design with 272 blocks, Clique took around 0.1 second to complete as compared to 2 seconds for the backtrack algorithm. For a non–parallelizable design having 6642 blocks, Clique took around four hours to complete as compared to Backtrack which did not complete in 15 hours. Apart from performing very poorly when D has a parallelism, Clique may require a considerable amount of memory. It is recommended that Backtrack be tried first, and if it does not complete in a reasonable amount of time, then Clique should be used.

`AllParallelisms(D)`

Returns all parallelisms of the uniform incidence structure D. This function is to be preferred to the function `AllResolutions(D, 1)` when D is uniform. This is the case since the implementation of `AllParallelClasses` implies finding cliques in graphs (see the algorithm Clique described in the function `HasParallelism`), while `AllResolutions` performs a full backtrack search.

`IsParallelism(D, P)`

Returns `true` if and only if the set P of blocks (or sets) is a parallelism of the incidence structure D.

`HasParallelClass(D)`

Returns `true` if and only if the uniform incidence structure D has a parallel class.

`IsParallelClass(D, B, C)`

Returns `true` if and only if the uniform incidence structure D has a parallel class containing the blocks B and C. If such a parallel class does exist, one is returned as the second value of the function.

`AllParallelClasses(D)`

Returns all parallel classes of the uniform incidence structure D.

Example H147E8

Some of the above functions are illustrated here.

```
> D := IncidenceStructure< 6 | {1,2,3}, {4,5,6}, {1,3,4}, {2,5,6}>;
> bool, R, lambda := HasResolution(D);
> bool;
true
> R;
{
  {
    {1, 2, 3},
    {4, 5, 6},
    {1, 3, 4},
    {2, 5, 6}
  }
}
> lambda;
2
> HasResolution(D,2);
true {
  {
    {1, 2, 3},
    {4, 5, 6},
    {1, 3, 4},
    {2, 5, 6}
  }
}
> AllResolutions(D);
[
  {
    {
      {1, 2, 3},
      {4, 5, 6},
      {1, 3, 4},
      {2, 5, 6}
    }
  },
  {
    {
      {1, 2, 3},
      {4, 5, 6}
    },
    {
      {1, 3, 4},
      {2, 5, 6}
    }
  }
]
```

```
> HasParallelism(D);
true {
  {
    {1, 2, 3},
    {4, 5, 6}
  },
  {
    {1, 3, 4},
    {2, 5, 6}
  }
}
> V := PointSet(D);
> S := { PowerSet(PowerSet(V)) |
>        { {1, 2, 3}, {4, 5, 6} }, { {1, 3, 4}, {2, 5, 6} } };
> IsParallelism(D, S);
true
> B := BlockSet(D);
> S := { { B.1, B.2 }, {B.3, B.4 }};
> IsParallelism(D, S);
true
> AllParallelClasses(D);
{
  {
    {1, 2, 3},
    {4, 5, 6}
  },
  {
    {1, 3, 4},
    {2, 5, 6}
  }
}
```

147.10 Conversion Functions

`IncidenceStructure(I)`

Given an incidence structure I (of any type), return the same structure as a "true" incidence structure (i.e., belonging to the category Inc).

`NearLinearSpace(I)`

Given an incidence structure I (of any type), return the same structure as a near–linear space (i.e., belonging to the category `IncNsp`), if possible.

`LinearSpace(I)`

Given an incidence structure I (of any type), return the same structure as a linear space (i.e., belonging to the category `IncLsp`), if possible.

`Design(I, t)`

Given an incidence structure I (of any type), return the same structure as a t–design (i.e., belonging to the category `Dsgn`), if possible.

Example H147E9

The following example illustrates some of the functions of the previous two sections.

```
> I := IncidenceStructure< 8 | {1, 2, 3, 4}, {1, 3, 5, 6}, {1, 4, 7, 8},
>   {2, 5, 6, 7}, {2, 5, 7, 8}, {3, 4, 6, 8} >;
> IsDesign(I, 1);
true 3
> IsDesign(I, 2);
false
> D := Design(I, 1);
> D;
1-(8, 4, 3) Design with 6 blocks
> IsSteiner(D, 1);
false
> IsNearLinearSpace(I);
false
```

147.11 Identity and Isomorphism

Incidence structures $D_1 = (P_1, B_1, I_1)$ and $D_2 = (P_2, B_2, I_2)$ are *identical* if $P_1 = P_2$, $B_1 = B_2$ and $I_1 = I_2$.

`D eq E`

Return `true` if the incidence structures D and E are identical, otherwise `false`.

`D ne E`

Return `true` if the incidence structures D and E are not identical, otherwise `false`.

`IsIsomorphic(D, E: parameters)`

`AutomorphismGroups` MonStgElt *Default : "None"*

Return `true` if the incidence structures D and E are isomorphic, otherwise `false`. If they are isomorphic, then an isomorphism is returned as the second value. The function first computes none, one, or both of the automorphism groups of the left and right incidence structures. In difficult examples, this may significantly speed up testing for isomorphism. The parameter `AutomorphismGroups`, with valid string values `"Both"`, `"Left"`, `"Right"`, `"None"`, may be used to specify which of the automorphism groups should be constructed if not already known. The default is `"None"`.

147.12 The Automorphism Group of an Incidence Structure

The automorphism group A of an incidence structure D is always presented as a permutation group G acting on the standard support. The reasons for this include the fact that the support chosen when computing the automorphism group is often quite complicated. Further, if the group is represented as acting on a set of objects relating to the incidence structure, printed permutations are often unreadable. The support used when constructing the automorphism group of an incidence structure D is either the point set of D (when D is simple), or the disjoint union of the point set with the block set (when D has repeated blocks).

The automorphism group G of D does not act directly on D. Instead, G-sets are used to transfer the action of G to various sets associated with D. The two most important G-sets, corresponding to action of G on the point set and on the block set, are returned by each of the functions provided for constructing automorphism group or some specified subgroup of it.

In some circumstances, rather than viewing automorphisms as group elements, it is desirable to view them is as mappings of D into itself. Associated with each incidence structure is a mapping structure, Aut(D), which denotes the set of automorphisms of D. Note that Aut(D) is the *parent* of the automorphisms of G so that the function `Aut(D)` simply creates a shell structure rather than the actual automorphism group of D. A transfer map is provided to convert a permutation of the automorphism group G into a mapping belonging to $Aut(D)$.

147.12.1 Construction of Automorphism Groups

`AutomorphismGroup(D)`

Construct the automorphism group G of the incidence structure D. The set on which G acts depends upon whether or not the design is simple. Suppose D has v points and b blocks. If the incidence structure D is simple, the automorphism group is constructed in its action on the points of D. It is returned acting on the standard support $\{1,\ldots,\text{v}\}$, where i corresponds to the i-th point of D. If D is not simple, it is constructed in its action of the disjoint union of the point set and block set of D.

Again, it is returned acting on the standard support {1,...,v+b}, where $1 \leq i \leq v$ corresponds to the i-th point of D and $v+1 \leq i \leq v+b$ corresponds to the $(i-v)$-th point of D. The function returns:

(i) The automorphism group G of D as described above;

(ii) A G-set Y corresponding to the action of G on the point set of D.

(iii) A G-set W corresponding to the action of G on the block set of D.

(iv) The *Aut* structure S for D; and

(v) A transfer map $t : G \to S$.

Given a permutation g of G, $t(g)$ is the *mapping* of D into itself that corresponds to the automorphism group element g. The G-sets Y and W should be used whenever it is necessary to have G act on the points or lines of D.

`AutomorphismSubgroup(D)`

A cyclic subgroup H of the automorphism group G of the incidence structure D. The purpose of this function is to terminate the search for automorphisms of D as soon as a non–trivial automorphism is found. The function returns:

(i) The automorphism group H of D as described above;

(ii) The *Aut* structure S for D; and

(iii) A transfer map $t : G \to S$.

Given a permutation g of H, $t(g)$ is the *mapping* of D into itself that corresponds to the automorphism group element g.

`AutomorphismGroupStabilizer(D, k)`

The subgroup H of the automorphism group G of the incidence structure D, which stabilizes the first k base points of G. This function is provided so as to return a subgroup of G that is sometimes easier to compute than all of G. The function returns:

(i) The automorphism group H of D as described above;

(ii) The *Aut* structure S for D; and

(iii) A transfer map $t : G \to S$.

Given a permutation g of H, $t(g)$ is the *mapping* of D into itself that corresponds to the automorphism group element g.

`PointGroup(D)`

The automorphism group of the incidence structure D given in its action on the point set of D, together with the points G-set.

`BlockGroup(D)`

The automorphism group of the incidence structure D given in its action on the block set of D.

`Aut(D)`

The power structure A of all automorphisms of the incidence structure D, together with the transfer map $t : Sym(n) \to A$, where the points of $Sym(n)$ are assumed to be in one–to–one correspondence with the natural support for the automorphism group of D.

Example H147E10

We construct a $3-(16,8,3)$ Hadamard design and investigate its automorphism group. The first step is to construct a Hadamard matrix of order 16.

```
> R := MatrixRing(Integers(), 4);
> H := R ! [1,1,1,-1, 1,1,-1,1, 1,-1,1,1, -1,1,1,1];
> L := TensorProduct(H, -H);
> D, P, B := HadamardRowDesign(L, 1);
> D;
3-(16, 8, 3) Design with 30 blocks
> G, pg, bg, A, t := AutomorphismGroup(D);
> G;
Permutation group G acting on a set of cardinality 16
Order = 322560 = 2^10 * 3^2 * 5 * 7
    (9, 15)(10, 16)(11, 13)(12, 14)
    (5, 11, 7, 9)(6, 12, 8, 10)(13, 15)(14, 16)
    (5, 9, 14)(6, 10, 13)(7, 11, 16)(8, 12, 15)
    (1, 2)(7, 8)(9, 13, 10, 14)(11, 16, 12, 15)
    (2, 5)(4, 7)(10, 13)(12, 15)
    (3, 9)(4, 10)(7, 13)(8, 14)
    (3, 8, 15, 12)(4, 7, 16, 11)(5, 9)(6, 10)
    (5, 6)(7, 8)(13, 14)(15, 16)
    (3, 12, 4, 11)(7, 16, 8, 15)(9, 10)(13, 14)
    (3, 7)(4, 8)(11, 15)(12, 16)
    (3, 8)(4, 7)(9, 10)(11, 15)(12, 16)(13, 14)
    (9, 10)(11, 12)(13, 14)(15, 16)
    (9, 13)(10, 14)(11, 15)(12, 16)
> CompositionFactors(G);
    G
    |  Alternating(8)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

147.12.2 Action of Automorphisms

As noted at the beginning of the section, the automorphism group G of an incidence structure D is given in its action on the standard support and it does not act directly on D. The action of G on D is obtained using the G-set mechanism. The two basic G-sets associated with D correspond to the action of G on the set of points P and the set of blocks B of D. These two G-sets are given as return values of the function `AutomorphismGroup` or may be constructed directly. Additional G-sets associated with D may be built using the G-set constructors. Given a G-set Y for G, the action of G on Y may be studied using the permutation group functions that allow a G-set as an argument. In this section, only a few of the available functions are described: see the section on G-sets for a complete list.

`Image(g, Y, y)`

Let G be a subgroup of the automorphism group for the incidence structure D and let Y be a G-set for G. Given an element y belonging either to Y or to a G-set derived from Y, find the image of y under G.

`Orbit(G, Y, y)`

Let G be a subgroup of the automorphism group for the incidence structure D and let Y be a G-set for G. Given an element y belonging either to Y or to a G-set derived from Y, construct the orbit of y under G.

`Orbits(G, Y)`

Let G be a subgroup of the automorphism group for the incidence structure D and let Y be a G-set for G. This function constructs the orbits of the action of G on Y.

`Stabilizer(G, Y, y)`

Let G be a subgroup of the automorphism group for the incidence structure D and let Y be a G-set for G. Given an element y belonging either to Y or to a G-set derived from Y, construct the stabilizer of y in G.

`Action(G, Y)`

Given a subgroup G of the automorphism group of the incidence structure D, and a G-set Y for G, construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. The function returns:

(a) The natural homomorphism $\phi : G \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of G).

`ActionImage(G, Y)`

Given a subgroup G of the automorphism group of the incidence structure D, and a G-set Y for G, construct the permutation group L giving the action of G on the set Y.

`ActionKernel(G, Y)`

Given a subgroup G of the automorphism group of the incidence structure D, and a G-set Y for G, construct the kernel of the action of G on the set Y.

`IsPointTransitive(D)`

Returns `true` iff the automorphism group of the incidence structure D acts transitively on the point set of D.

`IsBlockTransitive(D)`

Returns `true` iff the automorphism group of the incidence structure D acts transitively on the block set of D.

Example H147E11

In the following example, the automorphism group of the Witt design on 12 points is constructed.

```
> D, P, B := WittDesign(12);
> A, PY, BY := AutomorphismGroup(D);
> A;
Permutation group Aut acting on a set of cardinality 12
Order = 95040 = 2^6 * 3^3 * 5 * 11
    (1, 2)(5, 8)(6, 11)(10, 12)
    (2, 3)(5, 12)(6, 8)(10, 11)
    (3, 4)(5, 10)(6, 11)(8, 12)
    (4, 7)(5, 8)(6, 12)(10, 11)
    (5, 11, 12, 6)(7, 8, 9, 10)
    (5, 9, 12, 7)(6, 10, 11, 8)
    (5, 12)(6, 11)(7, 9)(8, 10)
> a := A!(1, 2, 3, 4)(5, 6, 12, 11);
> 4^a;
1
> {1, 2, 3}^a;
{ 2, 3, 4 }
> Image(a, BY, Block(D, 3));
{1, 2, 3, 8, 11, 12}
> S2 := SylowSubgroup(A, 2);
> S2;
Permutation group S2 acting on a set of cardinality 12
Order = 64 = 2^6
   (1, 11, 9, 4)(6, 10, 12, 7)
   (1, 9)(2, 12)(3, 7)(4, 11)(5, 10)(6, 8)
   (1, 4)(3, 5)(7, 10)(9, 11)
> Stabilizer(S2, BY, Block(D, 4));
Permutation group acting on a set of cardinality 12
Order = 2
    (1, 9)(3, 5)(6, 7)(10, 12)
> Stabilizer(S2, 6);
Permutation group acting on a set of cardinality 12
```

```
Order = 8 = 2^3
    (1, 4)(3, 5)(7, 10)(9, 11)
    (1, 4, 9, 11)(2, 5, 8, 3)
> 3^S2;
GSet{ 2, 3, 5, 6, 7, 8, 10, 12 }
> IsPointTransitive(D);
true
> IsBlockTransitive(D);
true
```

We calculate the action of the automorphism group of the design on its blocks.

```
> H := ActionImage(A, BY);
> #H;
95040
> H;
Permutation group H acting on a set of cardinality 132
Order = 95040 = 2^6 * 3^3 * 5 * 11
```

147.13 Incidence Structures, Graphs and Codes

`IncidenceStructure(G)`

Construct the incidence structure D corresponding to the graph G, where the blocks of D correspond to the edges of G.

`PointGraph(D)`

The point graph G of the incidence structure D. The graph G has the same point set as D and two vertices u and v of G are adjacent whenever there is a block of D containing both u and v.

`BlockGraph(D)`

The block graph of the incidence structure D, i.e., the point graph of the dual of D.

`IncidenceGraph(D)`

The incidence graph of the incidence structure D. This bipartite graph has as vertex set the union of the point set P and block set B of D. A vertex $p \in P$ is adjacent to a vertex $b \in B$ whenever $p \in b$.

`LinearCode(D, K)`

Given an incidence structure D with v points and a finite field K, this function returns the linear code C of length v generated by the characteristic functions of the blocks of D considered as vectors of the K-space $K^{(v)}$.

Example H147E12

The linear code of the Witt 5-(24, 8, 1) design over GF(2) is the extended Golay code over GF(2).

```
> D := WittDesign(24);
> C := LinearCode(D, GF(2));
> C eq GolayCode(GF(2), true);
true
```

147.14 Automorphisms of Matrices

Matrices may be regarded as defining a design with the entry of a matrix in a particular row and column defining the incidence of the row and column. The automorphism group of a matrix is the set of permutations of the rows and columns of the matrix that leave the matrix unchanged.

`M ^ x`

The action of a permutation on a matrix by permuting rows and columns. If M has r rows and c columns then x must have degree $r + c$ and fix the set $R = \{1..r\}$. The action of x on R gives the permutation of the rows of the matrix. The remainder of x gives the action on columns.

`AutomorphismGroup(M)`

Computes the group of all permutations x such that $M^x = M$. Currently this is done by constructing a graph from M, and applying the `AutomorphismGroup` function to the graph.

`IsIsomorphic(M, N)`

Finds a permutation x such that $M^x = N$, if such exists. If a permutation is found return values are true and the permutation, otherwise returns false. Currently this is done by constructing graphs from M and N, and applying the `IsIsomorphic` function to the graphs.

Example H147E13

We construct a matrix from a Fano polytope and compute its automorphism group.

```
> M := VertexFacetHeightMatrix(PolytopeSmoothFanoDim3(10)); M;
[1 0 0 0 2 1 2 2]
[0 0 0 1 1 2 2 2]
[0 0 1 2 0 2 2 1]
[2 0 1 0 2 0 2 1]
[0 2 0 1 1 2 0 2]
[1 2 0 0 2 1 0 2]
[0 2 1 2 0 2 0 1]
[2 2 1 0 2 0 0 1]
```

```
[2 2 2 1 1 0 0 0]
[1 2 2 2 0 1 0 0]
[1 0 2 2 0 1 2 0]
[2 0 2 1 1 0 2 0]
> A := AutomorphismGroup(M); A;
Permutation group A acting on a set of cardinality 20
Order = 24 = 2^3 * 3
  (2, 4)(3, 12)(5, 8)(7, 9)(13, 18)(15, 16)(17, 20)
  (1, 5)(2, 6)(3, 8)(4, 7)(9, 11)(10, 12)(13, 16)(14, 19)(17, 18)
  (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)(11, 12)(13, 16)(17, 18)
> Nrows(M), Ncols(M);
12, 8
> Orbits(A);
[
  GSet{@ 14, 19 @},
  GSet{@ 13, 18, 16, 17, 15, 20 @},
  GSet{@ 1, 5, 2, 8, 6, 4, 3, 7, 12, 9, 10, 11 @}
]
```

The automorphism group is transitive on the rows of the matrix. Now dualize the matrix and test for isomorphism.

```
> D := Matrix(12, 8, [2-x:x in Eltseq(M)]);
> f, x := IsIsomorphic(M, D); f, x;
true (2, 4)(3, 12)(5, 8)(7, 9)(14, 19)(15, 17)(16, 20)
> M^x eq D;
true
```

147.15 Bibliography

[**Hal86**] Marshall Hall. *Combinatorial Theory.* New York: Wiley, 2nd edition, 1986.

148 HADAMARD MATRICES

Chapter 148
HADAMARD MATRICES

148.1 Introduction

A *Hadamard matrix* is an $n \times n$ matrix, all of whose entries are ± 1, such that every pair of rows and every pair of columns differ in exactly $\frac{n}{2}$ places. Two such matrices are considered equivalent if one can be transformed into the other by performing row swaps, column swaps, row negations or column negations. The problem of deciding whether two Hadamard matrices are equivalent is hard.

MAGMA contains several specialised routines for working with Hadamard matrices; of special note is the introduction of a canonical form for such matrices (based on Brendan McKay's `nauty` program). This leads to a much faster equivalence algorithm than previously available.

148.2 Equivalence Testing

`IsHadamard(H)`

Returns `true` if and only if the matrix H is a Hadamard matrix.

`HadamardNormalize(H)`

Given a Hadamard matrix H, returns a normalized matrix equivalent to H. This matrix is created by negating rows and columns to ensure that the first row and column consist entirely of ones.

`HadamardCanonicalForm(H)`

Given a Hadamard matrix H, returns a Hadamard-equivalent matrix H' and transformation matrices X and Y such that $H' = XHY$. The matrix H' is canonical in the sense that all matrices that are Hadamard-equivalent to H (and no others) will produce the same matrix H'.

`HadamardInvariant(H)`

Returns a sequence S of integers giving the 4-profile of the Hadamard matrix H. All Hadamard-equivalent matrices have the same 4-profile, but so may some inequivalent ones. Thus this test may determine inequivalence of two matrices more cheaply than performing a full equivalence test, but cannot establish equivalence.

`IsHadamardEquivalent(H, J : parameters)`

`Al` MONSTGELT *Default : "nauty"*

Returns `true` if and only if the Hadamard matrices H and J are Hadamard equivalent. The parameter `Al` may be set to either `"Leon"` or `"nauty"` (the default). If the `"nauty"` option is chosen and the matrices are equivalent then transformation matrices X and Y are also returned, such that $J = XHY$.

`HadamardMatrixToInteger(H)`

Returns an integer that encodes the entries in the given Hadamard matrix in more compact form. The intended use is to save time when testing for equality against the same set of matrices many times.

`HadamardMatrixFromInteger(x, n)`

Returns a Hadamard matrix of degree n whose encoded form is the integer x. This function is the inverse of `HadamardMatrixToInteger()`.

Example H148E1

We demonstrate the use of some of these functions on certain degree 16 Hadamard matrices. For convenience, we create them from the more compact integer form.

```
> S := [
>   47758915483058652629300889924143904114798786457834842901517979108472281628672,
>   52517743516350345514775635367035460577743730272737765852723792818755052170805,
>   69809270372633075610047556428719374057869882804054059134346034969950931648512,
>   72098012275487127961355071358205554032515600906148326841367820166804453457 92
> ];
> T := [ HadamardMatrixFromInteger(x, 16) : x in S ];
> &and [ IsHadamard(m) : m in T ];
true
```

Now we can test them for equivalence; we start by checking the 4-profiles.

```
> [ HadamardInvariant(m) : m in T ];
[
    [ 1680, 0, 140 ],
    [ 1680, 0, 140 ],
    [ 1344, 448, 28 ],
    [ 1344, 448, 28 ]
]
```

We see that the only possible equivalencies are between the first two and the last two, since equivalent matrices must have the same 4-profile.

```
> equiv,X,Y := IsHadamardEquivalent(T[1], T[2]);
> equiv;
true
> T[2] eq X*T[1]*Y;
true
> equiv,X,Y := IsHadamardEquivalent(T[3], T[4]);
> equiv;
false
```

So we have three inequivalent matrices. An alternative way to determine the inequivalent matrices would have been to use the canonical forms.

```
> #{ HadamardCanonicalForm(m) : m in T };
```

3

148.3 Associated 3–Designs

`HadamardRowDesign(H, i)`

Given an $n \times n$ Hadamard matrix H (with $n \geq 4$) and an integer i with $1 \leq i \leq n$, returns the Hadamard 3–design corresponding to the ith row of H.

`HadamardColumnDesign(H, i)`

Given an $n \times n$ Hadamard matrix H (with $n \geq 4$) and an integer i with $1 \leq i \leq n$, returns the Hadamard 3–design corresponding to the ith column of H.

Example H148E2

There is only one Hadamard equivalence class of 8×8 Hadamard matrices. We construct one matrix, and two of its designs.

```
> R := MatrixRing(Integers(), 8);
> H := R![1,  1,  1,  1,  1,  1,  1,  1,
>         1,  1,  1,  1, -1, -1, -1, -1,
>         1,  1, -1, -1,  1,  1, -1, -1,
>         1,  1, -1, -1, -1, -1,  1,  1,
>         1, -1,  1, -1,  1, -1, -1,  1,
>         1, -1,  1, -1, -1,  1,  1, -1,
>         1, -1, -1,  1, -1,  1, -1,  1,
>         1, -1, -1,  1,  1, -1,  1, -1];
> IsHadamard(H);
true
> DR := HadamardRowDesign(H, 3);
> DR: Maximal;
3-(8, 4, 1) Design with 14 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8 @}
Blocks:
    {1, 2, 5, 6},
    {1, 2, 7, 8},
    {1, 2, 3, 4},
    {1, 4, 5, 7},
    {1, 4, 6, 8},
    {1, 3, 6, 7},
    {1, 3, 5, 8},
    {3, 4, 7, 8},
    {3, 4, 5, 6},
    {5, 6, 7, 8},
    {2, 3, 6, 8},
    {2, 3, 5, 7},
```

```
    {2, 4, 5, 8},
    {2, 4, 6, 7}
> HadamardColumnDesign(H, 8);
3-(8, 4, 1) Design with 14 blocks
```

148.4 Automorphism Group

`HadamardAutomorphismGroup(H : parameters)`

`Al`	MonStgElt	*Default : "nauty"*

Given a Hadamard matrix H of degree n, returns the automorphism group of H as a permutation group of degree $4n$. The parameter `Al` may be set to either `"Leon"` or `"nauty"` (the default).

148.5 Databases

Magma contains two databases of Hadamard matrices; the first database includes all inequivalent matrices of degree at most 28, and examples of matrices of all degrees up to 256. The representatives used are the canonical forms (as output by `HadamardCanonicalForm`), and matrices of a given degree are ordered lexicographically (with 1 considered to be less than -1 for the purposes of this ordering).

A database of skew-symmetric Hadamard matrices also exists; in this case the matrices are not stored in canonical form, since the canonical forms are not skew-symmetric. With the exception of the creation routines, the intrinsics below apply to both databases.

`HadamardDatabase()`

Returns the database of Hadamard matrices.

`SkewHadamardDatabase()`

Returns the database of skew-symmetric Hadamard matrices.

`Matrix(D, n, k)`

Returns the kth matrix of degree n in the database D.

`Matrices(D, n)`

Returns the sequence of all matrices of degree n that are stored in the database D.

`DegreeRange(D)`

Returns the smallest and largest degrees of matrices in the database D.

`Degrees(D)`

Returns the sequence of degrees for which there is at least one matrix of that degree in the database D.

`NumberOfMatrices(D, n)`

Return the number of matrices of degree n in the database D.

Example H148E3

```
> D := HadamardDatabase();
> Matrix(D, 16, 3);
[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
[ 1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1]
[ 1  1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
[ 1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1]
[ 1  1 -1 -1 -1  1  1 -1  1 -1 -1  1 -1 -1  1  1]
[ 1  1 -1 -1  1 -1 -1  1 -1  1  1 -1 -1 -1  1  1]
[ 1  1 -1 -1 -1 -1  1  1 -1 -1  1  1  1  1 -1 -1]
[ 1 -1 -1  1 -1 -1  1  1  1  1 -1 -1  1 -1 -1  1]
[ 1 -1 -1  1  1  1 -1 -1 -1 -1  1  1  1 -1 -1  1]
[ 1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1]
[ 1 -1  1 -1  1 -1 -1  1  1 -1 -1  1 -1  1 -1  1]
[ 1 -1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1 -1  1]
[ 1 -1  1 -1 -1  1 -1  1 -1  1 -1  1  1 -1  1 -1]
[ 1 -1 -1  1  1 -1  1 -1 -1  1 -1  1 -1  1  1 -1]
[ 1 -1 -1  1 -1  1 -1  1  1 -1  1 -1 -1  1  1 -1]
[ 1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1  1]
> NumberOfMatrices(D, 20);
3
> Degrees(D);
[ 1, 2, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72,
76, 80, 84, 88, 92, 96, 100, 104, 108, 112, 116, 120, 124, 128, 132, 136, 140,
144, 148, 152, 156, 160, 164, 168, 172, 176, 180, 184, 188, 192, 196, 200, 204,
208, 212, 216, 220, 224, 228, 232, 236, 240, 244, 248, 252, 256 ]
> [ NumberOfMatrices(D, n) : n in Degrees(D) ];
[ 1, 1, 1, 1, 1, 5, 3, 60, 487, 23, 218, 20, 500, 55, 562, 2, 3, 1, 2, 1, 2, 1,
3, 2, 1, 1, 1, 1, 2, 2, 1, 3, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
```

148.5.1 Updating the Databases

The databases are noticeably incomplete (for example, it is known that more than 60 000 Hadamard matrices exist for degree 32 whereas only 23 are in the database; also, only matrices of degrees 36, 44, or 52 are present in the skew database). The MAGMA group welcomes contributions of matrices not equivalent to ones already present and will add them to future versions of the databases.

The functions in this section can be used to create new versions of the databases. This allows users to include matrices not already present without having to wait for new official versions of the databases. These functions use a record whose format is not described, as this record should only be manipulated by these functions.

`HadamardDatabaseInformation(D : parameters)`

`Canonical` BoolElt *Default :* `true`

This takes an existing Hadamard database and extracts the information in it to an internal form which will be used by the other intrinsics. This internal form is returned. The parameter `Canonical` indicates whether the entries in the database are known to be the canonical forms or not. It defaults to `true`, which is correct for the standard database; it should be set to `false` if working with the skew database or a user-created database where the entries are not canonical.

Note that the value of the `Canonical` parameter also controls whether the database created from this data should store the canonical forms or the original ones. If you want to extract the matrices from a canonical database but store them in a non-canonical one, you should create the non-canonical database first — using either this intrinsic or `HadamardDatabaseInformationEmpty` — with `Canonical` set to `false` and then add the matrices from the database with `UpdateHadamardDatabase` and `Canonical` set to `true`.

`HadamardDatabaseInformationEmpty(: parameters)`

`Canonical` BoolElt *Default :* `true`

Returns the internal data that would correspond to an empty database. The parameter `Canonical` is used to indicate whether the entries in the database should be written out in canonical form or not.

This allows the creation of a new database, or a slice of an existing one, without including the entirety of a previous one.

`UpdateHadamardDatabase(~R, S : parameters)`

`Canonical` BoolElt *Default :* `false`

This augments the record R containing the database information with the matrices in the sequence S. The matrices are only added if they are inequivalent with matrices already there. Note that this requires finding the canonical forms, which can be expensive. If the matrices in S are known to be in canonical form already then the parameter `Canonical` should be set to `true`.

If the matrices of matching degree already in R are not known to be canonical then their canonical forms must also be computed. Again, this can take significant time. This is particularly troublesome for the skew database, where the canonical forms are not stored. For one way to deal with this, see the description of `WriteRawHadamardData` below.

`WriteHadamardDatabase(S, ~R)`

This creates the database files *name*`.dat` and *name*`.ind` from the database data R, where *name* is taken from the string S. Since canonical forms may need to be computed during this process, the data is passed with a variable reference so that this computation is not lost. (For instance, if one writes out the database, then adds some more matrices and writes it out again.)

`WriteRawHadamardData(S, R)`

This saves the data in R to the file whose name is given by the string S. When loaded, this file will define a single variable `data` which will behave identically to R. This is desirable for the non-canonical databases since the canonical forms will not have to be recomputed.

This routine destroys the original contents, if any, of the file.

`SetVerbose("HadamardDB", v)`

This procedure changes the verbose printing level for the Hadamard database update routines. The verbose value v should be an integer in the range 0 to 3. This can provide reassuring indications of progress when a long update process is underway.

Example H148E4

Here are some examples of how one might use these routines. We assume that the file `matrixfile` contains code that will define a sequence S of Hadamard matrices when loaded.

```
> D := HadamardDatabase();
> #D;
2004
> data := HadamardDatabaseInformation(D);
> SetVerboseLevel("HadamardDB", 1);
> load "matrixfile";
> UpdateHadamardDatabase(~data, S);
3 new matrices added
> WriteHadamardDatabase("~/data/hadamard", ~data);
Sorting matrices by canonical forms
Sorting by degrees
Computing size information
Writing data file
Data file written (2007 matrices total)

Writing index file
Index file written
```

This has created the files `hadamard.dat` and `hadamard.ind` (in the directory `~/data`) containing the updated data for the Hadamard database. The file stems should be `hadamard` for the standard database, or `hadamard_skew` for the database of skew-symmetric matrices; in this case we wanted the former. In order to use these files instead of the standard database, the library root must be changed. Database files are looked for in the `data` subdirectory of the library root, so in this example the library root should be changed to `~` in order to get the newly created database.

```
> SetLibraryRoot("~");
> D := HadamardDatabase();
> #D;
```

```
2007
```

Of course, the library root is used by other databases, so it would be a good idea to set it back to its original value after creating the modified database. A better approach would be to encapsulate this process in a function (which could be put into the user startup file).

```
> function MyHadamardDatabase()
>     oldlibroot := GetLibraryRoot();
>     SetLibraryRoot("~");
>     D := HadamardDatabase();
>     SetLibraryRoot(oldlibroot);
>     return D;
> end function;
>
> D := MyHadamardDatabase();
> #D;
2007
```

If working with the database of skew-symmetric Hadamard matrices, it will probably be desirable to use the raw data instead, since the computation of the canonical forms takes a lot of time (about 23 minutes on a 750MHz machine). This does have to be done, but by using the raw forms it will only have to be done once, rather than each time.

```
> D := SkewHadamardDatabase();
> #D;
638
> data := HadamardDatabaseInformationEmpty(: Canonical := false);
> for n in Degrees(D) do
>     UpdateHadamardDatabase(~data, Matrices(D, n));
> end for;
> WriteRawHadamardData("skewraw.m", data);
```

Now that the raw data has been created, it can be used in other sessions to update the database (or the same session, of course).

```
> load "skewraw.m";
> load "matrixfile";
> UpdateHadamardDatabase(~data, S);
> WriteHadamardDatabase("~/data/hadamard_skew", ~data);
> WriteRawHadamardData("skewraw.m", data);
```

Note that the updated raw form is saved as well as the database. Now the new data can be accessed in the same way as the previous database was.

```
> SetLibraryRoot("~");
> D := SkewHadamardDatabase();
> #D;
641
```

As before, for maximum convenience access to the new database should also be put into a function in the startup file.

149 GRAPHS

Chapter 149
GRAPHS

149.1 Introduction

A *simple graph* is a graph in which each edge joins two distinct vertices and two distinct vertices are joined by at most one edge. A *simple digraph*, whose edges are directed, is defined in an analogous manner. Thus, loops and multiple edges are not permitted in simple graphs and simple digraphs. A graph (digraph) with loops and/or multiple edges joining a fixed pair of vertices is called a *multigraph* (resp. *multidigraph*). A multidigraph whose edges are assigned a capacity is more commonly called a *network*.

In this chapter the term "graph" is used when referring to a simple undirected graph, while the term "digraph" is used when referring to a simple directed graph. Sometimes the term "graph" is used as the generic term for the incidence structure on vertices and edges. Such uses should be clear from the context in which they occur.

There are five MAGMA graph objects: the undirected simple graph of type `GrphUnd`, the directed simple graph of type `GrphDir`, the undirected multigraph of type `GrphMultUnd`, the directed multigraph of type `GrphMultDir`, and the network of type `GrphNet`. The simple graphs are all of type `Grph`, while the multigraphs (including the network) are of type `GrphMult`. There is a caveat here with respect to simple digraphs and loops: for historical reasons, MAGMA allows loops in digraphs.

Simple graphs and digraphs are covered in this chapter, while multigraphs, multidigraphs and networks are covered in Chapter 150. All types of graphs may have vertex labellings and/or edge labellings. In addition, assigning weights and/or capacities to graph edges is also possible, thus allowing to run shortest-paths and flow-based algorithms over the graphs.

Importantly, from the present version (V2.11) onwards, all the standard graph construction functions (see Subsections 149.6.1 and 149.6.2) respect the graph's support set and vertex and edge decorations. That is, the resulting graph will have a support set and vertex and edge decorations compatible with the original graph and the operation performed on that graph.

MAGMA employs two distinct data structures for representing graphs. A graph may be represented in the form of an adjacency matrix (the dense representation) or in the form of an adjacency list (the sparse representation). The latter is better suited for sparse graphs and for algorithms which have been designed with the adjacency list representation in mind. An advantage of the sparse representation is the possibility of creating much larger (sparse) graphs than would be possible using the dense representation, since memory requirements for the adjacency list representation is linear in the number of edges, while memory requirements for the adjacency matrix representation is quadratic in the number of vertices (order) of the graph. This is covered in detail in Section 149.2.1.

Further, multigraphs and multidigraphs (and networks) may only be represented by an adjacency list since they may contain multiple edges. Users have control over the choice of representation when creating simple graphs or digraphs. If no indication is given, simple graphs and digraphs are *always* created with the dense representation. At the time of the present release (V2.11), a significant part of MAGMA functions are able to work directly with either of the representations. However, wherever necessary, a graph will be converted internally to whichever representation is required by a given function. We emphasize that this process is completely transparent to users and *requires no intervention on their part.*

For a concise and comprehensive overview on graph theory and algorithms, the reader is referred to [Eve79]; in [TCR90] they'll find a clear exposure of some graph algorithms, and [RAO93] is the recommended reference for flow problems in graphs.

149.2 Construction of Graphs and Digraphs

Any enumerated or indexed set S may be given as the vertex-set of a graph. The graph constructor will take a copy V of S, convert V into an indexed set if necessary, and flag its type as `GrphVertSet`. A graph may be specifically created as a sparse graph. If no indication is given then the graph is *always* created with the dense representation, that is, as an adjacency matrix.

149.2.1 Bounds on the Graph Order

Memory allocation for any elementary object (that is, accessed via a single pointer) cannot exceed 2^{32} bytes. For this reason there is a bound on the graph order. This bound is dependent upon the (internal) graph representation chosen at creation: the dense representation (as a packed adjacency matrix), or the sparse representation (as an adjacency list). The former is quadratic in the number of vertices of the graph, the latter is linear in the number of the edges. Thus a large graph with a low edge density will require less memory space than the same graph with a dense representation.

The bounds on the graph order n are as follows:

- for the dense representation, $n \leq 65535$,
- for the sparse representation, $n \leq 134217722$.

These bounds are maximal, that is, they assume — for the sparse representation — that the number of edges is zero. To help users determine the likely size of the graph they want to construct, we provide the following function.

`GraphSizeInBytes(n, m : parameters)`

`IsDigraph`	BOOL	*Default :* `false`
`SparseRep`	BOOL	*Default :* `false`

Computes the memory requirement in bytes of a graph of order n and size m. By default it is assumed that the graph is undirected and has a dense representation.

Example H149E1

One may verify that the dense representation cannot be used for a value of n (graph order) larger than 65535.

```
> n := 65536;
> m := 0;
> assert GraphSizeInBytes(n, m) gt 2^32;
```

It is possible to construct such a graph with a sparse representation:

```
> GraphSizeInBytes(n, m : SparseRep := true);
2097580
```

However, assuming that the graph of order $n = 65535$ has a size of $m = 33538101$ we see that such a graph cannot be constructed as its total memory requirement is again larger than 2^{32}:

```
> m := 33538101;
> assert GraphSizeInBytes(n, m : SparseRep := true) gt 2^32;
```

149.2.2 Construction of a General Graph

`Graph< n | edges :` *parameters* `>`

`Graph< S | edges :` *parameters* `>`

`SparseRep` BOOL *Default :* `false`

Construct the graph G with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The graph G, the vertex-set V of G; and the edge-set E of G. If SparseRep is `true` then the resulting graph will have a sparse representation.

The elements of E are specified by the list `edges`, where the items of `edges` may be objects of the following types:

(a) A pair $\{v_i, v_j\}$ of vertices in V. The edge from v_i to v_j will be added to the edge-set for G.

(b) A tuple of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the edges $\{v_i, u_1\}, \ldots, \{v_i, u_r\}$ will be added to G.

(c) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of neighbours for the vertex v_i. The edges $\{v_i, u_i\}$, $1 \leq i \leq n$, $u_i \in N_i$, are added to G.

In addition to these three basic ways of specifying the *edges* list, the items in *edges* may also be:

(d) An edge e of a graph or digraph or multigraph or multidigraph or network of order n. If e is an edge from u to v in a directed graph H, then the undirected edge $\{u, v\}$ will be added to G.

(e) An edge-set E of a graph or digraph or multigraph or multidigraph or network of order n. Every edge e in E will be added to G.

(f) A graph or a digraph or a multigraph or a multidigraph or a network H of order n. Every edge e in H's edge-set is added to G.

(g) A $n \times n$ symmetric (0, 1)-matrix A. The matrix A will be interpreted as the adjacency matrix for a graph H on n vertices and the edges of H will be added will be added to G.

(h) A set of

(i) Pairs of the form $\{v_i, v_j\}$ of vertices in V.

(ii) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i.

(iii) Edges of a graph or digraph or multigraph or multidigraph or network of order n.

(iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order n.

(j) A sequence of

(i) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i.

`IncidenceGraph(A)`

Let A be a $n \times m$ $(0, 1)$ matrix having exactly two $1s$ in each column. Then a graph G of order n and size m having A as its incidence matrix will be constructed. The rows of A will correspond to the vertices of G while the columns will correspond to the edges.

Example H149E2

The Petersen graph will be constructed in various different ways to illustrate the use of the constructor. Firstly, it will be constructed by listing the edges:

```
> P := Graph< 10 | { 1, 2 }, { 1, 5 }, { 1, 6 }, { 2, 3 }, { 2, 7 },
>            { 3, 4 }, { 3, 8 }, { 4, 5 }, { 4, 9 }, { 5, 10 },
>            { 6, 8 }, { 6, 9 }, { 7, 9 }, { 7, 10 }, { 8, 10 } >;
```

We next construct the Petersen graph by listing the neighbours of each vertex:

```
> P := Graph< 10 | [ { 2, 5, 6 }, { 1, 3, 7 }, { 2, 4, 8 }, { 3, 5, 9 },
>              { 1, 4, 10 }, { 1, 8, 9 }, { 2, 9, 10 }, { 3, 6, 10 },
>              { 4, 6, 7 }, { 5, 7, 8 } ] >;
```

We repeat the above construction but now the graph is created as a sparse graph:

```
> PS := Graph< 10 | [ { 2, 5, 6 }, { 1, 3, 7 }, { 2, 4, 8 }, { 3, 5, 9 },
```

```
>              { 1, 4, 10 }, { 1, 8, 9 }, { 2, 9, 10 }, { 3, 6, 10 },
>              { 4, 6, 7 }, { 5, 7, 8 } ] : SparseRep := true >;
> assert PS eq P;
```

Here is yet another way of constructing the sparse graph, from the dense graph:

```
> PS := Graph< 10 | P : SparseRep := true >;
> assert PS eq P;
```

We finally construct the graph in terms of an adjacency matrix:

```
> M := MatrixRing( Integers(), 10 );
> P := Graph< 10 | M![ 0,1,0,0,1,1,0,0,0,0,
>                      1,0,1,0,0,0,1,0,0,0,
>                      0,1,0,1,0,0,0,1,0,0,
>                      0,0,1,0,1,0,0,0,1,0,
>                      1,0,0,1,0,0,0,0,0,1,
>                      1,0,0,0,0,0,0,1,1,0,
>                      0,1,0,0,0,0,0,0,1,1,
>                      0,0,1,0,0,1,0,0,0,1,
>                      0,0,0,1,0,1,1,0,0,0,
>                      0,0,0,0,1,0,1,1,0,0] >;
> P;
        Graph
    Vertex  Neighbours

       1     2 5 6 ;
       2     1 3 7 ;
       3     2 4 8 ;
       4     3 5 9 ;
       5     1 4 10 ;
       6     1 8 9 ;
       7     2 9 10 ;
       8     3 6 10 ;
       9     4 6 7 ;
      10     5 7 8 ;
```

Example H149E3

A more sophisticated example is the construction of Tutte's 8-cage using the technique described in P. Lorimer [Lor89]. The graph is constructed so that it has $G = P\Gamma L(2, 9)$ in its representation of degree 30 as its automorphism group. The vertices of the graph correspond to the points on which G acts. The neighbours of vertex 1 are the points lying in the unique orbit N_1 of length 3 of the stabilizer of 1. The edges for vertex i are precisely the points N_1^g where g is an element of G such that $1^g = i$.

```
> G := PermutationGroup< 30 |
>     (1, 2)(3, 4)(5, 7)(6, 8)(9, 13)(10, 12)(11, 15)(14, 19) (16, 23)
>         (17, 22)(18, 21)(20, 27)(24, 29)(25, 28)(26, 30),
>     (1, 24, 28, 8)(2, 9, 17, 22)(3, 29, 19, 15)(4, 5, 21, 25)
```

```
>         (6, 18, 7, 16)(10, 13, 30, 11)(12, 14)(20, 23)(26, 27) >;
> N1 := rep{ o : o in Orbits(Stabilizer(G, 1)) | #o eq 3 };
> tutte := Graph< 30 | <1, N1>^G >;
```

149.2.3 Construction of a General Digraph

`Digraph< n | edges :` *parameters* `>`
`Digraph< S | edges :` *parameters* `>`

`SparseRep` BOOL *Default :* `false`

Construct the digraph G with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The graph G, the vertex-set V of G; and the edge-set E of G. If SparseRep is `true` then the resulting graph will have a sparse representation.

The elements of E are specified by the list `edges`, where the items of `edges` may be objects of the following types:

(a) A pair $[v_i, v_j]$ of vertices in V. The directed edge from v_i to v_j will be added to the edge-set for G.

(b) A tuple of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the directed edges $[v_i, u_1], \ldots, [v_i, u_r]$ will be added to G.

(c) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of neighbours for the vertex v_i. The directed edges $[v_i, u_i]$, $1 \leq i \leq n$, $u_i \in N_i$, are added to G.

In addition to these three basic ways of specifying the *edges* list, the items in *edges* may also be:

(d) An edge e of a graph or digraph or multigraph or multidigraph or network of order n. If e is an edge $\{u, v\}$ in an undirected graph then both directed edges $[u, v]$ and $[v, u]$ are added to G.

(e) An edge-set E of a graph or digraph or multigraph or multidigraph or network of order n. Every edge e in E will be added to G according to the rule set out for a single edge.

(f) A graph or a digraph or a multigraph or a multidigraph or a network H of order n. Every edge e in H's edge-set is added to G according to the rule set out for a single edge.

(g) A $n \times n$ (0, 1)-matrix A. The matrix A will be interpreted as the adjacency matrix for a digraph H on n vertices and the edges of H will be added will be added to G.

(h) A set of

(i) Pairs of the form $[v_i, v_j]$ of vertices in V.

(ii) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

(iii) Edges of a graph or digraph or multigraph or multidigraph or network of order n.

(iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order n.

(j) A sequence of
(i) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

`IncidenceDigraph(A)`

Let A be a $n \times m$ matrix such that each column contains at most one entry equal to $+1$ and at most one entry equal to -1 (all others being zero). Then a digraph G of order n and size m having A as its incidence matrix will be constructed. The rows of A will correspond to the vertices of G while the columns will correspond to the edges. If column k of A contains $+1$ in row i and -1 in row j, the directed edge $v_i v_j$ will be included in G. If only row i has a non-zero entry (either 1 or -1), then the loop $v_i v_i$ will be included in G.

Example H149E4

The digraph D with vertices $\{v_1, v_2, v_3, v_4, v_5\}$ and edges $(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_3, v_2)$, and (v_4, v_3) will be constructed, firstly by listing its edges and secondly by giving its adjacency matrix.

```
> D1 := Digraph< 5 | [1, 2], [1, 3], [1, 4],
>                   [3, 2], [4, 3] >;
> D1;
Digraph
Vertex  Neighbours
1       2 3 4 ;
2       ;
3       2 ;
4       3 ;
5       ;
```

The same digraph with a sparse representation:

```
> D1 := Digraph< 5 | [1, 2], [1, 3], [1, 4],
>                   [3, 2], [4, 3] : SparseRep := true>;
```

We next construct the digraph by giving its adjacency matrix:

```
> M := MatrixRing(Integers(), 5);
> D2 := Digraph< 5 |
>           M![ 0,1,1,1,0,
>               0,0,0,0,0,
>               0,1,0,0,0,
```

```
>               0,0,1,0,0,
>               0,0,0,0,0 ] >;
> IsIsomorphic(D1, D2);
true
```

149.2.4 Operations on the Support

Let S be the set over which the graph G is defined. The set S is referred to as the *support* of G.

`Support(G)`

`Support(V)`

The indexed set used in the construction of G (or the graph for which V is the vertex-set), or the standard set $\{@1, \ldots, n@\}$ if it was not given.

`ChangeSupport(G, S)`

If G is a graph having n vertices and S is an indexed set of cardinality n, return a new graph H equal to G but whose support is S. That is, H is structurally equal to G and its vertex and edge decorations are the *same* as those for G.

`ChangeSupport(~G, S)`

The procedural version of the above function.

`StandardGraph(G)`

The graph H equal to G but defined on the standard support. That is, H is structurally equal to G and its vertex and edge decorations are the same as those for G.

Example H149E5

The Odd Graph, O_n, has as vertices all n-subsets of a $2n-1$ set with two vertices adjacent if and only if they are disjoint. We construct O_3 and then form its standard graph.

```
> V := Subsets({1..5}, 2);
> O3 := Graph< V | { {u,v} : u,v in V | IsDisjoint(u, v) } >;
> O3;
Graph
Vertex  Neighbours

{ 1, 5 }        { 2, 4 } { 2, 3 } { 3, 4 } ;
{ 2, 5 }        { 1, 3 } { 1, 4 } { 3, 4 } ;
{ 1, 3 }        { 2, 5 } { 2, 4 } { 4, 5 } ;
{ 1, 4 }        { 2, 5 } { 3, 5 } { 2, 3 } ;
{ 2, 4 }        { 1, 5 } { 1, 3 } { 3, 5 } ;
{ 3, 5 }        { 1, 4 } { 2, 4 } { 1, 2 } ;
```

```
{ 2, 3 }        { 1, 5 } { 1, 4 } { 4, 5 } ;
{ 1, 2 }        { 3, 5 } { 3, 4 } { 4, 5 } ;
{ 3, 4 }        { 1, 5 } { 2, 5 } { 1, 2 } ;
{ 4, 5 }        { 1, 3 } { 2, 3 } { 1, 2 } ;

> Support(O3);
{@
    { 1, 5 },
    { 2, 5 },
    { 1, 3 },
    { 1, 4 },
    { 2, 4 },
    { 3, 5 },
    { 2, 3 },
    { 1, 2 },
    { 3, 4 },
    { 4, 5 }
@}
> SO3 := StandardGraph(O3);
> SO3;
Graph
Vertex  Neighbours

1       5 7 9 ;
2       3 4 9 ;
3       2 5 10 ;
4       2 6 7 ;
5       1 3 6 ;
6       4 5 8 ;
7       1 4 10 ;
8       6 9 10 ;
9       1 2 8 ;
10      3 7 8 ;
> Support(SO3);
{@ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 @}
```

149.2.5 Construction of a Standard Graph

Some of the construction functions listed below can also be used to create a graph with a sparse representation. Are concerned only those functions creating relatively sparse graphs.

`BipartiteGraph(m, n)`

The complete bipartite graph, $K_{m,n}$, with partite sets of cardinality m and n.

`CompleteGraph(n)`

The complete graph K_n on n vertices.

`KCubeGraph(n : parameters)`

`SparseRep` BOOL *Default :* `false`

The graph of the n-dimensional cube, Q_n. The graph can be created as a sparse graph if so requested.

`MultipartiteGraph(Q)`

Given a sequence Q of positive integers $m_1, \ldots, m_r$, construct the complete multipartite graph $K_{m_1,\ldots,m_r}$.

`EmptyGraph(n : parameters)`

`SparseRep` BOOL *Default :* `false`

The graph on n vertices with no edges. It can be created as a sparse graph if so requested.

`NullGraph( : parameters)`

`SparseRep` BOOL *Default :* `false`

The graph with no vertices and no edges. It can be created as a sparse graph if so requested.

`PathGraph(n : parameters)`

`SparseRep` BOOL *Default :* `false`

The path graph on n vertices, i.e. the graph on vertices $v_1, \ldots, v_n$, with v_i and v_j $(1 \leq i < j \leq n)$ adjacent if and only if $j = i + 1$. The graph can be created as a sparse graph if so requested.

`PolygonGraph(n : parameters)`

`SparseRep` BOOL *Default :* `false`

Given an integer $n \geq 3$, define the polygon graph on n vertices, i.e. the graph on vertices $v_1, \ldots, v_n$, with v_i and v_j $(1 \leq i < j \leq n)$ adjacent if and only if $j = i + 1$, or $i = 1$ and $j = n$. The graph can be created as a sparse graph if so requested.

`RandomGraph(n, r : parameters)`

`SparseRep` BOOL *Default :* `false`

A random graph on n vertices such that the probability that an arbitrary pair of vertices is adjacent is given by the real number r, where r lies in the interval $[0, 1]$. The graph can be created as a sparse graph if so requested.

`RandomTree(n : parameters)`

`SparseRep` BOOL *Default :* `false`

A random tree on n vertices. It can be created as a sparse graph if so requested.

149.2.6 Construction of a Standard Digraph

As for standard graphs (Subsection 149.2.5) some of the construction functions listed below can be used to create digraphs with a sparse representation.

`CompleteDigraph(n)`

The complete symmetric digraph on n vertices.

`EmptyDigraph(n : parameters)`

`SparseRep` BOOL *Default :* `false`

The null digraph on n vertices. It can be created as sparse if so requested.

`RandomDigraph(n, r : parameters)`

`SparseRep` BOOL *Default :* `false`

A random digraph on n vertices such that the probability that there exists an edge directed from vertex u to vertex v, where u and v arbitrary vertices, is given by the real number r, where r lies in the interval $[0, 1]$. The digraph can be created as sparse if so requested.

Example H149E6

The complete symmetric digraph on 5 vertices is created as follows:

```
> kd5 := CompleteDigraph(5);
> kd5;
Digraph
Vertex  Neighbours

1       2 3 4 5 ;
2       1 3 4 5 ;
3       1 2 4 5 ;
4       1 2 3 5 ;
5       1 2 3 4 ;
```

We next construct a random digraph on 5 vertices such that the probability that there is an edge directed from an arbitrary vertex to any other arbitrary vertex is 0.75.

```
> rd5 := RandomDigraph(5, 0.75);
> rd5;
Digraph
Vertex  Neighbours

1       2 5 ;
2       2 3 4 5 ;
3       2 3 4 5 ;
4       1 2 3 5 ;
5       2 3 4 ;
```

149.3 Graphs with a Sparse Representation

As mentioned in the Introduction 149.1 of this chapter it is possible to construct graphs having a sparse representation. That is, graphs which are represented by means of an adjacency list. This has an obvious advantage for graphs with a low edge density, in that it allows to construct much larger graphs than if they were represented by means of an adjacency matrix (the dense representation). See Section 149.2.1 for more details on this issue.

Another advantage of the sparse representation is for graph algorithms which are linear in the number of edges (the planarity tester and the triconnectivity tester), and more generally, for those algorithms based on the adjacency list representation (the flow-based algorithms and the shortest-paths algorithms). Further, the sparse representation is required when creating multigraphs (see Chapter 150) since they may have multiple edges.

A significant number, but not all, of the existing MAGMA functions have been written for both representations. Should a function designed for a matrix representation of the graph be applied to a graph with a sparse representation, then the graph is automatically converted *without any user intervention*. This is also true when the reverse conversion is required.

Without being exhaustive, we will list here the functions which can deal with both graph representations without having to perform an (internal) conversion.

- Nearly all construction functions (149.2),
- All functions in 149.4, 149.6.1, 149.6.2,
- Some functions in 149.6.3 and 149.7,
- All functions in 149.8,
- The basic invariants and predicates in 149.10 and 149.11,
- Almost all functions in 149.12,
- All functions in 149.13.1 and all but the last in 149.13.2,
- The functions in 149.14.1,
- All functions in 149.17.

The functions in 149.5, 149.13.3, 149.13.4, 149.13.5, and 149.21, and all the functions listed in Chapter 150, deal with a graph as an adjacency list.

The functions in 149.9, 149.14.2, 149.16, 149.18, 149.19, 149.20, 149.22, and 149.23 deal with a graph as an adjacency matrix.

We emphasize again that should a conversion of the graph's representation take place due to internal specifications, the conversion takes place *automatically*, that is, *without user intervention*.

The only problem that may occur is when such a conversion concerns a (very) large graph and thus may possibly result in a lack of memory to create the required representation. This is where users might require control over which internal representation is used so as to minimise the likelihood of both representations coexisting. This is achieved by tuning the creation functions (as described in Section 149.2), and the following functions which determine the nature of a graph's representation.

When conversion of the graph's representation into the alternative one occurs the original representation is not deleted.

`HasSparseRep(G)`

`HasDenseRep(G)`

`HasSparseRepOnly(G)`

`HasDenseRepOnly(G)`

`HasDenseAndSparseRep(G)`

This set of functions determine the nature of a graph's representation.

Example H149E7

We give four examples, each illustrating one of the four possible cases that may arise. First, when the graph's dense representation remains unchanged:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} }>;
> HasDenseRepOnly(G);
true
> A := AutomorphismGroup(G);
> HasDenseRepOnly(G);
true
```

The same example using a sparse graph representation:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} } : SparseRep := true>;
> HasSparseRepOnly(G);
true
> A := AutomorphismGroup(G);
> HasDenseAndSparseRep(G);
true
```

Next the case when the graph's sparse representation remains unchanged:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} } : SparseRep := true>;
> HasSparseRepOnly(G);
true
> IsPlanar(G);
true
> HasSparseRepOnly(G);
true
```

Finally the same example using a dense graph representation:

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} }>;
> HasDenseRepOnly(G);
true
> IsPlanar(G);
true
> HasDenseAndSparseRep(G);
```

```
true
```

149.4 The Vertex–Set and Edge–Set of a Graph

149.4.1 Introduction

Let G be a graph on n vertices and m edges whose vertex-set is $V = \{v_1, \ldots, v_m\}$ and edge-set is $E = \{e_1, \ldots, e_m\}$. A graph created by MAGMA consists of three objects: the *vertex-set* V, the *edge-set* E and the graph G itself. The vertex-set and edge-set of a graph are *enriched* sets and consequently constitute types. The vertex-set and edge-set are returned as the second and third arguments, respectively, by all functions which create graphs. Alternatively, a pair of functions are provided to extract the vertex-set and edge-set of a graph G. The main purpose of having vertex-sets and edge-sets as types is to provide a convenient mechanism for referring to vertices and edges of a graph. Here, the functions applicable to vertex-sets and edge-sets are described.

149.4.2 Creating Edges and Vertices

`EdgeSet(G)`

Given a graph G, return the edge-set of G.

`Edges(G)`

A set E whose elements are the edges of the graph G. Note that this creates an indexed set and not the edge-set of G, in contrast to the function `EdgeSet`.

`VertexSet(G)`

Given a graph G, return the vertex-set of G.

`Vertices(G)`

A set V whose elements are the vertices of the graph G. In contrast to the function `VertexSet`, this function returns the collection of vertices of G in the form of an indexed set.

`V ! v`

Given the vertex-set V of the graph G and an element v of the support of V, create the corresponding vertex of G.

`V . i`

Given a vertex-set V and an integer i such that $1 \leq i \leq \#V$, create the vertex v_i of V.

`Index(v)`

Given a vertex v of some graph G, return the index of v in the (indexed) vertex-set of G.

`E ! {u, v}`

Given the edge-set E of the graph G and objects u, v belonging to the support of G, which correspond to adjacent vertices, create the edge uv of G.

`E ! [u, v]`

Given the edge-set E of the digraph G and objects u, v belonging to the support of G, which correspond to adjacent vertices, create the edge uv of G.

`E . i`

Given an edge-set E and an integer i such that $1 \leq i \leq \#E$, create the i-th edge of E.

Example H149E8

The construction of vertices and edges is illustrated using the Odd Graph, O_3. and then form its standard graph.

```
> S := Subsets({1..5}, 2);
> O3, V, E := Graph< S | { {u,v} : u,v in S | IsDisjoint(u, v) } >;
> VertexSet(O3);
Vertex-set of O3
> Vertices(O3);
{@ { 1, 5 }, { 2, 5 }, { 1, 3 }, { 1, 4 }, { 2, 4 }, { 3, 5 },
{ 2, 3 }, { 1, 2 }, { 3, 4 }, { 4, 5 } @}
> EdgeSet(O3);
Edge-set of O3
> Edges(O3);
{@ {{ 1, 5 }, { 2, 4 }}, {{ 1, 5 }, { 2, 3 }}, {{ 1, 5 }, { 3, 4 }},
{{ 2, 5 }, { 1, 3 }}, {{ 2, 5 }, { 1, 4 }}, {{ 2, 5 }, { 3, 4 }},
{{ 1, 3 }, { 2, 4 }}, {{ 1, 3 }, { 4, 5 }}, {{ 1, 4 }, { 3, 5 }},
{{ 1, 4 }, { 2, 3 }}, {{ 2, 4 }, { 3, 5 }}, {{ 3, 5 }, { 1, 2 }},
{{ 2, 3 }, { 4, 5 }}, {{ 1, 2 }, { 3, 4 }}, {{ 1, 2 }, { 4, 5 }} @}
> u := V!{1, 2};
> u, Type(u);
{1, 2} GrphVert
> Index(u);
8
> x := E!{ {1,2}, {3,4}};
> x, Type(x);
{{ 1, 2 }, { 3, 4 }} GrphEdge
```

149.4.3 Operations on Vertex-Sets and Edge-Sets

For each of the following operations, S and T may be interpreted as either the vertex-set or the edge-set of the graph G. The variable s may be interpreted as either a vertex or an edge. The edge-set and vertex-set support all the standard set operations.

`#S`

The cardinality of the set S.

`s in S`

Return `true` if the vertex (edge) s lies in the vertex-set (edge-set) S, otherwise `false`.

`s notin S`

Return `true` if the vertex (edge) s does not lie in the vertex-set (edge-set) S, otherwise `false`.

`S subset T`

Return `true` if the vertex-set (edge-set) S is contained in the vertex-set (edge-set) T, otherwise `false`.

`S notsubset T`

Return `true` if the vertex-set (edge-set) S is not contained in the vertex-set (edge-set) T, otherwise `false`.

`S eq T`

Return `true`if the vertex-set (edge-set) S is equal to the vertex-set (edge-set) T.

`s eq t`

Returns `true` if the vertex (edge) s is equal to the vertex (edge) t.

`S ne T`

Returns `true` if the vertex-set (edge-set) S is not equal to the vertex-set (edge-set) T.

`s ne t`

Returns `true` if the vertex (edge) s is not equal to the vertex (edge) t.

`ParentGraph(S)`

Return the graph G for which S is the vertex-set (edge-set).

`ParentGraph(s)`

Return the graph G for which s is a vertex (edge).

`Random(S)`

Choose a random element from the vertex-set (edge-set) S.

`Representative(S)`

`Rep(S)`

Choose some element from the vertex-set (edge-set) S.

```
for x in S do ...  end for;
```

The vertex-set (edge-set) S may appear as the range in the `for`-statement.

```
for random x in S do ... end for;
```

The vertex-set (edge-set) S may appear as the range in the `for random` - statement.

149.4.4 Operations on Edges and Vertices

`EndVertices(e)`

Given an edge e belonging to the graph G, return a set containing the two end-vertices of e. If G is a digraph return the two end-vertices in a sequence.

`InitialVertex(e)`

Given an undirected or directed edge e from vertex u to vertex v, return vertex u. This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

`TerminalVertex(e)`

Given an undirected or directed edge e from vertex u to vertex v, return vertex v. This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

`IncidentEdges(u)`

Given a vertex u of a graph G, return the set of all edges incident with the vertex u. If G is directed, then the set consists of all the edges incident into u and from v.

149.5 Labelled, Capacitated and Weighted Graphs

A vertex labelling of a graph G is a partial map f from the vertex-set V of G into a set L. An edge labelling of a graph G is a partial map f from the edge-set E of G into a set L. A labelled graph is built by assigning labels successively to vertices or edges after the (unlabelled) graph has been constructed.

Similarly, a capacitated graph G is a partial map from its edge-set into Z^+, and a weighted graph G is a partial map from its edge-set into R, R any ring with a total order. Those two last features are particularly convenient when running shortest-paths and flow algorithms. Edge capacities and edge weights are assigned to edges once the graph has been constructed. Any graph edge may carry a label, together with a capacity and/or a weight.

All the functions for decorating graph vertices and edges are fully documented in Section 150.4 in Chapter 150. A few examples are also given there.

149.6 Standard Constructions for Graphs

The two main ways in which to construct a new graph from an old one are either by taking a subgraph of or by modifying the original graph. Another method is to take the quotient graph. When the first two methods are employed, the support set and vertex and edge decorations are retained in the resulting graph.

149.6.1 Subgraphs and Quotient Graphs

`sub< G | list >`

Construct the graph H as a subgraph of G. The function returns three values: The graph H, the vertex-set V of H; and the edge-set E of H. If G has a support set and/or if G has vertex/edge decorations, then these attributes are transferred to the subgraph H.

The elements of V and of E are specified by the list *list* whose items can be objects of the following types:

(a) A vertex of G. The resulting subgraph will be the subgraph induced on the subset of `VertexSet(`G`)` defined by the vertices in *list*.

(b) An edge of G. The resulting subgraph will be the subgraph with vertex-set `VertexSet(`G`)` whose edge-set consists of the edges in *list*.

(c) A set of
   (i) Vertices of G.
   (ii) Edges of G.

It is easy to recover the map that maps the vertices of the subgraph to the vertices of the supergraph and vice-versa. We give an example below.

`quo< G | P >`

Given a graph G and a partition P of the vertex-set $V(G)$ of G, construct the quotient graph Q of G defined by P. The partition is given as a set of subsets of $V(G)$. Suppose the cells of the partition P are $P_1, \ldots, P_r$. The vertices of Q correspond to the cells P_i. Vertices v_i and v_j of Q are adjacent if and only if there is an edge in G joining a vertex in P_i with a vertex in P_j.

Example H149E9

We start with a simple example which demonstrates how to map the vertices of a subgraph onto the vertices of the supergraph and vice-versa: This is achieved by simple coercion into the appropriate vertex-set.

```
> K5, V := CompleteGraph(5);
> K3, V1 := sub< K5 | { V | 3, 4, 5 } >;
> IsSubgraph(K5, K3);
true
>
> V!V1!1;
3
>
> V1!V!4;
2
>
>  V1!V!1;
>> V1!V!1;
     ^
Runtime error in '!': Illegal coercion
LHS: GrphVertSet
RHS: GrphVert
```

A 1-factor of a graph G is a set of disjoint edges which form a spanning forest for G. In the following example, we construct the graph that corresponds to K_6 with a 1-factor removed.

```
> K6, V, E := CompleteGraph(6);
> K6;
Graph
Vertex  Neighbours

1       2 3 4 5 6 ;
2       1 3 4 5 6 ;
3       1 2 4 5 6 ;
4       1 2 3 5 6 ;
5       1 2 3 4 6 ;
6       1 2 3 4 5 ;
> F1 := { E | {1,2}, {3, 4}, {5, 6} };
> G1, V1, E1 := K6 - F1;
> G1;
Graph
```

```
Vertex  Neighbours

1       3 4 5 6 ;
2       3 4 5 6 ;
3       1 2 5 6 ;
4       1 2 5 6 ;
5       1 2 3 4 ;
6       1 2 3 4 ;
```

Example H149E10

Taking the complete graph K_9, we form its quotient with respect to the partition of the vertex-set into three 3-sets.

```
> K9, V, E := CompleteGraph(9);
> P := { { V | 1, 2, 3}, { V | 4, 5, 6}, { V | 7, 8, 9} };
> Q := quo< K9 | P >;
> Q;
Graph
Vertex  Neighbours

1       2 3 ;
2       1 3 ;
3       1 2 ;
```

149.6.2 Incremental Construction of Graphs

Unless otherwise specified, each of the functions described in this section returns three values:

(i) The graph G;

(ii) The vertex-set V of G;

(iii) The edge-set E of G.

149.6.2.1 Adding Vertices

`G + n`

Given a graph G and a non-negative integer n, adds n new vertices to G. The existing vertex and edge decorations are retained, but the support will become the standard support.

`G +:= n`
`AddVertex(~G)`
`AddVertices(~G, n)`

The procedural version of the previous function. `AddVertex` adds one vertex only to G.

`AddVertex(~G, l)`

Given a graph G and a label l, adds a new vertex with label l to G. The existing vertex and edge decorations are retained, but the support will become the standard support.

`AddVertices(~G, n, L)`

Given a graph G and a non-negative integer n, and a sequence of n labels, adds n new vertices to G with labels from L. The existing vertex and edge decorations are retained, but the support will become the standard support.

149.6.2.2 Removing Vertices

`G - v`
`G - U`

Given a graph G and a vertex v or a set of vertices U of G, removes v or the vertices in U from G. The support and vertex and edge decorations are retained.

`G -:= v`
`G -:= U`
`RemoveVertex(~G, v)`
`RemoveVertices(~G, U)`

The procedural versions of the previous functions.

149.6.2.3 Adding Edges

`G + { u, v }`
`G + [ u, v ]`

Given a graph G and a pair of vertices of G, add the edge to G described by this pair. If G is undirected then the edge must be given as a set of (two) vertices, if G is directed the edge is given as a sequence of (two) vertices. The support and existing vertex and edge decorations are retained. This set of functions has two return values: The first is the modified graph and the second is the newly created edge.

```
G + { { u, v } }
```
```
G + { [ u, v ] }
```

Given a graph G and a set of pairs of vertices of G, add the edges to G described by these pairs. If G is undirected then edges must be given as a set of (two) vertices, if G is directed edges are given as a sequence of (two) vertices. The support and existing vertex and edge decorations are retained.

```
G +:= { u, v }
```
```
G +:= [ u, v ]
```
```
G +:= { { u, v } }
```
```
G +:= { [ u, v ] }
```

The procedural versions of the previous four functions.

```
AddEdge(G, u, v)
```

Given a graph G, two vertices of G u and v, returns a new edge between u and v. The support and existing vertex and edge decorations are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

```
AddEdge(G, u, v, l)
```

Given a graph G, two vertices of G u and v, and a label l, adds a new edge with label l between u and v. The support and existing vertex and edge decorations are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

```
AddEdge(~G, u, v)
```
```
AddEdge(~G, u, v, l)
```

Procedural versions of previous functions adding edges to a graph.

```
AddEdges(G, S)
```

Given a graph G, a set S of pairs of vertices of G, adds the edges specified in S. The elements of S must be sets or sequences of two vertices of G, depending on whether G is undirected or directed respectively. The support and existing vertex and edge decorations are retained.

```
AddEdges(G, S, L)
```

Given a graph G, a sequence S of pairs of vertices of G, and a sequence L of labels of same length, adds the edges specified in S with its corresponding label in L. The elements of S must be sets or sequences of two vertices of G, depending on whether G is undirected or directed respectively. The support and existing vertex and edge decorations are retained.

`AddEdges(~G, S)`

`AddEdges(~G, S, L)`

Procedural versions of previous functions adding edges to a graph.

149.6.2.4 Removing Edges

`G - e`

`G - { e }`

Given a graph G and an edge e or a set S of edges of G, removes e or the edges in S from G. The resulting graph will have vertex-set $V(G)$ and edge-set $E(G) \setminus \{e\}$ or $E(G) \setminus S$. The support and existing vertex and edge decorations are retained.

`G - { { u, v } }`

`G - { [u, v] }`

Given a graph G and a set S of pairs $\{u, v\}$ or $[u, v]$, u, v vertices of G, form the graph having vertex-set $V(G)$ and edge-set $E(G)$ minus the edges from u to v for all pairs in S. An edge is represented as a set if G is undirected, as a set otherwise. The support and existing vertex and edge decorations are retained.

`G -:= e`

`G -:= { e }`

`G -:= { { u, v } }`

`G -:= { [u, v] }`

`RemoveEdge(~G, e)`

`RemoveEdges(~G, S)`

`RemoveEdge(~G, u, v)`

The procedural versions of the previous functions.

149.6.3 Constructing Complements, Line Graphs; Contraction, Switching

Unless otherwise stated, each of the functions described here apply to both graphs and digraphs. Further, each of the functions returns three values:

(i) The graph G;

(ii) The vertex-set V of G;

(iii) The edge-set E of G.

`Complement(G)`

The complement of the graph G.

`Contract(e)`

Given an edge $e = \{u, v\}$ of the graph G, form the graph obtained by identifying the vertices u and v, and removing any multiple edges (and loops if the graph is undirected) from the resulting graph. The graph's support and vertex/edges decorations are retained.

`Contract(u, v)`

Given vertices u and v belonging to the graph G, return the graph obtained by identifying the vertices u and v, and removing any multiple edges (and loops if the graph is undirected) from the resulting graph. The graph's support and vertex/edges decorations are retained.

`Contract(S)`

Given a set S of vertices belonging to the graph G, return the graph obtained by identifying all of the vertices in S, and removing any multiple edges (and loops if the graph is undirected) from the resulting graph. The graph's support and vertex/edges decorations are retained.

`InsertVertex(e)`

Given an edge e in the graph G, insert a new vertex of degree 2 in e. If applicable, the two new edges thus created that replace e will have the same capacity and weight as e. They will be unlabelled. The vertex labels and the edge decorations of G are retained, but the resulting graph will have standard support.

`InsertVertex(T)`

Given an a set T of edges belonging to the graph G, insert a vertex of degree 2 in each edge belonging to the set T. Vertex and edge decorations are handled as in the single edge case.

`LineGraph(G)`

The line graph of the non-empty graph G.

`Switch(u)`

Given a vertex u in an undirected graph G, construct an undirected graph H from G, where H has the same vertex-set as G and the same edge-set, except that the vertices that were the neighbours of u in G become the non-neighbours of u in H, and the vertices that were non-neighbours of u in G become the neighbours of u in H. The support and vertex labels are *not* retained in the resulting graph.

`Switch(S)`

Given a set S of vertices belonging to the undirected graph G, apply the function `Switch(u)`, to each vertex of S in turn. The support and vertex labels are *not* retained in the resulting graph.

Example H149E11

We illustrate the use of some of these operations by using them to construct the Grötzch graph. This graph may be built by taking the complete graph $K5$, choosing a cycle of length 5 (say, 1-3-5-2-4), inserting a vertex of degree two on each edge of this cycle and finally connecting each of these vertices to a new vertex.

```
> G := CompleteGraph(5);
> E := EdgeSet(G);
> H := InsertVertex({ E | { 1, 3 }, { 1, 4 }, { 2, 4 }, { 2, 5 }, { 3, 5 } } );
> L := Union(H, CompleteGraph(1));
> V := VertexSet(L);
> L := L + {  { V.11, V.6 }, { V.11, V.7 }, { V.11, V.8 }, { V.11, V.9 },
>            { V.11, V.10 } };
> L;
Graph
Vertex  Neighbours

1       2 5 6 7 ;
2       1 3 8 10 ;
3       2 4 6 9 ;
4       3 5 7 8 ;
5       1 4 9 10 ;
6       1 3 11 ;
7       1 4 11 ;
8       2 4 11 ;
9       3 5 11 ;
10      2 5 11 ;
11      6 7 8 9 10 ;
```

149.7 Unions and Products of Graphs

The support and vertex/edge decorations of the original graphs are *not* retained in the graph resulting from applying any of the union functions below.

`Union(G, H)`

`G join H`

Given graphs G and H with disjoint vertex sets $V(G)$ and $V(H)$, respectively, construct their union, i.e. the graph with vertex-set $V(G) \cup V(H)$, and edge-set $E(G) \cup E(H)$.

`EdgeUnion(G, H)`

Given graphs G and H having the same number of vertices, construct their edge union K. This construction identifies the i-th vertex of G with the i-th vertex of H for all i. The edge union has the same vertex-set as G (and hence as H) and vertices u and v of K are adjacent if and only if either u and v are adjacent in G or u and v are adjacent in H.

`CompleteUnion(G, H)`

Given graphs G and H with disjoint vertex-sets $V(G)$ and $V(H)$, respectively, construct the complete union of G and H. This graph consists of the union of G and H (`Union(G, H)`), together with edges uv, for all u in $V(G)$ and all v in $V(H)$.

`CartesianProduct(G, H)`

Given graphs G and H with disjoint vertex-sets $V(G)$ and $V(H)$, respectively, form the product $K = G \times H$ of G and H. The product has vertex-set $V(G) \times V(H)$. Two vertices $u = (u_1, u_2)$ and $v = (v_1, v_2)$ of K are adjacent when either

(a) $u_1 = v_1$ and u_2 adj v_2, or

(b) $u_2 = v_2$ and u_1 adj v_1.

`LexProduct(G, H)`

Given graphs G and H with disjoint vertex-sets $V(G)$ and $V(H)$, respectively, form the lexicographic product K of G and H. The lexicographic product has vertex-set $V(G) \times V(H)$. Two vertices $u = (u_1, u_2)$ and $v = (v_1, v_2)$ of K are adjacent when either

(a) u_1 adj v_1, or

(b) $u_1 = v_1$ and u_2 adj v_2.

`TensorProduct(G, H)`

Given graphs G and H with disjoint vertex-sets $V(G)$ and $V(H)$, respectively, form the tensor product K of G and H. This graph has vertex-set $V(G) \times V(H)$. Two vertices $u = (u_1, u_2)$ and $v = (v_1, v_2)$ of K are adjacent when u_1 adj v_1 and u_2 adj v_2.

`G ^ n`

Given a graph G and a positive integer n, construct the n-th power K of G. This graph has the same vertex-set as G, and vertices u and v of K are adjacent if and only if the distance between u and v in G is less than or equal to n.

149.8 Converting between Graphs and Digraphs

Note that the two functions `UnderlyingGraph` and `UnderlyingDigraph` may also be used when one needs to get a copy of a graph G without G's support and vertex/edge decorations.

`OrientatedGraph(G)`

Given a graph G, produce a digraph D whose vertex-set is the same as that of G and whose edge-set consists of the edges of G, each given a direction. The edges of D are always directed from the lower numbered vertex to the higher numbered vertex. Thus, if G contains the edge $\{u, v\}$, then D will have the edge $[u, v]$ if $u < v$, otherwise the edge $[u, v]$. The support and vertex/edge decorations of G are not retained.

`UnderlyingGraph(D)`

The underlying graph G of the graph D; G has the same vertex-set as D. If D is undirected, then G is a copy of D without D's support and vertex/edge decorations. If D is directed, then two vertices u and v are adjacent in G if and only if, in D, there is either an edge directed from u to v or from v to u. The support and vertex/edge decorations of G are not retained.

`UnderlyingDigraph(G)`

The underlying digraph D of the graph G; D has the same vertex-set as G. If G is directed, then D is a copy of G without D's support and vertex/edge decorations. If G is undirected, then if vertices u and v are adjacent in G then, in D, there will be both an edge directed from u to v and an edge directed from v to u. The support and vertex/edge decorations of G are not retained.

149.9 Construction from Groups, Codes and Designs

149.9.1 Graphs Constructed from Groups

`CayleyGraph(A)`

`CayleyGraph(A :` *parameter*`)`

`UnlabelledCayleyGraph(A)`

`Labelled` BoolElt *Default :* `true`

Given a finite group A defined on generating set X, construct the Cayley graph C of A relative to the generating set X. This graph is defined as follows: The vertices correspond to the elements of A and two vertices u, v are adjacent if and only if there exists an element x in X such that $u * x = v$.

The optional parameter `Labelled` (`Labelled := true` by default) can be set to `false` to prevent the graph being labelled. If this is not done, then the vertices of

C will be labelled with the appropriate elements of A and the (directed) edge from u to v will be labelled with the appropriate element x as defined above.

`SchreierGraph(A, B)`

`UnlabelledSchreierGraph(A, B)`

Given a finite group A defined on the generating set X and a subgroup B of A, construct the Schreier coset graph S for A over B, relative to X. The graph S is defined as follows: The vertices correspond to the cosets of B in A, and two vertices u, v are adjacent in S if and only if there exists an element x in X such that $u*x = v$.

The graph is available in both a labelled and an unlabelled version.

`OrbitalGraph(P, u, T)`

Let P be a transitive permutation group acting on the set $\Omega = \{1, \ldots, n\}$. Let u be an element of Ω and let $T = \{t_1, \ldots, t_r\}$ be a subset of Ω. This function constructs the underlying graph G of the digraph corresponding to the union of P-orbits containing the pairs $(u, t_1), \ldots, (u, t_r)$. Thus, if T defines a self-paired orbit Δ of the stabilizer in P of the point u, this function constructs the orbital graph associated with Δ.

`ClosureGraph(P, G)`

Let P be a permutation group acting on the set $\Omega = \{1, \ldots, n\}$. Let G be a graph (digraph) with vertices $v_1, \ldots, v_n$. This function adds the minimum number of edges to G so as to produce a graph (digraph) H which is left invariant by the group P.

`PaleyGraph(q)`

The Paley graph of $\mathbf{F}_q$ where q must be a prime power equivalent to 1 mod 4. Vertices are in bijection with elements of $\mathbf{F}_q$ and distinct elements are adjacent when their difference is a square in the field.

`PaleyTournament(q)`

The Paley tournament of $\mathbf{F}_q$ where q must be a prime power equivalent to 3 mod 4. Vertices are in bijection with elements of $\mathbf{F}_q$ and there is an edge from u to v when $u \neq v$ and $v - u$ is a square in the field.

149.9.2 Graphs Constructed from Designs

`IncidenceGraph(D)`

Given an incidence structure $D = (X, \mathcal{B})$, construct the incidence graph G of D. The vertices of G is $X \cup \mathcal{B}$. The adjacency rules are as follows: No two vertices of X are adjacent; no two vertices of $\mathcal{B}$ are adjacent; a vertex $x \in X$ is adjacent to a vertex $B \in \mathcal{B}$ if and only if $x \in B$.

`PointGraph(D)`

Given an incidence structure $D = (X, \mathcal{B})$, construct the point graph G of D. The vertex-set of G is X. Vertices $x \in X$, $y \in X$ are adjacent in G if there is a block $B \in \mathcal{B}$ such that $x \in B$ and $y \in B$.

`BlockGraph(D)`

The block graph of the incidence structure D; i.e. the point graph of the dual of D.

`IncidenceGraph(P)`

Given a plane P with point-set V and line-set L, construct the incidence graph G of P. The vertex-set of G is $V \cup L$. The adjacency rules are as follows: No two vertices of V are adjacent; no two vertices of L are adjacent; a vertex $v \in V$ is adjacent to a vertex $a \in L$ if and only if v lies on a.

`PointGraph(P)`

Given a plane P with point-set V and line-set L, construct the point graph G of P. The vertex-set of G is V. Vertices $u, v \in V$ are adjacent in G iff there is a line in L that contains them both.

`LineGraph(P)`

Given a plane P with point-set V and line-set L, construct the point graph G of P. The vertex-set of G is L. Lines $a, b \in L$ are adjacent in G iff there is a vertex in V that lies on them both.

`HadamardGraph(H :` *parameters*`)`

`Labels` BoolElt *Default :* `false`

The graph of the ± 1 matrix H as described in Brendan D. McKay's note "Hadamard equivalence via graph isomorphism" (with self-loops omitted). The parameter `Labels` is set to `false` by default, but when set to `true`, the vertices associated with rows are labelled "row" and the others "col". Those labelled "row" are those given loops in McKay's paper.

149.9.3 Miscellaneous Graph Constructions

`Converse(G)`

Returns the converse H of the directed graph G: if $[u, v]$ is an edge of G then $[v, u]$ is an edge of H.

`OddGraph(n)`

The nth odd graph. Vertices are $(n-1)$–subsets of a $(2n-1)$-set with vertices adjacent if and only if the $(n-1)$–subsets are disjoint.

`TriangularGraph(n)`

The nth triangular graph. Vertices are 2-subsets of a n-set with vertices adjacent if and only if the 2-subsets are unequal and not disjoint.

`SquareLatticeGraph(n)`

The nth square lattice graph. This is the cartesian product of the nth complete graph with itself.

`ClebschGraph()`
`ShrikhandeGraph()`
`GewirtzGraph()`

Return the named graph.

`ChangGraphs()`

Return a sequence of the three Chang graphs.

149.10 Elementary Invariants of a Graph

`Order(G)`
`NumberOfVertices(G)`

The number of vertices of the graph G.

`Size(G)`
`NumberOfEdges(G)`

The number of edges of the graph G.

`CharacteristicPolynomial(G)`

The characteristic polynomial (over the integers) of the graph G; i.e. the characteristic polynomial of the adjacency matrix of G.

`Spectrum(G)`

The spectrum of the graph G; i.e. the roots of the characteristic polynomial of G. The roots are returned as a set of tuples, each containing a root and its multiplicity.

149.11 Elementary Graph Predicates

Let G and H be two graphs. For clarity, we list here the conditions under which G is equal to H and H is a subgraph of G. The conditions take into account the fact that the graphs may have a support and that their vertex and edges may be decorated.

The graphs G and H are equal if and only if:

- they are of the same type,
- they are structurally identical,
- they have the same support,
- they have identical vertex and edge labels,
- if applicable, the total capacity from u to v in G is equal to the total capacity from u to v in H.

Also, H is a subgraph of G if and only if:

- they are of the same type,
- H is a structural subgraph of G,
- any vertex `v` in H has the same support as the vertex `VertexSet(G)!v` in G,
- any vertex `v` in H has the same label as the vertex `VertexSet(G)!v` in G,
- any edge `e` in H has the same label as the edge `EdgeSet(G)!e` in G,
- if applicable, the total capacity from u to v in G is at least as large as the total capacity from u to v in H.

Note that the truth value of the above two tests is not dependent on the weights of the edges of the graphs, should these edges be weighted. One could argue that given this shortcoming, the tests should not be dependent on the capacities of the edges either. We welcome users' comments on this matter.

`u adj v`

Let u and v be two vertices of the same graph G. If G is undirected, returns `true` if and only if u and v are adjacent. If G is directed, returns `true` if and only if there is an edge directed from u to v.

`e adj f`

Let e and f be two edges of the same graph G. If G is undirected, returns `true` if and only if e and f share a common vertex. If G is directed, returns `true` if and only if the terminal vertex of e (f) is the initial vertex of f (e).

`u notadj v`

The negation of the `adj` predicate applied to vertices.

`e notadj f`

The negation of the `adj` predicate applied to edges.

`u in e`

Let u be a vertex and e an edge of a graph G. Returns `true` if and only if u is an end-vertex of e.

`u notin e`

The negation of the `in` predicate applied to a vertex with respect to an edge.

`G eq H`

Returns `true` if the graphs G and H are identical, otherwise `false`. G and H are identical if and only if:

- they are structurally identical,
- they have the same support,
- they have identical vertex and edge labels,
- if applicable, the total capacity from u to v in G is equal to the total capacity from u to v in H.

Thus, as an example, if G and H are structurally identical graphs, and the vertices of G are labelled while the vertices of H are not, then `G eq H` returns `false`.

`IsSubgraph(G, H)`

Returns `true` if H is a subgraph of G, otherwise `false`. H is a subgraph of G if and only if:

- H can be determined to be a structural subgraph of G,
- any vertex `v` in H has the same support as the vertex `VertexSet(G)!v` in G,
- any vertex `v` in H has the same label as the vertex `VertexSet(G)!v` in G,
- any edge `e` in H has the same label as the edge `EdgeSet(G)!e` in G.
- if applicable, the total capacity from u to v in G is at least as large as the total capacity from u to v in H.

`IsBipartite(G)`

Returns `true` if the graph G is a bipartite graph, otherwise `false`.

`IsComplete(G)`

Returns `true` if the graph G on n vertices is the complete graph on n vertices, otherwise `false`.

`IsEulerian(G)`

Returns `true` if the graph G is an eulerian graph, otherwise `false`. A eulerian graph is a graph whose vertices have all even degree. An eulerian digraph is a digraph whose vertices have same in and outdegree. That is, if D is a digraph, D is eulerian if and only if `OutDegree(v)` equals `InDegree(v)` for all vertices v of D.

`IsForest(G)`

Returns `true` if the graph G is a forest, i.e. does not possess any cycles, otherwise `false`.

`IsEmpty(G)`

Returns `true` if the graph G is an empty graph, otherwise `false`. A graph is empty if its edge-set E is empty.

`IsNull(G)`

Returns `true` if the graph G is a null graph, otherwise `false`. A graph is null if its vertex-set V is empty.

`IsPath(G)`

Returns `true` if the graph G is a path graph, otherwise `false`.

`IsPolygon(G)`

Returns `true` if the graph G is a polygon graph, otherwise `false`.

`IsRegular(G)`

Returns `true` if the graph G is a regular graph, otherwise `false`.

`IsTree(G)`

Returns `true` if the graph G is a tree, otherwise `false`.

149.12 Adjacency and Degree

149.12.1 Adjacency and Degree Functions for a Graph

`Degree(u)`

Given a vertex u of the graph G, return the degree of u, ie the number of edges incident to u.

`Alldeg(G, n)`

Given a graph G, and a non-negative integer n, return the set of all vertices of G that have degree equal to n.

`MaximumDegree(G)`

`Maxdeg(G)`

The maximum of the degrees of the vertices of the graph G. This function returns two values: the maximum degree, and a vertex of G having that degree.

`MinimumDegree(G)`

`Mindeg(G)`

The minimum of the degrees of the vertices of the graph G. This function returns two values: the minimum degree, and a vertex of G having that degree.

`DegreeSequence(G)`

Given a graph G such that the maximum degree of any vertex of G is r, return a sequence D of length $r+1$, such that $D[i]$, $1 \leq i \leq r+1$, is the number of vertices in G having degree $i-1$.

`Valence(G)`

Given a regular graph G, return the valence of G (the degree of any vertex).

`Neighbours(u)`

`Neighbors(u)`

Given a vertex u of the graph G, return the set of vertices of G that are adjacent to u.

`IncidentEdges(u)`

The set of all edges incident with the vertex u.

`Bipartition(G)`

Given a bipartite graph G, return its two partite sets in the form of a pair of subsets of $V(G)$.

`MinimumDominatingSet(G)`

A dominating set S of a graph G is such that the vertices of S together with the vertices adjacent to vertices in S form the vertex-set of G. A dominating set S is minimal if no proper subset of S is a dominating set. A minimum dominating set is a minimal dominating set of smallest size. The algorithm implemented is a backtrack algorithm (see [Chr75] p. 41).

149.12.2 Adjacency and Degree Functions for a Digraph

`InDegree(u)`

The number of edges directed into the vertex u belonging to the directed graph G.

`OutDegree(u)`

The number of edges of the form uv where u is a vertex belonging to the directed graph G.

`Degree(u)`

Given a vertex u belonging to the digraph G, return the total degree of u, i.e. the sum of the in–degree and out–degree for u.

`Alldeg(G, n)`

Given a digraph G, and a non–negative integer n, return the set of all vertices of G that have total degree equal to n.

`MaximumInDegree(G)`

`Maxindeg(G)`

The maximum indegree of the vertices of the digraph G. This function returns two values: the maximum indegree, and the first vertex of G having that degree.

`MaximumOutDegree(G)`

`Maxoutdeg(G)`

The maximum outdegree of the vertices of the digraph G. This function returns two values: the maximum outdegree, and the first vertex of G having that degree.

`MinimumInDegree(G)`

`Minindeg(G)`

The minimum indegree of the vertices of the digraph G. This function returns two values: the minimum indegree, and the first vertex of G having that degree.

`MinimumOutDegree(G)`

`Minoutdeg(G)`

The minimum outdegree of the vertices of the digraph G. This function returns two values: the minimum outdegree, and the first vertex of G having that degree.

`MaximumDegree(G)`

`Maxdeg(G)`

The maximum total degree of the vertices of the digraph G. This function returns two values: the maximum total degree, and the first vertex of G having that degree.

`MinimumDegree(G)`

`Mindeg(G)`

The minimum total degree of the vertices of the digraph G. This function returns two values: the minimum total degree, and the first vertex of G having that degree.

`DegreeSequence(G)`

Given a digraph G such that the maximum degree of any vertex of G is r, return a sequence D of length $r+1$, such that $D[i]$, $1 \leq i \leq r+1$, is the number of vertices in G having degree $i-1$.

`InNeighbours(u)`

`InNeighbors(u)`

Given a vertex u of the digraph G, return the set containing all vertices v such that vu is an edge in the digraph, i.e. the starting points of all edges that are directed into the vertex u.

`OutNeighbours(u)`

`OutNeighbors(u)`

Given a vertex u of the digraph G, return the set of vertices v of G such that uv is an edge in the graph G, i.e. the set of vertices v that are the end vertices of edges directed from u to v.

`IncidentEdges(u)`

The set of all edges incident with the vertex u.

149.13 Connectedness

149.13.1 Connectedness in a Graph

`IsConnected(G)`

Returns `true` if and only if the graph G is connected.

`Components(G)`

The connected components of the graph G. These are returned in the form of a sequence of subsets of the vertex-set of G.

`Component(u)`

The subgraph corresponding to the connected component of the graph G containing vertex u. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

`IsSeparable(G)`

Returns `true` if and only if the graph G is connected and has at least one cut vertex.

`IsBiconnected(G)`

Returns `true` if and only if the graph G is biconnected.

`CutVertices(G)`

The set of cut vertices for the connected graph G (a set of vertices).

`Bicomponents(G)`

The biconnected components of the graph G. These are returned in the form of a sequence of subsets of the vertex-set of G. The graph may be disconnected.

149.13.2 Connectedness in a Digraph

`IsStronglyConnected(G)`

Returns `true` if and only if the digraph G is strongly connected.

`IsWeaklyConnected(G)`

Returns `true` if and only if the digraph G is weakly connected.

`StronglyConnectedComponents(G)`

The strongly connected components of the digraph G. These are returned in the form of a sequence of subsets of the vertex-set of G.

`Component(u)`

The subgraph corresponding to the connected component of the digraph G containing vertex u. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

149.13.3 Graph Triconnectivity

The linear-time triconnectivity algorithm by Hopcroft and Tarjan [HT73] has been implemented with corrections of our own and from C. Gutwenger and P. Mutzel [GM01]. This algorithm requires that the graph has a sparse representation. The input graph may be disconnected or have cut vertices.

The algorithm completely splits the graph into components that can later be reconstructed as triconnected graphs: It is our belief that the word "tricomponents" is misused in all the papers discussing Hopcroft and Tarjan's algorithm, including [HT73] and [GM01]. Let us clarify this point.

Assume that G is a biconnected but not triconnected graph with vertex-set V and with one separation pair (a, b) which splits $V - \{a, b\}$ into V_1 and V_2. That is, for any vertices $u \in V_1$ and $v \in V_2$, every path from u to v in G contains at least one of a or b.

Now, the algorithm will correctly find the separation pair (a, b) and return the two subsets of V, $V_1 + \{a, b\}$ and $V_2 + \{a, b\}$. Let G_1 and G_2 be the subgraphs of G induced by $V_1 + \{a, b\}$ and $V_2 + \{a, b\}$ respectively. We say that the algorithm *splits* G into G_1 and G_2.

But G_1 and G_2 are triconnected if and only if there is an edge (a, b) in G. We call G_1 and G_2 the *split components* of G.

More generally, if $G_1, \ldots, G_m$ are the split components of a graph G, one can make the G_i triconnected by adding the edges (a, b) to them for all the separation pairs (a, b) of G with $a, b \in G_i$ for some i.

The algorithm finds all the cut vertices and/or the separation pairs of a graph, unless only the boolean test is applied. When this is the case the algorithm stops as soon as a cut vertex or separation pair is found.

The triconnectivity algorithm only applies to undirected graphs.

`IsTriconnected(G)`

Returns `true` if and only if the graph G is triconnected.

`Splitcomponents(G)`

The split components of the graph G. These are returned in the form of a sequence of subsets of the vertex-set of G. The graph may be disconnected. The second return value returns the cut vertices and the separation pairs.

`SeparationVertices(G)`

The cut vertices and/or the separation pairs of the graph G as a sequence of vertices or pairs of vertices of G. The graph may be disconnected. The second return value returns G's split components.

Example H149E12

```
> G := Graph< 11 |
> {1, 3}, {1, 4}, {1, 7}, {1, 11}, {1, 2},
> {2, 4}, {2, 3},
> {3, 4},
> {4, 7}, {4, 11}, {4, 7}, {4, 5},
> {5, 10}, {5, 9}, {5, 8}, {5, 6},
> {6, 8}, {6, 7},
> {7, 8},
> {9, 11}, {9, 10},
> {10, 11}: SparseRep := true >;
> TS, PS := Splitcomponents(G);
> TS;
[
  { 5, 6, 7, 8 },
  { 5, 9, 10, 11 },
  { 1, 4, 5, 7, 11 },
  { 1, 4 },
  { 1, 2, 3, 4 }
]
> PS;
[
  [ 5, 7 ],
  [ 5, 11 ],
```

```
  [ 1, 4 ]
]
```

Let `G1` be the subgraph induced by `TS[1]`. We will verify that it is not triconnected and we make it triconnected by adding the edge `[5, 7]`, since `[5, 7]` is a separation pair of `G` whose vertices belong to `G1`.

```
> G1 := sub< G | TS[1] >;
> IsTriconnected(G1);
false
> AddEdge(~G1, Index(VertexSet(G1)!VertexSet(G)!5),
>       Index(VertexSet(G1)!VertexSet(G)!7));
> IsTriconnected(G1);
true
```

149.13.4 Maximum Matching in Bipartite Graphs

The maximum matching algorithm is a flow-based algorithm. We have implemented two maximum flow algorithms, the Dinic algorithm, and a push-relabel algorithm. The latter is almost always the most efficient, it may be outperformed by Dinic only in the case of very sparse graphs. For a full discussion of these algorithms, the reader is referred to Section 151.4 in Chapter 150. Example H151E4 in this same chapter actually is an implementation in MAGMA code of the maximum matching algorithm.

`MaximumMatching(G)`

`Al` MONSTGELT *Default : "PushRelabel"*

A maximum matching in the bipartite graph G. The matching is returned as a sequence of edges of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

Example H149E13

We begin by creating a random bipartite graph. The matching algorithm is flow-based and is one of the algorithms which deals with graphs as adjacency lists. For this reason we might as well create the graph with a sparse representation (see Section 149.3).

```
> n := 5;
> m := 7;
> d := 3;
> G := EmptyGraph(n);
> H := EmptyGraph(m);
> G := G join H;
> for u in [1..n] do
>     t := Random(0, d);
>     for tt in [1..t] do
>         v := Random(n+1, n+m);
```

```
>         AddEdge(~G, u, v);
>     end for;
> end for;
> G := Graph< Order(G) | G : SparseRep := true >;
>
> IsBipartite(G);
true
> Bipartition(G);
[
  { 1, 12, 2, 3, 4, 5, 7 },
  { 11, 6, 8, 9, 10 }
]
> MaximumMatching(G);
[ {1, 11}, {4, 10}, {2, 9}, {5, 8}, {3, 6} ]
```

149.13.5 General Vertex and Edge Connectivity in Graphs and Digraphs

As for the maximum matching algorithm (Subsection 149.13.4) the vertex and edge connectivity algorithms are flow-based algorithms. We have implemented two flow algorithms, Dinic and a push relabel method. In general Dinic outperforms the push relabel method only in the case of very sparse graphs. This is particularly visible when computing the edge connectivity of a graph. For full details on those flow algorithms, refer to Section 151.4 in Chapter 150. For details on how vertex and edge connectivity are implemented, see [Eve79].

The general connectivity algorithms apply to both undirected and directed graphs.

`VertexSeparator(G)`

`Al` MonStgElt *Default :* "*PushRelabel*"

If G is an undirected graph, a *vertex separator* for G is a smallest set of vertices S such that for any u, $v \in V(G)$, every path connecting u and v passes through at least one vertex of S.

If G is a directed graph, a vertex separator for G is a smallest set of vertices S such that for any u, $v \in V(G)$, every directed path from u to v passes through at least one vertex of S.

`VertexSeparator` returns the vertex separator of G as a sequence of vertices of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`VertexConnectivity(G)`

`Al` MonStgElt *Default :* "*PushRelabel*"

Returns the vertex connectivity of the graph G, the size of a minimum vertex separator of G. Also returns a vertex separator for G as the second value. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`IsKVertexConnected(G, k)`

`Al` MonStgElt *Default :* "*PushRelabel*"

Returns `true` if the graph G's vertex connectivity is at least k, `false` otherwise. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`EdgeSeparator(G)`

`Al` MonStgElt *Default :* "*PushRelabel*"

If G is an undirected graph, an *edge separator* for G is a smallest set of edges T such that for any u, $v \in V(G)$, every path connecting u and v passes through at least one edge of T.

If G is a directed graph, an edge separator for G is a smallest set of edges T such that for any u, $v \in V(G)$, every directed path from u to v passes through at least one edge of T.

`EdgeSeparator` returns the edge separator of G as a sequence of edges of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`EdgeConnectivity(G)`

`Al` MonStgElt *Default :* "*PushRelabel*"

Returns the edge connectivity of the graph G, the size of a minimum edge separator of G. Also returns as the second value an edge separator for G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`IsKEdgeConnected(G, k)`

`Al` MonStgElt *Default :* "*PushRelabel*"

Returns `true` if the graph G's edge connectivity is at least k, `false` otherwise. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

Example H149E14

We compute the vertex and edge connectivity for a small undirected graph and we perform a few checks. The connectivity algorithm is one of the algorithms which deals with graphs as adjacency lists. For this reason we might as well create the graph with a sparse representation (see Section 149.3).

```
> G := Graph< 4 | {1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4}
> : SparseRep := true >;
> IsConnected(G);
true
>
> vc := VertexConnectivity(G);
> S := VertexSeparator(G);
```

```
> vc;
2
> S;
[ 3, 2 ]
>
> H := G;
> vs := [ Integers() | Index(x) : x in S ];
> RemoveVertices(~H, vs);
> IsConnected(H);
false
> for k in [0..vc] do
>     IsKVertexConnected(G, k);
> end for;
true
true
true
> for k in [vc+1..Order(G)-1] do
>     IsKVertexConnected(G, k);
> end for;
false
>
>
>
> ec := EdgeConnectivity(G);
> T :=  EdgeSeparator(G);
> ec;
2
> T;
[ {1, 2}, {1, 3} ]
>
> H := G;
> M := [ { Index(e[1]), Index(e[2]) }
> where e := SetToSequence(EndVertices(e)) : e in T ];
> RemoveEdges(~H, M);
> IsConnected(H);
false
> for k in [0..ec] do
>     IsKEdgeConnected(G, k);
> end for;
true
true
true
> for k in [ec+1..Order(G)-1] do
>     IsKEdgeConnected(G, k);
> end for;
false
```

149.14 Distances, Paths and Circuits in a Graph

The distance functions apply to graphs as well as to digraphs.

149.14.1 Distances, Paths and Circuits in a Possibly Weighted Graph

There are a few distance and path functions that take into account the fact that the edges of the graph under consideration may be weighted. If the graph is not weighted, then the distance between two vertices is taken to be the length of the shortest path between the vertices. For more details on distance and paths functions in weighted graphs, refer to Section 150.12 in Chapter 150.

`Reachable(u, v)`

Return `true` if and only if vertices u and v of a graph G are in the same component of G.

`Distance(u, v)`

Given two vertices u and v belonging to a graph G, return the length of a shortest path from u to v. If the graph is weighted, the function returns the total weight of the path from u to v. Results in an error if there is no path in G from u to v.

`Geodesic(u, v)`

Given vertices u and v belonging to the graph G, return a sequence of vertices $u = v_1, v_2, \ldots, v_n = v$ such that the sequence of edges $v_1v_2, v_2v_3, \ldots v_{n-1}v_n$ forms a shortest path joining u and v. Results in an error if there is no path in G from u to v.

149.14.2 Distances, Paths and Circuits in a Non-Weighted Graph

All the distance and path functions listed below do not take account of the fact that the graph edges may be weighted. *The distance between two vertices is understood in its usual meaning of being the length of the shortest path between the vertices.*

`Diameter(G)`

The length of the longest path in the graph G. If G is not connected, the value -1 is returned. To see how the automorphism group of G is computed see Subsection 149.22.3.

`DiameterPath(G)`

A sequence of vertices defining a longest path if the graph G is connected, or the empty sequence if G is not connected. To see how the automorphism group of G is computed see Subsection 149.22.3.

`Ball(u, n)`

Given a vertex u belonging to a graph G, and a non-negative integer n, return the set of vertices of G that are at distance n or less from u.

`Sphere(u, n)`

Given a vertex u belonging to the graph G, and a non-negative integer n, return the set of vertices of G that are at distance n from u.

`DistancePartition(u)`

Given a vertex u belonging to the graph G, return the partition $P_0 \cup P_1 \cup \ldots \cup P_d$ of the vertex-set of G, where P_i is the set of vertices lying at distance i from vertex u. The partition is returned as a sequence of sets. Any vertices not connected to u are treated as being at infinite distance from u and are returned as the last set of the partition.

`IsEquitable(G, P)`

Given a partition P of the vertex-set $V(G)$ of the graph G, return the value `true` if P is an equitable partition.

`EquitablePartition(P, G)`

Given a partition P of the vertex-set $V(G)$ of the graph G, return the coarsest partition of $V(G)$ that refines P and which is equitable.

`Girth(G)`

The girth of graph G, i.e. the length of a shortest cycle. To see how the automorphism group of G is computed see Subsection 149.22.3.

`GirthCycle(G)`

A cycle of shortest length in the graph G. To see how the automorphism group of G is computed see Subsection 149.22.3.

149.15 Maximum Flow, Minimum Cut, and Shortest Paths

Whenever the edges of a graph are assigned a capacity it is possible to compute the maximum flow and the minimum cut of the graph. Whenever the edges of a graph are assigned a weight it is possible to find shortest paths in the graph. If one wants to perform these computations on a graph where none of its edges have been assigned a capacity or a weight, then the edges are taken to have a capacity or weight of one, and the computations are carried out accordingly.

The flow-based and shortest-paths algorithms and the resulting functionalities are fully documented in Chapter 150, Sections 151.4 and 150.12 respectively.

149.16 Matrices and Vector Spaces Associated with a Graph or Digraph

`AdjacencyMatrix(G)`

Returns the adjacency matrix for the (p, q) graph G as an element of the matrix ring $M_p(Z)$.

`DistanceMatrix(G)`

Returns the distance matrix A for the (p, q) graph G as an element of the matrix ring $M_p(Z)$. The (i, j)-th entry of A gives the distance between vertices v_i and v_j of G.

`IncidenceMatrix(G)`

Returns the incidence matrix M for the (p, q) graph G as an element of the matrix bimodule $M^{p\times q}(Z)$.

If G is a graph, then entry (i, j) of M is 1 if the vertex v_i of G lies on the edge e_j of G. Otherwise entry (i, j) is zero.

If G is a digraph, entry (i, j) of M is 1 if vertex v_i is the initial vertex of the edge e_j, and -1 if v_i is the final vertex of the edge e_j. Otherwise entry (i, j) is zero. If e_j is a loop, then entry (i, j) may be either 1 or -1.

`IntersectionMatrix(G, P)`

Given an ordered equitable partition $P = P_1 \cup P_2 \cup \ldots \cup P_r$ of the vertex-set of the graph G, return the intersection matrix T for the partition. Thus, entry $T[i, j]$ is the number of vertices of the set P_j that are adjacent to a vertex of the set P_i.

149.17 Spanning Trees of a Graph or Digraph

`SpanningTree(G)`

Given a connected (undirected) graph G, construct a spanning tree for G rooted at an arbitrary vertex of G. The spanning tree is returned as a subgraph of G. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

`SpanningForest(G)`

Given a graph G, construct a spanning forest for G. The forest is returned as a subgraph of G. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

`BreadthFirstSearchTree(u)`

`BFSTree(u)`

Given a vertex u belonging to the graph G, return a breadth-first search for G rooted at the vertex u. The tree is returned as a subgraph of G. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph. Note that G may be disconnected.

`DepthFirstSearchTree(u)`

`DFSTree(u)`

Given a vertex u belonging to the graph G, return a depth-first search tree T for G rooted at the vertex u. The tree T is returned as a subgraph of G. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph. Note that G may be disconnected.

The fourth return argument returns, for each vertex u of G, the tree order of u, that is, the order in which the vertex u has been visited while performing the depth-first search. If T does not span G then the vertices of G not in T are given tree order from `Order(T) + 1` to `Order(G)`.

149.18 Directed Trees

In the following functions, the graph is assumed to be a directed tree. This means it is a tree containing a root vertex and all edges are directed away from that vertex.

`IsRootedTree(G)`

Returns `true` exactly when the directed graph G is a tree having a vertex v such that all edges are directed away from v. In this case, the root vertex v is returned as a second value.

`Root(G)`

The root vertex of a rooted tree.

`IsRoot(v)`

Returns `true` if and only if the graph containing the vertex v is directed as a rooted tree with v as root.

`RootSide(v)`

When the graph containing the vertex v is directed as a rooted tree, this returns the unique neighbouring vertex to v which is closer to the root vertex. If v is the root vertex, it is returned itself.

`VertexPath(u,v)`

A sequence of vertices comprising a path in a directed graph from the vertex u to the vertex v. The path does not necessarily respect the edge directions. Indeed it will first trace back to a common ancestor of u and v and then follow edge directions to v.

`BranchVertexPath(u,v)`

The sequence of vertices on the vertex path from the vertex u to the vertex v having valency at least 3.

149.19 Colourings

The functions given here are not applicable to digraphs.

`ChromaticNumber(G)`

The chromatic number of the graph G, that is, the minimum number of colours required to colour the vertices of G such that no two adjacent vertices share the same colour.

`OptimalVertexColouring(G)`

An optimal vertex colouring of the graph G as a sequence of sets of vertices of G. Each set in the sequence contains the vertices which are given the same colour. Some optimal vertex colouring is returned, others may exist.

`ChromaticIndex(G)`

The chromatic index of the graph G, that is, the minimum number of colours required to colour the edges of G such that no two adjacent edges share the same colour.

`OptimalEdgeColouring(G)`

An optimal edge colouring of the graph G as a sequence of sets of edges of G. Each set in the sequence contains the edges which are given the same colour. Some optimal edge colouring is returned, others may exist.

`ChromaticPolynomial(G)`

The chromatic polynomial of the graph G. For a given natural number x, the chromatic polynomial $p_G(x)$ gives the number of colourings of G with colours $1, 2, \ldots, x$.

Example H149E15

We illustrate the use of these functions with the following graph:

```
> G:=Graph< 5 | [{2,5}, {1,3,5}, {2,4,5}, {3,5}, {1,2,3,4} ]>;
> ChromaticNumber(G);
3
> OptimalVertexColouring(G);
[
  { 1, 3 },
  { 2, 4 },
  { 5 }
]
> ChromaticIndex(G);
4
> OptimalEdgeColouring(G);
[
  { {1, 2}, {3, 5} },
  { {2, 3}, {1, 5} },
  { {3, 4}, {2, 5} },
  { {4, 5} }
]
> ChromaticPolynomial(G);
$.1^5 - 7*$.1^4 + 18*$.1^3 - 20*$.1^2 + 8*$.1
```

149.20 Cliques, Independent Sets

The functions given here are not applicable to digraphs.

A *clique* of a graph G is a complete subgraph of G. A *maximal clique* is a clique which is not contained in any other clique. A *maximum clique* is a maximal clique of largest size.

An *independent set* of G is an empty subgraph of G. A *maximal* (*maximum*) *independent set* is defined is the same way as a maximal (maximum) clique. Note that an independent set of size k in the graph G is a clique of size k in the complement graph of G.

The clique and independent set functions presented below are implemented using one of two algorithms, called here *BranchAndBound* and *Dynamic*. The algorithms *BranchAndBound* and *Dynamic* are briefly described here.

BranchAndBound : The algorithm is an implementation of the branch and bound algorithm by Bron and Kerbosch [BK73].

The algorithm is designed to find maximal cliques and it has been adapted here to also find cliques of specific size which may not be maximal. It attempts to build a maximal clique by trying to expend the current maximal clique. Some heuristics are built into the algorithm which enable to prune the search tree.

Dynamic : The algorithm finds a clique of size exactly k, not necessarily maximal, in the graph G by using recursion and dynamic programming. If a clique of size k exists in G,

then, for a given vertex v of G, there is either a clique of size $k-1$ in the subgraph induced by the neighbours of v, or there is a clique of size k in the graph $G-v$. This is the recursive step. The *Dynamic* algorithm applies a different strategy (sorting of vertices and selection of vertices to consider) according to the order and density of the subgraph considered at the current level of recursion. This is achieved by dynamic programming, hence the name. This algorithm is due to Wendy Myrvold [WM].

Comment: When comparing both algorithms in the situation where the problem is to find a maximum clique one observes that in general *BranchAndBound* does better. However *Dynamic* outperforms *BranchAndBound* when the graphs under consideration are large (more then 400 vertices) random graphs with high density (larger than 0.5%). So far, it can only be said that the comparative behaviour of both algorithms is highly dependent on the structure of the graphs.

`HasClique(G, k)`

Returns `true` if and only if the graph G has a maximal clique of size k. If `true`, returns such a clique as the second argument.

`HasClique(G, k, m` : *parameters*`)`

`Al` MonStgElt *Default :* "*BranchAndBound*"

If m is `true`, the function is `true` if and only if the graph G has a maximal clique of size k. If m is `false`, the function is `true` if and only if the graph G has a — not necessarily maximal — clique of size k. If the function is `true`, an appropriate clique is returned as the second argument. When m is `false`, the parameter `Al` enables the user to select the algorithm which is to be used. When m is `true`, the parameter `Al` is ignored.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the description given at the beginning of this section.

The default is `"BranchAndBound"`.

`HasClique(G, k, m, f` : *parameters*`)`

`Al` MonStgElt *Default :* "*BranchAndBound*"

If m is `true` and $f = 0$, the function is `true` if and only if the graph G has a maximal clique of size k. If m is `true` and $f = 1$, the function is `true` if and only if the graph G has a maximal clique of size larger than or equal to k. If m is `true` and $f = -1$, the function is `true` if and only if the graph G has a maximal clique of size smaller than or equal to k. If m is `false`, the function is `true` if and only if the graph G has a — not necessarily maximal — clique of size k. If the function is `true`, an appropriate clique is returned as the second argument. When m is `false`, the parameter `Al` enables the user to select the algorithm which is to be used. When m is `true` the parameter `Al` is ignored, and when m is `false`, the flag f is ignored.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the description given at the beginning of this section.

The default is "BranchAndBound".

`MaximumClique(G : parameters)`

`Al` MonStgElt *Default :* "*BranchAndBound*"

Finds a maximum clique in the graph G. The clique is returned as a set of vertices.

The parameter `Al` enables the user to select the algorithm which is to be used.

`Al := "BranchAndBound"`: See the description at the beginning of this section.

`Al := "Dynamic"`: Two steps are required here.

Step 1 Finding a lowerbound on the size of a maximum clique. This is achieved by using the *dsatur* colouring (*dsatur* stands for saturation degree) due to Brélaz [Bre79]. The *dsatur* colouring gives a reasonably good approximation to the size of a maximum clique, usually with a penalty of 2 to 3.

Step 2 Assume that the lowerbound found in Step 1 if l. Then a maximum clique is found by finding the largest possible clique of size $k \geq l$ using the Dynamic algorithm.

The default is "BranchAndBound".

`CliqueNumber(G : parameters)`

`Al` MonStgElt *Default :* "*BranchAndBound*"

Finds the size of a maximum clique in the graph G. The parameter `Al` enables the user to select the algorithm which is to be used. `Al := "BranchAndBound"`: See the description given at the beginning of this section.

`Al := "Dynamic"`: See the function `MaximumClique`.

The default is "BranchAndBound".

`AllCliques(G : parameters)`

`Limit` RngIntElt *Default :* 0

Returns all maximal cliques of the graph G as a sequence of sets of vertices. If `Limit` is set to a positive integer, returns `Limit` maximal cliques of G.

`AllCliques(G, k : parameters)`

`Limit` RngIntElt *Default :* 0

Returns all maximal cliques of size k of the graph G as a sequence of sets of vertices. If `Limit` is set to a positive integer, returns `Limit` maximal cliques of size k of G.

`AllCliques(G, k, m : parameters)`

`Limit`	RngIntElt	*Default :* 0
`Al`	MonStgElt	*Default :* "*BranchAndBound*"

If m is `true`, returns all maximal cliques of size k in the graph G. If m is `false`, returns all — not necessarily maximal — cliques of size k. When m is `false`, the parameter `Al` enables the user to select the algorithm which is to be used. When m is `true`, the parameter `Al` is ignored.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the description given at the beginning of this section. The default is `"BranchAndBound"`.

Except in the case where m is `false` and `Al` is `"Dynamic"`, the parameter `Limit`, if set to a positive integer, limits the number of cliques returned.

Maximal independent sets or independent sets of a given size k in a graph G can be easily found by finding maximal cliques or cliques of size k in the complement of G. Only two functions which are concerned with independent sets are provided: one finds a maximum independent set and the other returns the independence number of a graph.

`MaximumIndependentSet(G: parameters)`

`Al`	MonStgElt	*Default :* "*BranchAndBound*"

Finds a maximum independent set in the graph G. The maximum independent set is returned as a set of vertices.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the function `MaximumClique`.

The default is `"BranchAndBound"`.

`IndependenceNumber(G: parameters)`

`Al`	MonStgElt	*Default :* "*BranchAndBound*"

Finds the size of a maximum independent set in the graph G.

The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "BranchAndBound"` or `Al := "Dynamic"`. See the function `CliqueNumber`.

The default is `"BranchAndBound"`.

Example H149E16

We illustrate the use of the clique functions with the following graph:

```
> G := Graph< 9 | [ {4,5,6,7,8,9}, {4,5,6,7,8,9}, {4,5,6,7,8,9},
>                   {1,2,3,7,8,9}, {1,2,3,7,8,9}, {1,2,3,7,8,9},
>                   {1,2,3,4,5,6}, {1,2,3,4,5,6}, {1,2,3,4,5,6} ]>;
> HasClique(G, 2);
false
> HasClique(G, 2, true);
false
> HasClique(G, 2, false);
```

```
true { 1, 4 }
> HasClique(G, 2, true: Al := "Dynamic");
false
> HasClique(G, 2, false: Al := "Dynamic");
true { 1, 9 }
> HasClique(G, 2, true, 1);
true { 1, 4, 7 }
> MaximumClique(G);
{ 1, 4, 7 }
> AC := AllCliques(G);
> #AC;
27
> AC3 := AllCliques(G,3);
> #AC3;
27
> AC eq AC3;
true
> AllCliques(G, 2, true);
[]
> AllCliques(G, 2, false);
[
  { 1, 4 },
  { 1, 5 },
  { 1, 6 },
  { 1, 7 },
  { 1, 8 },
  { 1, 9 },
  { 2, 4 },
  { 2, 5 },
  { 2, 6 },
  { 2, 7 },
  { 2, 8 },
  { 2, 9 },
  { 3, 4 },
  { 3, 5 },
  { 3, 6 },
  { 3, 7 },
  { 3, 8 },
  { 3, 9 },
  { 4, 7 },
  { 4, 8 },
  { 4, 9 },
  { 5, 7 },
  { 5, 8 },
  { 5, 9 },
  { 6, 7 },
  { 6, 8 },
  { 6, 9 }
```

]

149.21 Planar Graphs

A linear-time algorithm due to Boyer and Myrvold [BM01] has been implemented to test if a graph is planar, and to identify a Kuratowski subgraph if the graph is non-planar. This algorithm requires that the graph has a sparse representation.

The interest of this new algorithm is that it is very easy to implement when compared to previous linear-time planarity testers. In addition, the algorithm also isolates a Kuratowski subgraph, if any, in linear-time.

The concept underlying the tester is deceptively simple: the idea is to perform a depth first search of the graph and then to try to embed (by post-order traversal of the tree) the back edges from a vertex v to its descendants in such a way that they do not cross and such that vertices with connections to ancestors of v remain along the external face.

To decide the order in which to embed back edges from v to its descendants, the external faces of components rooted by children of v are explored top-down, avoiding paths containing vertices with ancestor connections. To decide whether a vertex u has ancestor connections and must therefore be kept on the external face, one only needs the least ancestor directly adjacent to u and the lowpoint information for the children of u that are separable from the parent of u in the partial embedding.

Each back edge $[w, v]$ is added as soon as the descendant endpoint w is encountered. If w is not in the same biconnected component as v, then the biconnected components encountered during the traversal are merged at their respective cut vertices since the new edge $[w, v]$ biconnects them. The merge is performed such that vertices with ancestor connections remain on the external face of the newly formed biconnected component.

Traversal is terminated when all back edges from v to its descendants are added or when the external face traversal is prevented from reaching the descendant endpoint of a back edge. This occurs when both external face paths extending from the least numbered vertex in a biconnected component contain vertices with ancestor connections that block further traversal to a vertex that is the descendant endpoint w of a back edge $[w, v]$. In this case, a Kuratowski subgraph is isolated based primarily on ancestor connections, external face paths and depth first search tree paths.

MAGMA is deeply grateful for the valuable help and collaboration from the authors of [BM01], in particular John Boyer.

`IsPlanar(G)`

Tests whether the (undirected) graph G is planar. The graph may be disconnected. If the graph is non-planar then a Kuratowski subgraph of G is returned: That is, a subgraph of G homeomorphic to K_5 or $K_{3,3}$. The support and vertex/edge decorations of G are *not* retained in this (structural) subgraph.

`Obstruction(G)`

Returns a Kuratoswki obstruction if the graph is non-planar, or the empty graph if the graph is planar. The Kuratoswki is a (structural) subgraph of G; the support and vertex/edge decorations of G are not transferred to it.

`IsHomeomorphic(G : parameters)`

`Graph` MonStgElt *Default :*

Tests if a graph is homeomorphic to either K_5 or $K_{3,3}$. The parameter `Graph` must be set to either "K5" or "K33"; it has no default setting.

`Faces(G)`

Returns the faces of the planar graph G as sequences of the edges bordering the faces of G. If G is disconnected, then the face defined by an isolated vertex v is given as $[v]$.

`Face(u, v)`

Returns the face of the planar graph G bordered by the directed edge $[u, v]$ as an ordered list of edges of G.

Note that a directed edge and an orientation determine a face uniquely: We can assume without loss of generality that the plane is given a clockwise orientation. Then given a directed edge $e = [u_1, v_1]$, the face defined by e is the ordered set of edges $[u_1, v_1], [u_2, v_2], \ldots, [u_m, v_m]$ such that $v_i = u_{i+1}$ for all i, $1 \leq i < m$, $v_m = u_1$, and for each $v_i = u_{i+1}$, the neighbours of v_i, u_i and v_{i+1}, are *consecutive* vertices in v_i's adjacency list whose order is *anti-clockwise*.

`Face(e)`

Let e be the edge u, v of the planar graph G (recall that G is undirected). Then `Face`(u, v) returns the face bordered by the directed edge $[u, v]$ as a sequence of edges of G.

`NFaces(G)`

`NumberOfFaces(G)`

Returns the number of faces of the planar graph G. In the case of a disconnected graph, an isolated vertex counts for one face.

`Embedding(G)`

Returns the graph G's planar embedding as a sequence S where $S[i]$ is a sequence of edges incident from vertex i.

`Embedding(v)`

Returns the ordered list of edges (in clockwise order say) incident from vertex v.

`PlanarDual(G)`

Constructs the dual G' of a planar graph G. The numbering of the vertices of G' corresponds to the order of the faces as returned by `Faces(G)`.

Example H149E17

We start with a small disconnected planar graph G: Notice how one of the faces of G is defined by the isolated vertex 4.

```
> G := Graph< 5 | [ {2, 3, 5}, {1, 5}, {1}, {}, {1, 2} ] >;
> G;
Graph
Vertex  Neighbours
1       2 3 5 ;
2       1 5 ;
3       1 ;
4       ;
5       1 2 ;
> IsConnected(G);
false
> IsPlanar(G);
true
> Faces(G);
[
  [ {1, 2}, {2, 5}, {5, 1} ],
  [ {1, 5}, {5, 2}, {2, 1}, {1, 3}, {3, 1} ],
  [ 4 ]
]
> Embedding(G);
[
  [ {1, 2}, {1, 5}, {1, 3} ],
  [ {2, 5}, {2, 1} ],
  [ {3, 1} ],
  [],
  [ {5, 1}, {5, 2} ]
]
> Embedding(VertexSet(G)!1);
[ {1, 2}, {1, 5}, {1, 3} ]
```

Now let's turn to a simple non-planar graph whose obstruction is $K_{3,3}$:

```
> G := Graph< 6 | [ {3, 4, 5}, {3, 4, 5, 6}, {1, 2, 5, 6},
> {1, 2, 5, 6}, {1, 2, 3, 4, 6}, {2, 3, 4, 5} ] >;
> G;
Graph
Vertex  Neighbours
1       3 4 5 ;
2       3 4 5 6 ;
3       1 2 5 6 ;
4       1 2 5 6 ;
5       1 2 3 4 6 ;
6       2 3 4 5 ;
> IsPlanar(G);
false
```

```
> b, O := IsPlanar(G);
> IsSubgraph(G, O);
true
> O;
Graph
Vertex  Neighbours
1       4 5 3 ;
2       3 5 4 ;
3       1 6 2 ;
4       2 6 1 ;
5       6 1 2 ;
6       5 4 3 ;
> IsHomeomorphic(O : Graph := "K33");
true
```

149.22 Automorphism Group of a Graph or Digraph

The automorphism group functionality is an interface to B. McKay's `nauty` V 2.2 programme. For a paper describing `nauty` see [McK81]. There also exists a user's manual [McK] describing `nauty`'s essential features.

149.22.1 The Automorphism Group Function

`AutomorphismGroup(G : parameters)`

Compute the automorphism group of the graph G. This returns the group A, the vertex and edge G-sets (in that order), the power structure $\mathcal{A}$ of all automorphisms of G, and the transfer map from A to $\mathcal{A}$. Note that G may be directed or undirected. For small graphs (i.e. having less than 500 vertices) the canonically labelled graph is also returned.

The automorphism group computation may be driven by several user parameters. There are four principal parameters: `Canonical`, `Stabilizer`, `Invariant` and `Print`. Some of these parameters have associated parameters.

`Canonical` BoolElt *Default :* `false`

If the parameter `Canonical` is set to `true`, then the canonically labelled graph will be returned. If the graph has fewer than 500 vertices then the default value for this parameter is `true`, while if the graph has 500 or more vertices, the default value is `false`. It is important to note that the canonically labelled graph is *dependent* upon all the parameters used to compute the automorphism group.

`Stabilizer` [{ GrphVert }] *Default :*

If the parameter `Stabilizer` is assigned a partition P of the vertex-set of G as its value, then the subgroup of the automorphism group of G that preserves the

partition P will be computed. The partition given as `Stabilizer` need not be a partition of the full vertex-set of G. Any vertices not covered by the sets in `Stabilizer` are assumed to belong to an extra partition cell.

`Invariant` MonStgElt *Default :*

Use the named invariant to assist the auto group computation. The invariant is specified by a string which is the name of a C function computing an invariant, as on pages 16–17 of the tt nauty manual [McK]. The default value for `Invariant` is `"Null"` when G is an undirected graph, or `"adjacencies"` when G is a digraph. For a more detailed discussion on invariants see Subsection 149.22.2.

There are three optional associated parameters to `Invariant`:

`Minlevel` RngIntElt *Default :* 0

`Maxlevel` RngIntElt *Default :*

`Arg` RngIntElt *Default :*

An expression, depending upon the type of invariant. When G is undirected these three parameters take the values 0, 1, and 0 respectively. When G is a digraph they take the values 0, 2, and 0 respectively. When `Invariant` is one of "indsets", "cliques", "cellcliq", or "cellind", the default value for `Arg` is 3. Again, see Subsection 149.22.2 for more information.

`Print` RngIntElt *Default :* 0

The integer valued parameter `Print` is used to control informative printing according to the following table:

`Print := 0` : No output (default).
`Print := 1` : Statistics are printed.
`Print := 2` : Summary upon completion of each level.
`Print := 3` : Print the automorphisms as they are discovered.

`IgnoreLabels` BoolElt *Default :* `false`

By default, the automorphism group computation respects vertex labels (colours). That is, elements of the group will only take a vertex to an identically labelled vertex. The Boolean value `IgnoreLabels` will treat all vertices as identically labelled for the purposes of this computation.

149.22.2 `nauty` Invariants

Invariants can be used to supplement the built–in partition refinement code in `nauty`, to assist in the processing of difficult graphs. Invariants are used with three parameters:

`Minlevel`: the minimum depth in the search tree at which an invariant is to be applied (default 0)

`Maxlevel`: the maximum depth in the search tree at which an invariant is to be applied (default 1 for undirected graphs and 2 for digraphs)

`Arg`: an integer argument which has a different meaning (or no meaning) for each invariant

The root of the tree, corresponding to the partition provided by the user, has level 1. Note that the invariants use an existing partition and attempt to refine it. Thus, it can be that an invariant fails to refine the partition at the root of the search tree but succeeds further down the tree. In particular, all invariants necessarily fail at the root of the tree if the graph is vertex-transitive.

The invariants are :

`"default"`

Default — No invariant is used.

`"twopaths"`

Each vertex is classified according to the vertices reachable from along a path of length 2. This is a cheap invariant sometimes useful for regular graphs. `Arg` is not used.

`"adjtriang"`

This counts the number of common neighbours between pairs of adjacent (if `Arg` = 0) or non–adjacent (if `Arg` = 1) vertices. This is a fairly cheap invariant which can often work for strongly–regular graphs.

`"triples"`

This considers in detail the neighbourhoods of each triple of vertices. It often works for strongly–regular graphs that `"adjtriang"` fails on, but is more expensive. `Arg` is not used.

`"quadruples"`

Similar to triples but using quadruples of vertices. Much more expensive but can work on graphs with highly regular structure that `"triples"` fails on. `Arg` is not used.

`"celltrips"`

Like `"triples"`, but only triples inside one cell are used. One to try for the bipartite incidence graphs of structures like block designs (but won't work for projective planes). `Arg` is not used.

`"cellquads"`

Like `"quadruples"`, but only quadruples inside one cell are used. Originally designed for some exceptional graphs derived from Hadamard matrices (where applying it at level 2 is best). `Arg` is not used.

`"cellquins"`

Considers 5–tuples of vertices inside a cell. Very expensive. We don't know of a good application. `Arg` is not used.

`"cellfano"`

`"cellfano2"`

These are intended for the bipartite graphs of projective planes and similar highly regular structures. `"cellfano2"` is cheaper, so try it first. `Arg` is not used. The method is similar to counting Fano subplanes.

`"distances"`

This uses the distance matrix of the graph. It is good at partitioning general regular graphs (but not strongly regular graphs). Moderately cheap. `Arg` is unused.

`"indsets"`

This uses the independent sets of size `Arg` (default 3). Can be successful for strongly regular graphs (for `Arg` ≥ 4) or regular bipartite graphs.

`"cliques"`

This uses the cliques of size `Arg` (default 3). Can be successful for strongly regular graphs (for `Arg` ≥ 4).

`"cellind"`

This uses the independent sets of size `Arg` (default 3) that lie entirely within one cell of the partition. Try applying at level 2 for difficult vertex–transitive graphs.

`"cellcliq"`

This uses the cliques of size `Arg` (default 3) that lie entirely within one cell of the partition. Try applying at level 2 for difficult vertex–transitive graphs.

`"adjacencies"`

This is a simple invariant that corrects for a deficiency in the built–in partition refinement procedure for directed graphs. It is the default in MAGMA for directed graphs. Apply at a few levels, say levels $1 - 5$. `Arg` is unused.

These invariants are further described on pages 16–17 of the `nauty` manual [McK]. Also, we provide a facility enabling users to test if a specific `Invariant` achieves a partition refinement for a graph G.

`IsPartitionRefined(G:` *parameters*`)`

Returns `true` if and only the invariant in `Invariant` could refine the graph G's vertex-set partition. If no invariant is set in `Invariant` then the refinement procedure which is tested is taken to be the default one.

Four parameters can be passed:

`Stabilizer`	[{ VERT }]	*Default :*
`Invariant`	MONSTGELT	*Default :*
`Arg`	RNGINTELT	*Default :*
`IgnoreLabels`	BOOLELT	*Default :* `false`

The parameters have the same meaning as for `AutomorphismGroup`.

149.22.3 Graph Colouring and Automorphism Group

For any calls to `nauty` via MAGMA functions the graph colouring of a graph G (as set by `AssignLabels` or a similar function) is taken as an *intrinsic* property of G. Consequently, the default automorphism group computed by `nauty` is the group for the coloured graph G. In particular this implies that, when G is coloured, this default automorphism group is *not* G's full automorphism group.

There are several MAGMA functions which make use of the automorphism group of a graph, and therefore rely on a call to `nauty`. We list them here:

1 `Diameter`, `DiameterPath`, `IsDistanceRegular`, `GirthCycle`, `Girth`

2 `IsDistanceTransitive`, `IsTransitive`, `IsPrimitive`, `IsSymmetric`, `IsIsomorphic`, `IsEdgeTransitive`, `EdgeGroup`, `OrbitsPartition`, `CanonicalGraph`

The functions in Group 1 make use of the automorphism group of the graph only as a means to improve efficiency. If some group has already been computed for the graph under consideration then this group will be used. Otherwise the graph's default group is computed.

The functions in Group 2 all use the graph's default group. If a group A of the graph G has already been computed but is not G's default group (for example if a stabilizer of G was given when computing A) then the graph's default group will be computed. Consequently, as an example, if G is coloured, `IsTransitive(`G`)` will (always) return `false`.

For all these functions it is possible to drive the group computation by setting an appropriate `Invariant` parameter. This is particularly useful when it is known that some invariant will speed up the group computation. This is achieved by first computing the group using `AutomorphismGroup` while setting `Invariant`. This parameter is then remembered for any further (re)computation of the default automorphism group, unless reset by a subsequent call to `AutomorphismGroup`. For convenience, the function `IsIsomorphic` accepts a parameter suite very similar to that of the `AutomorphismGroup` function: It is particularly useful when stabilizers need to be set.

149.22.4 Variants of Automorphism Group

`CanonicalGraph(G)`

Given a graph G, return the canonically labelled graph isomorphic to G. To see which automorphism group is computed see Subsection 149.22.3. Users should be aware that the canonical graph is *dependent* upon the invariant used when computing the automorphism group. Thus, the computation of the automorphism group of G using different invariants will produce *different* canonical labellings.

`EdgeGroup(G)`

Returns the automorphism group of the graph G in its action on the edges of G, and the G-set of edges of G. To see which automorphism group is computed see Subsection 149.22.3.

`IsIsomorphic(G, H :` *parameters*`)`

This function returns `true` if the graphs G and H are isomorphic. If G and H are isomorphic, a second value is returned, namely a mapping between the vertices of G and the vertices of H giving the isomorphism.

The parameters accepted by `IsIsomorphic` are almost the same as for `AutomorphismGroup`. The exceptions are:

`Stabilizer` [{ VERT }] *Default :*

This sets the stabilizer set for *both* graphs G and H. If one wishes to set two distinct stabilizers then one would also use:

`Stabilizer2` [{ VERT }] *Default :*

This sets the stabilizer set for the second graph H only.

`IgnoreLabels` BOOLELT *Default :* `false`

This indicates whether or not to ignore the graph labelling for *both* graphs G and H.

Graph isomorphism is understood as mapping colours to colours and stabilizers to stabilizers. Consequently comparing two graphs for isomorphism will always return `false`if one graph is coloured while the other one is not. Similarly two graphs can't be isomorphic if each is coloured using a different set of colours. Moreover, it is a `nauty` requirement that in order to achieve mapping of stabilizers they must be compatible: that is, they *must* have same-sized cells in the same order.

In general, to see which and how the graphs' automorphism groups are computed see Subsection 149.22.3.

Example H149E18

We provide a very simple example of the use of `AutomorphismGroup` which shows how the automorphism group and the corresponding canonical graph differ depending on the labelling of the graph and of the stabilizer given, if any.

```
> G := Graph<5 | { {1,2}, {2,3}, {3,4}, {4,5}, {5,1} }>;
>
> AssignLabels(VertexSet(G), ["a", "b", "a", "b", "b"]);
> A, _, _, _, _, C1 := AutomorphismGroup(G);
> A;
Permutation group A acting on a set of cardinality 5
Order = 2
  (1, 3)(4, 5)
> C1;
Graph
Vertex  Neighbours
1       3 5 ;
2       4 5 ;
3       1 4 ;
4       2 3 ;
5       1 2 ;
> B, _, _, _, _, C2 := AutomorphismGroup(G : IgnoreLabels := true);
> B;
Permutation group B acting on a set of cardinality 5
Order = 10 = 2 * 5
  (2, 5)(3, 4)
  (1, 2)(3, 5)
> C2;
```

```
Graph
Vertex  Neighbours
1       2 3 ;
2       1 4 ;
3       1 5 ;
4       2 5 ;
5       3 4 ;
> C, _, _, _, _, C3 := AutomorphismGroup(G : Stabilizer :=
> [ { VertexSet(G) | 1, 2 } ]);
> C;
Permutation group C acting on a set of cardinality 5
> #C;
1
> C3;
Graph
Vertex  Neighbours
1       3 4 ;
2       3 5 ;
3       1 2 ;
4       1 5 ;
5       2 4 ;
```

We now check that the graph returned by `CanonicalGraph` is identical to C_1:

```
> C4 := CanonicalGraph(G);
> C4 eq C1;
true
```

Example H149E19

These two examples should help clarify how `IsIsomorphic` behaves for graphs for which a colouring is defined. The first example shows that two graphs cannot be isomorphic if one is coloured while the other is not. This example also demonstrates the use of stabilizers.

```
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["a", "a", "b", "b", "b"]);
> G2 := CompleteGraph(5);
> IsIsomorphic(G1, G2);
false
> IsIsomorphic(G1, G2 : IgnoreLabels := true);
true
>
> V1 := Vertices(G1);
> V2 := Vertices(G2);
> S1 := { V1 | 1, 2, 3};
> S2 := { V2 | 3, 4, 5};
>
> IsIsomorphic(G1, G2 : Stabilizer := [S1],
>       Stabilizer2 := [S2]);
```

```
false
> IsIsomorphic(G1, G2 : Stabilizer := [S1],
>       Stabilizer2 := [S2], IgnoreLabels := true);
true
> IsIsomorphic(G1, G2 : Stabilizer := [S1],
> IgnoreLabels := true);
true
>
> SS1 := [ { V1 | 1}, {V1 | 2, 3} ];
> SS2 := [ { V2 | 3, 4}, { V2 | 1} ];
> IsIsomorphic(G1, G2 : Stabilizer := SS1, Stabilizer2 := SS2,
> IgnoreLabels := true);
false
>
> SS1 := [ {V1 | 2, 3}, { V1 | 1} ];
> IsIsomorphic(G1, G2 : Stabilizer := SS1, Stabilizer2 := SS2,
> IgnoreLabels := true);
true
```

The second example shows that two graphs are isomorphic if and only if colours map to the *same* colours.

```
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["b", "b", "b", "a", "a"]);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G2), ["a", "a", "b", "b", "b"]);
> IsIsomorphic(G1, G2);
true
>
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["a", "b", "b", "c", "c"]);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G2), ["a", "c", "b", "b", "c"]);
> IsIsomorphic(G1, G2);
true
>
> G1 := CompleteGraph(5);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["a", "a", "b", "b", "b"]);
> AssignLabels(VertexSet(G2), ["b", "b", "a", "a", "a"]);
> IsIsomorphic(G1, G2);
false
> IsIsomorphic(G1, G2 : IgnoreLabels := true);
true
>
> G1 := CompleteGraph(5);
> AssignLabels(VertexSet(G1), ["b", "b", "b", "a", "a"]);
> G2 := CompleteGraph(5);
> AssignLabels(VertexSet(G2), ["b", "b", "c", "c", "c"]);
```

```
> IsIsomorphic(G1, G2);
false
> IsIsomorphic(G1, G2 : IgnoreLabels := true);
true
```

149.22.5 Action of Automorphisms

The automorphism group A of a graph G is given in its action on the standard support and it does not act directly on G. The action of A on G is obtained using the G-set mechanism. The two basic G-sets associated with the graph correspond to the action of A on the set of vertices V and the set of edges V of G. These two G-sets are given as return values of the function `AutomorphismGroup` or may be constructed directly. Additional G-sets associated with a graph may be built using the G-set constructors. Given a G-set Y for A, the action of A on Y may be studied using the permutation group functions that allow a G-set as an argument In this section, only a few of the available functions are described: see the section on G-sets for a complete list.

`Image(a, Y, y)`

Let a be an element of the automorphism group A for the graph G and let Y be a G-set for A. Given an element y belonging either to Y or to a G-set derived from Y, find the image of y under a.

`Orbit(A, Y, y)`

Let A be a subgroup of the automorphism group for the graph G and let Y be a G-set for A. Given an element y belonging either to Y or to a G-set derived from Y, construct the orbit of y under A.

`Orbits(A, Y)`

Let A be a subgroup of the automorphism group for the graph G and let Y be a G-set for G. This function constructs the orbits of the action of A on Y.

`Stabilizer(A, Y, y)`

Let A be a subgroup of the automorphism group for the graph G and let Y be a G-set for A. Given an element y belonging either to Y or to a G-set derived from Y, construct the stabilizer of y in A.

`Action(A, Y)`

Given a subgroup A of the automorphism group of the graph G, and a G-set Y for A, construct the homomorphism $\phi : A \to L$, where the permutation group L gives the action of A on the set Y. The function returns:

(a) The natural homomorphism $\phi : A \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of A).

`ActionImage(A, Y)`

Given a subgroup A of the automorphism group of the graph structure G, and a G-set Y for A, construct the permutation group L giving the action of A on the set Y.

`ActionKernel(A, Y)`

Given a subgroup A of the automorphism group of the graph G, and a G-set Y for A, construct the kernel of the action of A on the set Y.

Example H149E20

We construct the Clebsch graph (a strongly regular graph) and investigate the action of its automorphism group.

```
> S := { 1 .. 5 };
> V := &join[ Subsets({1..5}, 2*k) : k in [0..#S div 2]];
> E := { {u,v} : u,v in V | #(u sdiff v) eq 4 };
> G, V, E := StandardGraph( Graph< V | E >);
> G;
Graph
Vertex  Neighbours
1       4 6 8 10 12 ;
2       5 6 9 12 14 ;
3       9 10 12 13 16 ;
4       1 7 9 14 16 ;
5       2 7 8 10 16 ;
6       1 2 13 15 16 ;
7       4 5 12 13 15 ;
8       1 5 9 11 13 ;
9       2 3 4 8 15 ;
10      1 3 5 14 15 ;
11      8 12 14 15 16 ;
12      1 2 3 7 11 ;
13      3 6 7 8 14 ;
14      2 4 10 11 13 ;
15      6 7 9 10 11 ;
16      3 4 5 6 11 ;
> A, AV, AE := AutomorphismGroup(G);
> A;
Permutation group aut acting on a set of cardinality 16
Order = 1920 = 2^7 * 3 * 5
    (2, 15)(5, 11)(7, 14)(10, 12)
    (3, 11)(8, 10)(9, 14)(13, 15)
    (2, 11)(5, 15)(6, 8)(9, 16)
    (2, 7)(4, 6)(9, 13)(14, 15)
    (1, 2, 13, 15)(3, 11, 4, 5)(7, 10, 12, 14)(8, 9)
> CompositionFactors(A);
    G
```

```
    |  Cyclic(2)
    *
    |  Alternating(5)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
> IsPrimitive(A);
true
```

From the composition factors we guess that the group is $Sym(5)$ extended by the elementary abelian group of order 16. Let us verify this and relate it to the graph. We begin by trying to get at the group of order 16. We ask for the Fitting subgroup.

```
> F := FittingSubgroup(A);
> F;
Permutation group F acting on a set of cardinality 16
Order = 16 = 2^4
    (1, 13)(2, 11)(3, 4)(5, 15)(6, 8)(7, 10)(9, 16)(12, 14)
    (1, 10)(2, 9)(3, 12)(4, 14)(5, 8)(6, 15)(7, 13)(11, 16)
    (1, 11)(2, 13)(3, 5)(4, 15)(6, 14)(7, 9)(8, 12)(10, 16)
    (1, 12)(2, 6)(3, 10)(4, 7)(5, 16)(8, 11)(9, 15)(13, 14)
```

Since A is primitive, this looks as if it is an elementary abelian regular normal subgroup:

```
> EARNS(A) eq F;
true
```

Thus, A has a regular normal subgroup of order 16. Further, it acts transitively on the vertices. The complement of N is the stabilizer of a point.

```
> S1 := Stabilizer(A, AV, V!1);
> #S1;
120
> IsTransitive(S1);
false
> O := Orbits(S1);
> O;
[
    GSet{ 1 },
    GSet{ 2, 3, 5, 7, 9, 11, 13, 14, 15, 16 },
    GSet{ 4, 6, 8, 10, 12 }
]
> IsSymmetric(ActionImage(S1, O[3]));
```

```
true
```

So the stabilizer of the vertex 1 is Sym(5). Note that it acts faithfully on the 5 neighbours of 1.

149.23 Symmetry and Regularity Properties of Graphs

`IsTransitive(G)`
`IsVertexTransitive(G)`

Returns `true` if the automorphism group of the graph G is transitive, otherwise `false`. To see which automorphism group is computed see Subsection 149.22.3.

`IsEdgeTransitive(G)`

Returns `true` if the automorphism group of the graph G is transitive on the edges of G (i.e. if the edge group of G is transitive). To see which automorphism group is computed see Subsection 149.22.3.

`OrbitsPartition(G)`

Given a graph G, return the partition of its vertex-set corresponding to the orbits of its automorphism group in the form of a set system. To see which automorphism group is computed see Subsection 149.22.3.

`IsPrimitive(G)`

Returns `true` if the graph G is primitive, i.e. if its automorphism group is primitive. To see which automorphism group is computed see Subsection 149.22.3.

`IsSymmetric(G)`

Returns `true` if the graph G is symmetric, i.e. if for all pairs of vertices u, v and w, t such that u adj v and w adj t, there exists an automorphism a such $u^a = w$ and $v^a = t$. To see which automorphism group is computed see Subsection 149.22.3.

`IsDistanceTransitive(G)`

Returns `true` if the connected graph G is distance transitive i.e. if for all vertices u, v, w, t of G such that $d(u, v) = d(w, t)$, there is an automorphism a in A such that $u^a = w$ and $v^a = t$. To see which automorphism group is computed see Subsection 149.22.3.

`IsDistanceRegular(G)`

Returns `true` if the graph G is distance regular, otherwise `false`. To see how the automorphism group of G is computed see Subsection 149.22.3.

`IntersectionArray(G)`

The intersection array of the distance regular graph G. This is returned as a sequence $[k, b(1), \ldots, b(d-1), 1, c(2), \ldots, c(d)]$ where k is the valency of the graph, d is the diameter of the graph, and the numbers $b(i)$ and $c(i)$ are defined as follows: Let $N_j(u)$ denote the set of vertices of G that lie at distance j from vertex u. Let u and v be a pair of vertices satisfying $d(u, v) = j$.

Then $c(j)$ = number of vertices in $N_{j-1}(v)$ that are adjacent to u, $(1 \leq j \leq d)$, and $b(j)$ = number of vertices in $N_{j+1}(v)$ that are adjacent to u $(0 \leq j \leq d-1)$.

Example H149E21

We illustrate the use of some of the symmetry functions by applying then to the graph of the 8-dimensional cube.

```
> g := KCubeGraph(8);
> IsVertexTransitive(g);
true
> IsEdgeTransitive(g);
true
> IsSymmetric(g);
true
> IsDistanceTransitive(g);
true
> IntersectionArray(g);
[ 8, 7, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 7, 8 ]
```

We also see that the functions using the graph's automorphism group are dependent upon the graph being coloured or not:

```
> q := 9;
> P := FiniteProjectivePlane(q);
> X := IncidenceGraph(P);
>
> Order(X);
182
> Valence(X);
10
> Diameter(X);
3
> Girth(X);
6
> O1 := OrbitsPartition(X);
> IsSymmetric(X);
true
>
> Labels := [ "a" : i in [1..96] ];
> #Labels;
96
```

```
> AssignLabels(VertexSet(X), Labels);
> O2 := OrbitsPartition(X);
> O2 eq O1;
false
> IsSymmetric(X);
false
```

149.24 Graph Databases and Graph Generation

MAGMA provides interfaces to some databases of certain graphs of interest. These databases are not provided by default with MAGMA, but may be downloaded from the optional databases section of the MAGMA website.

149.24.1 Strongly Regular Graphs

A catalogue of strongly regular graphs is available. This catalogue has been put together from various sources by B. McKay and can be found at

`http://cs.anu.edu.au/~bdm/data/`

Graphs in the database are indexed by a sequence of four parameters. They are, in order: the order of the graph, its degree, the number of common neighbours to each pair of adjacent vertices, and the number of common neighbours to each pair of non-adjacent vertices.

`StronglyRegularGraphsDatabase()`

Opens the database of strongly regular graphs.

`Classes(D)`

Returns all the parameter sequences used to index the graphs in the database D.

`NumberOfClasses(D)`

Returns the number of "classes" of graphs in the database D.

`NumberOfGraphs(D)`

Returns the number of graphs in the database D.

`NumberOfGraphs(D, S)`

Returns the number of graphs in the database D with parameter sequence S.

`Graphs(D, S)`

Returns (in a sequence) all the graphs in the database D with parameter sequence S.

`Graph(D, S, i)`

Returns the ith graph in the database D with parameter sequence S.

`RandomGraph(D)`

Returns a random graph in the database D.

`RandomGraph(D, S)`

Returns a random graph in the database D with parameter sequence S.

`for G in D do ... end for;`

The database of strongly regular graphs may appear as the range in the `for`-statement.

Example H149E22

The following few statements illustrate the basic access functions to the database of strongly regular graphs.

```
> D := StronglyRegularGraphsDatabase();
> Cs := Classes(D);
> Cs;
[
  [ 25, 8, 3, 2 ],
  [ 25, 12, 5, 6 ],
  [ 26, 10, 3, 4 ],
  [ 27, 10, 1, 5 ],
  [ 28, 12, 6, 4 ],
  [ 29, 14, 6, 7 ],
  [ 35, 16, 6, 8 ],
  [ 35, 14, 4, 6 ],
  [ 36, 15, 6, 6 ],
  [ 37, 18, 8, 9 ],
  [ 40, 12, 2, 4 ]
]
> assert NumberOfClasses(D) eq #Cs;
>
> NumberOfGraphs(D);
43442
>
> for i in [1..#Cs] do
>     NumberOfGraphs(D, Cs[i]);
> end for;
1
15
10
1
4
41
```

```
3854
180
32548
6760
28
>
>  gs := Graphs(D, Cs[2]);
>
>  g :=  Graph(D, Cs[2], Random(1, NumberOfGraphs(D, Cs[2])));
```

149.24.2 Small Graphs

An enumeration of small graphs with various properties has been created by B. McKay and may be found at

`http://cs.anu.edu.au/~bdm/data/graphs.html`

Certain of these databases are available from within MAGMA: Simple graphs, Eulerian graphs, planar connected graphs, and self-complementary graphs.

149.24.2.1 Creation of Small Graph Databases

`SmallGraphDatabase(n : parameters)`

`IncludeDisconnected` BOOL *Default :* `false`

Opens the database of simple graphs with n vertices, $2 \leq n \leq 10$. If the optional parameter `IncludeDisconnected` is set to true then the database will also include non-connected graphs.

`EulerianGraphDatabase(n : parameters)`

`IncludeDisconnected` BOOL *Default :* `false`

Opens the database of Eulerian graphs with n vertices, $3 \leq n \leq 11$. If the optional parameter `IncludeDisconnected` is set to true then the database will also include non-connected graphs. The allowed range of n for non-connected graphs is $2 \leq n \leq 12$.

`PlanarGraphDatabase(n)`

Opens the database of planar connected graphs with n vertices, $2 \leq n \leq 11$.

`SelfComplementaryGraphDatabase(n)`

Opens the database of self-complementary graphs with n vertices, $n \in \{4, 5, 8, 9, 12, 13, 16, 17, 20\}$. For $n = 20$ this is not a complete enumeration.

149.24.2.2 Access functions

`#D`

Returns the number of graphs in the database D.

`Graph(D, i)`

Returns the ith graph in the database D.

`Random(D)`

Returns a random graph from the database D.

```
for G in D do ...  end for;
```

A database of small graphs may appear as the range in the `for`-statement.

149.24.3 Generating Graphs

We provide an interface to a graph generation programme, also due to B. McKay (see [McK98]). For the time being, users wanting to benefit from this facility must download the generation programme themselves directly from

`http://cs.anu.edu.au/~bdm/nauty/`

Important restriction: Since this program runs within a Unix pipe it is only available to users running MAGMA on a Unix platform.

And a note of caution: When the graph generation programme is used to generate reasonably large graphs ($n > 17$) it can be observed that the procedure of closing down the pipe (ie. closing the stream) may take some time. This will happen when the closing down attempt is made before the programme has completed the generation of all the graphs.

`GenerateGraphs(n : parameters)`

Opens a pipe to the graph generation programme to generate all graphs of order n. Only available on Unix platforms.

The `GenerateGraphs` function allows the user to drive the generation programme via a set of parameters. These parameters are described below.

Once the generation programme has been downloaded from

`http://cs.anu.edu.au/~bdm/nauty/`

and compiled (using `make geng`), the resulting executable (named `geng` – it is *compulsory* that the resulting executable's name be `geng`) can be placed anywhere in the user's directory tree. The environment variable `MAGMA_NAUTY` *must* then be set to the path where the executable/command `geng` is to be found.

`FirstGraph` RNGINTELT *Default :* 1

Reading of the generated graphs starts at the `FirstGraph`-th graph.

`MinEdges` RNGINTELT *Default :*

Generate graphs with minimum number of edges `MinEdges`.

`MaxEdges` RngIntElt *Default :*

Generate graphs with maximum number of edges `MaxEdges`.

`Classes` RngIntElt *Default :* 1

Divide the generated graphs into disjoint `Classes` classes of very approximately equal size.

`Class` RngIntElt *Default :* 1

When generated graphs are divided into disjoint `Classes` classes, write only the `Class`th class.

`Connected` BoolElt *Default :* `false`

Only generate connected graphs.

`Biconnected` BoolElt *Default :* `false`

Only generate biconnected graphs.

`TriangleFree` BoolElt *Default :* `false`

Only generate triangle–free graphs.

`FourCycleFree` BoolElt *Default :* `false`

Only generate 4–cycle–free graphs.

`Bipartite` BoolElt *Default :* `false`

Only generate bipartite graphs.

`MinDeg` RngIntElt *Default :*

Specify a lower bound for the minimum degree.

`MaxDeg` RngIntElt *Default :*

Specify an upper bound for the maximum degree.

`Canonical` BoolElt *Default :* `false`

Canonically label output graphs.

`SparseRep` BoolElt *Default :* `false`

If `true`, generate the graphs in Sparse6 format (see below).

`NextGraph(F: parameters)`

`SparseRep` Bool *Default :* `false`

Returns `true` if and only if file F is not at the end. In this case the next graph is returned as well.

Since `NextGraph` can in principle take as an argument any Magma object of type `File`, the following restriction applies to the `File` F: The graphs in F *must* be in either of the output formats Graph6 or Sparse6. Details on the Graph6 and Sparse6 format can be found at

`http://cs.anu.edu.au/~bdm/data/formats.html`

If SparseRep is `true` then the resulting graph will have a sparse representation. This of course is of special interest if the graphs read from F are also in Sparse6 format.

Example H149E23

The following statements should help clarify the usage of the graph generation programme.

```
> F := GenerateGraphs (12:
>     FirstGraph:= 10,
>     Connected:= true,
>     Biconnected:= true,
>     TriangleFree:= true,
>     FourCycleFree:= true,
>     Bipartite:= true,
>     MinDeg:= 1,
>     MaxDeg:= 9
> );
```

We'll read all the graphs from the 10th graph onwards (one can check that 28 graphs have been generated):

```
> count := 0;
> while true do
>     more := NextGraph(F);
>     if more then
>       count +:= 1;
>     else
>       break;
>     end if;
> end while;
> count;
19
```

If one wants to work with sparse graphs, it is recommended to proceed as follows:

```
> F := GenerateGraphs (6: SparseRep := true);
> count := 0;
> while true do
>     more := NextGraph(F: SparseRep := true);
>     if more then
>       count +:= 1;
>     else
>       break;
>     end if;
> end while;
> count;
156
```

149.24.4 A General Facility

In order to give users more flexibility in dealing with certain graph files the MAGMA function `OpenGraphFile` is provided. It allows one to open either a graph file or a pipe to a graph generation programme. Since in both cases (file or Unix pipe) the outcome is the access to a stream of graphs, we henceforth refer to the graphs to be read as a *graph stream*.

The usual restriction: The `OpenGraphFile` which opens a pipe is only available to users running MAGMA on a Unix platform.

Accessing and reading the graph stream: Reading the graph stream is achieved by the above described access function `NextGraph`. As mentioned there, the graphs in the graph stream *must* be in either of the output formats Graph6 or Sparse6. This is why `OpenGraphFile` is restricted to streams of graphs in format Graph6 or Sparse6.

Details on the Graph6 and Sparse6 format can be found at

`http://cs.anu.edu.au/~bdm/data/formats.html`.

`OpenGraphFile(s, f, p)`

Opens a graph file/pipe at position p. If the stream to be opened is a Unix pipe then the string s *must* have the format "cmd command" where command stands for the command to run including necessary parameters. If the stream to be opened is a file the string s has format "filename".

The integer f indicates that the record length is fixed, which is true for streams in Graph6 format with every graph having the same order. This permits rapid positioning to position p in that case. If in doubt, use $f = 0$. Also, positioning to 0 or positioning to 1 has the same effect of positioning to the start of the stream.

Opening a pipe is only available on Unix platforms. The file/pipe F *must* contain graphs in Graph6 and Sparse6 format.

Example H149E24

As an example one could download one of the files found at

`http://cs.anu.edu.au/~bdm/data/`

Assuming this has been done, one can then proceed to read the graphs in the file:

```
> F := OpenGraphFile("/home/paule/graph/bdm_data/sr251256.g6", 0, 0);
>
> count := 0;
> more, g := NextGraph(F);
> while more do
>    count +:= 1;
>    more, g := NextGraph(F);
> end while;
> count;
15
```

Alternatively one could also drive the graph generation programme (or any other suitable programme for that matter) described in 149.24.3 using the `OpenGraphFile` access function.

The graph generation programme's name is `geng` (which can be found at `http://cs.anu.edu.au/~bdm/nauty/`) and has a help facility:

```
>  F := OpenGraphFile("cmd /home/paule/graph/bdm_pgr/nauty/geng -help", 0, 0);
Usage: geng [-cCmtfbd#D#] [-uygsnh] [-lvq] [-x#X#] n [mine[:maxe]] [res/mod]
[file]
 Generate all graphs of a specified class.
      n    : the number of vertices (1..32)
 mine:maxe : a range for the number of edges
              #:0 means '# or more' except in the case 0:0
   res/mod : only generate subset res out of subsets 0..mod-1
     -c    : only write connected graphs
     -C    : only write biconnected graphs
     -t    : only generate triangle-free graphs
     -f    : only generate 4-cycle-free graphs
     -b    : only generate bipartite graphs
                (-t, -f and -b can be used in any combination)
     -m    : save memory at the expense of time (only makes a
                difference in the absence of -b, -t, -f and n <= 30).
     -d#   : a lower bound for the minimum degree
     -D#   : a upper bound for the maximum degree
     -v    : display counts by number of edges
     -l    : canonically label output graphs
     -u    : do not output any graphs, just generate and count them
     -g    : use graph6 output (default)
     -s    : use sparse6 output
     -y    : use the obsolete y-format instead of graph6 format
     -h    : for graph6 or sparse6 format, write a header too
     -q    : suppress auxiliary output (except from -v)
  See program text for much more information.
```

Finally, here is a typical run of this graph generation programme:

```
>  F := OpenGraphFile(
>  "cmd /home/paule/graph/bdm_pgr/nauty/geng 15 -cCtfb -v", 0, 0);
>A geng -Ctfbd2D14 n=15 e=15-22
>C 4 graphs with 16 edges
>C 45 graphs with 17 edges
>C 235 graphs with 18 edges
>C 294 graphs with 19 edges
>C 120 graphs with 20 edges
>C 13 graphs with 21 edges
>C 1 graphs with 22 edges
>Z 712 graphs generated in 1.38 sec
```

149.25 Bibliography

[**BK73**] C. Bron and J. Kerbosch. Finding All Cliques of an Undirected Graph. *Communications of the ACM 9*, 16(9):575–577, 1973.

[**BM01**] J. Boyer and W. Myrvold. Simplified O(n) Planarity Algorithms. submitted, 2001.

[**Bre79**] D. Brelaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(9):251–256, 1979.

[**Chr75**] N. Christofides. *Graph Theory, An Algorithm Approach.* Academic Press, 1975.

[**Eve79**] Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.

[**GM01**] C. Gutwenger and P. Mutzel. A Linear Time Implementation of SPQR-Trees. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *LNCS*, pages 70–90. Springer-Verlag, 2001.

[**HT73**] J.E. Hopcroft and R.E. Tarjan. Dividing a Graph into Triconnected Components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[**Lor89**] P. Lorimer. The construction of Tutte's 8-cage and the Conder graph. *J. of Graph Theory*, 13(5):553–557, 1989.

[**McK**] B. D. McKay. `nauty` User's Guide (Version 2.2). URL:http://cs.anu.edu.au/~bdm/nauty/nug.pdf.

[**McK81**] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30: 45–87, 1981.

[**McK98**] B. D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26:306–324, 1998.

[**RAO93**] T.L. Magnanti R.K. Ahuja and J.B. Orlin. *Network Flows, Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[**TCR90**] C.E. Leiserson T.H. Cormen and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[**WM**] N. Walker W. Myrvold, T. Prsa. A Dynamic Programming Approach for Timing and Designing Clique Algorithms. available at URL:http://www.csr.uvic.ca/~wendym/.

150 MULTIGRAPHS

Chapter 150
MULTIGRAPHS

150.1 Introduction

Multigraphs and *multidigraphs* are graphs and digraphs which may have multiple (i.e., parallel) edges and loops. This is in contrast to simple graphs (see Section 149.1 in the chapter on graphs). In the context of this chapter we use the term "graph" as a generic term for the general vertex-edge incident structure, and the term "multigraph" as a generic term whenever we want to emphasize the possible existence of multiple edges and loops. Thus, graphs and multigraphs may be directed or undirected. The meaning of the terms "graph" and "multigraph" should be made clear from the context in which they occur.

Multigraphs are represented in the form of an adjacency list; for more information on this topic we refer the reader to Section 149.3. As is the case for simple graphs, the vertices and edges of a multigraph may be given "decorations". More precisely, they may be labelled; in addition the edges may also be assigned a capacity (if one is interested in flow problems) and/or a weight (for investigation of shortest path problems).

For convenience, MAGMA provides users with *networks*: in the MAGMA context, a network is understood as being a multidigraph whose edges are *always* given a capacity. Any MAGMA function that takes a network as an argument will usually take any graph whose edges have a capacity as an argument. Networks are covered in Chapter 151; a few functionalities specific to networks will, however, be discussed in the general multigraph context whenever there is a need to highlight some differences between networks and general multi(di)graphs.

In particular, almost all the standard graph construction functions (see Section 150.5) preserve the graph's support set and vertex and edge decorations. That is, the resulting graph will have a support and vertex and edge decorations compatible with the original graph and the operation performed on that graph.

A MAGMA multigraph object has type `GrphMultUnd`, a multidigraph has type `GrphMultDir`, and a network has type `GrphNet`. All three objects are of type `GrphMult`.

The multigraph facilities represent a significant subset of the (simple) graph functionality. Consequently, some sections of this chapter are very similar to their counterparts in Chapter 149. For the sake of clarity, this chapter describes the complete multigraph functionality.

150.2 Construction of Multigraphs

In this implementation, the order n of a multigraph or multidigraph is bounded by 134217722. See Section 149.2.1 in Chapter 149 for more details.

150.2.1 Construction of a General Multigraph

Undirected multigraphs are constructed in a similar way to graphs (Subsection 149.2.2).

```
MultiGraph< n | edges  >
```

```
MultiGraph< S | edges  >
```

Construct the multigraph G with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The multigraph G, the vertex-set V of G; and the edge-set E of G.

The elements of E are specified by the list *edges*, where the items of *edges* may be objects of the following types:

(a) A pair $\{v_i, v_j\}$ of vertices in V. The undirected edge $\{v_i, v_j\}$ from v_i to v_j will be added to the edge-set for G.

(b) A tuple of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the edges $\{v_i, u_1\}, \ldots, \{v_i, u_r\}$ will be added to G.

(c) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of neighbours for the vertex v_i. The edges $\{v_i, u_i\}$, $u_i \in N_i$, are added to G.

In addition to these three basic ways of specifying the *edges* list, the items in *edges* may also be:

(d) An edge e of a graph or digraph or multigraph or multidigraph or network of order n. If e is an edge from u to v, then the edge $\{u, v\}$ is added to G.

(e) An edge-set E of a graph or digraph or multigraph or multidigraph or network of order n. Every edge e in E will be added to G according to the rule set out for a single edge.

(f) A graph or a digraph or a multigraph or a multidigraph or a network H of order n. Every edge e in H's edge-set is added to G according to the rule set out for a single edge.

(g) A set of

- (i) Pairs of the form $\{v_i, v_j\}$ of vertices in V.
- (ii) Tuples of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i.
- (iii) Edges of a graph or digraph or multigraph or multidigraph or network of order n.

(iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order n.

(h) A sequence of

(i) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of neighbours for the vertex v_i.

Example H150E1

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours
1       2 3 2 ;
2       3 2 2 1 1 ;
3       2 1 ;
```

150.2.2 Construction of a General Multidigraph

Multidigraphs are constructed in the same way as digraphs (Subsection 149.2.3).

```
MultiDigraph< n | edges  >
```

```
MultiDigraph< S | edges  >
```

Construct the multidigraph G with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The multidigraph G, the vertex-set V of G; and the edge-set E of G.

The elements of E are specified by the list *edges*, where the items of *edges* may be objects of the following types:

(a) A pair $[v_i, v_j]$ of vertices in V. The directed edge $[v_i, v_j]$ from v_i to v_j will be added to the edge-set for G.

(b) A tuple of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the edges $[v_i, u_1], \ldots, [v_i, u_r]$ will be added to G.

(c) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of out-neighbours for the vertex v_i. All the edges $[v_i, u_i]$, $u_i \in N_i$, are added to G.

In addition to these four basic ways of specifying the *edges* list, the items in *edges* may also be:

(d) An edge e of a graph or digraph or multigraph or multidigraph or network of order n. If e is an edge from u to v, then the edge $[u, v]$ is added to G. Thus, if e is an undirected edge from u to v, both edges $[u, v]$ and $[v, u]$ are added to G.

(e) An edge-set E of a graph or digraph or multigraph or multidigraph or network of order n. Every edge e in E will be added to G according to the rule set out for a single edge.

(f) A graph or a digraph or a multigraph or a multidigraph or a network H of order n. Every edge e in H's edge-set is added to G according to the rule set out for a single edge.

(g) A set of

(i) Pairs of the form $[v_i, v_j]$ of vertices in V.

(ii) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

(iii) Edges of a graph or digraph or multigraph or multidigraph or network of order n.

(iv) Graphs or digraphs or multigraphs or multidigraphs or networks of order n.

(h) A sequence of

(i) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

Example H150E2

```
> G := MultiDigraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multidigraph
Vertex  Neighbours
1       2 3 2 ;
2       3 2 ;
3       ;
```

150.2.3 Printing of a Multi(di)graph

A multi(di)graph is displayed by listing, for each vertex, all of its adjacent vertices. If the multigraph has multiple edges from u to v, then the adjacency list of u contains as many copies of the vertex v as there are edges from u to v.

The vertices in the adjacency list are not ordered, they appear in the order in which they were created. See the previous examples H150E1 and H150E2.

150.2.4 Operations on the Support

The support of a multi(di)graph is subject to exactly the same operations as simple graphs (see Subsection 149.2.4).

`Support(G)`

`Support(V)`

The indexed set used in the construction of G (or the graph for which V is the vertex-set), or the standard set $\{@1, \ldots, n@\}$ if it was not given.

`ChangeSupport(G, S)`

If G is a graph having n vertices and S is an indexed set of cardinality n, return a new graph H equal to G but whose support is S. That is, H is structurally equal to G and its vertex and edge decorations are the *same* as those for G (see Sections 150.4.1 and 150.4.2).

`ChangeSupport(~G, S)`

The procedural version of the above function.

`StandardGraph(G)`

Returns a graph H that is isomorphic to G but defined on the standard support. That is, H is structurally equal to G and its vertex and edge decorations are the same as those for G.

Example H150E3

```
> S := {@ "a", "b", "c" @};
> G := MultiGraph< S | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours
c       b a b ;
b       a b b c c ;
a       b c ;
> StandardGraph(G);
Multigraph
Vertex  Neighbours
1       2 3 2 ;
2       3 2 2 1 1 ;
3       2 1 ;
```

150.3 The Vertex–Set and Edge–Set of Multigraphs

Much of the functionality for simple graphs (see Section 149.4) also applies to multigraphs. We will not repeat here the functions pertaining to the vertex-set and edge-set of a graph, but concentrate instead on the edges. Indeed, there is a difference in the manner in which multigraph edges are created and accessed when compared to simple graph edges.

Since multigraphs may have multiple edges from u to v, it is necessary to know how many of these edges there are, and how each can be accessed. More importantly a scheme is required to uniquely identify an edge.

Recall that a multigraph is represented by means of an adjacency list, so any edge from u to v can be identified by its index in this list. If there is more than one edge from u to v, then a list of indices will identify each edge from u to v. Thus, the index of the edge in the adjacency list will be the means by which this edge can be *uniquely* identified.

This has a bearing on how an edge can be coerced into a multigraph: Successful coercion will require two vertices u and v so that v is a neighbour of u, and a valid index i in the multigraph's adjacency list. That is, the position i in the list is the index of an edge from u to v.

`EdgeIndices(u, v)`

`Indices(u, v)`

Given vertices u and v of a multigraph G, returns the indices of the possibly multiple edge from u to v.

`EdgeMultiplicity(u, v)`

`Multiplicity(u, v)`

Given vertices u and v of a multigraph G, returns the multiplicity of the possibly multiple edge from u to v. Returns 0 if u is not adjacent to v.

`Edges(u, v)`

Given vertices u and v of a multigraph G, returns all the edges from u to v as a sequence of elements of the edge-set of G.

`IncidentEdges(u)`

Given a vertex u of an undirected multigraph G, returns all the edges incident to u as a set of elements of the edge-set of G. If G is a multidigraph, the function returns all the edges incident to *and* from u as a set of elements of the edge-set of G.

`E ! < { u, v }, i >`

Given the edge-set E of the undirected multigraph G and objects u, v belonging to the support of G corresponding to adjacent vertices, returns the edge from u to v which corresponds to the edge with index i in the adjacency list of G. This requires that the edge at i in the adjacency list of G is an edge from u to v.

`E ! < [ u, v ], i >`

Given the edge-set E of the multidigraph G and objects u, v belonging to the support of G corresponding to adjacent vertices, returns the edge from u to v which corresponds to the edge with index i in the adjacency list of G. This requires that the edge at i in the adjacency list of G is an edge from u to v.

`E . i`

Let E be the edge-set of G. If G is a simple graph (digraph) then this function is as described in Subsection 149.4.2.

If G is a multigraph or multidigraph, then this function returns the edge at index i in the adjacency list of G, provided i is a valid index.

`EndVertices(e)`

Given an edge e in an undirected multigraph G, returns the end vertices of e as a set of vertices $\{u, v\}$. If G is a multidigraph, returns e's end vertices as a sequence of vertices $[u, v]$.

`InitialVertex(e)`

Given an edge $e = \{u, v\}$ or $e = [u, v]$, returns vertex u. This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

`TerminalVertex(e)`

Given an edge $e = \{u, v\}$ or $e = [u, v]$, returns vertex v. This is useful in the undirected case since it indicates, where relevant, the direction in which the edge has been traversed.

`Index(e)`

Given an edge e in a multi(di)graph G, returns the index of e in the adjacency list of G.

`s eq t`

Returns `true` if the edge s is equal to the edge t. It s and t are edges in a multi(di)graph G, returns `true` if and only if s and t have the same index in the adjacency list of G.

Example H150E4

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
```

The graph G has a loop at vertex 2:

```
> Edges(G);
{@ < {1, 2}, 1 >, < {1, 2}, 5 >, < {1, 3}, 3 >, < {2, 2}, 7 >, < {2,
3}, 9 > @}
> E := EdgeSet(G);
> E.7;
< {2, 2}, 7 >
> assert InitialVertex(E.7) eq TerminalVertex(E.7);
```

The graph G has a multiple edge from vertex 1 to vertex 2:

```
> u := VertexSet(G)!1;
> v := VertexSet(G)!2;
> I := EdgeIndices(u, v);
> I;
[ 5, 1 ]
> assert #I eq Multiplicity(u, v);
>
> E := EdgeSet(G);
> e1 := E!< { 1, 2 }, I[1] >;
> e2 := E!< { 1, 2 }, I[2] >;
> e1, e2;
< {1, 2}, 5 > < {1, 2}, 1 >
> EndVertices(e1);
{ 1, 2 }
> EndVertices(e2);
{ 1, 2 }
> assert Index(e1) eq I[1];
> assert Index(e2) eq I[2];
>
> assert e1 eq E.I[1];
> assert not e2 eq E.I[1];
> assert e2 eq E.I[2];
> assert not e1 eq E.I[2];
```

We have seen that these two edges can be accessed directly:

```
> Edges(u, v);
[ < {1, 2}, 5 >, < {1, 2}, 1 > ]
```

150.4 Vertex and Edge Decorations

150.4.1 Vertex Decorations: Labels

In this implementation, it is only possible to assign labels as vertex decorations.

A vertex labelling of a graph G is a partial map from the vertex-set of G into a set L of labels.

`AssignLabel(~G, u, l)`

Assigns the label l to the vertex u in the graph G.

`AssignLabels(~G, S, L)`

Assigns the labels in L to the corresponding vertices in G in the sequence or indexed set S. If for vertex u the corresponding entry in L is not defined then any existing label of u is removed.

`AssignVertexLabels(~G, L)`

Assigns the labels in L to the corresponding vertices in the graph G.

`IsLabelled(u)`

Returns `true` if and only if the vertex u has a label.

`IsLabelled(V)`

Returns `true` if and only if the vertex-set V is labelled.

`IsVertexLabelled(G)`

Returns `true` if and only if vertices of the graph G are labelled.

`Label(u)`

The label of the vertex u. An error is raised if u is not labelled.

`Labels(S)`

The sequence L of labels of the vertices in the sequence S. If an element of S has no label, then the corresponding entry in L is undefined.

`Labels(V)`

The sequence L of labels of the vertices in the vertex-set V. If an element of V has no label, then the corresponding entry in L is undefined.

`VertexLabels(G)`

The sequence L of labels of the vertices of G. If a vertex of G has no label, then the corresponding entry in L is undefined.

`DeleteLabel(~G, u)`

Removes the label of the vertex u.

Remove the labels of the vertices in S.

```
DeleteVertexLabels(~G)
```

Remove the labels of the vertices in the graph G.

150.4.2 Edge Decorations

Simple graph or multigraph edges can be assigned three different types of decorations:

- a label,
- a capacity,
- a weight.

Edge labels can be objects of any MAGMA type. Edge capacities must be non-negative integers (and loops must be assigned a zero capacity). Edge weights must be elements of a totally ordered ring. Not all edges need to be assigned labels when labelling edges, nor do all edges need to be assigned capacities or weights when assigning either decoration. In the latter case, if some edges have been assigned a capacity or a weight, then the capacity or weight of any remaining unassigned edge is always taken to be zero.

One may want to assign capacities to edges in order to apply a network-flow algorithm to the graph (Section 151.4). By assigning weights to edges one may also be able to run a shortest-path algorithm (Section 150.12).

150.4.2.1 Assigning Edge Decorations

```
AssignLabel(~G, e, l)
```

```
AssignCapacity(~G, e, c)
```

```
AssignWeight(~G, e, w)
```

Assigns the label l or the capacity c or the weight w to the edge e in the graph G. The capacity of any edge must be a non-negative integer except in the case of a loop when it must be zero. The weight w must be an element from a totally ordered ring.

```
AssignLabels(~G, S, D)
```

```
AssignCapacities(~G, S, D)
```

```
AssignWeights(~G, S, D)
```

Assigns the labels or capacities or weights in the sequence D to the corresponding edges in the sequence or indexed set S. If for some edge e the corresponding entry in D is not defined then any existing label or capacity or weight of e is removed. The same constraints regarding capacity and weight apply as in the single edge assignment case.

`AssignEdgeLabels(~G, D)`

`AssignCapacities(~G, D)`

`AssignWeights(~G, D)`

Assigns the labels or capacities or weights in the sequence D to the corresponding edges in graph G. Let E be the edge-set of G and let d be the decoration at position i in D, that is, $d = D[i]$. Then the corresponding edge e in E which will be decorated is $E.i$ (see `E.i`).

If for some edge e the corresponding entry in D is not defined then any existing label or capacity or weight of e is removed. The same constraints regarding capacity and weight apply as in the single edge assignment case.

150.4.2.2 Testing for Edge Decorations

While it is the case that an edge is considered to be labelled if and only if it has been assigned a label, an edge may have a default capacity (weight) of zero. Let e be an edge of a graph G and assume that e has not been assigned a capacity (weight).

If any other edge of G has been assigned a capacity (weight), then the edge-set of G is considered to be capacitated (weighted). In which case the capacity (weight) of e has a default value of zero.

If no other edge of G has been assigned a capacity (weight), then the edge-set of G is considered to be uncapacitated (unweighted). In which case asking for the capacity (weight) of e results in an error.

This is in contrast to the labelling situation where there is no concept of a "default" label. The edge-set of G is considered to be labelled if and only if at least one edge of G has been labelled, while any edge of G is considered to be labelled if and only if e has been labelled.

`IsLabelled(e)`

Returns `true` if and only if the edge e has a label.

`IsLabelled(E)`

Returns `true` if and only if the edge-set is labelled; that is, if and only if at least one edge of E has been assigned a label.

`IsEdgeLabelled(G)`

Returns `true` if and only if the edge-set of G is labelled.

`IsCapacitated(E)`

Returns `true` if and only if the edge-set is capacitated; that is, if and only if at least one edge of E has been assigned a capacity.

`IsEdgeCapacitated(G)`

Returns `true` if and only if the edge-set of G is capacitated.

`IsWeighted(E)`

Returns `true` if and only if the edge-set is weighted; that is, if and only if at least one edge of E has been assigned a weight.

`IsEdgeWeighted(G)`

Returns `true` if and only if the edge-set of G is weighted.

150.4.2.3 Reading Edge Decorations

An edge may have a default capacity and weight of zero, see Subsubsection 150.4.2.2 for more details.

`Label(e)`

The label of the edge e. An error is raised if e has not been assigned a capacity.

`Capacity(e)`

The capacity of the edge e. Let G be the parent graph of e. An error is raised if the edge-set of G is uncapacitated. If the edge-set of G is capacitated but e has not been assigned a capacity, then `Capacity(e)` returns zero as the default value.

`Weight(e)`

The weight of the edge e. Let G be the parent graph of e. An error is raised if the edge-set of G is unweighted. If the edge-set of G is weighted but e has not been assigned a weight, then `Weight(e)` returns zero as the default value.

`Labels(S)`
`Capacities(S)`
`Weights(S)`

The sequence D of labels or capacities or weights of the edges in the sequence S. Let E be the edge-set of the parent graph of the edges. If E is unlabelled or uncapacitated or unweighted then D is the null sequence. If an element of S has no label then the corresponding entry in D is undefined. If an element of S has no capacity or weight while E is capacitated or weighted then the corresponding entry in D has the default value of zero.

`Labels(E)`
`Capacities(E)`
`Weights(E)`

The sequence D of labels or capacities or weights of the edges in the edge-set E. If E is unlabelled or uncapacitated or unweighted then D is the null sequence. If an element of E has no label then the corresponding entry in D is undefined. If an element of E has no capacity or weight while E is capacitated or weighted then the corresponding entry in D has the default value of zero. The corresponding entry i in D of any edge e is such that $e = E.i$ (see `E.i`).

`EdgeLabels(G)`

`EdgeCapacities(G)`

`EdgeWeights(G)`

The sequence D of labels or capacities or weights of the edges in edge-set E of the graph G. If E is unlabelled or uncapacitated or unweighted then D is the null sequence. If an element of E has no label then the corresponding entry in D is undefined. If an element of E has no capacity or weight while E is capacitated or weighted then the corresponding entry in D has the default value of zero. The corresponding entry i in D of any edge e is such that $e = E.i$ (see `E.i`).

150.4.2.4 Deleting Edge Decorations

`DeleteLabel(~G, e)`

`DeleteCapacity(~G, e)`

`DeleteWeight(~G, e)`

Removes the label or capacity or weight of the edge e in the graph G.

`DeleteLabels(~G, S)`

`DeleteCapacities(~G, S)`

`DeleteWeights(~G, S)`

Remove the labels or capacities or weights of the edges (of the graph G) in S.

`DeleteEdgeLabels(~G)`

`DeleteCapacities(~G)`

`DeleteWeights(~G)`

Remove the labels or capacities or weights of the edges in the edge set of the graph G.

150.4.3 Unlabelled, or Uncapacitated, or Unweighted Graphs

Starting with a graph G, the functions below return a graph that is isomorphic as a simple graph to G, but without the vertex and edge labels of G, (or without the edge capacities or weights of G in the case of a network). Should one require a copy of a graph without the support of G, see Subsection 149.2.4. Should one require a copy of a graph without the support of G *and* without the vertex/edge decorations of G, see Subsection 150.6.3.

`UnlabelledGraph(G)`

Return the (vertex and edge) unlabelled graph structurally identical to G, whose edges have the same capacities and weights as those in G. The support of G is also retained in the resulting graph.

`UncapacitatedGraph(G)`

Return the uncapacitated graph structurally identical to G, whose vertices and edges have the same labels, and whose edges have the same weights as those in G. The support of G is also retained in the resulting graph.

`UnweightedGraph(G)`

Return the unweighted graph structurally identical to G, whose vertices and edges have the same labels, and whose edges have the same capacities as those in G. The support of G is also retained in the resulting graph.

Example H150E5

The labelling operations are illustrated by constructing a 2-colouring of the complete bipartite graph $K_{3,4}$. Use is made of the function `Distance`(u, v) which returns the distance between vertices u and v.

```
> K34, V, E := BipartiteGraph(3, 4);
> L := [ IsEven(Distance(V!1, v)) select "red" else "blue" : v in Vertices(K34) ];
> AssignLabels(Vertices(K34), L);
> VertexLabels(K34);
[ red, red, red, blue, blue, blue, blue ]
```

Example H150E6

Another illustration is the creation of the Cayley graph of a group. In this example Sym(4) is used.

```
> G<a,b> := FPGroup(Sym(4));
> I, m := Transversal(G, sub<G | 1>);
> S := Setseq(Generators(G));
> N := [ {m(a*b) : b in S} : a in I ];
> graph := StandardGraph(Digraph< I | N >);
> AssignLabels(VertexSet(graph), IndexedSetToSequence(I));
> V := VertexSet(graph);
> E := EdgeSet(graph);
> for i in [1..#I] do
>     AssignLabels([ E![V | i, Index(I, m(I[i]*s))] : s in S], S);
> end for;
```

In this graph, [1,2,5,4] is a cycle. So the corresponding edge labels should multiply to the identity.

```
> &*Labels([ EdgeSet(graph) | [1,2], [2,5], [5,4], [4,1] ]);
a^4
> G;
Finitely presented group G on 2 generators
Relations
    b^2 = Id(G)
    a^4 = Id(G)
    (a^-1 * b)^3 = Id(G)
```

Example H150E7

We turn to multigraphs to illustrate a point with respect to the listing of the edge decorations. We assign random labels, capacities and weights to a multigraph.

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> E := EdgeSet(G);
> I := Indices(G);
>
> for i in I do
>     AssignLabel(~G, E.i, Random([ "a", "b", "c", "d" ]));
>     if not InitialVertex(E.i) eq TerminalVertex(E.i) then
>         AssignCapacity(~G, E.i, Random(1, 3));
>     end if;
>     AssignWeight(~G, E.i, Random(1, 3));
> end for;
>
> EdgeLabels(G);
[ c, undef, c, undef, d, undef, c, undef, c ]
> EdgeCapacities(G);
[ 2, undef, 3, undef, 2, undef, 0, undef, 3 ]
> EdgeWeights(G);
[ 3, undef, 2, undef, 1, undef, 2, undef, 1 ]
```

Since G is undirected, the edge $\{v, u\}$ and the edge $\{v, u\}$ are the same object, and thus they have the same index:

```
> V := VertexSet(G);
> u := V!1;
> v := V!3;
> Indices(u, v);
[ 3 ]
> Indices(v, u);
[ 3 ]
```

However, since G is represented by means of an adjacency list, the undirected edge $\{u, v\}$ is stored twice in the list, and so there are two positions in the list associated with the edge. By convention, these positions are contiguous, but, more importantly from the user's perspective, the function `Index` that returns the index of the edge $\{u, v\}$ *always* returns the odd index associated with the edge.
This explains why, for an undirected multigraph, the sequence returned by a function like `EdgeLabels` will always have undefined elements.
Finally note that the loop $\{2, 2\}$, which was assigned no capacity, is shown to have capacity zero:

```
> E := EdgeSet(G);
> E.7;
< {2, 2}, 7 >
> Capacity(E.7);
0
```

This is in accordance with the definition of the default value for edge capacity and weight (see Subsubsection 150.4.2.2).

150.5 Standard Construction for Multigraphs

As noted in the Introduction 150.1, most of the functions listed in this section correctly handle a graph's support and vertex/edge decorations. That is, these attributes are inherited by the graph created as a result of applying such a function.

150.5.1 Subgraphs

The construction of subgraphs from multigraphs or multidigraphs is similar to the construction of subgraphs from simple graphs or digraphs (see Subsection 149.6.1).

Note that the support set, vertex labels and edge decorations are transferred from the supergraph to the subgraph.

`sub< G | list >`

Construct the multigraph H as a subgraph of G. The function returns three values: The multigraph H, the vertex-set V of H; and the edge-set E of H. If G has a support set and/or if G has vertex/edge labels, and/or edge capacities or edge weights then *all* these attributes are transferred to the subgraph H.

The elements of V and of E are specified by the list *list* whose items can be objects of the following types:

(a) A vertex of G. The resulting subgraph will be the subgraph induced on the subset of `VertexSet(`G`)` defined by the vertices in *list*.

(b) An edge of G. The resulting subgraph will be the subgraph with vertex-set `VertexSet(`G`)` whose edge-set consists of the edges in *list*.

(c) A set of

(i) Vertices of G.

(ii) Edges of G.

If the list of vertices and edges happens to contain duplicate elements, they will be ignored by the subgraph constructor. It is easy to recover the map that sends the vertices of the subgraph to the vertices of the supergraph and vice-versa: Simply coerce the vertex of the subgraph into the supergraph's vertex-set, and, if applicable, coerce the vertex of the supergraph into the subgraph's vertex-set.

Example H150E8

We create a multidigraph G with a support set; we assign labels to its vertices, and labels, capacities and weights to its edges. This demonstrates the fact that any subgraph of G retains the support set, as well as the vertex and edge decorations.

```
> S := {@ "a", "b", "c" @};
> G := MultiDigraph< S | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multidigraph
Vertex  Neighbours
a       b c b ;
b       c b ;
c       ;
>
> V := VertexSet(G);
> for u in V do
>     AssignLabel(~G, u, Random([ "X", "Y", "Z" ]));
> end for;
>
> E := EdgeSet(G);
> I := Indices(G);
> for i in I do
>     AssignLabel(~G, E.i, Random([ "D", "E", "F" ]));
>     if not InitialVertex(E.i) eq TerminalVertex(E.i) then
>         AssignCapacity(~G, E.i, Random(1, 3));
>     end if;
>     AssignWeight(~G, E.i, Random(1, 3));
> end for;
>
> VertexLabels(G);
[ Z, Y, Z ]
> EdgeLabels(G);
[ E, F, D, E, E ]
> EdgeCapacities(G);
[ 2, 1, 3, 0, 2 ]
> EdgeWeights(G);
[ 2, 1, 3, 1, 1 ]
>
> V := VertexSet(G);
> H := sub< G | V!1, V!2 >;
> H;
Multidigraph
Vertex  Neighbours
a       b b ;
b       b ;
>
> for u in VertexSet(H) do
>     assert Label(u) eq Label(V!u);
```

```
> end for;
>
> for e in EdgeSet(H) do
>     u := InitialVertex(e);
>     v := TerminalVertex(e);
>
>     assert SequenceToSet(Labels(Edges(u, v)))
>     eq SequenceToSet(Labels(Edges(V!u, V!v)));
>     assert SequenceToSet(Capacities(Edges(u, v)))
>     eq SequenceToSet(Capacities(Edges(V!u, V!v)));
>     assert SequenceToSet(Weights(Edges(u, v)))
>     eq SequenceToSet(Weights(Edges(V!u, V!v)));
> end for;
```

Note that since G is a multidigraph it is not possible to coerce an edge of H into the edge-set of G. This is because edge coercion for multi(di)graphs involves the coercion of both the end-vertices *and* the index of the edge.
A correspondence is established between the edges of H and the edges of G by retrieving *all* the edges in H and G having same end-vertices.

150.5.2 Incremental Construction of Multigraphs

The complete functionality for adding and removing vertices or edges in simple graphs (see Subsection 149.6.2) is also available for multigraphs.

Note that some functions adding an edge to a multigraph also return the newly created edge. This feature is useful when there is a need to determine the index of this edge in the adjacency list of the modified multigraph. Also, there is the possibility of removing *all* the edges from u to v for given vertices u and v of the multigraphs.

Further, whenever a vertex or an edge is added or removed from a graph, the existing vertex labels and the edge labels or capacities or weights are retained. The support of the graph is retained in all cases except when adding a new vertex.

Unless otherwise specified, each of the functions described in this section returns three values:

(i) The multigraph G;

(ii) The vertex-set V of G;

(iii) The edge-set E of G.

150.5.2.1 Adding Vertices

`G + n`

Given a non-negative integer n, adds n new vertices to the multigraph G. The existing vertex labels and edge labels or capacities or weights are retained, but the support will become the standard support.

`G +:= n`

`AddVertex(~G)`

`AddVertices(~G, n)`

The procedural version of the previous function. `AddVertex` adds one vertex only to G.

`AddVertex(~G, l)`

Given a graph G and a label l, adds a new vertex with label l to G. The existing vertex labels and edge labels or capacities or weights are retained, but the support will become the standard support.

`AddVertices(~G, n, L)`

Given a graph G and a non-negative integer n, and a sequence L of n labels, adds n new vertices to G with labels from L. The existing vertex labels and edge labels or capacities or weights are retained, but the support will become the standard support.

150.5.2.2 Removing Vertices

`G - v`

`G - U`

Given a vertex v of G, or a set U of vertices of G, removes v (or the vertices in U) from G. The support, vertex labels, and edge labels or capacities or weights are retained.

`G -:= v`

`G -:= U`

`RemoveVertex(~G, v)`

`RemoveVertices(~G, U)`

The procedural versions of the previous functions.

150.5.2.3 Adding Edges

`G + { u, v }`

`G + [ u, v ]`

Given a graph G and a pair of vertices of G, add the edge to G described by this pairs. If G is undirected then the edge must be given as a set of (two) vertices, if G is directed the edge is given as a sequence of (two) vertices. If G is a network, then the edge is added with a capacity of 1 (0 if a loop). The support, vertex labels, and edge labels or capacities or weights are retained. This set of functions has two return values: The first is the modified graph and the second is the newly created edge. This feature is especially useful when adding parallel edges.

```
G + { { u, v } }
G + [ { u, v } ]
G + { [ u, v ] }
G + [ [ u, v ] ]
```

Given a graph G and a set or a sequence of pairs of vertices of G, add the edges to G described by these pairs. If G is undirected then the edges must be given as a set of (two) vertices, if G is directed the edges are given as a sequence of (two) vertices. If G is a network, then the edges are added with a capacity of 1 (0 if a loop). The support, vertex labels, and edge labels or capacities or weights are retained.

```
G +:= { u, v }
G +:= [ u, v ]
G +:= [ u, v ]
G +:= { { u, v } }
G +:= [ { u, v } ]
G +:= { [ u, v ] }
G +:= [ [ u, v ] ]
```

The procedural versions of the previous four functions.

```
AddEdge(G, u, v)
```

Given a graph G, two vertices of G u and v, returns a new edge between u and v. If G is a network, the edge is added with a capacity of 1 (0 if loop). The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge. This feature is especially useful when adding parallel edges.

```
AddEdge(G, u, v, l)
```

Given a graph G which is not a network, two vertices of G u and v, and a label l, adds a new edge with label l between u and v. The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

```
AddEdge(G, u, v, c)
```

Given a network G, two vertices of G u and v, and a non-negative integer c, adds a new edge from u to v with capacity c. The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

`AddEdge(G, u, v, c, l)`

Given a network G, two vertices of G u and v, a non-negative integer c, and a label l, adds a new edge from u to v with capacity c and label l. If G is a network, then add a new edge with label l and capacity c between u and v. The support, vertex labels, and edge labels or capacities or weights are retained. This function has two return values: The first is the modified graph and the second is the newly created edge.

`AddEdge(~G, u, v)`

`AddEdge(~G, u, v, l)`

`AddEdge(~G, u, v, c)`

`AddEdge(~G, u, v, c, l)`

Procedural versions of the previous functions for adding edges to a graph.

`AddEdges(G, S)`

Given a graph G, and a sequence or set S of pairs of vertices of G, the edges specified in S are added to G. The elements of S must be sets or sequences of two vertices of G, depending upon whether G is undirected or directed respectively.

If G is a network, the edges are added with a capacity of 1 (0 if a loop). The support, vertex labels, and edge labels or capacities or weights are retained.

`AddEdges(G, S, L)`

Given a graph G, a sequence S of pairs of vertices of G, and a sequence L of labels of the same length, the edges specified in S are added to G with its corresponding label as given in L. The elements of S must be sets or sequences of two vertices of G, depending upon whether G is undirected or directed respectively. If G is a network, the edges are added with a capacity of 1 (0 if loop). The support, vertex labels, and edge labels or capacities or weights are retained.

`AddEdges(~G, S)`

`AddEdges(~G, S, L)`

These are procedural versions of the previous functions for adding edges to a graph.

150.5.2.4 Removing Edges

`G - e`

`G - { e }`

Given an edge e or a set S of edges of a multigraph G, this function creates a graph that corresponds to G with the edge e (respectively, set of edges S) removed. The resulting multigraph will have vertex-set $V(G)$ and edge-set $E(G) \setminus \{e\}$ (respectively, $E(G) \setminus S$). The support, vertex labels and edge labels are retained on the remaining edges.

```
G - { { u, v } }
```

```
G - { [u, v] }
```

Given a graph G and a set S of pairs $\{ u, v\}$ or $[u, v]$, u, v vertices of G, this function forms the graph having vertex-set $V(G)$ and edge-set $E(G) - S$. That is, the graph returned is the same as G except that *all* the edges specified by pairs in S have been removed. An edge is represented as a set if G is undirected, and as a sequence otherwise. The support, vertex labels and edge labels are retained on the remaining edges.

```
G -:= e
```

```
G -:= { e }
```

```
G -:= { { u, v } }
```

```
G -:= { [u, v] }
```

```
RemoveEdge(~G, e)
```

```
RemoveEdges(~G, S)
```

```
RemoveEdge(~G, u, v)
```

These operations represent procedural versions of the previous functions. Whenever an edge is represented as a pair $\{u, v\}$ or $[u, v]$ of vertices of G, it is assumed that *all* the edges from u to v are to be removed from G.

150.5.3 Vertex Insertion, Contraction

As in the case of simple graphs (see Subsection 149.6.3) it is possible to insert a vertex in a multigraph edge. The new edges thus created will be unlabelled with their capacities and weights set as follows: (These rules apply regardless as to whether the edge-set is capacitated and/or weighted).

Let e be an edge from u to v with capacity c and weight w in a multigraph G. After the insertion of a new vertex x in e, the edge e will be replaced by two edges, one from u to x and the other from x to v, *both* having capacity c and weight w.

These rules apply regardless as to whether the edge-set is capacitated and/or weighted.

The contraction operation can only be applied to a pair of vertices, since contracting a single multigraph edge which might have parallel edges is meaningless. The graph's support and vertex/edges decorations are retained when contracting its edges.

Each of the functions described below returns three values:

(i) The multigraph G;

(ii) The vertex-set V of G;

(iii) The edge-set E of G.

`InsertVertex(e)`

Given an edge e of the multigraph G, this function inserts a new vertex of degree 2 in e. If appropriate, the two new edges that replace e will have the same capacity and weight as e. They will be unlabelled. The vertex labels and the edge decorations of G are retained, but the resulting graph will have standard support.

`InsertVertex(T)`

Given a set T of edges belonging to the multigraph G, this function inserts a vertex of degree 2 in each edge belonging to the set T.

`Contract(e)`

Given an edge $e = \{u, v\}$ of the graph G, form the graph obtained by removing the edge e and then identifying the vertices u and v. New parallel edges and new loops may result from this operation but any new loop will be assigned zero capacity. With the above exception, the edge decorations are retained, as are the support and the vertex labels of G.

`Contract(u, v)`

Given vertices u and v belonging to the multigraph G, this function returns the multigraph obtained by identifying the vertices u and v. New parallel edges and new loops may result from this operation but any new loop will be assigned zero capacity. With the above exception, the edge decorations are retained, as are the support and the vertex labels of G.

`Contract(S)`

Given a set S of vertices belonging to the multigraph G, this function returns the multigraph obtained by identifying all of the vertices in S.

150.5.4 Unions of Multigraphs

Of the union operations available for simple graphs (see Section 149.7) only `Union` and `EdgeUnion` have been implemented for multigraphs. It is straightforward to write MAGMA code for other union functions with multigraphs.

In contrast with the other standard graph constructions, the support, vertex labels and edge decorations are generally *not* handled by the functions listed below. Thus, the resulting graph will always have standard support and no vertex labels nor edge decorations. The one exception occurs in the case of networks, where edge capacities are properly handled.

`Union(G, H)`

`G join H`

Given multi(di)graphs G and H with disjoint vertex sets $V(G)$ and $V(H)$ respectively, this function constructs their union, i.e. the multi(di)graph with vertex-set $V(G) \cup V(H)$, and edge-set $E(G) \cup E(H)$. The resulting multi(di)graph has standard support and no vertex labels nor edge decorations.

`Union(N, H)`

`N join H`

Given networks N and H with disjoint vertex sets $V(N)$ and (H) respectively, construct their union, i.e. the network with vertex-set $V(N) \cup V(H)$, and edge-set $E(N) \cup E(H)$. The resulting network has standard support and capacities but neither vertex labels, edge labels nor weights retained.

`& join S`

The union of the multigraphs or networks in the sequence or the set S.

`EdgeUnion(G, H)`

Given multi(di)graphs G and H having the same number of vertices, construct their edge union K. This construction identifies the i-th vertex of G with the i-th vertex of H for all i. The edge union has the same vertex-set as G (and hence as H) and there is an edge from u to v in K if and only if there is an edge from u to v in either G or H. The resulting multi(di)graph has standard support but neither vertex labels nor edge decorations.

`EdgeUnion(N, H)`

Given networks N and H having the same number of vertices, construct their edge union K. This construction identifies the i-th vertex of N with the i-th vertex of H for all i. The edge union has the same vertex-set as N (and hence as H) and there is an edge $[u, v]$ with capacity c in K if and only if either there is an edge $[u, v]$ with capacity c in N or if there is an edge $[u, v]$ with capacity c in H. The resulting network has standard support and the inherited edge capacities but neither vertex labels, edge labels nor weights.

150.6 Conversion Functions

Conversion functions do not preserve a graph's support and vertex/edge decorations. That is, the resulting graph has standard support and no vertex/edge decorations. A slight exception to this rule occurs when the resulting graph is a network (of type `GrphNet`) and is described in detail in `UnderlyingNetwork`.

150.6.1 Orientated Graphs

The rules followed in building an orientated graph from an undirected graph are the same as those described for simple graphs (see Section 149.8).

`OrientatedGraph(G)`

Given a multigraph G, produce a multidigraph D whose vertex-set is the same as that of G and whose edge-set consists of the edges of G, each given a direction. The edges of D are always directed from the lower numbered vertex to the higher numbered vertex. Thus, if G contains the edge $\{u, v\}$, then D will have the edge $[u, v]$ if $u < v$, otherwise the edge $[u, v]$. If G has a loop at u, then D will have a directed loop at u.

150.6.2 Converse

`Converse(G)`

Given a multidigraph G with edge-set E, produce a multidigraph D whose vertex-set is the same as that of G and whose edge-set is $\{[u, v] : [v, u] \in E\}$.

150.6.3 Converting between Simple Graphs and Multigraphs

Any simple (di)graph can be converted into a multi(di)graph and any multi(di)graph can be converted into a simple (di)graph. The resulting graph has standard support and neither vertex labels nor edge decorations, unless it is a network, in which case all the edges in the resulting graph are assigned a capacity of 1 (0 if loops).

Let G be a graph and e an edge of G from u to v and let H be the graph resulting from the conversion. If G and H are both undirected or both directed then e is also an edge of H. If G is undirected while H is undirected then both edges $[u, v]$ and $[v, u]$ are edges of H. If G is directed while H is directed then the edge $\{u, v\}$ is an edge of H.

Since these conversion functions do not retain the original graph's support and vertex/edge decorations, they may also be used when requiring a copy of a graph G without G's support and vertex/edge decorations.

`UnderlyingGraph(G)`

The underlying simple graph of the graph G. The support and vertex/edge decorations of G are not retained.

`UnderlyingDigraph(G)`

The underlying simple digraph of the graph G. The support and vertex/edge decorations of G are not retained.

`UnderlyingMultiGraph(G)`

The underlying multigraph of the graph G. The support and vertex/edge decorations of G are not retained.

`UnderlyingMultiDigraph(G)`

The underlying multidigraph of the graph G. The support and vertex/edge decorations of G are not retained.

`UnderlyingNetwork(G)`

The underlying network of the graph G. The support and vertex/edge decorations of G are not retained except when G is a network in which case only the edge capacities are retained. If G is not a network, then all the edge capacities are set to 1 (0 for loops).

150.7 Elementary Invariants and Predicates for Multigraphs

Most but not all of the invariants and predicates that apply to simple graphs (see Sections 149.10 and 149.11) also apply to multigraphs. We list them below.

Let G and H be two graphs. For clarity, we list here once again the conditions under which G is equal to H and H is a subgraph of G.

The graphs G and H are equal if and only if:

- they are of the same type,
- they are structurally identical,
- they have the same support,
- they have identical vertex and edge labels,
- if applicable, the total capacity from u to v in G is equal to the total capacity from u to v in H.

Also, H is a subgraph of G if and only if:

- they are of the same type,
- H is a structural subgraph of G,
- any vertex `v` in H has the same support as the vertex `VertexSet(G)!v` in G,
- any vertex `v` in H has the same label as the vertex `VertexSet(G)!v` in G,
- any edge `e` in H has the same label as the edge `EdgeSet(G)!e` in G,
- if applicable, the total capacity from u to v in G is at least as large as the total capacity from u to v in H.

Note that the truth value of the above two tests is not dependent on the weights of the edges of the graphs, should these edges be weighted.

Finally, we have introduced a few predicates to help users determine if a general graph is simple or not, undirected or not.

`Order(G)`

`NumberOfVertices(G)`

The number of vertices of the graph G.

`Size(G)`

`NumberOfEdges(G)`

The number of edges of the graph G.

`u adj v`

Let u and v be two vertices of the same graph G. If G is undirected, returns `true` if and only if u and v are adjacent. If G is directed, returns `true` if and only if there is an edge directed from u to v.

`e adj f`

Let e and f be two edges of the same graph G. If G is undirected, returns `true` if and only if e and f share a common vertex. If G is directed, returns `true` if and only if the terminal vertex of e (f) is the initial vertex of f (e).

`u notadj v`

The negation of the `adj` predicate applied to vertices.

`e notadj f`

The negation of the `adj` predicate applied to edges.

`u in e`

Let u be a vertex and e an edge of a graph G. Returns `true` if and only if u is an end-vertex of e.

`u notin e`

The negation of the `in` predicate applied to a vertex with respect to an edge.

`G eq H`

Returns `true` if and only if the graphs G and H are equal, that is if and only if they are structurally equal and are compatible with respect to their support, vertex and edge labels, and edge capacities (see the introduction to this section).

`IsSubgraph(G, H)`

Returns `true` if and only if H is a subgraph of G, that is, if and only if H is a structural subgraph of G and the graphs are compatible with respect to their support, vertex and edge labels, and edge capacities (see the introduction to this section).

`IsBipartite(G)`

Returns `true` if and only if the graph G is bipartite.

`Bipartition(G)`

Given a bipartite graph G, return its two partite sets in the form of a pair of subsets of $V(G)$.

`IsRegular(G)`

Returns `true` if and only if G is a regular graph.

`IsComplete(G)`

Returns `true` if and only if the graph G, on n vertices, is the complete graph on n vertices.

`IsEmpty(G)`

Returns `true` if and only if the edge-set of the graph is empty.

`IsNull(G)`

Returns `true` if and only if the vertex-set of the graph is empty.

`IsSimple(G)`

Returns `true` if and only if G is a simple graph.

`IsUndirected(G)`

Returns `true` if and only if G is a undirected graph.

`IsDirected(G)`

Returns `true` if and only if G is a directed graph.

150.8 Adjacency and Degree

The adjacency and degree functionalities that apply to simple graphs (see 149.12) similarly apply to multigraphs.

150.8.1 Adjacency and Degree Functions for Multigraphs

`Degree(u)`

Given a vertex u of a graph G, return the degree of u, ie the number of edges incident to u.

`Alldeg(G, n)`

Given a multigraph G, and a non-negative integer n, return the set of all vertices of G that have degree equal to n.

`MaximumDegree(G)`

`Maxdeg(G)`

The maximum of the degrees of the vertices of the multigraph G. This function returns two values: the maximum degree, and a vertex of G having that degree.

`MinimumDegree(G)`

`Mindeg(G)`

The minimum of the degrees of the vertices of the multigraph G. This function returns two values: the minimum degree, and a vertex of G having that degree.

`DegreeSequence(G)`

Given a multigraph G such that the maximum degree of any vertex of G is r, return a sequence D of length $r+1$, such that $D[i]$, $1 \leq i \leq r+1$, is the number of vertices in G having degree $i-1$.

`Neighbours(u)`

`Neighbors(u)`

Given a vertex u of a graph G, return the set of vertices of G that are adjacent to u.

`IncidentEdges(u)`

Given a vertex u of a graph G, return the set of all edges incident with the vertex u.

150.8.2 Adjacency and Degree Functions for Multidigraphs

`InDegree(u)`

The number of edges directed into the vertex u belonging to a multidigraph.

`OutDegree(u)`

The number of edges of the form $[u, v]$ where u is a vertex belonging to a multidigraph.

`MaximumInDegree(G)`

`Maxindeg(G)`

The maximum indegree of the vertices of the multidigraph G. This function returns two values: the maximum indegree, and the first vertex of G having that degree.

`MinimumInDegree(G)`

`Minindeg(G)`

The minimum indegree of the vertices of the multidigraph G. This function returns two values: the minimum indegree, and the first vertex of G having that degree.

`MaximumOutDegree(G)`

`Maxoutdeg(G)`

The maximum outdegree of the vertices of the multidigraph G. This function returns two values: the maximum outdegree, and the first vertex of G having that degree.

`MinimumOutDegree(G)`

`Minoutdeg(G)`

The minimum outdegree of the vertices of the multidigraph G. This function returns two values: the minimum outdegree, and the first vertex of G having that degree.

`Degree(u)`

Given a vertex u belonging to the multidigraph G, return the total degree of u, i.e. the sum of the in–degree and out–degree for u.

`MaximumDegree(G)`

`Maxdeg(G)`

The maximum total degree of the vertices of the multidigraph G. This function returns two values: the maximum total degree, and the first vertex of G having that degree.

`MinimumDegree(G)`

`Mindeg(G)`

The minimum total degree of the vertices of the multidigraph G. This function returns two values: the minimum total degree, and the first vertex of G having that degree.

`Alldeg(G, n)`

Given a multidigraph G, and a non–negative integer n, return the set of all vertices of G that have total degree equal to n.

`DegreeSequence(G)`

Given a multidigraph G such that the maximum degree of any vertex of G is r, return a sequence D of length $r + 1$, such that $D[i]$, $1 \leq i \leq r + 1$, is the number of vertices in G having degree $i - 1$.

`InNeighbours(u)`

`InNeighbors(u)`

Given a vertex u of a multidigraph G, return the set containing all vertices v such that $[v, u]$ is an edge in G, i.e. the initial vertex of all edges that are directed into the vertex u.

`OutNeighbours(u)`

`OutNeighbors(u)`

Given a vertex u of the multidigraph G, return the set of vertices v of G such that $[u, v]$ is an edge in G, i.e. the set of vertices v that are terminal vertices of edges directed from u to v.

`IncidentEdges(u)`

Given a vertex u of a graph G, return the set of all edges incident with the vertex u, that is, the set of all edges incident into u and incident from u.

150.9 Connectedness

All the functions relating to connectivity issues are the same for simple graphs and multigraphs. See Section 149.13.1 and its subsections for specific details about the algorithms underlying these functions.

150.9.1 Connectedness in a Multigraph

`IsConnected(G)`

Returns `true` if and only if the undirected graph G is a connected graph.

`Components(G)`

The connected components of the undirected graph G. These are returned in the form of a sequence of subsets of the vertex-set of G.

`Component(u)`

The subgraph corresponding to the connected component of the multigraph G containing vertex u. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

`IsSeparable(G)`

Returns `true` if and only if the graph G is connected and has at least one cut vertex.

`IsBiconnected(G)`

Returns `true` if and only if the graph G is biconnected. The graph G must be undirected.

`CutVertices(G)`

The set of cut vertices for the connected undirected graph G (as a set of vertices).

`Bicomponents(G)`

The biconnected components of the undirected graph G. These are returned in the form of a sequence of subsets of the vertex-set of G. The graph may be disconnected.

150.9.2 Connectedness in a Multidigraph

`IsStronglyConnected(G)`

Returns `true` if and only if the multidigraph G is strongly connected.

`IsWeaklyConnected(G)`

Returns `true` if and only if the multidigraph G is weakly connected.

`StronglyConnectedComponents(G)`

The strongly connected components of the multidigraph G. These are returned in the form of a sequence of subsets of the vertex-set of G.

`Component(u)`

The subgraph corresponding to the connected component of the multidigraph G containing vertex u. The support and vertex/edge decorations are *not* retained in the resulting (structural) subgraph.

150.9.3 Triconnectivity for Multigraphs

We refer the reader to Subsection 149.13.3 of the graphs chapter for details about the triconnectivity algorithm implemented here, and especially about the meaning of the `Splitcomponents` function. The triconnectivity algorithm applies to undirected graphs only.

`IsTriconnected(G)`

Returns `true` if and only if G is triconnected. The graph G must be undirected.

`Splitcomponents(G)`

The split components of the undirected graph G. These are returned in the form of a sequence of subsets of the vertex-set of G. The graph may be disconnected. The second return value returns the cut vertices and the separation pairs as sequences of one or two vertices respectively.

`SeparationVertices(G)`

The cut vertices and/or the separation pairs of the undirected graph G as a sequence of sequences of one and/or two vertices respectively. The graph may be disconnected. The second return value returns the split components of G .

150.9.4 Maximum Matching in Bipartite Multigraphs

We refer the reader to Subsection 149.13.4 for information about the maximum matching algorithm.

`MaximumMatching(G : parameters)`

`Al` MonStg *Default : "PushRelabel"*

A maximum matching in the bipartite graph G. The matching is returned as a sequence of edges of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

150.9.5 General Vertex and Edge Connectivity in Multigraphs and Multidigraphs

See Subsection 149.13.5 for details about the algorithms underlying the functions listed below. These functions apply to both undirected and directed graphs.

`VertexSeparator(G : parameters)`

`Al` MonStg *Default : "PushRelabel"*

If G is an undirected graph, a *vertex separator* for G is a smallest set of vertices S such that for any u, $v \in V(G)$, every path connecting u and v passes through at least one vertex of S.

If G is a directed graph, a vertex separator for G is a smallest set of vertices S such that for any u, $v \in V(G)$, every directed path from u to v passes through at least one vertex of S.

`VertexSeparator` returns the vertex separator of G as a sequence of vertices of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`VertexConnectivity(G :` *parameters*`)`

`Al` MonStg *Default :* "*PushRelabel*"

Returns the vertex connectivity of the graph G, the size of a minimum vertex separator of G. Also returns a vertex separator for G as the second value. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`IsKVertexConnected(G, k :` *parameters*`)`

`Al` MonStg *Default :* "*PushRelabel*"

Returns `true` if the vertex connectivity of the graph G is at least k, `false` otherwise. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`EdgeSeparator(G :` *parameters*`)`

`Al` MonStg *Default :* "*PushRelabel*"

If G is an undirected graph, an *edge separator* for G is a smallest set of edges T such that for any u, $v \in V(G)$, every path connecting u and v passes through at least one edge of T.

If G is a directed graph, an edge separator for G is a smallest set of edges T such that for any u, $v \in V(G)$, every directed path from u to v passes through at least one edge of T.

`EdgeSeparator` returns the edge separator of G as a sequence of edges of G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`EdgeConnectivity(G :` *parameters*`)`

`Al` MonStg *Default :* "*PushRelabel*"

Returns the edge connectivity of the graph G, the size of a minimum edge separator of G. Also returns as the second value an edge separator for G. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`IsKEdgeConnected(G, k :` *parameters*`)`

`Al` MonStg *Default :* "*PushRelabel*"

Returns `true` if the edge connectivity of the graph G is at least k, `false` otherwise. The parameter `Al` enables the user to select the algorithm which is to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

Example H150E9

We demonstrate that the connectivity functions deal properly with multiple edges.

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2, 3} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours
1       3 2 3 2 ;
2       3 2 2 1 1 ;
3       2 1 1 ;
> EdgeConnectivity(G);
3
> EdgeSeparator(G);
[ < {3, 2}, 11 >, < {1, 2}, 5 >, < {1, 2}, 1 > ]
```

150.10 Spanning Trees

All the trees returned by the functions described below are returned as structural subgraphs of the original graph whose support and vertex and edge decorations are not retained in the resulting tree.

`SpanningTree(G)`

Given a connected undirected graph G, construct a spanning tree for G rooted at an arbitrary vertex of G. The spanning tree is returned as a structural subgraph of G, without the support, vertex or edge decorations of G.

`SpanningForest(G)`

Given a graph G, construct a spanning forest for G. The forest is returned as a structural subgraph of G, without the support, vertex or edge decorations of G.

`BreadthFirstSearchTree(u)`

`BFSTree(u)`

Given a vertex u belonging to the graph G, return a breadth-first search for G rooted at the vertex u. The tree is returned as a structural subgraph of G, without the support, vertex or edge decorations of G. Note that G may be disconnected.

`DepthFirstSearchTree(u)`

`DFSTree(u)`

Given a vertex u belonging to the graph G, return a depth-first search tree T for G rooted at the vertex u. The tree T is returned as a structural subgraph of G, without the support, vertex or edge decorations of G. Note that G may be disconnected.

The fourth return argument returns, for each vertex u of G, the tree order of u, that is, the order in which the vertex u has been visited while performing the depth-first search. If T does not span G then the vertices of G not in T are given tree order from `Order(T) + 1` to `Order(G)`.

150.11 Planar Graphs

The planarity algorithm implemented in MAGMA tests whether an undirected graph or multigraph is planar. If the graph is planar, then an embedding of the graph is produced, otherwise a Kuratowski subgraph is identified. For a thorough discussion of this algorithm, its implementation and complexity, the reader is referred to Section 149.21.

`IsPlanar(G)`

Tests whether the (undirected) graph G is planar. The graph may be disconnected. If the graph is non-planar then a Kuratowski subgraph of G is returned: That is, a subgraph of G homeomorphic to K_5 or $K_{3,3}$. The support and vertex/edge decorations of G are *not* retained in this (structural) subgraph.

`Obstruction(G)`

Returns a Kuratoswki obstruction if the graph is non-planar, or the empty graph if the graph is planar. The Kuratoswki graph is returned as a (structural) subgraph of G; the support and vertex/edge decorations are not retained.

`IsHomeomorphic(G:` *parameters*`)`

`Graph`	MONSTG	*Default :*

Tests if a graph is homeomorphic to either K_5 or $K_{3,3}$. The parameter `Graph` must be set to either "K5" or "K33"; it has no default setting.

`Faces(G)`

Returns the faces of the planar graph G as sequences of the edges bordering the faces of G. If G is disconnected, then the face defined by an isolated vertex v is given as $[v]$.

`Face(u, v)`

Returns the face of the planar graph G bordered by the directed edge $[u, v]$ as an ordered list of edges of G.

Note that a directed edge and an orientation determine a face uniquely: We can assume without loss of generality that the plane is given a clockwise orientation. Then given a directed edge $e = [u_1, v_1]$, the face defined by e is the ordered set of edges $[u_1, v_1], [u_2, v_2], \ldots, [u_m, v_m]$ such that $v_i = u_{i+1}$ for all i, $1 \leq i < m$, $v_m = u_1$, and for each $v_i = u_{i+1}$, the neighbours of v_i, u_i and v_{i+1}, are *consecutive* vertices in v_i's adjacency list whose order is *anti-clockwise*.

`Face(e)`

Let e be the edge u, v of the planar graph G (recall that G is undirected). Then `Face(`u, v`)` returns the face bordered by the directed edge $[u, v]$ as a sequence of edges of G.

`NFaces(G)`

`NumberOfFaces(G)`

Returns the number of faces of the planar graph G. In the case of a disconnected graph, an isolated vertex counts for one face.

`Embedding(G)`

Returns the planar embedding of the graph G as a sequence S where $S[i]$ is a sequence of edges incident from vertex i.

`Embedding(v)`

Returns the ordered list of edges (in clockwise order say) incident from vertex v.

Example H150E10

The purpose of the example is to show the embedding and faces of a graph with multiples edges and loops.

```
> G := MultiGraph< 3 | < 1, {2, 3} >, < 1, {2} >, < 2, {2, 3} > >;
> G;
Multigraph
Vertex  Neighbours
1       2 3 2 ;
2       3 2 2 1 1 ;
3       2 1 ;
> IsPlanar(G);
true
> Faces(G);
[
  [ < {1, 2}, 5 >, < {2, 1}, 1 > ],
  [ < {1, 2}, 1 >, < {2, 2}, 7 >, < {2, 3}, 9 >, < {3, 1}, 3 > ],
  [ < {1, 3}, 3 >, < {3, 2}, 9 >, < {2, 1}, 5 > ],
```

```
  [ < {2, 2}, 7 > ]
]
> Embedding(G);
[
  [ < {1, 2}, 5 >, < {1, 2}, 1 >, < {1, 3}, 3 > ],
  [ < {2, 3}, 9 >, < {2, 2}, 7 >, < {2, 1}, 1 >, < {2, 1}, 5 > ],
  [ < {3, 1}, 3 >, < {3, 2}, 9 > ]
]
```

Example H150E11

We show how to construct the dual graph D of a planar graph G and how to find all its minimal cuts. The vertex set of the dual is the set of faces F of G where face f_i is adjacent to face f_2 if and only if f_1 and f_2 share a common edge in G. For the purpose of this example a cut of a graph G is defined as a set of edges which disconnects G.
Let us construct a small planar graph G and its dual D. For clarity, the support of D will be the standard support (we could have chosen it to be the set of faces of G).

```
> G := MultiGraph< 4 | {1, 2}, {1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4}>;
> IsPlanar(G);
true
> Faces(G);
[
  [ < {1, 3}, 5 >, < {3, 2}, 7 >, < {2, 1}, 3 > ],
  [ < {1, 2}, 3 >, < {2, 1}, 1 > ],
  [ < {1, 2}, 1 >, < {2, 4}, 9 >, < {4, 3}, 11 >, < {3, 1}, 5 > ],
  [ < {3, 4}, 11 >, < {4, 2}, 9 >, < {2, 3}, 7 > ]
]
> F := {@ SequenceToSet(f) : f in Faces(G) @};
> D := MultiGraph< #F | >;
> mapG2D := [ 0 : i in [1..Max(Indices(G))] ];
> mapD2G := [ 0 : i in [1..Max(Indices(G))] ];
> for u in VertexSet(D) do
>     for v in VertexSet(D) do
>        if Index(v) le Index(u) then
>           continue;
>        end if;
>        M := F[ Index(u) ] meet F[ Index(v) ];
>        for e in M do
>           D, edge :=
>           AddEdge(D, VertexSet(D)!u, VertexSet(D)!v);
>
>           mapG2D[Index(e)] := Index(edge);
>           mapD2G[Index(edge)] := Index(e);
>        end for;
>     end for;
> end for;
>
```

```
> e_star := map< EdgeSet(G) -> EdgeSet(D) |
> x :-> EdgeSet(D).mapG2D[Index(x)],
> y :-> EdgeSet(G).mapD2G[Index(y)] >;
```

The map `e-star` is the bijection from G's edge-set into D's edge-set:

```
> for e in EdgeSet(G) do
>     e, "   ", e @ e_star;
> end for;
< {1, 3}, 5 >     < {1, 3}, 3 >
< {1, 2}, 3 >     < {1, 2}, 1 >
< {1, 2}, 1 >     < {2, 3}, 7 >
< {2, 4}, 9 >     < {3, 4}, 11 >
< {2, 3}, 7 >     < {1, 4}, 5 >
< {3, 4}, 11 >     < {3, 4}, 9 >
>
> for e in EdgeSet(D) do
>     e, "   ", e @@ e_star;
> end for;
< {1, 4}, 5 >     < {2, 3}, 7 >
< {1, 3}, 3 >     < {1, 3}, 5 >
< {1, 2}, 1 >     < {1, 2}, 3 >
< {2, 3}, 7 >     < {1, 2}, 1 >
< {3, 4}, 11 >     < {2, 4}, 9 >
< {3, 4}, 9 >     < {3, 4}, 11 >
```

If G is biconnected, then any of its faces is bounded by a cycle. From Euler's formula giving the number of faces in a graph, we deduce that the boundaries of the internal faces of G, which form a chordless cycle, form a basis for the cycle space of G.
It is a well-known fact that, if G is connected and planar, a set of edges E is the set of edges of a cycle in G if and only if $E^* = \{e^* : e \in E\}$ is a minimal cut in D. For more details, see [Die00, $ 4].
From this we conclude that we can compute the minimal cuts generating the cut space of the dual graph D. We verify that G is biconnected, we compute the cut corresponding to each face of G, and verify that it is a minimal cut. All the cuts together form a generating set of the cut space of D. Had we not included the cut corresponding to the external face of G, we would have a basis of the cut space.

```
> IsBiconnected(G);
true
> for f in F do
>     Cut := { e @ e_star : e in f };
>     H := D;
>     RemoveEdges(~H, Cut);
>     assert not IsConnected(H);
>
>     for e in Cut do
>        C := Exclude(Cut, e);
>        H := D;
>        RemoveEdges(~H, C);
```

```
>         assert IsConnected(H);
>     end for;
> end for;
```

150.12 Distances, Shortest Paths and Minimum Weight Trees

Two standard algorithms have been implemented for finding single-source shortest paths in weighted graphs. One is Dijsktra's algorithm for graphs without negative weight cycles, the second is Bellman-Ford's for those graphs with negative weight cycles. Dijsktra's algorithm is implemented either by a priority queue (binary heap) or a Fibonacci heap. The Fibonacci heap is asymptotically faster for sparse graphs. But for most practical purposes (graphs of small order) the binary heap outperforms the Fibonacci heap and is therefore chosen as the default data structure wherever relevant.

Johnson's algorithm has been chosen for the all-pairs shortest paths computation. Indeed, it outperforms the simpler Floyd's algorithm, especially as the graphs get larger.

Finally, Prim's algorithm is used to implement the minimum weight tree computation for undirected graphs. (The tree is a spanning tree if and only if the graph is connected.)

All the functions described below apply to general graphs whose edges are assigned a weight. If the graph under consideration is not weighted, then all its edges are assumed to have weight one. To assign weights to the edges of a graph, see Subsection 150.4.2. Note that all the functions described below accept negatively weighted edges.

`Reachable(u, v : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Return `true` if and only there is a path from vertex u to vertex v. If true, also returns the distance between u and v.

`Distance(u, v : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Given vertices u and v in a graph G, computes the distance from u to v. Results in an error if there is no path in G from u to v.

`Distances(u : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Given a vertex u in a graph G, computes the sequence D of distances from u to v, v any vertex in G. Given any vertex v in G, let i be `Index(v)`. Then $D[i]$, if defined, is the distance from u to v. If there is no path from u to v, then the sequence element $D[i]$ is undefined.

`PathExists(u, v : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Return `true` if and only there is a path from vertex u to vertex v in the parent graph G. If so, also returns a shortest path from u to v as a sequence of edges of G.

`Path(u, v : parameters)`

`ShortestPath(u, v : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Given vertices u and v in a graph G, computes a shortest path from u to v as a sequence of edges of G. Results in an error if there is no path in G from u to v.

`Paths(u : parameters)`

`ShortestPaths(u : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Given a vertex u in a graph G, computes the sequence P of shortest paths from u to v, for every vertex v in G. Given any vertex v in G, let i be `Index(v)`. If there exists a path from u to v then $P[i]$ is a sequence of edges giving a shortest path from u to v. If there is no path from u to v, then the sequence element $P[i]$ is undefined.

`GeodesicExists(u, v : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Return `true` if and only there is a path from vertex u to vertex v in the parent graph G. If true, also returns a shortest path from u to v as a sequence of vertices of G.

`Geodesic(u, v : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Given vertices u and v in a graph G, this function computes a shortest path from u to v as a sequence of vertices of G. An error results if there is no path in G from u to v.

`Geodesics(u : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Given a vertex u in a graph G, this function computes the sequence P of shortest paths from u to v, for every vertex v in G. Given any vertex v in G, let i be `Index(v)`. If there exists a path from u to v then $P[i]$ is a sequence of vertices specifying a shortest path from u to v. If there is no path from u to v, then the sequence element $P[i]$ is undefined.

`HasNegativeWeightCycle(u : parameters)`

Return `true` if and only if there is a negative-weight cycle reachable from vertex u.

`HasNegativeWeightCycle(G)`

Return `true` if and only if the graph G has negative-weight cycles.

`AllPairsShortestPaths(G : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Computes the all-pairs shortest paths. Let u and v be two vertices of a graph G and let $i = Index(u)$ and $j = Index(v)$. Let S_1 and S_2 be the two sequences returned by `AllPairsShortestPaths`, and let $s_1 = S_1[i]$ and $s_2 = S_2[i]$. Then $s_1[j]$, if defined, gives the distance from u to v and $s_2[j]$, if defined, gives the vertex preceding v in the shortest path from u to v. An error results if the graph G has a negative-weight cycle.

`MinimumWeightTree(u : parameters)`

`UseFibonacciHeap` BOOL *Default :* `false`

Returns a minimum weight tree rooted at vertex u, u any vertex of an *undirected* graph G, as a subgraph of G. The tree spans G if and only if G is connected. The support of G as well as the vertex and edge decorations in G *are* transferred to the tree.

Example H150E12

We create a weighted multidigraph.

```
> G := MultiDigraph< 5 | [1, 2], [1, 2], [1, 3], [2, 4], [3, 5], [3, 4], [4, 5] >;
> E := EdgeSet(G);
> AssignWeight(~G, E.1, 1);
> AssignWeight(~G, E.2, 5);
> AssignWeight(~G, E.3, 10);
> AssignWeight(~G, E.4, 1);
> AssignWeight(~G, E.5, -5);
> AssignWeight(~G, E.6, 1);
> AssignWeight(~G, E.7, 2);
>
>
> V := VertexSet(G);
> E := EdgeSet(G);
> for e in E do
>     e, "  ", Weight(e);
> end for;
< [1, 3], 3 >    10
< [1, 2], 2 >    5
< [1, 2], 1 >    1
< [2, 4], 4 >    1
< [3, 4], 6 >    1
< [3, 5], 5 >    -5
```

```
< [4, 5], 7 >    2
```

We verify that it has no negative weight cycle reachable from vertex 1, and that there is a path from vertex 1 to vertex 5.

```
> HasNegativeWeightCycle(V!1);
false
> b, d := Reachable(V!1, V!5);
> assert b;
> P := Path(V!1, V!5);
> G := Geodesic(V!1, V!5);
>
> d;
4
> P;
[ < [1, 2], 1 >, < [2, 4], 4 >, < [4, 5], 7 > ]
> G;
[ 1, 2, 4, 5 ]
```

Finally, we verify that the shortest path found has length 4.

```
> dP := 0;
> for e in P do
>     dP +:= Weight(e);
> end for;
> assert dP eq d;
```

Note that had we taken instead an undirected graph, we would have had to assign only positive weights to the edges of the graph: any undirected edge $\{u, v\}$ with negative weight results in the negative weight cycles $\{u, u\}$ and $\{v, v\}$.

Example H150E13

We create a weighted multigraph with one edge assigned a negative weight.

```
> G := MultiGraph< 5 | {1, 2}, {1, 2}, {1, 3}, {2, 4}, {3, 5}, {3, 4}, {4, 5} >;
> E := EdgeSet(G);
> AssignWeight(~G, E.1, 1);
> AssignWeight(~G, E.3, 5);
> AssignWeight(~G, E.5, 10);
> AssignWeight(~G, E.7, 1);
> AssignWeight(~G, E.9, -5);
> AssignWeight(~G, E.11, 1);
> AssignWeight(~G, E.13, 2);
>
> V := VertexSet(G);
> E := EdgeSet(G);
> for e in E do
>     e, "  ", Weight(e);
> end for;
< {1, 3}, 5 >    10
```

```
< {1, 2}, 3 >    5
< {1, 2}, 1 >    1
< {2, 4}, 7 >    1
< {3, 4}, 11 >    1
< {3, 5}, 9 >    -5
< {4, 5}, 13 >    2
```

We compute a minimum weight spanning tree rooted at vertex 1.

```
> T := MinimumWeightTree(V!1);
> ET := EdgeSet(T);
> for e in ET do
>     e, "  ", Weight(e);
> end for;
< {1, 2}, 1 >    1
< {2, 4}, 5 >    1
< {3, 5}, 7 >    -5
< {3, 4}, 3 >    1
```

We compute any other spanning tree rooted at vertex 1 (say a depth first tree using DFSTree), and verify that the weights of the edges in G corresponding to the edges of the tree sum up to a total weight which is no smaller than the weight of the minimum weight spanning tree.

```
> DFST := DFSTree(V!1);
> EDT := EdgeSet(DFST);
> for e in EDT do
>     u := InitialVertex(e);
>     v := TerminalVertex(e);
>     w := Min([ Weight(edge) : edge in Edges(V!u, V!v) ]);
>     e, "  ", w;
> end for;
< {1, 3}, 1 >    10
< {2, 4}, 7 >    1
< {3, 4}, 3 >    1
< {4, 5}, 5 >    2
```

150.13 Bibliography

[**Die00**] Reinhard Diestel. *Graph Theory, Second Edition.* Springer, 2000.

151 NETWORKS

Chapter 151
NETWORKS

151.1 Introduction

Networks are an essential tool in modelling communication systems and dependence problems. A network is generally defined as a directed graph whose arcs are associated with a cost and a capacity; it may have multiple (i.e., parallel) edges.

The fundamental network flow problem is the minimum cost flow problem, that is, determining a maximum flow at minimum cost from a specified source to a specified sink. Specializations of this problem are the shortest path problem (where there is no capacity constraint) and the maximum flow problem (where there is no cost constraint). Some of the related problems are the minimum spanning tree problem (finding a spanning tree whose sum of the costs of its arcs is minimum), the matching problem (a pairing of the edges of the graph according to some criteria), and the multicommodity flow problem (where arcs may carry several flows of different nature). For a comprehensive monograph on networks, their implementation and applications, see [RAO93].

For convenience we provide users with the MAGMA network object with type `GrphNet`. It differs from the MAGMA multidigraph object having type `GrphMultDir` in one respect only: its edges are always assumed to be capacitated. The edges of a network have a capacity of one by default, unless they are specifically assigned a capacity. The loops in a network always have capacity zero. Since networks are a specialisation of multidigraphs, all the functions applying to multidigraphs also apply to networks. Below we outline those few functions that specifically concern networks, namely, their construction.

151.2 Construction of Networks

Networks are constructed in a similar way to multidigraphs (Subsection 150.2.2). In this implementation the order n of a network is bounded by 134217722. See Section 149.2.1 for more details on this.

Let N be the network to be constructed. In all cases, whenever an edge $[u, v]$, $u \neq v$, is to be added to N, its capacity will be set to 1 (0 if a loop) unless either its capacity is explicitly given at construction time, or it is the edge of a network, in which case the capacity of the edge remains as it was in the original network.

As an example, if D is a digraph, then the edges of the network N constructed as `N := Network< Order(D) | D >;` will be all the edges of D whose capacity is set as 1 (or 0 if they are loops).

`Network< n | edges >`

`Network< S | edges >`

Construct the network N with vertex-set $V = \{@v_1, v_2, \ldots, v_n@\}$ (where $v_i = i$ for each i if the first form of the constructor is used, or the ith element of the enumerated or indexed set S otherwise), and edge-set $E = \{e_1, e_2, \ldots, e_q\}$. This function returns three values: The network N, the vertex-set V of N; and the edge-set E of N.

The elements of E are specified by the list *edges*, where the items of *edges* may be objects of the following types:

(a) A pair $[v_i, v_j]$ of vertices in V. The directed edge $[v_i, v_j]$ from v_i to v_j with capacity 1 (or 0 if it is a loop) will be added to the edge-set for N.

(b) A tuple of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i. The elements of the sets N_i must be elements of V. If $N_i = \{u_1, u_2, \ldots, u_r\}$, the edges $[v_i, u_1], \ldots, [v_i, u_r]$ will be added to N, all with capacity 1 (or 0 if they are loops).

(c) A tuple of the form $\langle [v_i, v_j], c\rangle$ where v_i, v_j are vertices in V and c the non-negative capacity of the directed edge $[v_i, v_j]$ added to N.

(d) A sequence $[N_1, N_2, \ldots, N_n]$ of n sets, where N_i will be interpreted as a set of out-neighbours for the vertex v_i. All the edges $[v_i, u_i]$, $u_i \in N_i$, are added to N with capacity 1 (or 0 if they are loops).

In addition to these four basic ways of specifying the *edges* list, the items in *edges* may also be:

(e) An edge e of a graph (or di/multi/multidigraph) or network of order n. If e is an edge of a network H, then it will be added to N with the capacity it has in H. If e is not a network edge, then it will be added to N with capacity 1, or 0 if it is a loop.

(f) An edge-set E of a graph (or di/multi/multidigraph) or network of order n. Every edge e in E will be added to N according to the rule set out for a single edge.

(g) A graph (or di/multi/multidigraph) or network H of order n. Every edge e in H's edge-set is added to N according to the rule set out for a single edge.

(h) A $n \times n$ (0, 1)-matrix A. The matrix A will be interpreted as the adjacency matrix for a digraph H on n vertices and the edges of H will be included among the edges of N with capacity 1 (0 if loops).

(i) A set of

  (i) Pairs of the form $[v_i, v_j]$ of vertices in V.

  (ii) Tuples of the form $\langle v_i, N_i\rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

  (iii) A tuple of the form $\langle [v_i, v_j], c\rangle$ where v_i, v_j are vertices in V and c a non-negative capacity.

(iv) Edges of a graph (or di/multi/multidigraph) or network of order n.

(v) Graphs (or di/multi/multidigraphs) or networks of order n.

(j) A sequence of

(i) Tuples of the form $\langle v_i, N_i \rangle$ where N_i will be interpreted as a set of out-neighbours for the vertex v_i.

(ii) A tuple of the form $\langle [v_i, v_j], c \rangle$ where v_i, v_j are vertices in V and c a non-negative capacity.

Example H151E1

We construct a network from a digraph, and observe that the edges that are not loops have a capacity of 1:

```
> SetSeed(1, 0);
> n := 5;
> d := 0.2;
> D := RandomDigraph(n, d : SparseRep := true);
> N := Network< n | D >;
> D;
Digraph
Vertex  Neighbours
1       2 1 ;
2       3 2 ;
3       ;
4       5 ;
5       2 ;
> N;
Network
Vertex  Neighbours
1       1 [ 0 ] 2 [ 1 ] ;
2       2 [ 0 ] 3 [ 1 ] ;
3       ;
4       5 [ 1 ] ;
5       2 [ 1 ] ;
```

151.2.1 Magma Output: Printing of a Network

MAGMA displays a network N in the form of a list of vertices, each accompanied by a list of its outgoing capacitated edges (each followed by the capacity of the edge in brackets). Thus, in the previous example H151E1, it can be verified that all edges have capacity 1 (since the network was constructed from a digraph) except those edges that are loops.

If the network has multiple edges from u to v, then *each* edge from u to v, or rather its end-point v, is printed followed by the capacity of that edge. Also, the end-points in the adjacency list are not ordered and appear in the order in which they were created. The next example illustrates these two points.

Example H151E2

We construct a network from a set of tuples $< [vertex, vertex], capacity >$ and we exhibit a multiple edge.

```
> n := 5;
> C := 5;
> M := 3;
> T := [];
> for i in [1..12] do
>     u := Random(1, n);
>     v := Random(1, n);
>     m := Random(1, M);
>     for j in [1..m] do
>        c := Random(0, C);
>        if u eq v then
>            Append(~T, < [u, u], 0 >);
>        else
>            Append(~T, < [u, v], c >);
>        end if;
>     end for;
> end for;
> T;
[
  <[ 5, 4 ], 1>, <[ 5, 4 ], 3>, <[ 5, 4 ], 2>, <[ 5, 4 ], 1>,
  <[ 5, 4 ], 5>, <[ 1, 3 ], 2>, <[ 1, 3 ], 2>, <[ 5, 5 ], 0>,
  <[ 5, 5 ], 0>, <[ 2, 1 ], 2>, <[ 4, 2 ], 2>, <[ 4, 2 ], 5>,
  <[ 4, 2 ], 1>, <[ 4, 1 ], 3>, <[ 4, 1 ], 4>, <[ 4, 1 ], 3>,
  <[ 2, 3 ], 1>, <[ 2, 3 ], 3>, <[ 4, 3 ], 5>, <[ 4, 3 ], 3>,
  <[ 4, 3 ], 4>, <[ 2, 2 ], 0>, <[ 2, 2 ], 0>, <[ 5, 4 ], 0>,
  <[ 4, 4 ], 0>
]
> N := Network< n | T >;
> N;
Network
Vertex  Neighbours
1       3 [ 2 ] 3 [ 2 ] ;
2       2 [ 0 ] 2 [ 0 ] 3 [ 3 ] 3 [ 1 ] 1 [ 2 ] ;
3       ;
4       4 [ 0 ] 3 [ 4 ] 3 [ 3 ] 3 [ 5 ] 1 [ 3 ] 1 [ 4 ] 1 [ 3 ] 2
        [ 1 ] 2 [ 5 ] 2 [ 2 ] ;
5       4 [ 0 ] 5 [ 0 ] 5 [ 0 ] 4 [ 5 ] 4 [ 1 ] 4 [ 2 ] 4 [ 3 ] 4
        [ 1 ] ;
> Edges(N);
{@ < [1, 3], 6 >, < [1, 3], 7 >, < [2, 1], 10 >, < [2, 2], 22 >,
< [2, 2], 23 >, < [2, 3], 17 >, < [2, 3], 18 >, < [4, 1], 14 >,
< [4, 1], 15 >, < [4, 1], 16 >, < [4, 2], 11 >, < [4, 2], 12 >,
< [4, 2], 13 >, < [4, 3], 19 >, < [4, 3], 20 >, < [4, 3], 21 >,
< [4, 4], 25 >, < [5, 4], 1 >, < [5, 4], 2 >, < [5, 4], 3 >,
```

```
< [5, 4], 4 >, < [5, 4], 5 >, < [5, 4], 24 >, < [5, 5], 8 >,
< [5, 5], 9 > @}
```

151.3 Standard Construction for Networks

151.3.1 Subgraphs

The construction of a sub-network is very similar to the construction of a sub-multidigraph (see Subsection 150.5.1). Additional flexibility is available for setting edge capacities in the subgraph.

There are two constraints when building a subgraph (see the introduction to Section 150.7). Let N be a network, and H a subgraph of N. Then, given any vertices u, v of H, the edge multiplicity from u to v is no greater in H than it is in N, and the total capacity from u to v in H is no greater than the total capacity from u to v in N. Failure to satisfy these constraints will result in a run-time error when constructing a sub-network.

Assume that we intend to add an edge from u to v in H. Assume also that the total capacity from u to v in N is C_N, that the total capacity from u to v in H is C_H before adding the edge from u to v in H, and that the total capacity from u to v in H is C'_H after adding the edge from u to v in H. In order to satisfy the "capacity constraint" it is enough that $C'_H \leq C_N$, i.e. that $C_H + c \leq C_N$ where c is the capacity of the edge one wants to add in H.

There are two methods for adding an edge from u to v. Firstly, adding the edge $[u, v]$ to H without specifying its capacity in H assumes that the edge $[u, v]$ will be added with capacity C_N. This implies that C_H is zero, since we require that $C_H + C_G \leq C_N$. Secondly, adding the edge $[u, v]$ to H when specifying its capacity as c assumes that the edge $[u, v]$ will be added with capacity c such that $C_H + c \leq C_N$.

Note that the support set, vertex labels and edge labels and weights, if applicable, are transferred from the network to the sub-network.

`sub< N | list >`

Construct the network H as a subgraph (sub-network) of N. The function returns three values: The network H, the vertex-set V of H; and the edge-set E of H. If N has a support set and/or if N has vertex/edge labels, and/or edge weights then *all* these attributes are transferred to the subgraph H. Edge capacities are also transferred to H unless they are specifically set as explained below.

The elements of V and of E are specified by the list *list* whose items can be objects of the following types:

(a) A vertex of N. The resulting subgraph will be the subgraph induced on the subset of `VertexSet(`N`)` defined by the vertices in *list*.

(b) An edge of N. The resulting subgraph will be the subgraph with vertex-set `VertexSet(`N`)` whose edge-set consists of the edges in *list* subject to the multiplicity and capacity constraints being satisfied.

(c) A pair of $[v_i, v_j]$ of vertices of N. The resulting subgraph will be the subgraph with vertex-set `VertexSet(`N`)` whose edge-set consists of the edges $[v_i, v_j]$ in *list* whose capacity is assumed to be the total capacity from v_i to v_j in N. The multiplicity and capacity constraints must be satisfied.

(d) A tuple of the form $< [v_i, v_j], c >$ where v_i, v_j are vertices of N and c the non-negative capacity of the edge $[v_i, v_j]$ to be added to H. The resulting subgraph will be the subgraph with vertex-set `VertexSet(`N`)` whose edge-set consists of the edges as they are given in *list*, subject to the multiplicity and capacity constraints being satisfied.

(e) A set of

(i) Vertices of N.

(ii) Edges of N.

(iii) Pairs of vertices of N.

(iv) Tuples of the form $< [v_i, v_j], c >$ where v_i, v_j are vertices of N and c the non-negative capacity of the edge $[v_i, v_j]$ to be added to H.

Example H151E3

We start by constructing a network with some multiple edges.

```
> N := Network< 4 |
> < [1, 2], 2 >, < [1, 2], 3 >, < [1, 4], 5 >,
> < [2, 3], 1 >, < [2, 3], 3 >, < [3, 4], 1 >, < [3, 4], 6 > >;
> N;
Network
Vertex  Neighbours
1       4 [ 5 ] 2 [ 3 ] 2 [ 2 ] ;
2       3 [ 3 ] 3 [ 1 ] ;
3       4 [ 6 ] 4 [ 1 ] ;
4       ;
> V := VertexSet(N);
> E := EdgeSet(N);
```

We construct a subgraph H of N induced by some of N's vertices and we obtain the mapping from the vertices of N to the vertices of H and vice-versa.

```
> H := sub< N | V!1, V!3, V!4 >;
> assert IsSubgraph(N, H);
> H;
Network
Vertex  Neighbours
1       3 [ 5 ] ;
2       3 [ 1 ] 3 [ 6 ] ;
3       ;
> V!VertexSet(H)!1, VertexSet(H)!V!1;
1 1
> V!VertexSet(H)!2, VertexSet(H)!V!3;
```

```
3 2
> V!VertexSet(H)!3, VertexSet(H)!V!4;
4 3
```

The next statements illustrate the "capacity constraint": That is, given any pair $[u, v]$ of vertices of H, H a subgraph of N, the total capacity from u to v in H can not be greater than the total capacity from u to v in N. The subgraph constructor will fail whenever this rule cannot be satisfied. We give a few examples below.

```
> Edges(N);
{@ < [1, 2], 1 >, < [1, 2], 2 >, < [1, 4], 3 >, < [2, 3], 4 >, < [2, 3], 5 >,
< [3, 4], 6 >, < [3, 4], 7 > @}
> E.1, E.2;
< [1, 2], 1 > < [1, 2], 2 >
> Capacity(E.1);
2
> Capacity(E.2);
3
> Capacity(V!1, V!2);
5
>
> H := sub< N | E.1, E.1 >;
> H;
Network
Vertex  Neighbours
1       2 [ 2 ] 2 [ 2 ] ;
2       ;
3       ;
4       ;
```

Adding twice the edge $E.1$ to H is a valid operation since the resulting total capacity from 1 to 2 in H is 4 while it is 5 in N.

```
>  > H := sub< N | E.2, E.2 >;
>> H := sub< N | E.2, E.2 >;
           ^
Runtime error in sub< ... >: RHS argument 2 - Edge multiplicity and capacity
not compatible with subgraph constructor
>
```

Adding twice the edge $E.2$ (which has capacity 3) to H would have resulted in the total capacity from 1 to 2 in H to be 6, while it is 5 in N.

```
> H := sub< N | E!< [1, 2], 1 >, E!< [1, 2], 1 > >;
> H;
Network
Vertex  Neighbours
1       2 [ 2 ] 2 [ 2 ] ;
2       ;
3       ;
```

```
4       ;
```

This succeeded since the total capacity from 1 to 2 in H is now 4.

```
>  > H := sub< N | E!< [1, 2], 2 >, E!< [1, 2], 2 > >;
>> H := sub< N | E!< [1, 2], 2 >, E!< [1, 2], 2 > >;
           ^
Runtime error in sub< ... >: RHS argument 2 - Edge multiplicity and capacity
not compatible with subgraph constructor
>
```

Again, this operation failed as it would have resulted in the total capacity from 1 to 2 in H to be 6.

```
> H := sub< N | < [ V!1, V!2 ], 2 >, < [ V!1, V!2 ], 2 > >;
> H;
Network
Vertex  Neighbours
1       2 [ 2 ] 2 [ 2 ] ;
2       ;
3       ;
4       ;
```

This operation is valid since the total capacity from 1 to 2 in H is now 4.

```
>  > H := sub< N | < [ V!1, V!2 ], 2 >, < [ V!1, V!2 ], 4 > >;
>> H := sub< N | < [ V!1, V!2 ], 2 >, < [ V!1, V!2 ], 4 > >;
           ^
Runtime error in sub< ... >: RHS argument 2 - Tuple must be <[vertex,
vertex], capacity> with total edge multiplicity and capacity compatible with
subgraph constructor
>
```

This operation cannot succeed as the total capacity from 1 to 2 in H would be 6.

```
> H := sub< N | [ V!1, V!2 ] >;
> H;
Network
Vertex  Neighbours
1       2 [ 5 ] ;
2       ;
3       ;
4       ;
```

Adding the edge $[1, 2]$ without specifying its capacity implies that an edge from 1 to 2 is added with capacity 5 to H, which is the total capacity from 1 to 2 in N.

```
>  > H := sub< N | [ V!1, V!2 ], [ V!1, V!2 ] >;
>> H := sub< N | [ V!1, V!2 ], [ V!1, V!2 ] >;
           ^
Runtime error in sub< ... >: RHS argument 2 - Sequence must be
[vertex, vertex] with vertices of the LHS and must be unique
```

```
>
```

Adding twice the edge $[1, 2]$ without specifying its capacity would have resulted in the total capacity from 1 to 2 in H to be 10. This operation cannot succeed. Finally, let us illustrate the edge multiplicity constraint.

```
>  > H := sub< N | E.4, E.4, E.4 >;
>> H := sub< N | E.4, E.4, E.4 >;
           ^
Runtime error in sub< ... >: RHS argument 3 - Edge multiplicity and capacity
not compatible with subgraph constructor
>
```

Although the above statement satisfies the capacity constraint

```
> Capacity(E.4);
1
> Capacity(V!InitialVertex(E.4), V!TerminalVertex(E.4));
4
```

it cannot succeed since the edge multiplicity constraint is violated.

```
> EdgeMultiplicity(V!InitialVertex(E.4), V!TerminalVertex(E.4));
2
```

151.3.2 Incremental Construction: Adding Edges

Almost all the functions to add or remove either vertices or edges that are available for multidigraphs also apply to networks; they are not listed here, see Section 150.5.2 for details.

The only exception is the function `AddEdge(G, u, v, l)` which differs for multidigraphs and networks. It is replaced by the functions `AddEdge(G, u, v, c)` and `AddEdge(G, u, v, c, l)` which are specialised functions for adding capacitated edges to networks. There are a few more such specialised functions which are listed below, they all concern adding edges to a network.

Note that whenever an edge is added to a network using the general multidigraph functions, which do not allow specifying an edge capacity, the edge to be added is *always taken to have capacity 1 (or 0 if a loop).*

`N + < [ u, v ], c >`

Given two vertices u and v of a network N, and c a non-negative integer, adds an edge from u to v with capacity c. The support and edge capacities are retained. This function returns two values: The modified network, and the edge newly created (added). This feature is especially useful when adding parallel edges.

`N + { < [ u, v ], c > }`

`N + [ < [ u, v ], c > ]`

Given tuples of the form $< [u, v], c >$, u and v vertices of the network N and c a non-negative integer, adds the edges from u to v with capacity c. Tuples can be contained in a set or a sequence; the latter is useful when dealing with duplicates. The support and edge capacities are retained.

`N +:= < [ u, v ], c >`

`N +:= { < [ u, v ], c > }`

`N +:= [ < [ u, v ], c > ]`

The procedural versions of the previous three functions. Tuples can be contained in a set or a sequence; the latter is useful when dealing with duplicates.

`AddEdge(N, u, v, c)`

Given two vertices u and v of the network N, and c a non-negative integer, adds an edge from u to v with capacity c. The support and edge capacities are retained. This function returns the modified network and the newly created edge. This feature is especially useful when adding parallel edges.

`AddEdge(N, u, v, c, l)`

Given two vertices u and v of the network N, c a non-negative integer, and a label l, adds an edge from u to v with capacity c and label l. The support and edge capacities are retained. This function returns the modified network and the newly created edge. This feature is especially useful when adding parallel edges.

`AddEdges(N, S)`

Given a network N and a set or a sequence S of tuples, this function includes the edges specified in S. The tuples must be of the form $< [u, v], c >$, where u and v vertices of N and c a non-negative integer. The support and existing vertex and edge decorations are retained.

`AddEdge(~N, u, v, c)`

`AddEdge(~N, u, v, c, l)`

`AddEdges(~N, S)`

Procedural versions of previous functions adding edges to a network. Tuples can be contained in a set or a sequence; the latter is useful when dealing with duplicates.

151.3.3 Union of Networks

It is possible to construct a new network from the union of two networks. For more details, we refer the reader to Subsection 150.5.4.

151.4 Maximum Flow and Minimum Cut

All the functions described in this section apply to general graphs whose edges are given a capacity, that is, networks. If the graph under consideration is not capacitated, then all its edges are assumed to have capacity one. To assign capacities to the edges of a graph, see Subsection 150.4.2. To create a MAGMA network object (with type `GrphNet`) see Section 151.2.

The fundamental network flow problem is the minimum cost flow problem, that is, determining a maximum flow at minimum cost from a specified source to a specified sink. Specializations of this problem are the shortest path problem (where there is no capacity constraint) which is covered in Section 150.12, and the maximum flow problem (where there is no cost constraint). Some of the related problems are the minimum spanning tree problem, the matching problem (a pairing of the edges of the graph according to some criteria), and the multicommodity flow problem (where arcs may carry several flows of different nature). For a comprehensive monograph on network problems, their implementation and applications, see [RAO93].

Let G be a general graph or multigraph whose edges are capacitated. If G is undirected then consider any edge $\{u, v\}$ with capacity c as being equivalent to two directed edges $[u, v]$ and $[v, u]$, both with capacity c. We may thus assume without loss of generality that G is directed, and from now on G is called a network. Let V and E be G's vertex-set and edge-set respectively, and for every edge $[u, v]$ in G, denote its capacity by $c(u, v)$. If there is no edge $[u, v]$ in E we assume that $c(u, v) = 0$. By convention the capacity of an edge is a non-negative integer.

Distinguish two vertices of G, the *source* s and the *sink* t. A *flow* in G is an integer-valued function $f : V \times V \to \mathbf{Z}$ that satisfies the following properties:

(i) Capacity constraint: $\forall u, v \in V,\ f(u, v) \leq c(u, v)$.

(ii) Skew symmetry: $\forall u, v \in V,\ f(u, v) = -f(v, u)$.

(iii) Flow conservation: $\forall u \in V \setminus \{s, t\},\ \sum_{v \in V} f(u, v) = 0$.

Note that the flow could have been defined as a real-valued function if the edge capacity had been defined as real-valued.

The quantity $f(u, v)$, which can be positive or negative, is called the *net flow* from vertex u to vertex v. The *value* of a flow f is defined as $F = \sum_{v \in V} f(s, v)$, that is, the total net flow out of the source.

In the *maximum flow problem* we wish to find a flow of maximum value from s to t. A *cut* of the network G with source s and sink t is a partition of V into S and T such that $s \in S$ and $t \in T$. The *capacity* of the cut $c(S)$ determined by S is the sum of the capacity $c(u, v)$ of the edges $[u, v]$ such that $u \in S$ and $v \in T$. It is easy to see that $F \leq c(S)$, that is, the value of the flow from s into t cannot be larger than the capacity of any cut defined by s and t. In fact, $F = c(S)$ if and only if F is a maximum flow and S is a cut of minimum capacity.

We have implemented two algorithms for finding the maximum flow in a network G from a source s to a sink t. One is the Dinic algorithm, the other is an instance of the

generic push-relabel algorithm. We have already encountered both algorithms in Subsections 149.13.4 and 149.13.5.

The Dinic algorithm

The Dinic algorithm consists of two phases. In the first phase, one constructs a layered network which consists of all the "useful" edges of G: A useful edge $[u, v]$ has the property that $f(u, v) < c(u, v)$. The first and the last layer of this layered network always contain s and t respectively. If such a layered network cannot be constructed then the flow in G is maximum and the algorithm ends.

In the second phase of the Dinic algorithm, one finds a maximal flow by constructing paths from s to t (using a depth-first search) in the layered network. Note that a maximal flow may not be maximum: A maximal flow satisfies the condition that for every path P from s to t in the layered network, there is at least a *saturated* edge in P. A saturated edge is one for which its flow equals its capacity. For more details on the Dinic algorithm, see [Eve79].

The Dinic algorithm has a theoretical complexity of $\mathrm{O}(|V|^2|E|)$, which can be improved to $\mathrm{O}(|E|^{3/2})$ if G is a zero-one network (the capacities of the edges are either 0 or 1), or $\mathrm{O}(|V|^{2/3}|E|)$ if G is a zero-one network with no parallel edges. Prior to the advent of the push-relabel method (which we describe below), the Dinic algorithm was shown to be superior in practice to other methods. The MAGMA implementation of the Dinic algorithm only outperforms the push-relabel method for very sparse networks (a small number of edges relative to the number of vertices) whose edges have a small capacity and where the maximum flow is small. In other words, the Dinic method performs best on zero-one and very sparse networks.

The push-relabel method

The generic push-relabel algorithm does not construct a flow by constructing paths from s to v. Rather, it starts by pushing the maximum possible flow out from the source s into the neighbours of s, then pushing the excess flow at those vertices into their own neighbours. This is repeated until all vertices of G except s and t have an excess flow of zero (that is, the flow conservation property is satisfied). Of course this might mean that some flow is pushed back into s.

At initialisation all vertices are given a height of 0 except s which will have height $|V|$, and flow is pushed out of s. The maximum flow that can be pushed out of s is $\sum_{v \in V} c(s, v)$. The idea is to push flow into vertices only in a "downward" manner, that is, from a vertex with height $h + 1$ into a vertex with height h. If a vertex in $V \setminus \{s, t\}$ has height h, it will be relabelled if it has an excess flow and none of its neighbours to which some flow could be pushed has height $h - 1$. Its new label (height) will be $l + 1$ where l is the height of the lowest labelled neighbour towards which flow can be pushed. Flow may only be pushed along an edge $[u, v]$ while the capacity constraint $f(u, v) \leq c(u, v)$ is satisfied.

So the general notion is of pushing flow "downward" and "discharging" the vertices in $V \setminus \{s, t\}$ from all non-zero excess flow they may have accumulated, a process which may require "lifting" some vertices so that discharging can proceed. It is easy to show that the

height of any vertex in $V \setminus \{s, t\}$ is bounded by $2|V| - 1$, so the algorithm must terminate. When this happens, the flow is at a maximum.

The theoretical complexity of the generic push-relabel algorithm is $O(|V|^2|E|)$ which can be improved to at least $O(|V||E| \log(|V|^2/|E|)$ thanks to some heuristics.

One commonly used heuristic is the *global relabelling* procedure whereby the vertices in $V \setminus \{s, t\}$ are pre-labelled according to their distance from the sink. Global relabelling may occur at specified intervals during execution. Another very useful heuristic is *gap relabelling* which involves recognising those vertices that have become unreachable from the sink and labelling them accordingly so that they won't be chosen during later execution. Finally, another heuristic involves the order in which the vertices to be discharged are processed.

When G is a one-zero network, choosing the vertex with smallest height works best, while choosing the vertex with largest height is recommended for general networks. These heuristics are fully described and analysed by B.V. Cherkassky and A.V. Goldberg *et al.* in [CGM+98] and [CG97]. The MAGMA implementation incorporates all the above heuristics together with some new ones.

As a rule, whenever the push-relabel algorithm is applied to a zero-one network (such as occurs when computing the connectivity of a graph in 149.13.5 or finding a maximum matching in a bipartite network in 149.13.4) the vertex with smallest height is chosen as the next vertex to be processed. When the algorithm is used to compute a maximum flow in a network (as below) the vertex with largest height is chosen.

For the two functions `MinimumCut` and `MaximumFlow` described below, the `PushRelabel` algorithm is used. It is only in the case of a very sparse zero-one network that `Dinic` may outperform `PushRelabel`.

`MinimumCut(s, t` : *parameters*)

`Al` MONSTGELT *Default :* "*PushRelabel*"

Given vertices s and t of a network G, this function returns the subset S defining a minimum cut $\{S, T\}$ of V, where $s \in S$, $t \in T$, corresponding to the maximum flow F from s to t. The subset S is returned as a sequence of vertices of G. The maximum flow F is returned as a second value. The parameter `Al` enables the user to select the algorithm to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`MinimumCut(Ss, Ts` : *parameters*)

`Al` MONSTGELT *Default :* "*PushRelabel*"

Given two sequences S_s and T_s of vertices of a network G, this function returns the subset S defining a minimum cut $\{S, T\}$ of V, such that $S_s \subseteq S$ and $T_s \subseteq T$, and which corresponds to the maximum flow F from the vertices in S_s to the vertices in T_s. The subset S is returned as a sequence of vertices of G. The maximum flow F is returned as a second value. The parameter `Al` enables the user to select the algorithm to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`MaximumFlow(s, t : parameters)`

`Al` MonStgElt *Default : "PushRelabel"*

Given vertices s and t of a network G, this function returns the maximum flow F from s to t. The subset S defining a minimum cut $\{S, T\}$ of `VertexSet(N)`, $s \in S$, $t \in T$, corresponding to F is returned as the second value. The subset S is returned as a sequence of vertices of G. The parameter `Al` enables the user to select the algorithm to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`MaximumFlow(Ss, Ts : parameters)`

`Al` MonStgElt *Default : "PushRelabel"*

Given two sequences S_s and T_s of vertices of a network G, this function returns the maximum flow F from the vertices in S_s to the vertices in T_s. The subset S defining a minimum cut $\{S, T\}$ of `VertexSet(N)`, $S_s \subseteq S$ and $T_s \subseteq T$, corresponding to F is returned as the second value. The subset S is returned as a sequence of vertices of G. The parameter `Al` enables the user to select the algorithm to be used: `Al := "PushRelabel"` or `Al := "Dinic"`.

`Flow(e)`

Given an edge e in a network G, this function returns the flow of the edge e as an integer. In this instance we require that the edges of G be explicitly assigned capacities (see Subsection 150.4.2 or Section 151.2). Edges of G will carry a flow only if a flow has been constructed from a source to a sink in G. If no such flow has yet been constructed, all edges will have zero flow.

`Flow(u, v)`

Let u and v be adjacent vertices of a network G whose edges have been explicitly assigned capacities (see Subsection 150.4.2 or Section 151.2). `Flow(u, v)` returns the total net flow from u to v as an integer. The total net flow from u to v is defined as the total outgoing flow from u into v minus the total ingoing flow into u from v. Thus the flow satisfies the skew symmetry `Flow(u, v)` = - `Flow(v, u)`.

Edges of G will carry a flow only if a flow has been constructed from a source to a sink in G. If no such flow has yet been constructed, all edges will have zero flow.

Example H151E4

We illustrate the maximum flow algorithm by applying it to find a maximum matching in a bipartite graph. This example exactly replicates the implementation of the maximum matching algorithm (Section 149.13.4).
We construct a bipartite graph.

```
> G := Graph< 7 | [ {5, 7}, {6, 7}, {4, 5, 6},
> {3}, {1, 3}, {2, 3}, {1, 2} ] : SparseRep := true >;
> assert IsBipartite(G);
> P := Bipartition(G);
> P;
[
```

```
  { 1, 2, 3 },
  { 4, 5, 6, 7 }
]
```

We add two extra vertices to G, the source s and the sink t and we construct three sets of pairs of vertices. The first contains all the pairs $[s, u]$ where $u \in P[1]$, the second contains all the pairs $[u, t]$ where $u \in P[2]$, and the third contains all the pairs $[u, v]$ where $u \in P[1]$ and $v \in P[2]$. From these sets we construct a multidigraph with capacitated edges.

```
> G +:= 2;
> G;
Graph
Vertex  Neighbours
1       7 5 ;
2       7 6 ;
3       6 5 4 ;
4       3 ;
5       3 1 ;
6       3 2 ;
7       2 1 ;
8       ;
9       ;
> s := Order(G)-1;
> t := Order(G);
>
> E1 := { [s, Index(u)] : u in P[1] };
> E2 := { [Index(u), t] : u in P[2] };
> E3 := { [Index(u), Index(v)] : u in P[1], v in P[2] };
>
> N := MultiDigraph< Order(G) | E1, E2, E3 >;
> N;
Multidigraph
Vertex  Neighbours
1       7 6 5 4 ;
2       5 4 7 6 ;
3       7 6 5 4 ;
4       9 ;
5       9 ;
6       9 ;
7       9 ;
8       1 3 2 ;
9       ;
> E := EdgeSet(N);
> for e in E do
>     AssignCapacity(~N, e, 1);
```

```
> end for;
```

First note that all the edges of N have been assigned capacity 1. Also, the edges of N are those connecting s to all the vertices in $P[1]$, those connecting all the vertices of $P[2]$ to t, and those connecting all the vertices of $P[1]$ to all the vertices of $P[2]$.
We construct a maximum flow F from s to t and we check that the capacity of the cut is indeed F.

```
> V := VertexSet(N);
> F := MaximumFlow(V!s, V!t);
> S := MinimumCut(V!s, V!t);
> F;
3
> S;
[ 8 ]
>
> c := 0;
> for u in S do
>     for v in OutNeighbours(u) do
>        if not v in S then
>           c +:= Capacity(u, v);
>           assert Capacity(u, v) eq Flow(u, v);
>        end if;
>     end for;
> end for;
> assert c eq F;
```

We now exhibit a matching in G. It will be given by the edges in E_3 which are saturated, that is, whose flow is 1.

```
> M := [];
> for e in E3 do
>     u := V!e[1];
>     v := V!e[2];
>     if Flow(u, v) eq 1 then
>         E := Edges(u, v);
>         assert #E eq 1;
>         Append(~M, EndVertices(E[1]));
>     end if;
> end for;
> assert #M eq F;
> M;
[
  [ 1, 7 ],
  [ 3, 5 ],
  [ 2, 4 ]
```

```
]
```

Note that the statement

```
>         assert #E eq 1;
```

makes sense as the network N has no parallel edges.
We end the example by showing that the capacity constraint and the skew symmetry is satisfied by all edges in the network.

```
> for e in EdgeSet(N) do
>     u := EndVertices(e)[1];
>     v := EndVertices(e)[2];
>
>     assert Flow(u, v) le Capacity(u, v);
>     assert Flow(u, v) eq  -Flow(v, u);
> end for;
```

Also, flow conservation holds for all vertices in $V \setminus \{s, t\}$.

```
> s := V!s;
> t := V!t;
> for u in V do
>     if not u eq s and not u eq t then
>         f := 0;
>         for v in OutNeighbours(u) do
>             E := Edges(u, v);
>             f +:= Flow(E[1]);
>         end for;
>         for v in InNeighbours(u) do
>             E := Edges(v, u);
>             f -:= Flow(E[1]);
>         end for;
>
>         assert f eq 0;
>     end if;
> end for;
```

151.5 Bibliography

[**CG97**] B.V. Cherkassky and A.V. Golberg. On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19:390–410, 1997.

[**CGM+98**] B.V. Cherkassky, A.V. Golberg, Paul Martin, J.C. Setubal, and J. Stolfi. Augment or Push? A Computational Study of Bipartite Matching and Unit Capacity Flow Algorithms. Technical report, NEC Research Institute, 1998.

[**Eve79**] Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.

[**RAO93**] T.L. Magnanti R.K. Ahuja and J.B. Orlin. *Network Flows, Theory, Algorithms, and Applications.* Prentice Hall, 1993.

PART XXI
CODING THEORY

152 LINEAR CODES OVER FINITE FIELDS

Chapter 152
LINEAR CODES OVER FINITE FIELDS

152.1 Introduction

Let K be a finite field and let V be the vector space of n-tuples over K. The *Hamming-distance* between elements x and y of V, denoted $d(x, y)$, is defined by

$$d(x, y) := \#\{\ 1 \leq i \leq n \mid x_i \neq y_i\ \}.$$

The *minimum distance* d for a subset C of V is then

$$d = \min\{\ d(x, y) \mid x \in C, y \in C, x \neq y\ \}.$$

The subset C of V is called an (n, M, d) *code* if the minimum distance for the subset C is d and $|C| = M$. Then V is referred to as the *ambient space* of C.

The code C is called a $[n, k, d]$ *linear code* if C is a k-dimensional subspace of V. Currently MAGMA supports not only linear codes, but also codes over finite fields which are only linear over some subfield. These are known as *additive codes* and can be found in Chapter 156. This chapter deals only with linear codes.

In this chapter, the term "code" will refer to a linear code. MAGMA provides machinery for studying linear codes over finite fields $F_q = GF(q)$, over the integer residue classes $\mathbf{Z}_m = \mathbf{Z}/m\mathbf{Z}$, and over galois rings $GR(p^n, k)$.

This chapter describes those functions which are applicable to codes over F_q. The highlights of the facilities provided for such codes include:

- The construction of codes in terms of generator matrices, parity check matrices and generating polynomials (cyclic codes).
- A large number of constructions for particular families of codes, e.g., quadratic residue codes.
- Highly optimized algorithms for the calculation of the minimum weight.
- Various forms of weight enumerator including the Macwilliams transform.
- A database that gives the user access to every best known linear code over $GF(2)$ of length up to 256, and 98% of best known linear codes over $GF(4)$ of length up to 100.
- Machinery that allows the user to construct algebraic-geometric codes from a curve defined over F_q.
- The computation of automorphism groups for codes over small fields.

The reader is referred to [MS78] as a general reference on coding theory.

152.2 Construction of Codes

152.2.1 Construction of General Linear Codes

`LinearCode< R, n | L >`

Create a code as a subspace of $V = R^{(n)}$ which is generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of V.

(b) A set or sequence of elements of V.

(c) A sequence of n elements of K, defining an element of V.

(d) A set or sequence of sequences of type (c).

(e) A subspace of V.

(f) A set or sequence of subspaces of V.

Example H152E1

We define the ternary Golay code as a six-dimensional subspace of the vector space $K^{(11)}$, where K is $\mathbf{F}_3$. The ternary Golay code could be defined in a single statement as follows:

```
> K := FiniteField(3);
> C := LinearCode<K, 11 |
>    [1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1], [0, 1, 0, 0, 0, 0, 0, 1, 2, 2, 1],
>    [0, 0, 1, 0, 0, 0, 1, 0, 1, 2, 2], [0, 0, 0, 1, 0, 0, 2, 1, 0, 1, 2],
>    [0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 1], [0, 0, 0, 0, 0, 1, 1, 2, 2, 1, 0]>;
```

Alternatively, if we want to see the code as a subspace of $K^{(11)}$, we would proceed as follows:

```
> K := FiniteField(3);
> K11 := VectorSpace(K, 11);
> C := LinearCode(sub<K11 |
>    [1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1], [0, 1, 0, 0, 0, 0, 0, 1, 2, 2, 1],
>    [0, 0, 1, 0, 0, 0, 1, 0, 1, 2, 2], [0, 0, 0, 1, 0, 0, 2, 1, 0, 1, 2],
>    [0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 1], [0, 0, 0, 0, 0, 1, 1, 2, 2, 1, 0]>);
```

`LinearCode(U)`

Let V be the R-space $R^{(n)}$ and suppose that U is a subspace of V. The effect of this function is to define the linear code C corresponding to the subspace U. Suppose the code C being constructed has dimension k. The evaluation of this constructor results in the creation of the following objects:

(a) The generator matrix G for C, created as a $k \times n$ matrix belonging to the R-matrix space, $R^{(k \times n)}$.

(b) The parity check matrix H for C, created as an $(n-k) \times n$ matrix belonging to the R-matrix space, $R^{(n-k) \times n)}$.

`LinearCode(A)`

Given a $k \times n$ matrix A over the ring R, construct the linear code generated by the rows of A. Note that it is not assumed that the rank of A is k. The effect of this constructor is otherwise identical to that described above.

Example H152E2

We define a code by constructing a matrix in a K-matrix space and using its row space to generate the code:

```
> M := KMatrixSpace(FiniteField(5), 2, 4);
> G := M ! [1,1,1,2, 3,2,1,4];
> G;
[1 1 1 2]
[3 2 1 4]
> C := LinearCode(G);
> C;
[4, 2, 2] Linear Code over GF(5)
Generator matrix:
[1 0 4 0]
[0 1 2 2]
```

`PermutationCode(u, G)`

Given a finite permutation group G of degree n, and a vector u belonging to the n-dimensional vector space V over the ring R, construct the code C corresponding to the subspace of V spanned by the set of vectors obtained by applying the permutations of G to the vector u.

Example H152E3

We define G to be a permutation group of degree 7 and construct the code C as the F_2-code generated by applying the permutations of G to a certain vector:

```
> G := PSL(3, 2);
> G;
Permutation group G of degree 7
    (1, 4)(6, 7)
    (1, 3, 2)(4, 7, 5)
> V := VectorSpace(GF(2), 7);
> u := V ! [1, 0, 0, 1, 0, 1, 1];
> C := PermutationCode(u, G);
> C;
[7, 3, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1 1]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
```

152.2.2 Some Trivial Linear Codes

`ZeroCode(R, n)`

Given a ring R and positive integer n, return the $[n, 0, n]$ code consisting of only the zero code word, (where the minimum weight is by convention equal to n).

`RepetitionCode(R, n)`

Given a ring R and positive integer n, return the $[n, 1, n]$ code over K generated by the all-ones vector.

`ZeroSumCode(R, n)`

Given a ring R and positive integer n, return the $[n, n-1, 2]$ code over R such that for all codewords $(c_1, c_2, \ldots, c_n)$ we have $\sum_i c_i = 0$.

`UniverseCode(R, n)`

Given a ring R and positive integer n, return the $[n, n, 1]$ code consisting of all possible codewords.

`EvenWeightCode(n)`

Given a positive integer n, return the $[n, n-1, 2]$ code over $\mathbf{F}_2$ such that all vectors have even weight. This is equivalent to the zero sum code over $\mathbf{F}_2$.

`EvenWeightSubcode(C)`

Given a linear code C over $\mathbf{F}_2$, return the subcode of C containing the vectors of even weight.

`RandomLinearCode(K, n, k)`

Given a finite field K and positive integers n and k, such that $0 < k \leq n$, the function returns a random linear code of length n and dimension k over the field K. The method employed is to successively choose random vectors from $K^{(n)}$ until generators for a k-dimensional subspace have been found.

`CordaroWagnerCode(n)`

Construct the Cordaro–Wagner code of length n, This is the 2-dimensional repetition code over $\mathbf{F}_2$ of length n and having the largest possible minimum weight.

Example H152E4

Over any specific finite field K, the zero code of length n is contained in every code of length n, and similarly every code of length n is contained in the universe code of length n. This is illustrated over $GF(2)$ for length 6 codes with an arbitrary code of length 6 dimension 3.

```
> K := GF(2);
> U := UniverseCode(K, 6);
> U;
[6, 6, 1] Linear Code over GF(2)
> Z := ZeroCode(K, 6);
> Z;
[6, 0, 6] Linear Code over GF(2)
> R := RandomLinearCode(K, 6, 3);
> (Z subset R) and (R subset U);
true
```

152.2.3 Some Basic Families of Codes

In this section we describe how to construct three very important families of codes: cyclic codes, Hamming codes and Reed-Muller codes. We choose to present these very important families at this stage since they are easily understood and they give us a nice collection of codes for use in examples.

Many more constructions will be described in subsequent sections. In particular, variations and generalizations of the cyclic code construction presented here will be given.

`CyclicCode(n, g)`

Let K be a finite field. Given a positive integer n and a univariate polynomial $g(x) \in K[x]$ of degree $n-k$ such that $g(x) \mid x^n - 1$, construct the $[n,k]$ cyclic code generated by $g(x)$.

Example H152E5

We construct the length 23 Golay code over $GF(2)$ as a cyclic code by factorizing the polynomial $x^{23}-1$ over $GF(2)$ and constructing the cyclic code generated by one of the factors of degree 11.

```
> P<x> := PolynomialRing(FiniteField(2));
> F := Factorization(x^23 - 1);
> F;
[
    <x + 1, 1>,
    <x^11 + x^9 + x^7 + x^6 + x^5 + x + 1, 1>,
    <x^11 + x^10 + x^6 + x^5 + x^4 + x^2 + 1, 1>
]
> CyclicCode(23, F[2][1]);
[23, 12, 7] Cyclic Code over GF(2)
Generator matrix:
```

```
[1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0 0 1 1]
[0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1]
[0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0 1]
```

`HammingCode(K, r)`

Given a positive integer r, and a finite field K of cardinality q, construct the r-th order Hamming code over K of cardinality q. This code has length

$$n = (q^r - 1)/(q - 1).$$

Example H152E6

We construct the third order Hamming code over GF(2) together with its parity check matrix.

```
> H := HammingCode(FiniteField(2), 3);
> H;
[7, 4, 3] Hamming code (r = 3) over GF(2)
Generator matrix:
[1 0 0 0 0 1 1]
[0 1 0 0 1 1 0]
[0 0 1 0 1 0 1]
[0 0 0 1 1 1 1]
> ParityCheckMatrix(H);
[1 0 1 0 1 1 0]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
```

`SimplexCode(r)`

Given a positive integer r, construct the $[2^r - 1, r, 2^{r-1}]$ binary simplex code, which is the dual of a Hamming code.

`ReedMullerCode(r, m)`

Given positive integers r and m, where $0 \leq r \leq m$, construct the r-th order binary Reed–Muller code of length $n = 2^m$.

Example H152E7

We construct the first order Reed–Muller code of length 16 and count the number of pairs of vectors whose components are orthogonal.

```
> R := ReedMullerCode(1, 4);
> R;
[16, 5, 8] Reed-Muller Code (r = 1, m = 4) over GF(2)
Generator matrix:
[1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
> #{<v, w>: v, w in R | IsZero(InnerProduct(v, w))};
1024
```

152.3 Invariants of a Code

152.3.1 Basic Numerical Invariants

`Length(C)`

Given an $[n, k]$ code C, return the block length n of C.

`Dimension(C)`
`NumberOfGenerators(C)`

The dimension k of the $[n, k]$ linear code C.

`#C`

Given a code C, return the number of codewords belonging to C.

`InformationRate(C)`

The information rate of the $[n, k]$ code C. This is the ratio k/n.

152.3.2 The Ambient Space and Alphabet

`AmbientSpace(C)`

The ambient space of the code C, i.e. the generic R-space V in which C is contained.

`RSpace(C)`

`VectorSpace(C)`

Given an $[n, k]$ linear code C, defined as a subspace U of the n-dimensional space V, return U as a subspace of V with basis corresponding to the rows of the generator matrix for C.

`Generic(C)`

Given an $[n, k]$ code C, return the generic $[n, n, 1]$ code in which C is contained.

`Alphabet(C)`

`Field(C)`

The underlying ring (or alphabet) R of the code C.

152.3.3 The Code Space

`GeneratorMatrix(C)`

`BasisMatrix(C)`

The generator matrix for the linear code C, returned as an element of $\mathrm{Hom}(U, V)$ where U is the information space of C and V is the ambient space of C.

`Basis(C)`

The current vector space basis for the linear code C, returned as a sequence of elements of C.

`Generators(C)`

The current vector space basis for the linear code C, returned as a set of elements of C.

`C . i`

Given an $[n, k]$ code C and a positive integer i, $1 \leq i \leq k$, return the i-th element of the current basis of C.

152.3.4 The Dual Space

`Dual(C)`

The code that is dual to the code C.

`ParityCheckMatrix(C)`

The parity check matrix for the code C, returned as an element of $\mathrm{Hom}(V, U)$.

Example H152E8

We create a Reed–Muller code and demonstrate some simple relations.

```
> R := ReedMullerCode(1, 3);
> R;
[8, 4, 4] Reed-Muller Code (r = 1, m = 3) over GF(2)
Generator matrix:
[1 0 0 1 0 1 1 0]
[0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1]
> G := GeneratorMatrix(R);
> P := ParityCheckMatrix(R);
> P;
[1 0 0 1 0 1 1 0]
[0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1]
> G * Transpose(P);
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
> D := LinearCode(P);
> Dual(R) eq D;
true
```

`Hull(C)`

The Hull of a code is the intersection between itself and its dual.

152.3.5 The Information Space and Information Sets

`InformationSpace(C)`

Given an $[n, k]$ linear code C, return the k-dimensional R-space U which is the space of information vectors for the code C.

`InformationSet(C)`

Given an $[n, k]$ linear code C over a finite field, return the current information set for C. The information set for C is an ordered set of k linearly independent columns of the generator matrix, such that the generator matrix is the identity matrix when restricted to these columns. The information set is returned as a sequence of k integers, giving the numbers of the columns that correspond to the information set.

`AllInformationSets(C)`

Given an $[n, k]$ linear code C over a finite field, return all the possible information sets of C as a (sorted) sequence of sequences of column indices. Each inner sequence contains a maximal set of indices of linearly independent columns in the generator matrix of C.

`StandardForm(C)`

Given an $[n, k]$ linear code C over a finite field, return the standard form D of C. A code is in *standard form* if the first k components of the code words correspond to the information set. MAGMA returns one of the many codes in standard form which is isomorphic to C. (The same code is returned each time.) Thus, the effect of this function is to return a code D whose generators come from the generator matrix of C with its columns permuted, so that the submatrix consisting of the first k columns of the generator matrix for D is the identity matrix. Two values are returned:

(a) The standard form code D;

(b) An isomorphism from C to D.

Example H152E9

We construct a Reed–Muller code C and its standard form S and then map a codeword of C into S.

```
> C := ReedMullerCode(1, 4);
> C;
[16, 5, 8] Reed-Muller Code (r = 1, m = 4) over GF(2)
Generator matrix:
[1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
> InformationSet(C);
[ 1, 2, 3, 5, 9 ]
```

```
> #AllInformationSets(C);
2688
> S, f := StandardForm(C);
> S;
[16, 5, 8] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 1 1 0 1 1 0 1 0 0 1]
[0 1 0 0 0 1 1 0 1 1 0 1 0 1 0 1]
[0 0 1 0 0 1 0 1 1 0 1 1 0 0 1 1]
[0 0 0 1 0 0 1 1 1 0 0 0 1 1 1 1]
[0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1]
> u := C.1;
> u;
(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)
> f(u);
(1 0 0 0 0 1 1 1 0 1 1 0 1 0 0 1)
```

152.3.6 The Syndrome Space

`SyndromeSpace(C)`

Given an $[n, k]$ linear code C, return the $(n-k)$-dimensional vector space W, which is the space of syndrome vectors for the code C.

152.3.7 The Generator Polynomial

The operations in this section are restricted to cyclic codes.

`GeneratorPolynomial(C)`

Given a cyclic code C over a finite field, return the generator polynomial of C. The generator polynomial of C is a divisor of $x^n - 1$, where n is the length of C.

`CheckPolynomial(C)`

Given a cyclic code C over a finite field, return the check polynomial of C as an element of $K[x]$. If $g(x)$ is the generator polynomial of C and $h(x)$ is the check polynomial of C, then $g(x)h(x) = 0 \pmod{x^n - 1}$, where n is the length of C.

`Idempotent(C)`

Given a cyclic code C, return the (polynomial) idempotent of C. If $c(x)$ is the idempotent of C, then $c(x)^2 = 0 \pmod{x^n - 1}$, where n is the length of C.

Example H152E10

We find the generator and check polynomials for the third order Hamming code over GF(2).

```
> K<w> := GF(2);
> P<x> := PolynomialRing(K);
> H := HammingCode(K, 3);
> g := GeneratorPolynomial(H);
> g;
x^3 + x + 1
> h := CheckPolynomial(H);
> h;
x^4 + x^2 + x + 1
> g*h mod (x^7 - 1);
0
> forall{ c : c in H | h * P!Eltseq(c) mod (x^7-1) eq 0 };
true
> e := Idempotent(H);
> e;
x^4 + x^2 + x
> e^2;
x^8 + x^4 + x^2
```

152.4 Operations on Codewords

152.4.1 Construction of a Codeword

`C ! [a1, ..., an]`

`elt< C | a1, ..., an >`

Given a code C which is defined as a subset of the R-space $R^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to R, construct the codeword $(a_1, \ldots, a_n)$ of C. It is checked that the vector $(a_1, \ldots, a_n)$ is an element of C.

`C ! u`

Given a code C which is defined as a subset of the R-space $V = R^{(n)}$, and an element u belonging to V, create the codeword of C corresponding to u. The function will fail if u does not belong to C.

`C ! 0`

The zero word of the code C.

`Random(C)`

A random codeword of C.

152.4.2 Arithmetic Operations on Codewords

`u + v`

Sum of the codewords u and v, where u and v belong to the same linear code C.

`-u`

Additive inverse of the codeword u belonging to the linear code C.

`u - v`

Difference of the codewords u and v, where u and v belong to the same linear code C.

`a * u`

Given an element a belonging to the field K, and a codeword u belonging to the linear code C, return the codeword $a * u$.

`Normalize(u)`

Given an element u over a field, not the zero element, belonging to the linear code C, return $\frac{1}{a} * u$, where a is the first non-zero component of u. If u is the zero vector, it is returned as the value of this function. The net effect is that `Normalize(u)` always returns a vector v in the subspace generated by u, such that the first non-zero component of v is the unit of K.

`Syndrome(w, C)`

Given an $[n, k]$ linear code C over a finite field with parent vector space V, and a vector w belonging to V, construct the syndrome of w relative to the code C. This will be an element of the *syndrome space* of C.

152.4.3 Distance and Weight

`Distance(u, v)`

The Hamming distance between the codewords u and v, where u and v belong to the same code C.

`Weight(u)`

The Hamming weight of the codeword u, i.e., the number of non-zero components of u.

`LeeWeight(u)`

The Lee weight of the codeword u.

Example H152E11

We calculate all possible distances between code words of the non-extended Golay code over GF(3), and show the correspondence with all possible code word weights.

```
> C := GolayCode(GF(3),false);
> {Distance(v,w):v,w in C};
{ 0, 5, 6, 8, 9, 11 }
> {Weight(v):v in C};
{ 0, 5, 6, 8, 9, 11 }
```

152.4.4 Vector Space and Related Operations

`(u, v)`

`InnerProduct(u, v)`

Inner product of the vectors u and v with respect to the Euclidean norm, where u and v belong to the parent vector space of the code C.

`Support(w)`

Given a word w belonging to the $[n, k]$ code C, return its support as a subset of the integer set $\{1..n\}$. The support of w consists of the coordinates at which w has non-zero entries.

`Coordinates(C, u)`

Given an $[n, k]$ linear code C and a codeword u of C return the coordinates of u with respect to C. The coordinates of u are returned as a sequence $Q = [a_1, \ldots, a_k]$ of elements from the alphabet of C so that $u = a_1 * C.1 + \ldots + a_k * C.k$.

`Parent(w)`

Given a word w belonging to the code C, return the ambient space V of C.

`Rotate(u, k)`

Given a vector u, return the vector obtained from u by cyclically shifting its components to the right by k coordinate positions.

`Rotate(~u, k)`

Given a vector u, destructively rotate u by k coordinate positions.

`Trace(u, S)`

`Trace(u)`

Given a vector u with components in K, and a subfield S of K, construct the vector with components in S obtained from u by taking the trace of each component with respect to S. If S is omitted, it is taken to be the prime field of K.

Example H152E12

We create a specific code word in the length 5 even weight code, after a failed attempt to create a code word of odd weight. We then display its support, find its coordinates with respect to the basis and then confirm it by way of re-construction.

```
> C := EvenWeightCode(5);
>  C![1,1,0,1,0];
>> C![1,1,0,1,0];
    ^
Runtime error in '!': Result is not in the given structure
> c := C![1,1,0,1,1];
> c;
(1 1 0 1 1)
> Support(c);
{ 1, 2, 4, 5 }
> Coordinates(C,c);
[ 1, 1, 0, 1 ]
> C.1 + C.2 + C.4;
(1 1 0 1 1)
```

152.4.5 Predicates for Codewords

`u eq v`

The function returns `true` if and only if the codewords u and v are equal.

`u ne v`

The function returns `true` if and only if the codewords u and v are not equal.

`IsZero(u)`

The function returns `true` if and only if the codeword u is the zero vector.

152.4.6 Accessing Components of a Codeword

`u[i]`

Given a codeword u belonging to the code C defined over the ring R, return the i-th component of u (as an element of R).

`u[i] := x;`

Given an element u belonging to a subcode C of the full R-space $V = R^n$, a positive integer i, $1 \leq i \leq n$, and an element x of R, this function returns a vector in V which is u with its i-th component redefined to be x.

152.5 Coset Leaders

`CosetLeaders(C)`

Given a code C with ambient space V over a finite field, return a set of coset leaders (vectors of minimal weight in their cosets) for C in V as an indexed set of vectors from V. Note that this function is only applicable when V and C are small. This function also returns a map from the syndrome space of C into the coset leaders (mapping a syndrome into its corresponding coset leader).

Example H152E13

We construct a Hamming code C, encode an information word using C, introduce one error, and then decode by calculating the syndrome of the "received" vector and applying the CosetLeaders map to the syndrome to recover the original vector.
First we set C to be the third order Hamming Code over the finite field with two elements.

```
> C := HammingCode(GF(2), 3);
> C;
[7, 4, 3] Hamming code (r = 3) over GF(2)
Generator matrix:
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
```

Then we set L to be the set of coset leaders of C in its ambient space V and f to be the map which maps the syndrome of a vector in V to its coset leader in L.

```
> L, f := CosetLeaders(C);
> L;
{@
    (0 0 0 0 0 0 0),
    (1 0 0 0 0 0 0),
    (0 1 0 0 0 0 0),
    (0 0 1 0 0 0 0),
    (0 0 0 1 0 0 0),
    (0 0 0 0 1 0 0),
    (0 0 0 0 0 1 0),
    (0 0 0 0 0 0 1)
@}
```

Since C has dimension 4, the degree of the information space I of C is 4. We set i to be an "information vector" of length 4 in I, and then encode i using C by setting w to be the product of i by the generator matrix of C.

```
> I := InformationSpace(C);
> I;
Full Vector space of degree 4 over GF(2)
> i := I ! [1, 0, 1, 1];
> w := i * GeneratorMatrix(C);
```

```
> w;
(1 0 1 1 0 1 0)
```

Now we set r to be the same as w but with an error in the 7-th coordinate (so r is the "received vector").

```
> r := w;
> r[7] := 1;
> r;
(1 0 1 1 0 1 1)
```

Finally we let s be the syndrome of r with respect to C, apply f to s to get the coset leader l, and subtract l from r to get the corrected vector v. Finding the coordinates of v with respect to the basis of C (the rows of the generator matrix of C) gives the original information vector.

```
> s := Syndrome(r, C);
> s;
(1 1 1)
> l := f(s);
> l;
(0 0 0 0 0 0 1)
> v := r - l;
> v;
(1 0 1 1 0 1 0)
> res := I ! Coordinates(C, v);
> res;
(1 0 1 1)
```

152.6 Subcodes

152.6.1 The Subcode Constructor

`sub< C | L >`

Given an $[n, k]$ linear code C over R, construct the subcode of C, generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of C;

(b) A set or sequence of elements of C;

(c) A sequence of n elements of R, defining an element of C;

(d) A set or sequence of sequences of type (c);

(e) A subcode of C;

`Subcode(C, k)`

Given an $[n, k]$ linear code C and an integer t, $1 \leq t < n$, return a subcode of C of dimension t.

`Subcode(C, S)`

Given an $[n, k]$ linear code C and a set S of integers, each of which lies in the range $[1, k]$, return the subcode of C generated by the basis elements whose positions appear in S.

`SubcodeBetweenCode(C1, C2, k)`

Given a linear code C_1 and a subcode C_2 of C_1, return a subcode of C_1 of dimension k containing C_2.

`SubcodeWordsOfWeight(C, S)`

Given an $[n, k]$ linear code C and a set S of integers, each of which lies in the range $[1, n]$, return the subcode of C generated by those words of C whose weights lie in S.

Example H152E14

We give an example of how `SubcodeBetweenCode` may be used to create a code nested in between a subcode pair.

```
> C1 := RepetitionCode(GF(2),6);
> C1;
[6, 1, 6] Cyclic Code over GF(2)
Generator matrix:
[1 1 1 1 1 1]
> C3 := EvenWeightCode(6);
> C3;
[6, 5, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1]
[0 1 0 0 0 1]
[0 0 1 0 0 1]
[0 0 0 1 0 1]
[0 0 0 0 1 1]
> C1 subset C3;
true
> C2 := SubcodeBetweenCode(C3, C1, 4);
> C2;
[6, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 0]
[0 1 0 0 0 1]
[0 0 1 0 0 1]
[0 0 0 1 0 1]
> (C1 subset C2) and (C2 subset C3);
true
```

152.6.2 Sum, Intersection and Dual

For the following operators, C and D are codes defined as subsets (or subspaces) of the same R-space V.

`C + D`

The (vector space) sum of the linear codes C and D, where C and D are contained in the same K-space V.

`C meet D`

The intersection of the linear codes C and D, where C and D are contained in the same K-space V.

`Dual(C)`

The dual D of the linear code C. The dual consists of all codewords in the K-space V which are orthogonal to all codewords of C.

Example H152E15

Verify some simple results from the sum and intersection of subcodes with known basis.

```
> C := EvenWeightCode(5);
> C;
[5, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1]
[0 1 0 0 1]
[0 0 1 0 1]
[0 0 0 1 1]
> C1 := sub< C | C.1 >;
> C2 := sub< C | C.4 >;
> C3 := sub< C | { C.1 , C.4} >;
> (C1 + C2) eq C3;
true
> (C1 meet C3) eq C1;
true
```

Example H152E16

Verify the orthogonality of codewords in the dual for a `ReedSolomonCode`.

```
> K<w> := GF(8);
> R := ReedSolomonCode(K, 3);
> R;
[7, 5, 3] BCH code (d = 3, b = 1) over GF(2^3)
Generator matrix:
[  1   0   0   0   0 w^3 w^4]
[  0   1   0   0   0   1   1]
[  0   0   1   0   0 w^3 w^5]
```

```
[  0   0   0   1   0   w w^5]
[  0   0   0   0   1   w w^4]
> D := Dual(R);
> D;
[7, 2, 6] Cyclic Code over GF(2^3)
Generator matrix:
[  1   0 w^3   1 w^3   w   w]
[  0   1 w^4   1 w^5 w^5 w^4]
> {<u,v> : u in R, v in D | InnerProduct(u,v) ne 0};
{}
```

152.6.3 Membership and Equality

For the following operators, C and D are codes defined as a subset (or subspace) of the R-space V.

`u in C`

Return `true` if and only if the vector u of V belongs to the code C.

`u notin C`

Return `true` if and only if the vector u of V does not belong to the code C.

`C subset D`

Return `true` if and only if the code C is a subcode of the code D.

`C notsubset D`

Return `true` if and only if the code C is not a subcode of the code D.

`C eq D`

Return `true` if and only if the codes C and D are equal.

`C ne D`

Return `true` if and only if the codes C and D are not equal.

152.7 Properties of Codes

For the following operators, C and D are codes defined as a subset (or subspace) of the vector space V.

`IsCyclic(C)`

Return `true` if and only if the linear code C is a cyclic code.

`IsSelfDual(C)`

Return `true` if and only if the linear code C is self-dual. (i.e. C equals the dual of C).

`IsSelfOrthogonal(C)`

Return `true` if and only if the linear code C is self-orthogonal (i.e., C is contained in the dual of C).

`IsMaximumDistanceSeparable(C)`

`IsMDS(C)`

Returns `true` if and only if the linear code C is maximum-distance separable; that is, has parameters $[n, k, n - k + 1]$.

`IsEquidistant(C)`

Returns `true` if and only if the linear code C is equidistant.

`IsPerfect(C)`

Returns `true` if and only if the linear code C is perfect; that is, if and only if the cardinality of C is equal to the size of the sphere packing bound of C.

`IsNearlyPerfect(C)`

Returns `true` if and only if the binary linear code C is nearly perfect.

`IsEven(C)`

Returns `true` if and only if C is an even linear binary code, (i.e., all codewords have even weight). If `true`, then MAGMA will adjust the upper and lower minimum weight bounds of C if possible.

`IsDoublyEven(C)`

Returns `true` if and only if C is a doubly even linear binary code, (i.e., all codewords have weight divisible by 4). If `true`, then MAGMA will adjust the upper and lower minimum weight bounds of C if possible.

`IsProjective(C)`

Returns `true` if and only if the (non-quantum) code C is projective.

Example H152E17

We look at an extended quadratic residue code over $GF(2)$ which is self-dual, and then confirm it manually.

```
> C := ExtendCode( QRCode(GF(2),23) );
> C:Minimal;
[24, 12, 8] Linear Code over GF(2)
> IsSelfDual(C);
true
> D := Dual(C);
> D: Minimal;
[24, 12, 8] Linear Code over GF(2)
> C eq D;
true
```

Example H152E18

We look at the `CordaroWagnerCode` of length 6, which is self-orthogonal, and then confirm it manually.

```
> C := CordaroWagnerCode(6);
> C;
[6, 2, 4] Linear Code over GF(2)
Generator matrix:
[1 1 0 0 1 1]
[0 0 1 1 1 1]
> IsSelfOrthogonal(C);
true
> D := Dual(C);
> D;
[6, 4, 2] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1]
[0 1 0 1 0 1]
[0 0 1 1 0 0]
[0 0 0 0 1 1]
> C subset D;
true
```

152.8 The Weight Distribution

152.8.1 The Minimum Weight

In the case of a linear code, the minimum weight and distance are equivalent. It is clear that is substantially easier to determine the minimum weight of a (possibly non-linear) code than its minimum distance. The general principle underlying the minimum weight algorithm in MAGMA is the embedding of low-weight information vectors into the code space, in the hope that they will map onto low weight codewords.

Let C be a $[n, k]$ linear code over a finite field and G be its generator matrix. The minimum weight algorithm proceeds as follows: Starting with $r = 1$, all linear combinations of r rows of G are enumerated. By taking the minimum weight of each such combination, an upper bound, d_{upper}, on the minimum weight of C is obtained. A strictly increasing function, $d_{lower}(r)$, finds a lower bound on the minimum weight of the non-enumerated vectors for each computational step (the precise form of this function depends upon the algorithm being used). The algorithm terminates when $d_{lower}(r) \geq d_{upper}$, at which point the actual minimum weight is equal to d_{upper}.

The algorithm is used for non-cyclic codes, and is due to A.E. Brouwer and K.H. Zimmermann [BFK+98]. The key idea is to construct as many different generator matrices for the same code as possible, each having a different information set and such that the information sets are as disjoint as possible. By maximizing the number of information sets, $d_{lower}(r)$ can be made increasingly accurate. Each information set will provide a different embedding of information vectors into the code, and thus the probability of a low-weight information vector mapping onto a low-weight codeword is increased.

A well known improvement attributed to Brouwer exists for cyclic codes, requiring the enumeration fo only one information set. A generalisation of this improvement has been made by G. White to quasicyclic codes, and any codes whose known automorphisms have large cycles. Functionality is included in this section for inputting partial knowledge of the automorphism group to take advantage of this improvement.

Information sets are discarded if their ranks are too low to contribute to the lower bound calculation. The user may also specify a lower bound, `RankLowerBound`, on the rank of information sets initially created.

`MinimumWeight(C: parameters)`

`MinimumDistance(C: parameters)`

`Method`	MONSTGELT	*Default :* "*Auto*"
`RankLowerBound`	RNGINTELT	*Default :* 0
`MaximumTime`	RNGRESUBELT	*Default :* ∞
`Nthreads`	RNGINTELT	*Default :* 1

Determine the minimum weight of the words belonging to the code C, which is also the minimum distance between any two codewords. The parameter `RankLowerBound` sets a minimum rank on the informations sets used in the calculation, while the parameter `MaximumTime` sets a time limit (in seconds of "user time") after which the calculation is aborted.

If the base field is $\mathbf{F}_2$ and the parameter `Nthreads` is set to a positive integer n, then n threads will be used in the computation, if POSIX threads are enabled. One can alternatively use the procedure `SetNthreads` to set the global number of threads to a value n so that n threads are always used by default in this algorithm unless overridden by the `Nthreads` parameter.

Sometimes a brute force calculation of the entire weight distribution can be a faster way to get the minimum weight for small codes. When the parameter `Method` is set to the default `"Auto"` then the method is internally chosen. The user can specify which method they want using setting it to either `"Distribution"` or `"Zimmerman"`.

By setting the verbose flag `"Code"`, information about the progress of the computation can be printed. An example to demonstrate the interpretation of the verbose output follows:

```
> SetVerbose("Code", true);
> SetSeed(1);
> MinimumWeight(RandomLinearCode(GF(2),85,26));
Linear Code over GF(2) of length 85 with 26 generators. Is not cyclic
Lower Bound: 1, Upper Bound: 60
Constructed 4 distinct generator matrices
Relative Ranks:  26  26  26   7
Starting search for low weight codewords...
Enumerating using 1 generator at a time:
     New codeword identified of weight 32, time 0.000
     New codeword identified of weight 28, time 0.000
     New codeword identified of weight 27, time 0.000
     New codeword identified of weight 25, time 0.000
     Discarding non-contributing rank 7 matrix
     New Relative Ranks:  26  26  26
   Completed Matrix  1:  lower =  4, upper = 25. Time so far: 0.000
     New codeword identified of weight 23, time 0.000
   Completed Matrix  2:  lower =  5, upper = 23. Time so far: 0.000
   Completed Matrix  3:  lower =  6, upper = 23. Time so far: 0.000
Enumerating using 2 generators at a time:
     New codeword identified of weight 20, time 0.000
   Completed Matrix  1:  lower =  7, upper = 20. Time so far: 0.000
   Completed Matrix  2:  lower =  8, upper = 20. Time so far: 0.000
   Completed Matrix  3:  lower =  9, upper = 20. Time so far: 0.000
Enumerating using 3 generators at a time:
     New codeword identified of weight 19, time 0.000
   Completed Matrix  1:  lower = 10, upper = 19. Time so far: 0.000
   Completed Matrix  2:  lower = 11, upper = 19. Time so far: 0.000
   Completed Matrix  3:  lower = 12, upper = 19. Time so far: 0.000
Enumerating using 4 generators at a time:
```

```
    New codeword identified of weight 18, time 0.000
  Completed Matrix  1:  lower = 13, upper = 18. Time so far: 0.000
    New codeword identified of weight 17, time 0.000
  Completed Matrix  2:  lower = 14, upper = 17. Time so far: 0.010
  Completed Matrix  3:  lower = 15, upper = 17. Time so far: 0.010
Termination predicted with 5 generators at matrix 2
Enumerating using 5 generators at a time:
  Completed Matrix  1:  lower = 16, upper = 17. Time so far: 0.020
  Completed Matrix  2:  lower = 17, upper = 17. Time so far: 0.030
Final Results: lower = 17, upper = 17, Total time: 0.030
17
```

Verbose output can be invaluable on long minimum weight calculations.

The algorithm constructs different (equivalent) generator matrices, each of which have pivots in different column positions of the code, called its *information set*. A generator matrix's *relative rank* is the size of its information set independent from the previously constructed matrices.

The algorithm proceeds by enumerating all combinations derived from r generators, for each successive r. Once r exceeds the difference between the actual rank of a matrix (i.e., the dimension), and its relative rank, then the lower bound on the minimum weight will increment by 1 for that step.

The upper bound on the minimum weight is determined by the minimum weight of codewords that are enumerated. Once these bounds meet the computation is complete.

`MinimumWeightBounds(C)`

Return the currently known lower and upper bounds on the minimum weight of code C.

`ResetMinimumWeightBounds(C)`

Undefine the minimum weight of the code C if it is known, and reset any known bounds on its value.

`VerifyMinimumDistanceLowerBound(C, d)`

`RankLowerBound`	RngIntElt	*Default :* 0
`MaximumTime`	RngIntElt	*Default :* ∞

The minimum weight algorithm is executed until it determines whether or not d is a lower bound for the minimum weight of the code C. (See the description of the function `MinimumWeight` for information on the parameters `RankLowerBound` and `MaximumTime` and on the verbose output). Three values are returned. The first of these is a boolean value, taking the value `true` if and only if d is verified to be a lower bound for the minimum weight of C, (`false` if the calculation is aborted due to time restrictions). The second return value is the best available lower bound for the minimum weight of C, and the third is a boolean which is `true` if this value is the actual minimum weight of C.

`VerifyMinimumDistanceUpperBound(C, d)`

`VerifyMinimumWeightUpperBound(C, d)`

`RankLowerBound`	RngIntElt	*Default :* 0
`MaximumTime`	RngIntElt	*Default :* ∞

The minimum weight algorithm is executed until it determines whether or not d is an upper bound for the minimum weight of the code C. (See the description of the function `MinimumWeight` for information on the parameters `RankLowerBound` and `MaximumTime` and on the verbose output). Three values are returned. The first of these is a boolean value, taking the value `true` if and only if d is verified to be an upper bound for the minimum weight of C, (`false` if the calculation is aborted due to time restrictions). The second return value is the best available upper bound for the minimum weight of C, and the third is a boolean which is `true` if this value is the actual minimum weight of C.

`MinimumWord(C)`

Return one word of the code C having minimum weight.

`MinimumWords(C)`

`NumWords`	RngIntElt	*Default :*
`Method`	MonStgElt	*Default :* "*Auto*"
`RankLowerBound`	RngIntElt	*Default :* ∞
`MaximumTime`	RngReSubElt	*Default :* ∞

Given a linear code C, return the set of all words of C having minimum weight. If `NumWords` is set to a non-negative integer, then the algorithm will terminate after that total of words have been found. Similarly, if `MaximumTime` then the algorithm will abort if the specified time limit expires.

A variation of the Zimmermann minimum weight algorithm is generally used to collect the minimum words, although in some cases (such as small codes) a brute force enumeration may be used. When the parameter `Method` is set to the default `"Auto"` then the method is internally chosen. The user can specify which method they want using setting it to either `"Distribution"` or `"Zimmerman"`.

By setting the verbose flag `"Code"`, information about the progress of the computation can be printed.

Example H152E19

The function `BKLC(K, n, k)` returns the best known linear $[n, k]$-code over the field K. We use this function to construct the $[77, 34, 16]$ best known linear code and confirm a lower bound on its minimum weight (which is not as good as its actual minimum weight). We check to see whether the minimum weight of this code is at least 11 and in doing so we will actually get a slightly better bound, though it will be still less than the true minimum weight. Since the function `BLKC`

will set the true minimum weight, it is first necessary to reset the bounds so that the minimum weight data is lost.

```
> a := BKLC(GF(2),77,34);
> a:Minimal;
[77, 34, 16] Linear Code over GF(2)
> ResetMinimumWeightBounds(a);
> MinimumWeightBounds(a);
1 44
> a:Minimal;
[77, 34] Linear Code over GF(2)
> SetVerbose("Code",true);
> IsLB, d_lower, IsMinWeight := VerifyMinimumWeightLowerBound(a, 11);
Linear Code over GF(2) of length 77 with 34 generators. Is not cyclic
Lower Bound: 1, Upper Bound: 44
Using congruence d mod 4 = 0
Constructed 3 distinct generator matrices
Relative Ranks:  34  34   6
Starting search for low weight codewords...
     Discarding non-contributing rank 6 matrix
Enumerating using 1 generator at a time:
     New codeword identified of weight 20, time 0.000
     New codeword identified of weight 16, time 0.000
   Completed Matrix  1:  lower =  4, upper = 16. Time so far: 0.000
   Completed Matrix  2:  lower =  4, upper = 16. Time so far: 0.000
Enumerating using 2 generators at a time:
   Completed Matrix  1:  lower =  8, upper = 16. Time so far: 0.000
   Completed Matrix  2:  lower =  8, upper = 16. Time so far: 0.000
Enumerating using 3 generators at a time:
   Completed Matrix  1:  lower =  8, upper = 16. Time so far: 0.000
   Completed Matrix  2:  lower =  8, upper = 16. Time so far: 0.000
Enumerating using 4 generators at a time:
   Completed Matrix  1:  lower = 12, upper = 16. Time so far: 0.010
Final Results: lower = 12, upper = 16, Total time: 0.010
> IsLB;
true
> d_lower, IsMinWeight;
12 false
```

`IncludeAutomorphism(~C, p)`

`IncludeAutomorphism(~C, G)`

Given some automorphism p or group of automorphisms G of the code C, which can either be a permutation of the columns or a full monomial permutation of the code. Then include these automorphism in the known automorphisms subgroup. Automorphisms with long cycles that can aid the minimum weight calculation should be added in this way.

`KnownAutomorphismSubgroup(C)`

Return the maximally known subgroup of the full group of automorphisms of the code C.

152.8.2 The Weight Distribution

`WeightDistribution(C)`

Determine the weight distribution for the code C. The distribution is returned in the form of a sequence of tuples, where the i-th tuple contains the i-th weight, w_i say, and the number of codewords having weight w_i.

`WeightDistribution(C, u)`

Determine the weight distribution of the coset $C + u$ of the linear code C. The distribution is returned in the form of a sequence of tuples, where the i-th tuple contains the i-th weight, w_i say, and the number of codewords having weight w_i.

`DualWeightDistribution(C)`

The weight distribution of the dual code of C (see `WeightDistribution`).

Example H152E20

We construct the second order Reed–Muller code of length 64, and calculate the its weight distribution and that of its dual code.

```
> R := ReedMullerCode(2, 6);
> #R;
4194304
> WeightDistribution(R);
[ <0, 1>, <16, 2604>, <24, 291648>, <28, 888832>, <32, 1828134>, <36, 888832>,
<40, 291648>, <48, 2604>, <64, 1> ]
> D := Dual(R);
> #D;
4398046511104
> time WeightDistribution(D);
[ <0, 1>, <8, 11160>, <12, 1749888>, <14, 22855680>, <16, 232081500>, <18,
1717223424>, <20, 9366150528>, <22, 38269550592>, <24, 119637587496>, <26,
286573658112>, <28, 533982211840>, <30, 771854598144>, <32, 874731154374>,
<34, 771854598144>, <36, 533982211840>, <38, 286573658112>, <40,
```

```
119637587496>, <42, 38269550592>, <44, 9366150528>, <46, 1717223424>,
<48, 232081500>, <50, 22855680>, <52, 1749888>, <56, 11160>, <64, 1> ]
```

`PartialWeightDistribution(C, ub)`

Return the weight distribution of the code C up to the specified upper bound. This function uses the minimum weight collection to collect word sets.

152.8.3 The Weight Enumerator

`WeightEnumerator(C)`

The (Hamming) weight enumerator $W_C(x, y)$ for the linear code C. The weight enumerator is defined by

$$W_C(x, y) = \sum_{u \in C} x^{n-wt(u)} y^{wt(u)}.$$

`WeightEnumerator(C, u)`

The (Hamming) weight enumerator $W_{C+u}(x, y)$ for the coset $C + u$.

`CompleteWeightEnumerator(C)`

The complete weight enumerator $\mathcal{W}_C(z_0, \ldots, z_{q-1})$ for the linear code C where q is the size of the alphabet K of C. Let the q elements of K be denoted by $\omega_0, \ldots, \omega_{q-1}$. If K is a prime field, we let ω_i be i (i.e. take the natural representation of each number). If K is a non-prime field, we let ω_0 be the zero element of K and let ω_i be α^{i-1} for $i = 1 \ldots q-1$ where α is the primitive element of K. Now for a codeword u of C, let $s_i(u)$ be the number of components of u equal to ω_i. The complete weight enumerator is defined by

$$\mathcal{W}_C(z_0, \ldots, z_{q-1}) = \sum_{u \in C} {z_0}^{s_0(u)} \ldots {z_{q-1}}^{s_{q-1}(u)}.$$

`CompleteWeightEnumerator(C, u)`

The complete weight enumerator $\mathcal{W}_{C+u}(z_0, \ldots, z_{q-1})$ for the coset $C + u$.

Example H152E21

We construct the cyclic ternary code of length 11 with generator polynomial $t^5 + t^4 + 2t^3 + t^2 + 2$ and calculate both its weight enumerator and its complete weight enumerator. To ensure the polynomials print out nicely, we assign names to the polynomial ring indeterminates in each case. These names will persist if further calls to `WeightEnumerator` and `CompleteWeightEnumerator` over the same alphabet are made.

```
> R<t> := PolynomialRing(GF(3));
> C := CyclicCode(11, t^5 + t^4 + 2*t^3 + t^2 + 2);
> W<x, y> := WeightEnumerator(C);
> W;
x^11 + 132*x^6*y^5 + 132*x^5*y^6 + 330*x^3*y^8 + 110*x^2*y^9 + 24*y^11
> CW<u, v, w> := CompleteWeightEnumerator(C);
> CW;
u^11 + 11*u^6*v^5 + 55*u^6*v^3*w^2 + 55*u^6*v^2*w^3 + 11*u^6*w^5 +
      11*u^5*v^6 + 110*u^5*v^3*w^3 + 11*u^5*w^6 + 55*u^3*v^6*w^2 +
      110*u^3*v^5*w^3 + 110*u^3*v^3*w^5 + 55*u^3*v^2*w^6 + 55*u^2*v^6*w^3 +
      55*u^2*v^3*w^6 + v^11 + 11*v^6*w^5 + 11*v^5*w^6 + w^11
```

The vector $u = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)$ does not lie in the code C and can be taken as a coset leader. We determine the weight enumerator of the coset containing u.

```
> u := AmbientSpace(C)![0,0,0,0,0,0,0,0,0,0,1];
> Wu := WeightEnumerator(C, u);
> Wu;
x^10*y + 30*x^7*y^4 + 66*x^6*y^5 + 108*x^5*y^6 + 180*x^4*y^7 + 165*x^3*y^8 +
    135*x^2*y^9 + 32*x*y^10 + 12*y^11
```

152.8.4 The MacWilliams Transform

`MacWilliamsTransform(n, k, q, W)`

Let C be a hypothetical $[n, k]$ linear code over a finite field of cardinality q. Let W be the weight distribution of C (in the form as returned by the function `WeightDistribution`). This function applies the MacWilliams transform to W to obtain the weight distribution W' of the dual code of C. The transform is a combinatorial algorithm based on n, k, q and W alone. Thus C itself need not exist—the function simply works with the sequence of integer pairs supplied by the user. Furthermore, if W is not the weight distribution of an actual code, the result W' will be meaningless and even negative weights may be returned.

`MacWilliamsTransform(n, k, K, W)`

Let C be a hypothetical $[n, k]$ linear code over a finite field K. Let W be the complete weight enumerator of C (in the form as returned by the function `CompleteWeightEnumerator`). This function applies the MacWilliams transform to W to obtain the complete weight enumerator W' of the dual code of C. The transform is a combinatorial algorithm based on K, n, k, and W alone. Thus C itself need not exist—the function simply manipulates the given polynomial. Furthermore, if W is not the weight distribution of an actual code, the weight enumerator W' will be meaningless.

Example H152E22

Let us suppose there exists a $[31, 11]$ code C over F_2 that has complete weight enumerator

$$u^{31} + 186u^{20}v^{11} + 310u^{19}v^{12} + 527u^{16}v^{15} + 527u^{15}v^{16} + 310u^{12}v^{19} + 186u^{11}v^{20} + v^{31}$$

We compute the weight distribution and the complete weight enumerator of the dual of the hypothetical code C.

```
> W := [ <0, 1>, <11, 186>, <12, 310>, <15, 527>, <16, 527>,
>        <19, 310>, <20, 186>, <31, 1> ];
>  MacWilliamsTransform(31, 11, 2, W);
[ <0, 1>, <6, 806>, <8, 7905>, <10, 41602>, <12, 142600>, <14,
251100>, <16, 301971>, <18, 195300>, <20, 85560>, <22, 18910>, <24,
2635>, <26, 186> ]
> R<u, v> := PolynomialRing(Integers(), 2);
> CWE := u^31 + 186*u^20*v^11 + 310*u^19*v^12 + 527*u^16*v^15 + 527*u^15*v^16 +
>          310*u^12*v^19 + 186*u^11*v^20 + v^31;
> MacWilliamsTransform(31, 11, GF(2), CWE);
u^31 + 806*u^25*v^6 + 7905*u^23*v^8 + 41602*u^21*v^10 + 142600*u^19*v^12 +
      251100*u^17*v^14 + 301971*u^15*v^16 + 195300*u^13*v^18 + 85560*u^11*v^20 +
      18910*u^9*v^22 + 2635*u^7*v^24 + 186*u^5*v^26
```

152.8.5 Words

The functions in this section only apply to codes over finite fields.

`Words(C, w: parameters)`

`NumWords`	RngIntElt	*Default :*
`Method`	MonStgElt	*Default :* "*Auto*"
`RankLowerBound`	RngIntElt	*Default :* ∞
`MaximumTime`	RngReSubElt	*Default :* ∞

Given a linear code C, return the set of all words of C having weight w. If `NumWords` is set to a non-negative integer c, then the algorithm will terminate after that total

of words have been found. Similarly, if `MaximumTime` then the algorithm will abort if the specified time limit expires.

There are two methods for collecting words, one based on the Zimmermann minimum weight algorithm, and a brute force type calculation. When the parameter `Method` is set to the default `"Auto"` then the method is internally chosen. The user can specify which method they want using setting it to either `"Distribution"` or `"Zimmerman"`.

By setting the verbose flag `"Code"`, information about the progress of the computation can be printed.

`NumberOfWords(C, w)`

Given a linear code C, return the number of words of C having weight w.

`WordsOfBoundedWeight(C, l, u: parameters)`

`Cutoff`	RngIntElt	*Default :* ∞
`StoreWords`	BoolElt	*Default :* `true`

Given a linear code C, return the set of all words of C having weight between l and u, inclusive. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found.

If `StoreWords` is true then any words of a *single weight* generated will be stored internally.

`ConstantWords(C, i)`

Given a linear code C, return the set of all words of C which have weight i and which consist of zeros and ones alone.

`NumberOfConstantWords(C, i)`

Given a linear code C, return the number of words of C which have weight i and which consist of zeros and ones alone.

Example H152E23

We construct the words of weight 11 and also the constant (zero-one) words of weight 11 in the length 23 cyclic code over $\mathbf{F}_3$ that is defined by the generator polynomial $x^{11} + x^{10} + x^9 + 2x^8 + 2x^7 + x^5 + x^3 + 2$.

```
> R<x> := PolynomialRing(GF(3));
> f := x^11 + x^10 + x^9 + 2*x^8 + 2*x^7 + x^5 + x^3 + 2;
> C := CyclicCode(23, f);
> C;
[23, 12, 8] BCH code (d = 5, b = 1) over GF(3)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 1 0 1 0 2 2 1 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 2 0 2 1 2 1 1 0 1 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 1 2 0 2 1 2 1 1 0 1]
[0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 1 0 1 1 0 2 0 2]
```

```
[0 0 0 0 1 0 0 0 0 0 0 0 2 1 0 2 1 1 1 0 2 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 2 1 2 2 0 1 2 1 1 2]
[0 0 0 0 0 0 1 0 0 0 0 0 2 1 2 2 2 0 0 0 1 2 2]
[0 0 0 0 0 0 0 1 0 0 0 0 2 2 1 0 2 0 0 2 2 2 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 2 2 1 0 2 0 0 2 2 2]
[0 0 0 0 0 0 0 0 0 1 0 0 2 0 2 0 1 1 2 2 2 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 2 0 2 0 1 1 2 2 2 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 2 0 2 0 1 1 2 2 2]
> time WeightDistribution(C);
[ <0, 1>, <8, 1518>, <9, 2530>, <11, 30912>, <12, 30912>, <14, 151800>, <15,
91080>, <17, 148764>, <18, 49588>, <20, 21252>, <21, 3036>, <23, 48> ]
Time: 0.030
```

Note that the minimum distance is 8. We calculate all words of weight 11 and the constant words of weight 11.

```
1518
> time W11 := Words(C, 11);
Time: 0.350
> #W11;
30912
> ZOW11 := ConstantWords(C, 11);
> #ZOW11;
23
> ZOW11 subset W11;
true
```

152.8.6 Covering Radius and Diameter

`CosetDistanceDistribution(C)`

Given a linear code C, determine the coset distance distribution of C, relative to C. The distance between C and a coset D of C is the Hamming weight of a vector of minimum weight in D. The distribution is returned as a sequence of pairs comprising a distance d and the number of cosets that are distance d from C.

`CoveringRadius(C)`

The covering radius of the linear code C.

`Diameter(C)`

The diameter of the code C (the largest weight of the codewords of C).

Example H152E24

We construct the second order Reed–Muller code of length 32, and calculate its coset distance distribution.

```
> R := ReedMullerCode(2, 5);
> R:Minimal;
[32, 16, 8] Reed-Muller Code (r = 2, m = 5) over GF(2)
> CD := CosetDistanceDistribution(R);
> CD;
[ <0, 1>, <1, 32>, <2, 496>, <3, 4960>, <4, 17515>, <5, 27776>, <6, 14756> ]
```

From the dimension of the code we know C has 2^{16} cosets. The coset distance distribution tells us that there are 32 cosets at distance 1 from C, 496 cosets are distance 2, etc. We confirm that all cosets are represented in the distribution.

```
> &+ [ t[2] : t in CD ];
65536
> CoveringRadius(R);
6
> Diameter(R);
32
> WeightDistribution(R);
[ <0, 1>, <8, 620>, <12, 13888>, <16, 36518>, <20, 13888>, <24, 620>, <32, 1> ]
```

The covering radius gives the maximum distance of any coset from the code, and, from the coset distance distribution, we see that this maximum distance is indeed 6. We can confirm the value (32) for the diameter by examining the weight distribution and seeing that 32 is the largest weight of a codeword.

152.9 Families of Linear Codes

152.9.1 Cyclic and Quasicyclic Codes

`CyclicCode(u)`

Given a vector u belonging to the R-space $R^{(n)}$, construct the $[n, k]$ cyclic code generated by the right cyclic shifts of the vector u.

`CyclicCode(n, T, K)`

Given a positive integer n and a set or sequence T of primitive n-th roots of unity from a finite field L, together with a subfield K of L, construct the cyclic code C over K of length n, such that the generator polynomial for C is the polynomial of least degree having the elements of T as roots.

`QuasiCyclicCode(n, Gen)`

Constructs the quasi-cyclic code of length n with generator polynomials given by the sequence of polynomials in Gen. Created by `HorizontalJoin` of each `GeneratorMatrix` from the `CyclicCode`'s generated by the polynomials in Gen. Requires that $|Gen| \mid n$.

`QuasiCyclicCode(Gen)`

Constructs the quasi-cyclic code of length n generated by simultaneous cyclic shifts of the vectors in Gen.

`QuasiCyclicCode(n, Gen, h)`

Constructs the quasi-cyclic code of length n with generator polynomials given by the sequence of polynomials in Gen. The `GeneratorMatrix`'s are joined 2 dimensionally, with height h. Requires that $h \mid (|Gen|)$ and $(|Gen|/h) \mid n$.

`QuasiCyclicCode(Gen, h)`

Constructs the quasi cyclic code generated by simultaneous cyclic shifts of the vectors in Gen, arranging them two dimensionally with height h.

`ConstaCyclicCode(n, f, alpha)`

Return the length n code generated by consta-cyclic shifts by α of the coefficients of f.

`QuasiTwistedCyclicCode(n, Gen, alpha)`

Construct the quasi-twisted cyclic code of length n pasting together the constacyclic codes with parameter α generated by the polynomials in Gen.

`QuasiTwistedCyclicCode(Gen, alpha)`

Construct the quasi-twisted cyclic code generated by simultaneous constacyclic shifts w.r.t. α of the codewords in Gen.

Example H152E25

Let the m factors of $x^n - 1$ be $f_i(x)$,$i = 0, \ldots, m$ in any particular order. Then we can construct a chain of polynomials $g_k(x) = \prod_{i=0}^{k} f_i(x)$ such that $g_k(x) \mid g_{k+1}(x)$. This chain of polynomials will generate a nested chain of cyclic codes of length n, which is illustrated here for $n = 7$.

```
> P<x> := PolynomialRing(GF(2));
> n := 7;
> F := Factorization(x^n-1);
> F;
[
    <x + 1, 1>,
    <x^3 + x + 1, 1>,
    <x^3 + x^2 + 1, 1>
]
```

```
> Gens := [ &*[F[i][1]:i in [1..k]] : k in [1..#F] ];
> Gens;
[
    x + 1,
    x^4 + x^3 + x^2 + 1,
    x^7 + 1
]
> Codes := [ CyclicCode(n, Gens[k]) : k in [1..#Gens] ];
> Codes;
[
    [7, 6, 2] Cyclic Code over GF(2)
    Generator matrix:
    [1 0 0 0 0 0 1]
    [0 1 0 0 0 0 1]
    [0 0 1 0 0 0 1]
    [0 0 0 1 0 0 1]
    [0 0 0 0 1 0 1]
    [0 0 0 0 0 1 1],
     [7, 3, 4] Cyclic Code over GF(2)
    Generator matrix:
    [1 0 0 1 0 1 1]
    [0 1 0 1 1 1 0]
    [0 0 1 0 1 1 1],
    [7, 0, 7] Cyclic Code over GF(2)
]
> { Codes[k+1] subset Codes[k] : k in [1..#Codes-1] };
{ true }
```

152.9.2 BCH Codes and their Generalizations

`BCHCode(K, n, d, b)`

`BCHCode(K, n, d)`

Given a finite field $K = F_q$ and positive integers n, d and b such that $\gcd(n, q) = 1$, we define m to be the smallest integer such that $n \mid (q^m - 1)$, and α to be a primitive n-th root of unity in the degree m extension of K, $GF(q^m)$. This function constructs the BCH code of designated distance d as the cyclic code with generator polynomial

$$g(x) = \text{lcm}\{m_1(x), \cdots, m_{d-1}(x)\}$$

where $m_i(x)$ is the minimum polynomial of α^{b+i-1}. The BCH code is an $[n, \geq (n - m(d-1)), \geq d]$ code over K. If b is omitted its value is taken to be 1, in which case the corresponding code is a *narrow sense* BCH code.

Example H152E26

We construct a BCH code of length 13 over GF(3) and designated minimum distance 3

```
> C := BCHCode(GF(3), 13, 3);
> C;
[13, 7, 4] BCH code (d = 3, b = 1) over GF(3)
Generator matrix:
[1 0 0 0 0 0 0 1 2 1 2 2 2]
[0 1 0 0 0 0 0 1 0 0 0 1 1]
[0 0 1 0 0 0 0 2 2 2 1 1 2]
[0 0 0 1 0 0 0 1 1 0 1 0 0]
[0 0 0 0 1 0 0 0 1 1 0 1 0]
[0 0 0 0 0 1 0 0 0 1 1 0 1]
[0 0 0 0 0 0 1 2 1 2 2 2 1]
```

`GoppaCode(L, G)`

Let K be the field $GF(q)$, let $G(z) = G$ be a polynomial defined over the degree m extension field F of K (i.e. the field $GF(q^m)$) and let $L = [\alpha_1, \ldots, \alpha_n]$ be a sequence of elements of F such that $G(\alpha_i) \neq 0$ for all $\alpha_i \in L$. This function constructs the Goppa code $\Gamma(L, G)$ over K. If the degree of $G(z)$ is r, this is an $[n, k \geq n - mr, d \geq r + 1]$ code.

Example H152E27

We construct a Goppa code of length 31 over GF(2) with generator polynomial $G(z) = z^3 + z + 1$.

```
> q := 2^5;
> K<w> := FiniteField(q);
> P<z> := PolynomialRing(K);
> G := z^3 + z + 1;
> L := [w^i : i in [0 .. q - 2]];
> C := GoppaCode(L, G);
> C:Minimal;
[31, 16, 7] Goppa code (r = 3) over GF(2)
> WeightDistribution(C);
[ <0, 1>, <7, 105>, <8, 295>, <9, 570>, <10, 1333>, <11, 2626>,
 <12, 4250>, <13, 6270>, <14, 8150>, <15, 9188>, <16, 9193>,
 <17, 8090>, <18, 6240>, <19, 4270>, <20, 2590>, <21, 1418>,
 <22, 650>, <23, 195>, <24, 55>, <25, 36>, <26, 11> ]
```

`ChienChoyCode(P, G, n, S)`

Let P and G be polynomials over a finite field F, let n be an integer greater than one, and let S be a subfield of F. Suppose also that n is coprime to the cardinality of S, F is the splitting field of x^n-1 over S, P and G are both coprime to x^n-1 and both have degree less than n. This function constructs the Chien-Choy generalised BCH code with parameters P, G, n over S.

`AlternantCode(A, Y, r, S)`

`AlternantCode(A, Y, r)`

Let $A = [\alpha_1, \ldots, \alpha_n]$ be a sequence of n distinct elements taken from the degree m extension K of the finite field S, and let $Y = [y_1, \ldots, y_n]$ be a sequence of n non-zero elements from K. Let r be a positive integer. Given such A, Y, r, and S, this function constructs the alternant code $A(A, Y)$ over S. This is an $[n, k \geq n - mr, d \geq r + 1]$ code. If S is omitted, S is taken to be the prime subfield of K.

Example H152E28

We construct an alternant code over GF(2) based on sequences of elements in the extension field GF(2^4) of GF(2). The parameter r is taken to be 4, so the minimum weight 6 is greater than $r + 1$.

```
> q := 2^4;
> K<w> := GF(q);
> A := [w ^ i : i in [0 .. q - 2]];
> Y := [K ! 1 : i in [0 .. q - 2]];
> r := 4;
> C := AlternantCode(A, Y, r);
> C;
[15, 6, 6] Alternant code over GF(2)
Generator matrix:
[1 0 0 0 0 0 1 1 0 0 1 1 1 0 0]
[0 1 0 0 0 0 0 1 1 0 0 1 1 1 0]
[0 0 1 0 0 0 0 0 1 1 0 0 1 1 1]
[0 0 0 1 0 0 1 1 0 1 0 1 1 1 1]
[0 0 0 0 1 0 1 0 1 0 0 1 0 1 1]
[0 0 0 0 0 1 1 0 0 1 1 1 0 0 1]
```

`NonPrimitiveAlternantCode(n, m, r)`

Returns the $[n, k, d]$ non-primitive alternant code over $\mathbf{F}_2$, where $n - mr \leq k \leq n - r$ and $d \geq r + 1$.

`FireCode(h, s, n)`

Let K be the field $GF(q)$. Given a polynomial h in K[X], a nonnegative integer s, and a positive integer n, this function constructs a Fire code of length n with generator polynomial $h(X^s - 1)$.

`GabidulinCode(A, W, Z, t)`

Given sequences $A = [a_1, ...a_n]$, $W = [w_1, ...w_s]$, and $Z = [z_1, ...z_k]$, such that the $n+s$ elements of A and W are distinct and the elements of Z are non-zero, together with a positive integer t, construct the Gabidulin MDS code with parameters A, W, Z, t.

`SrivastavaCode(A, W, mu, S)`

Given sequences $A = [\alpha_1, ..., \alpha_n]$, $W = [w_1, ..., w_s]$ of elements from the extension field K of the finite field S, such that the elements of A are non-zero and the $n + s$ elements of A and W are distinct, together with an integer μ, construct the Srivastava code of parameters A, W, mu, over S.

`GeneralizedSrivastavaCode(A, W, Z, t, S)`

Given sequences $A = [\alpha_1, ..., \alpha_n]$, $W = [w_1, ..., w_s]$, and $Z = [z_1, ...z_k]$ of elements from the extension field K of the finite field S, such that the elements of A and Z are non-zero and the $n + s$ elements of A and W are distinct, together with a positive integer t, construct the generalized Srivastava code with parameters A, W, Z, t, over S.

152.9.3 Quadratic Residue Codes and their Generalizations

If p is an odd prime, the *quadratic residues modulo p* consist of the set of non-zero squares modulo p while the set of non-squares modulo p are termed the *quadratic nonresidues modulo p*.

`QRCode(K, n)`

Given a finite field $K = F_q$ and an odd prime n such that q is a quadratic residue modulo n, this function returns the quadratic residue code of length n over K. This corresponds to the cyclic code with generator polynomial $g_0(x) = \prod(x - \alpha^r)$, where α is a primitive n-th root of unity in some extension field of K, and the product is taken over all quadratic residues modulo p.

`GolayCode(K, ext)`

If the field K is $GF(2)$, construct the binary Golay code. If the field K is $GF(3)$, construct the ternary Golay code. If the boolean argument *ext* is `true`, construct the extended code in each case.

`DoublyCirculantQRCode(p)`

Given an odd prime p, this function returns the doubly circulant binary $[2p, p]$ code based on quadratic residues modulo p. A doubly circulant code has generator matrix of the form $[I \mid A]$, where A is a circulant matrix.

`DoublyCirculantQRCodeGF4(m, a)`

Given a prime power m that is greater than 2 and an integer a that is either 0 or 1, return a $[2m, m]$ doubly circulant linear code over GF(4). For details see [Gab02].

`BorderedDoublyCirculantQRCode(p, a, b)`

Given an odd prime p and integers a and b, this function returns the bordered doubly circulant binary $[2p + 1, p + 1]$ code based on quadratic residues modulo p. The construction is similar to that of a doubly circulant code except that the first p rows are extended by a mod 2 while the $p + 1$-th row is extended by b mod 2.

`TwistedQRCode(l, m)`

Given positive integers l and m, both coprime to 2, return a binary "twisted QR" code of length $l * m$.

`PowerResidueCode(K, n, p)`

Given a finite field $K = F_q$, a positive integer n and a prime p such that q is a p-th power residue modulo n, construct the p-th power residue code of length n.

Example H152E29

We construct a quadratic residue code of length 23 over GF(3).

```
> QRCode(GF(3), 23);
[23, 12, 8] Quadratic Residue code over GF(3)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 2 2 2 1 1 0 2 0 2 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 2 2 2 1 1 0 2 0 2 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 2 2 2 1 1 0 2 0 2]
[0 0 0 1 0 0 0 0 0 0 0 0 2 2 2 0 0 2 0 1 2 2 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 2 2 2 0 0 2 0 1 2 2]
[0 0 0 0 0 1 0 0 0 0 0 0 2 2 1 0 0 0 2 2 2 1 2]
[0 0 0 0 0 0 1 0 0 0 0 0 2 1 1 2 1 0 2 2 1 2 1]
[0 0 0 0 0 0 0 1 0 0 0 0 1 0 2 0 1 1 1 2 0 1 2]
[0 0 0 0 0 0 0 0 1 0 0 0 2 0 2 0 1 1 0 1 1 0 1]
[0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 2 1 2 0 2 1 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 2 1 2 0 2 1]
[0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 0 1 0 1 0 0 2]
```

152.9.4 Reed–Solomon and Justesen Codes

`ReedSolomonCode(K, d, b)`

`ReedSolomonCode(K, d)`

Given a finite field $K = F_q$ and a positive integer d, return the Reed–Solomon code of length $n = q - 1$ with design distance d. This corresponds to `BCHCode(K, q-1, d)`. For details see [MS78, p.294].

If b is given as a non-negative integer then the primitive element is first raised to the b-th power.

`ReedSolomonCode(n, d)`

`ReedSolomonCode(n, d, b)`

Given an integer n such that $q = n + 1$ is a prime power, and a positive integer d, return the Reed–Solomon code over F_q of length n and designed minimum distance d.

If b is given as a non-negative integer then the primitive element is first raised to the b-th power.

`GRSCode(A, V, k)`

Let $A = [\alpha_1, \ldots, \alpha_n]$ be a sequence of n distinct elements taken from the finite field K, and let $V = [v_1, \ldots, v_n]$ be a sequence of n non-zero elements from K. Let k be a non-negative integer. Given such A, V, and k, this function constructs the generalized Reed–Solomon code $GRS_k(A, V)$ over K. This is an $[n, k' \leq k]$ code. For details see [MS78, p.303].

`JustesenCode(N, K)`

Given an integer N such that $N = 2^m - 1$ and a positive integer K, construct the binary linear Justesen code of length $2mN$ and dimension mK. For details see [MS78, p.307].

Example H152E30

We construct a generalized Reed–Solomon code over GF(2) based on sequences of elements in the extension field GF(2^3) of GF(2). The parameter k is taken to be 3, so the dimension 3 is at most k.

```
> q := 2^3;
> K<w> := GF(q);
> A := [w ^ i : i in [0 .. q - 2]];
> V := [K ! 1 : i in [0 .. q - 2]];
> k := 3;
> C := GRSCode(A, V, k);
[7, 3, 5] GRS code over GF(2^3)
Generator matrix:
[  1   0   0 w^3   w   1 w^3]
[  0   1   0 w^6 w^6   1 w^2]
```

```
[  0   0   1 w^5 w^4   1 w^4]
```

152.9.5 Maximum Distance Separable Codes

`MDSCode(K, k)`

Given a finite field $GF(q = 2^m)$, this function constructs the $[q + 1, k, q - k + 2]$ maximum distance separable code.

152.10 New Codes from Existing

The operations described here produce a new code by modifying in some way the codewords of a given code.

152.10.1 Standard Constructions

`AugmentCode(C)`

Given an $[n, k]$ binary code C, construct a new code C' by including the all-ones vector with the words of C (provided that it is not already in C).

`CodeComplement(C, C1)`

Given a subcode $C1$ of C, return a code $C2$ such that $C = C1 + C2$.

`DirectSum(C, D)`

Given an $[n_1, k_1]$ code C and an $[n_2, k_2]$ code D, both over the same field F, construct the direct sum of C and D. The direct sum consists of all vectors $u|v$, where $u \in C$ and $v \in D$.

`DirectSum(Q)`

Given a sequence of codes $Q = [C_1, \ldots, C_r]$, all defined over the same field F, construct the direct sum of the C_i.

`DirectProduct(C, D)`

`ProductCode(C, D)`

Given an $[n_1, k_1]$ code C and an $[n_2, k_2]$ code D, both over the same ring R, construct the direct product of C and D. The direct product has length $n_1 \cdot n_2$, dimension $k_1 \cdot k_2$, and its generator matrix is the Kronecker product of the basis matrices of C and D.

`ExtendCode(C)`

Given an $[n, k, d]$ code C form a new code C' from C by adding the appropriate extra coordinate to each vector of C such that the sum of the coordinates of the extended vector is zero. (Thus if C is a binary code, the construction will add a 0 at the end of every codeword having even weight, and a 1 at the end of every codeword having odd weight.)

`ExtendCode(C, n)`

Return the code C extended n times.

`PadCode(C, n)`

Add n zeros to the end of each codeword of C.

`ExpurgateCode(C)`

Construct a new code by deleting all the code words of C having odd weight.

`ExpurgateCode(C, L)`

The sequence L consists of codewords from C. The result is obtained by deleting the words in L from C.

`ExpurgateWeightCode(C, w)`

Delete a subspace generated by a word of weight w.

`LengthenCode(C)`

Given an $[n, k]$ binary code C, construct a new code by first adding the all-ones codeword, and then extending it by adding an overall parity check.

`PlotkinSum(C1, C2)`

Given two codes over the same alphabet, return the code consisting of all vectors of the form $u|u + v$, where $u \in C1$ and $v \in C2$. Zeros are appended where needed to make up any length differences in the two codes. The result is a $[n_1 + \max\{n_1, n_2\}, k_1 + k_2, \min\{2 * d_1, d_2\}]$ code.

`PlotkinSum(C1, C2, C3:` *parameters*`)`

`a` FldFinElt *Default :* -1

Given three codes over the same alphabet, return the code consisting of all vectors of the form $u|u + a * v|u + v + w$, where $u \in C1$, $v \in C2$ and $w \in C3$. Zeros are appended where needed to make up any length differences in the three codes. The result for the default case of `a := -1` is a $[n_1 + \max\{n_1, n_2\} + \max\{n_1, n_2, n_3\}, k_1 + k_2 + k_3, \min\{3 * d_1, 2 * d_2, d_3\}]$ code.

`PunctureCode(C, i)`

Given an $[n, k]$ code C, and an integer i, $1 \leq i \leq n$, construct a new code C' by deleting the i-th coordinate from each code word of C.

`PunctureCode(C, S)`

Given an $[n, k]$ code C and a set S of distinct integers $\{i_1, \cdots, \mathrm{i}_r\}$ each of which lies in the range $[1, n]$, construct a new code C' by deleting the components $i_1, \cdots, i_r$ from each code word of C.

`ShortenCode(C, i)`

Given an $[n, k]$ code C and an integer i, $1 \leq i \leq n$, construct a new code from C by selecting only those codewords of C having a zero as their i-th component and deleting the i-th component from these codewords. Thus, the resulting code will have length $n - 1$.

`ShortenCode(C, S)`

Given an $[n, k]$ code C and a set S of distinct integers $\{i_1, \cdots, \mathrm{i}_r\}$, each of which lies in the range $[1, n]$, construct a new code from C by selecting only those codewords of C having zeros in each of the coordinate positions $i_1, \cdots, i_r$, and deleting these components. Thus, the resulting code will have length $n - r$.

Example H152E31

Using only two simple `RepetitionCode`'s and several standard constructions, we create a $[12, 4, 6]$ code. This is the best possible minimum weight for a code of this length and dimension, as is the minimum weight for all codes produced in this example.
Instead of printing each individual code out, the codes are named by convention as c_n_k_d, where n, k, d represent the `Length`,`Dimension` and `MinimumWeight` respectively.

```
> c_4_1_4 := RepetitionCode(GF(2),4);
> c_6_1_6 := RepetitionCode(GF(2),6);
> c_4_3_2 := Dual( c_4_1_4 );
> c_8_4_4 := PlotkinSum( c_4_3_2 , c_4_1_4 );
> c_7_4_3 := PunctureCode( c_8_4_4 , 8 );
> c_6_3_3 := ShortenCode(  c_7_4_3 , 7 );
> c_12_4_6 := PlotkinSum( c_6_3_3 , c_6_1_6 );
> c_12_4_6;
[12, 4, 6] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 1 0 0 1 1 0 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 1 1 0 0 1 1 1 1]
[0 0 0 0 0 0 1 1 1 1 1 1]
```

152.10.2 Changing the Alphabet of a Code

`ExtendField(C, L)`

Given an $[n, k, d]$ code C defined over the finite field K, and an extension field L of K, construct the code C' over L corresponding to C. The function also returns the embedding map from C into C'.

`LinearCode(C, S)`

Given an $[n, k, d]$ code C defined over the finite field K, and a subfield S of K such that the degree of K over S is m, construct a new $[N = mn, k, D \geq d]$ code C' over S by replacing each component of each codeword of C by its representation as a vector over S. The function also returns the isomorphism from C onto C'.

`SubfieldRepresentationCode(C, K)`

Given a linear code over $GF(q^m)$, return a code whose codewords are obtained from those of C by expanding each coordinate in $GF(q^m)$ as a vector of dimension m over $K = GF(q)$.

`SubfieldRepresentationParityCode(C, K)`

Given a linear code over $GF(q^m)$, return a code whose codewords are obtained from those of C by expanding each coordinate in $GF(q^m)$ as a vector of dimension $m+1$ over $K = GF(q)$, including a parity check bit.

`SubfieldSubcode(C, S)`
`RestrictField(C, S)`
`SubfieldSubcode(C)`
`RestrictField(C)`

Given an $[n, k, d]$ code C defined over the field K, and a subfield S of K, construct a new $[n, K \leq k, D \geq d]$ code C' over S consisting of the codewords of C which have all their components in S. If S is omitted, it is taken to be the prime subfield of K. The function also returns the restriction map from C to the subfield subcode C'.

`SubfieldCode(C, S)`

Given an $[n, k]$ code C defined over the field K, and a subfield S of K, such that K is a degree m extension of S, construct a new $[n, k']$ code over S by expanding each element of K as a column vector over S. The new code will have $k' \leq km$.

`Trace(C, F)`
`Trace(C)`

Given a code C defined over the field K, and a subfield F of K, construct a new code C' over F consisting of the traces with respect to F of each of the codewords of L. If F is omitted, it is taken to be the prime subfield of K.

152.10.3 Combining Codes

`C1 cat C2`

Given codes $C1$ and $C2$, both defined over the same field K, return the concatenation C of $C1$ and $C2$. If A and B are the generator matrices of $C1$ and $C2$, respectively, the concatenation of $C1$ and $C2$ is the code with generator matrix whose rows consist of each row of A concatenated with each row of B.

`Juxtaposition(C1, C2)`

Given an $[n_1, k, d_1]$ code $C1$ and an $[n_2, k, d_2]$ code $C2$ of the same dimension, where both codes are defined over the same field K, the function returns a $[n_1 + n_2, k, \geq d_1 + d_2]$ code whose generator matrix is `HorizontalJoin(A, B)`, where A and B are the generator matrices for codes $C1$ and $C2$, respectively.

`ConcatenatedCode(O, I)`

Given a $[N, K, D]$-code O defined over $GF(q^k)$ and a $[n, k, d]$-code I defined over $GF(q)$, construct the $[Nn, Kk, \delta \geq dD]$ concatenated code by taking O as outer code and I as inner code.

Example H152E32

We use the function `ConcatenatedCode` to construct a $[69, 19, 22]$ code over $GF(2)$, this being the best known code for this length and dimension. While it is theoretically possible for a code of minimum weight up to 24 to exist, the best binary $[69, 19]$ code at the time of writing (July 2001) has minimum weight 22.

```
> C1 := ShortenCode( QRCode(GF(4),29) , {24..29} );
> C1:Minimal;
[23, 9] Linear Code over GF(2^2)
> C2 := ConcatenatedCode( C1 , CordaroWagnerCode(3) );
> C2:Minimal;
[69, 18] Linear Code over GF(2)
> res := C2 + RepetitionCode(GF(2),69);
> res:Minimal;
[69, 19] Linear Code over GF(2)
> MinimumWeight(res);
22
> res:Minimal;
[69, 19, 22] Linear Code over GF(2)
```

`ConstructionX(C1, C2, C3)`

Let C_1, C_2 and C_3 be codes with parameters $[n_1, k_1, d_1]$, $[n_2, k_2, d_2]$ and $[n_3, k_3, d_3]$, respectively, where $C2$ is a union of b cosets of $C1$, (so $n_1 = n_2$ and $k_2 \leq k_1$) and $k_1 = k_2 + k_3$. The construction divides $C2$ into a union of cosets of $C1$ and attaches a different codeword of $C3$ to each coset. The new code has parameters $[n_1 + n_3, k_1, \geq \min\{\ d_2, d_1 + d_3\}]$. For further details see [MS78, p.581].

`ConstructionXChain(S, C)`

Given a sequence of codes S where all codes are subcodes of the first one, apply `ConstructionX` to $S[1], S[2]$ and C. Then compute the resulting subcodes from the other codes in S.

Example H152E33

We create a $[161, 29, 53]$ code by applying construction X to two BCH codes of length 127. To maximise the resulting minimum weight we take the best known $[34, 14]$ code as C_3. The construction sets a lower bound on the minimum weight, making the calculation of the true minimum weight much faster.

```
> SetPrintLevel("Minimal");
>
> C1 := BCHCode(GF(2), 127, 43);
> C2 := BCHCode(GF(2), 127, 55);
> C3 := BKLC(GF(2), 34, 14);
> C1; C2; C3;
[127, 29, 43] BCH code (d = 43, b = 1) over GF(2)
[127, 15, 55] BCH code (d = 55, b = 1) over GF(2)
[34, 14, 10] Linear Code over GF(2)
> CX := ConstructionX(C1, C2, C3);
> CX;
[161, 29] Linear Code over GF(2)
> time MinimumWeight(CX);
53
Time: 0.010
```

`ConstructionX3(C1, C2, C3, D1, D2)`

Given a chain of codes $C1 = [n, k_1, d_1]$, $C2 = [n, k_2, d_2]$, and $C3 = [n, k_3, d_3]$ with $k_3 < k_2 < k_1$, and suffix codes $D1 = [n_1, k_1 - k_2, e_1]$ and $D2 = [n_2, k_3 - k_2, e_2]$, construct a code $C = [n+n_1+n_2, k_1, \geq min\{d_3, d_1+e_1, d_2+e_2\}$. For further details see [MS78, p.583].

`ConstructionX3u(C1, C2, C3, D1, D2)`

Given two chains of codes $C1 = [n, k_1] \subset C2 = [n, k_2] \subset C3 = [n, k_3]$ and $D1 = [n', k_1 - k_3] \subset D2 = [n', k_2 - k_3]$, return the codes $C = [n+n', k_1] \subset C' = [n+n', k_2]$ using Construction X with $C1, C3$ and $D1$ resp. $C2, C3$ and $D2$.

Example H152E34

We construct a best known $[74, 43, 11]$ code using construction X3. From a chain of BCH subcodes, we take an subcode to get the appropriate length, then use construction X3 with the best possible codes $D1, D2$. Because the construction algorithm sets a lower bound on the minimum weight, then it is quick to calculate afterwards.

```
> SetPrintLevel("Minimal");
> C1 := ExtendCode( BCHCode(GF(2), 63, 7) );
> C2 := ExtendCode( BCHCode(GF(2), 63, 9) );
> C3 := ExtendCode( BCHCode(GF(2), 63, 11) );
> C1; C2; C3;
[64, 45, 8] Linear Code over GF(2)
[64, 39, 10] Linear Code over GF(2)
[64, 36, 12] Linear Code over GF(2)
> CC := SubcodeBetweenCode(C1, C2, 43);
> CC;
[64, 43] Linear Code over GF(2)
> MinimumWeight(CC);
8
> CX3 := ConstructionX3(CC, C2, C3,
>                 BKLC(GF(2), 7, 4), BKLC(GF(2), 3, 3));
> CX3;
[74, 43] Linear Code over GF(2)
> time MinimumWeight(CX3);
11
Time: 0.000
```

`ConstructionXX(C1, C2, C3, D2, D3)`

Let the parameters of codes C_1, C_2, C_3 be $[n_1, k, d_1], [n_1, k - l_2, d_2]$ and $[n_1, k - l_3, d_3]$ respectively, where C_2 and C_3 are subcodes of C_1. Codes D_2, D_3 must have dimensions l_2, l_3, with parameters $[n_2, l_2, \delta_2], [n_3, l_3, \delta_3]$ say. The construction breaks C_1 up into cosets of C_2 and C_3 with the relevant tails added from D_2 and D_3. If the intersection of C_1 and C_2 has minimum distance d_0 then the newly constructed code will have parameters $[n_1 + n_2 + n_3, k, min\{d_0, d_2 + \delta_2, d_3 + \delta_3, d_1 + \delta_2 + \delta_3\}]$. For further details see [All84].

Example H152E35

We construct a best known $[73, 38, 13]$ code C using construction XX. For C_1, C_2, C_3 we take three cyclic (or BCH) codes of length 63, while for D_1, D_2 we use two Best Known Codes.

```
> SetPrintLevel("Minimal");
> C1 := BCHCode(GF(2),63,10,57);
> P<x> := PolynomialRing(GF(2));
> p := x^28 + x^25 + x^22 + x^21 + x^20 + x^17 + x^16
>      + x^15 + x^9 + x^8 + x^6 + x^5 + x + 1;
```

```
> C2 := CyclicCode(63, p);
> C3 := BCHCode(GF(2), 63, 10, 58);
> C1; C2; C3;
[63, 38] BCH code (d = 10, b = 57) over GF(2)
[63, 35] Cyclic Code over GF(2)
[63, 32] BCH code (d = 10, b = 58) over GF(2)
> MinimumDistance(C1 meet C2);
12
```

So the minimum distance of the code produced by Construction XX must be at least 12.

```
> C := ConstructionXX(C1, C2, C3, BKLC(GF(2),3,3), BKLC(GF(2),7,6) );
> C;
[73, 38] Linear Code over GF(2)
MinimumDistance(C);
13
```

Thus the actual minimum distance is one greater than the lower bound guaranteed by Construction XX.

`ZinovievCode(I, O)`

The arguments are as follows: The first argument must be a sequence I containing an increasing chain of r codes with parameters,

$$[n, k_1, d_1]_q \subset [n, k_2, d_2]_q \subset \ldots \subset [n, k_r, d_r]_q$$

where $0 = k_0 < k_1 < k_2 < \ldots < k_r$, (the *inner codes*). The second argument must be a sequence O of r codes with parameters $[N, K_i, D_i]_{Q_i}$, where $Q_i = q^{e_i}$ and $e_i = k_i - k_{i-1}$ for $i = 1 \ldots r$ (the *outer codes*). The function constructs a generalised concatenated $[n * N, K, D]_q$ code is constructed, where $K = e_1 K_1 + \ldots + e_r K_r$ and $D = min(d_1 D_1, \ldots, d_r D_r)$. For further details see [MS78, p.590].

Example H152E36

We create a $[72, 41, 12]$ code over $GF(2)$ using the `ZinovievCode` function, which is the best known code for this length and dimension. While it is theoretically possible for a $[72, 41]$ code to have minimum weight up to 14, at the time of writing (July 2001) the best known code has minimum weight 12. The minimum weight is not calculated since it is a lengthy calculation.

```
> I1 := RepetitionCode(GF(2),8);
> I2 := I1 + LinearCode( KMatrixSpace(GF(2),3,8) !
>                 [0,1,0,0,0,1,1,1,0,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1]
>                      );
> I3 := Dual(I1);
> Inner := [I1, I2, I3];
> Inner:Minimal;
[
    [8, 1, 8] Cyclic Code over GF(2),
```

```
    [8, 4, 4] Linear Code over GF(2),
    [8, 7, 2] Cyclic Code over GF(2)
]
>
> O1 := Dual(RepetitionCode(GF(2),9));
> O2 := BCHCode(GF(8),9,3,4);
> O3 := BCHCode(GF(8),9,6,7);
> Outer := [O1, O2, O3];
> Outer:Minimal;
[
    [9, 8, 2] Cyclic Code over GF(2),
    [9, 7, 3] BCH code (d = 3, b = 4) over GF(2^3),
    [9, 4, 6] BCH code (d = 6, b = 7) over GF(2^3)
]
>
> C := ZinovievCode(Inner, Outer);
> C:Minimal;
[72, 41] Linear Code over GF(2)
```

`ConstructionY1(C)`

Apply construction $Y1$ to the code C. This construction applies the shortening operation at the positions in the support of a word of minimal weight in the dual of C. If C is a $[n, k, d]$ code, whose dual code has minimum weight d', then the returned code has parameters $[n - d', k - d' + 1, \geq d]$. For further details see [MS78, p.592].

`ConstructionY1(C, w)`

Apply construction $Y1$ to the code C. This construction applies the shortening operation at the positions in the support of a word of weight w in the dual of C. If C is a $[n, k, d]$ code, then the returned code has parameters $[n - w, k - w + 1, \geq d]$.

152.11 Coding Theory and Cryptography

One of the few public-key cryptosystems which does not rely on number theory is the McEliece cryptosystem, whose security depends on coding theory. An attack on the McEliece cryptosystem must determine the coset leader (of known weight) from a user defined error coset. In general it is assumed that the code in question has no known structure, and it treated as a random code.

The best known attacks on the McEliece cryptosystem are a series of probabilistic enumeration-based algorithms.

152.11.1 Standard Attacks

`McEliecesAttack(C, v, e)`

`MaxTime`	FldReElt	*Default :* ∞
`DirectEnumeration`	BoolElt	*Default :* `true`

Perform the original decoding attack described by McEliece when he defined his cryptosystem. Random information sets are tested for being disjoint for the support of the desired error vector. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

`LeeBrickellsAttack(C, v, e, p)`

`MaxTime`	FldReElt	*Default :* ∞
`DirectEnumeration`	BoolElt	*Default :* `true`

Perform the decoding attack described by Lee and Brickell. Random information sets are tested for having weight less than or equal to p. For most sized codes, the optimal input parameter for this attack is $p = 2$. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

`LeonsAttack(C, v, e, p, l)`

`MaxTime`	FldReElt	*Default :* ∞
`DirectEnumeration`	BoolElt	*Default :* `true`

Perform the decoding attack described by Leon. For random information sets of size k, a punctured code of length $k + l$ is investigated for codewords of weight less than or equal to p. For small codes (up to length around 200), the optimal input parameter for this attack is $p = 2$ with l somewhere in the range $3 - 6$. For larger code $p = 3$ can sometimes be faster, with values of l in the range $7 - 10$. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

`SternsAttack(C, v, e, p, l)`

`MaxTime`	FldReElt	*Default :* ∞
`DirectEnumeration`	BoolElt	*Default :* `true`

Perform the decoding attack described by Stern. For random information sets of size k, a punctured code of length $k+l$ is split into two subspaces. Each subspace is enumerated up to information weight p and collisions found with zero non-information weight. For small to mid-range codes (up to length around 500), the optimal input parameter for this attack is $p = 2$ with l somewhere in the range $9 - 13$. For larger code $p = 3$ can sometimes be faster, with values of l from 20 to much higher. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

`CanteautChabaudsAttack(C, v, e, p, l)`

`MaxTime`	FldReElt	*Default :* ∞
`DirectEnumeration`	BoolElt	*Default :* `true`

Perform the decoding attack described by Canteaut and Chabaud. For random information sets of size k, a punctured code of length $k+l$ is split into two subspaces. Using the enumeration technique identical to that of Stern's attack, a different linear algebra process steps through information sets more quickly. The price for this is less independent information sets. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

For most codes (up to length around 1000), the optimal input parameter for this attack is $p = 1$ with l somewhere in the range $6 - 9$. For very large codes $p = 2$ can sometimes be faster, with values of l from 20 to much higher.

If set to a non-zero positive value, the variable argument `MaxTime` aborts the computation if it goes to long. The argument `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

152.11.2 Generalized Attacks

All of the decoding attacks on the McEliece cryptosystem can be put into a uniform framework, consisting of repeated operation of a two stage procedure. Magma allows the user to choose any combination of the implemented methods, which include improvements on the standard attacks.

`DecodingAttack(C, v, e)`

`Enumeration`	MonStgElt	*Default :* "*Standard*"
`MatrixSequence`	MonStgElt	*Default :* "*Random*"

`NumSteps`	RngIntElt	*Default :* 1
`p`	RngIntElt	*Default :* 2
`l`	RngIntElt	*Default :*
`MaxTime`	FldReElt	*Default :* ∞
`DirectEnumeration`	BoolElt	*Default :* `true`

Perform a generalized decoding attack by specifying the enumeration and matrix sequence procedures to be used. This intrinsic attempts to enumerate a vector of weight e from the error coset $v + C$ for the given vector v.

The parameter `Enumeration` can take the values `"Standard"`, `"Leon"` or `"HashTable"`, and correspond to the methods used in Lee and Brickells, Leons and Sterns attacks respectively.

The parameter `MatrixSequence` can take on the values `"Random"` or `"Stepped"`, corresponding to either a completely random sequence of information sets or a sequence of sets differing in one place.

The integer valued `NumSteps` offers a generalization of the stepped matrix process, taking a sequence of sets which differ at the specified number of places.

The parameter `p` and `l` describe the enumeration process, and their exact meaning depends on the enumeration process in question. See the earlier descriptions of the standard attacks for a full description of their meanings.

For codes of lengths anywhere between $500 - 1000$, the best performance can be obtained using a multiply stepped matrix sequence, using around 10 steps at a time. This is in conjunction with the hashtable enumeration technique using $p = 2$ and l in the range $15 - 20$.

If set to a non-zero positive value, the variable parameter `MaxTime` aborts the computation if it goes to long. The parameter `DirectEnumeration` controls whether or not the coset is enumerated directly, or whether the larger code generated by $\langle C, v\rangle$ is enumerated.

152.12 Bounds

Magma supplies various functions for computing lower and upper bounds for parameters associated with codes. It also contains tables of best known bounds for linear codes. The functions in this section only apply to codes over finite fields.

152.12.1 Best Known Bounds for Linear Codes

A Magma database allows the user access to tables giving the best known upper and lower bounds of the `Length`, `Dimension`, and `MinimumWeight` of linear codes. Tables are currently available relating to codes over $GF(2)$ and $GF(4)$ with $1 \leq$ `Length` ≤ 256, over $GF(3)$ with $1 \leq$ `Length` ≤ 243, over $GF(5), GF(8)$, and $GF(9)$ with $1 \leq$ `Length` ≤ 130, and over $GF(7)$ with $1 \leq$ `Length` ≤ 100.

`BKLCLowerBound(F, n, k)`

Returns the best known lower bound on the maximum possible minimum weight of a linear code over finite field F having length n and dimension k.

`BKLCUpperBound(F, n, k)`

Returns the best known upper bound on the minimum weight of a linear code over finite field F of length n and dimension k.

`BLLCLowerBound(F, k, d)`

Returns the best known lower bound on the minimum possible length of a linear code over finite field F having dimension k and minimum weight at least d. If the required length is out of the range of the database then no bound is available and -1 is returned.

`BLLCUpperBound(F, k, d)`

Returns the best known upper bound on the minimum possible length of a linear code over finite field F of dimension k and minimum weight at least d. If the required length is out of the range of the database then no bound is available and -1 is returned.

`BDLCLowerBound(F, n, d)`

Returns the best known lower bound on the maximum possible dimension of a linear code over finite field F having length n and minimum weight at least d.

`BDLCUpperBound(F, n, d)`

Returns the best known upper bound on the dimension of a linear code over finite field F having length n and minimum weight at least d.

152.12.2 Bounds on the Cardinality of a Largest Code

`EliasBound(K, n, d)`

Return the Elias upper bound of the cardinality of a largest code of length n and minimum distance d over the field K.

`GriesmerBound(K, n, d)`

Return the Griesmer upper bound of the cardinality of a largest code of length n and minimum distance d over the field K.

`JohnsonBound(n, d)`

Return the Johnson upper bound of the cardinality of a largest binary code of length n and minimum distance d.

`LevenshteinBound(K, n, d)`

Return the Levenshtein upper bound of the cardinality of a largest code of length n and minimum distance d over the field K.

`PlotkinBound(K, n, d)`

Return the Plotkin upper bound on the cardinality of a (possibly non-linear) code of length n and minimum distance d over the field K. The bound is formed by calculating the maximal possible average distance between codewords.

For binary codes the bound exists for $n \leq 2d$, (d even), or $n \leq 2d+1$ (d odd). For codes over general fields the bound exists for $d > (1 - 1/\#K) * n$.

`SingletonBound(K, n, d)`

Return the Singleton upper bound of the cardinality of a largest code of length n and minimum distance d over the field K.

`SpherePackingBound(K, n, d)`

Return the Hamming sphere packing upper bound on the cardinality of a largest codes of length n and minimum distance d over the field K.

`GilbertVarshamovBound(K, n, d)`

Return the Gilbert–Varshamov lower bound of the cardinality of a largest code (possibly non-linear) of length n and minimum distance d over the field K.

`GilbertVarshamovLinearBound(K, n, d)`

Return the Gilbert–Varshamov lower bound of the cardinality of a largest linear code of length n and minimum distance d over the field K.

`VanLintBound(K, n, d)`

Return the van Lint lower bound of the cardinality of a largest code of length n and minimum distance d over the field K.

Example H152E37

We compare computed and stored values of best known upper bounds of the dimension of binary linear codes of length 20. The cardinality of a linear code of dimension k over $\mathbf{F}_q$ is q^k, and so the computed bounds on cardinality are compared with the stored bounds on dimension by taking logs.

```
> n:=20;
> K := GF(2);
> [ Ilog(#K, Minimum({GriesmerBound(K, n, d), EliasBound(K, n, d),
>                       JohnsonBound(n, d) , LevenshteinBound(K, n, d),
>                       SpherePackingBound(K, n, d)})) : d in [1..n] ];
[ 20, 19, 15, 14, 12, 11, 9, 8, 5, 4, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1 ]
> [ BDLCUpperBound(K, n, d) : d in [1..n] ];
[ 20, 19, 15, 14, 11, 10, 9, 8, 5, 4, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1 ]
```

152.12.3 Bounds on the Minimum Distance

`BCHBound(C)`

Given a cyclic code C, return the BCH bound for C. This a lower bound on the minimum weight of C.

`GriesmerMinimumWeightBound(K, n, k)`

Return the Griesmer upper bound of the minimum weight of a linear code of length n and dimension k over the field K.

152.12.4 Asymptotic Bounds on the Information Rate

`EliasAsymptoticBound(K, delta)`

Return the Elias asymptotic upper bound of the information rate for δ in $[0, 1]$ over the field K.

`McElieceEtAlAsymptoticBound(delta)`

Return the McEliece–Rodemich–Rumsey–Welch asymptotic upper bound of the binary information rate for δ in $[0, 1]$.

`PlotkinAsymptoticBound(K, delta)`

Return the Plotkin asymptotic upper bound of the information rate for δ in $[0, 1]$ over the field K.

`SingletonAsymptoticBound(delta)`

Return the Singleton asymptotic upper bound of the information rate for δ in $[0, 1]$ over any finite field.

`HammingAsymptoticBound(K, delta)`

Return the Hamming asymptotic upper bound of the information rate for δ in $[0, 1]$ over the field K.

`GilbertVarshamovAsymptoticBound(K, delta)`

Return the Gilbert–Varshamov asymptotic lower bound of the information rate for δ in $[0, 1]$ over the field K.

152.12.5 Other Bounds

`GriesmerLengthBound(K, k, d)`

Return the Griesmer lower bound of the length of a linear code of dimension k and minimum distance d over K.

152.13 Best Known Linear Codes

An $[n, k]$ linear code C is said to be a *best known linear* $[n, k]$ *code* (BKLC) if C has the highest minimum weight among all known $[n, k]$ linear codes.

An $[n, k]$ linear code C is said to be an *optimal linear* $[n, k]$ *code* if the minimum weight of C achieves the theoretical upper bound on the minimum weight of $[n, k]$ linear codes.

MAGMA currently has databases for best known linear codes over $GF(q)$ for $q = 2, 3, 4, 5, 7, 8, 9$. There is also a database of best known quantum codes that can be found in Chapter 157. The database for codes over $GF(2)$ contains constructions of best codes of length up to $n_{\max} = 256$. The codes of length up to $n_{\text{opt}} = 31$ are optimal. The database is complete in the sense that it contains a construction for every set of parameters. Thus the user has access to 33 152 best-known binary codes.

The database for codes over $GF(3)$ contains constructions of best codes of up to length $n_{\max} = 243$. The codes of length up to $n_{\text{opt}} = 21$ are optimal. The database is complete up to length $n_{\text{complete}} = 100$. Many of the codes constructed in this database are vast improvements on the previously known bounds for best codes over $GF(3)$. The database of codes over $GF(3)$ is a contribution of Markus Grassl, Karlsruhe.

The database for codes over $GF(4)$ contains constructions of best codes of up to length $n_{\max} = 256$. The codes of length up to $n_{\text{opt}} = 18$ are optimal. The database is over 65% complete with the first missing code coming at length 98. Many of the codes constructed in this database are vast improvements on the previously known bounds for best codes over $GF(4)$.

Similar databases for other small fields have been added in V2.14. They are contributions of Markus Grassl, Karlsruhe. The statistics of all databases are summarised in the following table.

	$GF(2)$	$GF(3)$	$GF(4)$	$GF(5)$	$GF(7)$	$GF(8)$	$GF(9)$
$n_{\max}$	256	243	256	130	100	130	130
n_{opt}	31	21	18	15	14	14	16
n_{complete}	256	100	97	80	68	76	93
total	33 152	29 889	33 152	8 645	5 150	8 645	8 645
missing	0	6 545	11 379	527	381	1 763	1 333
filled	100%	78.10%	65.67%	93.90%	92.60%	79.61%	84.58%

Compared to previous released versions of the MAGMA BKLC database, 1308 codes over $GF(2)$, 102 codes over $GF(3)$ and 160 codes over $GF(4)$ have been improved, and the maximal length for codes over $GF(3)$ and $GF(4)$ has been increased to 243 and 256, respectively.

Best known upper and lower bounds on the minimum weight for $[n, k]$ linear codes are also available (see section 152.12.1).

The MAGMA BKLC database makes use of the tables of bounds compiled by A. E. Brouwer [Bro98]. The online version of these tables [Bro] has been discontinued. Similar tables are now maintained by Markus Grassl [Gra]. Any improvements, errors, or problems with the MAGMA BKLC database should be reported to `codes@codetables.de`.

It should be noted that the MAGMA BKLC database is unrelated to the similar (but rather incomplete) BKLC database forming part of GUAVA, a share package in GAP3.

A significant number of entries in the MAGMA BKLC database provide better codes than the corresponding ones listed in Brouwer's tables.

The construction of the MAGMA BKLC database has been undertaken by John Cannon (Sydney), Markus Grassl (Karlsruhe) and Greg White (Sydney). The authors wish to express their appreciation to the following people who generously supplied codes, constructions or other assistance: Nuh Aydin, Anton Betten, Michael Braun, Iliya Bouyukliev, Andries Brouwer, Tat Chan, Zhi Chen, Rumen Daskalov, Scott Duplichan, Iwan Duursma, Yves Edel, Sebastian Egner, Peter Farkas, Damien Fisher, Philippe Gaborit, Willi Geiselmann, Stephan Grosse, Aaron Gulliver, Masaaki Harada, Ray Hill, Plamen Hristov, David Jaffe, Axel Kohnert, San Ling, Simon Litsyn, Pawel Lizak, Tatsuya Maruta, Masami Mohri, Masakatu Morii, Harald Niederreiter, Ayoub Otmani, Fernanda Pambianco, James B. Shearer, Neil Sloane, Roberta Sabin, Cen Tjhai, Ludo Tolhuizen, Martin Tomlinson, Gerard van der Geer, Henk van Tilborg, Chaoping Xing, Karl-Heinz Zimmermann, Johannes Zwanzger.

Given any two of the parameters: length, dimension, and minimum weight, then MAGMA will return the code with the best possible value of the omitted parameter. Given a specified length and minimum weight, for example, will result in a corresponding code of maximal possible dimension.

The user can display the method used to construct a particular BKLC code through use of a verbose mode, triggered by the verbose flag `BestCode`. When it is set to `true`, all of the functions in this section will output the steps involved in each code they construct. While some codes are defined by stored generator matrices, and some use constructions which are not general enough, or safe enough, to be available to the user, most codes are constructed using standard MAGMA functions. Note that having the verbose flag `Code` set to `true` at the same time can produce mixed and confusing output, since the database uses functions which have verbose outputs dependent on this flag.

`BKLC(K, n, k)`

`BestKnownLinearCode(K, n, k)`

Given a finite field K, a positive integer n, and a non-negative integer k such that $k \leq n$, return an $[n, k]$ linear code over K which has the largest minimum weight among all known $[n, k]$ linear codes. A second boolean return value signals whether or not the desired code exists in the database.

The databases currently available are over $GF(q)$ for $q = 2, 3, 4, 5, 7, 8, 9$ of length up to $n_{\max}$ as given in the table above.

If the verbose flag `BestCode` is set to true then the method used to construct the code will be printed.

`BLLC(K, k, d)`

`BestLengthLinearCode(K, k, d)`

Given a finite field K, and positive integers k and d, return a linear code over K with dimension k and minimum weight at least d which has the shortest length among

known codes. A second boolean return value signals whether or not the desired code exists in the database.

The databases currently available are over $GF(q)$ for $q = 2, 3, 4, 5, 7, 8, 9$ of length up to $n_{\max}$ as given in the table above.

If the verbose flag `BestCode` is set to true then the method used to construct the code will be printed.

`BDLC(K, n, d)`

`BestDimensionLinearCode(K, n, d)`

Given a finite field K, a positive integer n, and a positive integer d such that $d \leq n$, return a linear code over K with length n and minimum weight $\geq d$ which has the largest dimension among known codes. A second boolean return value signals whether or not the desired code exists in the database.

The databases currently available are over $GF(q)$ for $q = 2, 3, 4, 5, 7, 8, 9$ of length up to $n_{\max}$ as given in the table above.

If the verbose flag `BestCode` is set to true then the method used to construct the code will be printed.

Example H152E38

We look at some best known linear codes over $GF(2)$. Since the database over $GF(2)$ is completely filled, we can ignore the second boolean return value.

```
> C := BKLC(GF(2),23,12);
> C;
[23, 12, 7] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0 0 1 1]
[0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1]
[0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0 1]
> WeightDistribution(C);
[ <0, 1>, <7, 253>, <8, 506>, <11, 1288>, <12, 1288>, <15, 506>,
<16, 253>, <23, 1> ]
> BKLCLowerBound(GF(2),23,12), BKLCUpperBound(GF(2),23,12);
7 7
```

So we see that this code is optimal, in the sense that it meets the best known upper bound on its minimum weight. (All best known binary codes of length up to 31 are optimal).

However larger best known codes are not optimal, making it theoretically possible that better codes exist.

```
> C := BKLC(GF(2),145,36);
> C:Minimal;
[145, 36, 42] Linear Code over GF(2)
> BKLCLowerBound(GF(2),145,36), BKLCUpperBound(GF(2),145,36);
42 52
```

Example H152E39

We look at some best known codes over $GF(4)$. Since this database is only approximately 66% complete, it is necessary to check the second boolean return value to know if the database contained the desired code.

```
> F<w> := GF(4);
> C, has_code := BKLC(F, 14, 9);
> has_code;
true
> C;
[14, 9, 4] Linear Code over GF(2^2)
Generator matrix:
[  1   0   0   0   0   0   0   0   0   0   0 w^2   w   1]
[  0   1   0   0   0   0   0   0   0   0   1   1   1 w^2]
[  0   0   1   0   0   0   0   0   0   0 w^2   w   0   w]
[  0   0   0   1   0   0   0   0   0   0   w   1   1   w]
[  0   0   0   0   1   0   0   0   0   0   w   0   w w^2]
[  0   0   0   0   0   1   0   0   0   0 w^2   1   1   1]
[  0   0   0   0   0   0   1   0   0   0   1   w w^2   0]
[  0   0   0   0   0   0   0   1   0   0   0   1   w w^2]
[  0   0   0   0   0   0   0   0   1   0 w^2 w^2   0   1]
> BKLCLowerBound(F, 14, 9), BKLCUpperBound(F, 14, 9);
4 4
```

Since the database over $GF(4)$ is completely filled up to length 97 the boolean value was in fact unnecessary in this case. We see that the minimum weight of this code reaches the theoretical upper bound, as do all best known codes over $GF(4)$ up to length 18.
For longer lengths we have the possibility that the database may not contain the desired code.

```
> C, has_code := BKLC(F, 98, 57);
> has_code;
false
> C;
[98, 0, 98] Cyclic Linear Code over GF(2^2)
>
> C, has_code := BKLC(F, 98, 58);
> has_code;
true
> C:Minimal;
```

```
[98, 58, 16] Linear Code over GF(2^2)
```

Example H152E40

We search for best known codes using dimension and minimum weight, looking at codes over $GF(2)$ of dimension 85. Even though the database over $GF(2)$ is 100% filled up to length 256, the code required may be longer than that so we have to check the second boolean return value.

```
> C, has_code := BestLengthLinearCode(GF(2),85,23);
> has_code;
true
> C:Minimal;
[166, 85, 23] Linear Code over GF(2)
>
> C, has_code := BestLengthLinearCode(GF(2),85,45);
> has_code;
true
> C:Minimal;
[233, 85, 45] Linear Code over GF(2)
>
> C, has_code := BestLengthLinearCode(GF(2),85,58);
> has_code;
false
```

Example H152E41

For a given minimum weight, we find the maximal known possible dimensions for a variety of code lengths over $GF(4)$.
For lengths < 98 we know the database is filled so we do not need to check the second boolean return value.

```
> F<w> := GF(4);
> C := BDLC(F, 12, 8);
> C;
[12, 3, 8] Linear Code over GF(2^2)
Generator matrix:
[  1   0   0   w w^2   w w^2   w   w   1   w   w]
[  0   1   0   w w^2   1   0 w^2   w   0 w^2 w^2]
[  0   0   1   0   1 w^2 w^2 w^2   0 w^2   w   w]
>
> C := BDLC(F, 27, 8);
> C:Minimal;
[27, 15, 9] Linear Code over GF(2^2)
> C := BDLC(F, 67, 8);
> C:Minimal;
[67, 52, 8] Linear Code over GF(2^2)
```

But for lengths ≥ 98 there may be gaps in the database so to be safe we check the second value.

```
> C, has_code := BDLC(F, 99, 8);
```

```
> has_code;
true
> C:Minimal;
[99, 81, 8] Linear Code over GF(2^2)
> C, has_code := BDLC(F, 195, 8);
> has_code;
true
> C:Minimal;
[195, 174, 8] Linear Code over GF(2^2)
```

Example H152E42

We find the best known code of length 54 and dimension 36, then using the output of the verbose mode we re-create this code manually.

```
> SetPrintLevel("Minimal");
> SetVerbose("BestCode",true);
> a := BKLC(GF(2), 54, 36);
Construction of a [ 54 , 36 , 8 ] Code:
[1]:  [63, 46, 7] Cyclic Code over GF(2)
       CyclicCode of length 63 with generating polynomial x^17 +
       x^16 + x^15 + x^13 + x^12 + x^8 + x^6 + x^4 + x^3 + x^2 +
       1
[2]:  [64, 46, 8] Linear Code over GF(2)
       ExtendCode [1] by 1
[3]:  [54, 36, 8] Linear Code over GF(2)
       Shortening of [2] at { 55 .. 64 }
> a;
[54, 36, 8] Linear Code over GF(2)
>
> P<x> := PolynomialRing(GF(2));
> p := x^17 + x^16 + x^15 + x^13 + x^12 + x^8 + x^6 + x^4 +
>                                          x^3 + x^2 + 1;
> C1 := CyclicCode(63, p);
> C1;
[63, 46] Cyclic Code over GF(2)
> C2 := ExtendCode(C1);
> C2;
[64, 46] Linear Code over GF(2)
> C3 := ShortenCode(C2, {55 .. 64});
> C3;
[54, 36, 8] Linear Code over GF(2)
>
> C3 eq a;
true
```

152.14 Decoding

Magma supplies functions for decoding vectors from the ambient space of a linear code C. The functions in this section only apply to codes over finite fields.

`Decode(C, v: parameters)`

`Al` MonStgElt *Default :* "*Euclidean*"

Given a linear code C and a vector v from the ambient space V of C, attempt to decode v with respect to C. Currently the accessible algorithms are: syndrome decoding (which is demonstrated manually in the example in the Coset Leaders section above); and a Euclidean algorithm, which operates on alternant codes (BCH, Goppa, and Reed–Solomon codes, etc.). While the Euclidean algorithm cannot correct as many errors as can the syndrome algorithm, in general it is much faster, since the syndrome algorithm requires the coset leaders of the code and is also inapplicable as soon as the codimension of the code is moderately large. If the code is alternant, the Euclidean algorithm is used by default, but the syndrome algorithm will be used if the parameter `Al` is assigned the value `"Syndrome"`. For non-alternant codes, only syndrome decoding is possible, so the parameter `Al` is not relevant. If the decoding algorithm succeeds in computing a vector v' as the decoded version of v, then the function returns `true` and v'. (In the Euclidean case it may even happen that v' is not in C because there are too many errors in v to correct.) If the decoding algorithm does not succeed in decoding v, then the function returns `false` and the zero vector.

`Decode(C, Q: parameters)`

`Al` MonStgElt *Default :* "*Euclidean*"

Given a linear code C and a sequence Q of vectors from the ambient space V of C, attempt to decode the vectors of Q with respect to C. This function is similar to the function `Decode(C, v)` except that rather than decoding a single vector, it decodes a sequence of vectors and returns a sequence of booleans and a sequence of decoded vectors corresponding to the given sequence. The algorithm used and effect of the parameter `Al` are as for the function `Decode(C, v)`.

Example H152E43

We create a code C and a vector v of C and then perturb v to a new vector w. We then decode w to find v again.

```
> C := GolayCode(GF(2), false);
> v := C ! [1,1,1,1,0,0,0,1,0,0,1,1,0,0,0,1,0,0,0,1,1,1,1];
> w := v;
> w[5] := 1 - w[5];
> w[20] := 1 - w[20];
> v;
(1 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 1 1)
> w;
(1 1 1 1 1 0 0 1 0 0 1 1 0 0 0 1 0 0 0 0 1 1 1)
```

```
> v - w;
(0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0)
> b, d := Decode(C, w);
> b;
true
> d;
(1 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 1 1)
> d eq v;
true
> Decode(C, [w]);
[ true ]
[
    (1 1 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1 1 1 1)
]
```

152.15 Transforms

152.15.1 Mattson–Solomon Transforms

`MattsonSolomonTransform(f, n)`

Given f, a polynomial over a finite field containing a primitive n-th root of unity, return the Mattson–Solomon transform of parameter n.

`InverseMattsonSolomonTransform(A, n)`

Given A, a polynomial over a finite field containing a primitive n-th root of unity, return the inverse Mattson–Solomon transform of parameter n.

Example H152E44

We compute the Mattson–Solomon transform of parameter $n = 7$ of the polynomial x^4+x^2+x+1 over $\mathbf{F}_{2^{12}}$.

```
> n := 7;
> K := GF(2, 12);
> FP<x> := PolynomialRing(K);
> f := x^4 + x^2 + x + 1;
> A := MattsonSolomonTransform(f, n);
> A;
x^6 + x^5 + x^3
```

152.15.2 Krawchouk Polynomials

`KrawchoukPolynomial(K, n, k)`

Return the Krawchouk polynomial of parameters k and n in K over the rational field.

`KrawchoukTransform(f, K, n)`

Return the Krawchouk transform of the polynomial f over the rational field with respect to the vector space K^n.

`InverseKrawchouk(A, K, n)`

Return the inverse Krawchouk transform of the polynomial A over the rational field with respect to the vector space K^n.

152.16 Automorphism Groups

152.16.1 Introduction

Let C be an $[n, k]$ linear code and G a permutation group of degree n. Then G acts on C in the following way: for a codeword v of C and a permutation x of G, the image of v under x is obtained from v by permuting the coordinate positions of v according to x. We call this the *permutation* action of G on C.

If C is a non-binary code over a finite field, there is also a *monomial* action on C. Let K be the alphabet of C. A monomial permutation of monomial degree n is equivalent to a permutation s on $K^* \times \{1, \ldots, n\}$ which satisfies the following property:

$$(\alpha, i)^s = (\beta, j) \text{ implies } (\gamma\alpha, i)^s = (\gamma\beta, j)$$

for all $\alpha, \beta, \gamma \in K^*$ and $i, j \in \{1, \ldots, n\}$. The actual degree of s is $(q-1)n$. Note that s is completely determined by its action on the points $(1, i)$ for each i, and the matrix representation of s is also determined by its action on the elements $(1, i)$, for $1 \leq i \leq n$. To represent a monomial permutation of monomial degree n, we number the pair (α, i) by $(q-1)(i-1)+\alpha$ and then use a permutation s of degree $(q-1)n$.

The functions in this section allow one to investigate such actions. The algorithms in MAGMA to compute with such actions are backtrack searches due to Jeff Leon [Leo82][Leo97]. There are 4 algorithms which are provided for codes of length n over a field of cardinality q:

(a) *Automorphism group* or *Monomial group.* Computes the group of monomials which map a code into itself, where monomials are represented as permutations of degree $(q-1)n$ (so the group has degree $(q-1)n$). For this function q may be any small prime or 4.

(b) *Permutation group.* Computes the group of permutations which map a code into itself (so the group has degree n). For this function q may be any small prime or 4.

(c) *Equivalence test.* Computes whether there is a monomial permutation which maps a code to another code and, if so, returns the monomial as a permutation of degree $(q-1)n$. For this function, q may be any small prime or 4.

(d) *Isomorphism test.* Computes whether there is a permutation which maps a code to another code and returns the permutation (of degree n) if so. For this function q may only be 2.

For more information on permutation group actions and orbits, see Chapter 58.

152.16.2 Group Actions

`v ^ x`

Given a codeword v belonging to the $[n, k]$ code C and an element x belonging to a permutation group G, construct the vector w obtained from v by the action of x. If G has degree n, the permutation action is used; otherwise G should have degree $n(q-1)$ and the monomial action is used.

`v ^ G`

Given a codeword v belonging to the $[n, k]$ code C and a permutation group G (with permutation or monomial action on C), construct the vector orbit Y of v under the action of G. The orbit Y is a G-set for the group G.

`C ^ x`

Given an $[n, k]$ code C and an element x belonging to a permutation group G (with permutation or monomial action on C), construct the code consisting of all the images of the codewords of C under the action of x.

`C ^ G`

Given an $[n, k]$ code C and a permutation group G (with permutation or monomial action on C), construct the orbit Y of C under the action of G. The orbit Y is a G-set for the group G.

`S ^ x`

Given a set or sequence S of codewords belonging to the $[n, k]$ code C and an element x belonging to a permutation group (with permutation or monomial action on the codewords), construct the set or sequence of the vectors obtained by permuting the coordinate positions of v, for each v in S, according to the permutation x.

`S ^ x`

Given a set or sequence S of codes of length n and an element x belonging to a permutation group (with permutation or monomial action on the codes) construct the set or sequence of the codes consisting of all the images of the codewords of C under the action of x.

`Fix(C, G)`

Given an $[n, k]$ code C and a permutation group G of degree n, find the subcode of C which consists of those vectors of C which are fixed by the elements of G. That is, the subcode consists of those codewords that are fixed by the group G.

152.16.3 Automorphism Group

`AutomorphismGroup(C: parameters)`

`MonomialGroup(C: parameters)`

`Weight` RngIntElt *Default :* 0

The automorphism group A of the $[n, k]$ linear code C over the field K, where A is the group of all monomial-action permutations which preserve the code. Thus both permutation of coordinates and multiplication of components by non-zero elements from K is allowed, and the degree of A is $n(q-1)$ where q is the cardinality of K. A power structure P and transfer map t are also returned, so that, given a permutation g from A, one can create a map $f = t(g)$ which represents the automorphism g as a mapping $P : C \to C$.

If the code is known to have very few words of low weight, then it may take some time to compute the *support* of the code (a set of low weight words). The optional parameter `Weight` can be used to specify the set of vectors of the specified weight to be used as the support in the algorithm. This set should be of a reasonable size, (possibly hundreds for a large code), while also keeping the weight as small as possible.

Warning: If `Weight` specifies a set that is too small, then the algorithm risks getting stuck.

`PermutationGroup(C)`

The permutation group G of the $[n, k]$ linear code C over the field K, where G is the group of all permutation-action permutations which preserve the code. Thus only permutation of coordinates is allowed, and the degree of G is always n. A power structure P and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the automorphism g as a mapping $P : C \to C$.

`AutomorphismSubgroup(C)`

`MonomialSubgroup(C)`

A subgroup of the (monomial) automorphism group A of the code C. If the automorphism group of C is already known then the group returned is the full automorphism group, otherwise it will be a subgroup generated by one element. This allows one to find just one automorphism of C if desired. A power structure P and transfer map t are also returned, so that, given a permutation g from A, one can create a map $f = t(g)$ which represents the automorphism g as a mapping $P : C \to C$.

`AutomorphismGroupStabilizer(C, k)`

`MonomialGroupStabilizer(C, k)`

The subgroup of the (monomial) automorphism group A of the code C, which stabilizes the first k base points as chosen by the backtrack search. These base points may be different to those of the returned group. A power structure P and transfer map t are also returned, so that, given a permutation g from A, one can create a map $f = t(g)$ which represents the automorphism g as a mapping $P : C \to C$.

`Aut(C)`

The power structure A of all automorphisms of the code C (with monomial action), together with the transfer map t into A from the generic symmetric group associated with the automorphism group of C.

`Aut(C, T)`

The power structure A of all automorphisms of the code C, together with the transfer map t into A from the generic symmetric group associated with the automorphism group of C; the string T determines which action type should be used: `"Monomial"` or `"Permutation"`.

Example H152E45

We compute the automorphism group of the second order Reed–Muller code of length 64.

```
> C := ReedMullerCode(2, 6);
> aut := AutomorphismGroup(C);
> FactoredOrder(aut);
[ <2, 21>, [3, 4>, <5, 1>, <7, 2>, <31, 1> ]
> CompositionFactors(aut);
    G
    |  A(5, 2)                = L(6, 2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
```

Example H152E46

We compute the automorphism group of a BCH code using the set of vectors of minimal weight as the invariant set. We look first at its weight distribution to confirm that there is sufficient vectors.

```
> C := BCHCode(GF(2),23,2);
> C;
[23, 12, 7] BCH code (d = 2, b = 1) over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 1 1 0 0 0 1 1]
[0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1]
[0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 0 1]
> WeightDistribution(C);
[ <0, 1>, <7, 253>, <8, 506>, <11, 1288>, <12, 1288>, <15, 506>,
<16, 253>, <23, 1> ]
> AutomorphismGroup(C : Weight := MinimumWeight(C) );
Permutation group acting on a set of cardinality 23
Order = 10200960 = 2^7 * 3^2 * 5 * 7 * 11 * 23
    (6, 15, 12)(7, 20, 19)(8, 9, 17)(10, 13, 22)(11, 14, 23)(16,
        18, 21)
    (5, 17, 9)(6, 21, 20)(7, 16, 23)(10, 13, 22)(11, 12, 19)(14,
        18, 15)
    (5, 16, 21)(6, 23, 19)(7, 17, 12)(8, 14, 15)(9, 20, 11)(10,
        13, 22)
    (1, 2)(4, 12)(5, 7)(6, 17)(9, 10)(13, 21)(15, 18)(22, 23)
    (2, 3)(4, 21, 6, 20)(5, 10, 12, 17)(7, 13, 9, 11)(15, 18)(16,
        19, 22, 23)
    (3, 8)(4, 5, 20, 22)(6, 16, 21, 12)(7, 11, 13, 9)(10, 17, 19,
        23)(15, 18)
    (4, 8, 6, 21, 20)(5, 10, 14, 17, 12)(7, 22, 18, 16, 9)(11,
        23, 19, 13, 15)
```

152.16.4 Equivalence and Isomorphism of Codes

`IsIsomorphic(C, D: parameters)`

`IsEquivalent(C, D: parameters)`

`AutomorphismGroups`	MonStgElt	*Default :* "*Right*"
`Weight`	RngIntElt	*Default :* 0

Given $[n, k]$ codes C and D, this function returns `true` if and only if C is equivalent to D. If C is equivalent to D, an equivalence map f is also returned from C onto D. The equivalence is with respect to the monomial action. The function first computes none, one, or both of the automorphism groups of the left and right codes. This may assist the isomorphism testing.

The parameter `AutomorphismGroups`, with valid string values `Both`, `Left`, `Right`, `None`, may be used to specify which of the automorphism groups should be constructed first if not already known. The default is `Right`.

In rare cases this algorithm can get stuck, due to an insufficient set of invariant vectors. In this case, the optional parameter `Weight` can be used to specify this set to be the vectors of the specified weight. This set should be of a reasonable size, (possibly hundreds for large codes), while also keeping the weight as small as possible.

Warning: If `Weight` specifies a set that is too small, then the algorithm risks getting stuck.

152.17 Bibliography

[**All84**] W.O. Alltop. A method for extending binary linear codes. *IEEE Trans. Inform. Theory*, 30:871 – 872, 1984.

[**BFK+98**] A. Betten, H. Fripertinger, A. Kerber, A. Wassermann, and K.-H. Zimmermann. *Codierungstheorie – Konstruktion und Anwendung linearer Codes*. Springer-Verlag, Berlin–Heidelberg–New York, 1998.

[**Bro**] A. E. Brouwer. Bounds on the minimum distance of linear codes. URL:http://www.win.tue.nl/~aeb/voorlincod.html.

[**Bro98**] A. E. Brouwer. Bounds on the size of linear codes. In *Handbook of coding theory, Vol. I, II*, pages 295–461. North-Holland, Amsterdam, 1998.

[**Gab02**] Philippe Gaborit. Quadratic Double Circulant Codes over Fields. *Journal of Combinatorial Theory*, 97:85–107, 2002.

[**Gra**] Markus Grassl. Bounds on the minimum distance of linear codes. URL:http://www.codetables.de/.

[**Leo82**] Jeffrey S. Leon. Computing automorphism groups of error-correcting codes. *IEEE Trans. Inform. Theory*, IT-28:496–511, 1982.

[**Leo97**] Jeffrey S. Leon. Partitions, refinements, and permutation group computation. In Larry Finkelstein and William M. Kantor, editors, *Groups and Computation*

II, volume 28 of *Dimacs series in Discrete Mathematics and Computer Science*, pages 123–158, Providence R.I., 1997. Amer. Math. Soc.

[**MS78**] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. North Holland, New York, 1978.

153 ALGEBRAIC-GEOMETRIC CODES

Chapter 153
ALGEBRAIC-GEOMETRIC CODES

153.1 Introduction

Algebraic–Geometric Codes (AG–codes) are a family of linear codes described by Goppa in [Gop81a, Gop81b]. Let $\mathcal{X}$ be an irreducible projective plane curve of genus g, defined by a (absolutely irreducible) homogeneous polynomial $H(X, Y, Z)$ over a finite field $K = \mathbf{F}_q$. A *place* of $\mathcal{X}$ is the maximal ideal of a discrete valuation subring of $\overline{K}(\mathcal{X})$. We denote by v_P the valuation at place P. Its *degree* is the degree of its residue class field over K. A *divisor* is an element D of the free abelian group over the set of places of $\mathcal{X}$. Namely, such an element can be written additively:

$$D = \sum_{P \in \mathrm{Pl}(\mathcal{X})} n_P \cdot P \, ,$$

where all but finitely many $n_P \in \mathbf{Z}$ are zero. The set of places with nonzero multiplicity is the *support* of D, denoted by Supp$D$. The set of divisors can be equipped with a natural partial order $\leq$ defined by:

$$D = \sum_{P \in \mathrm{Pl}(\mathcal{X})} n_P \cdot P \leq D' = \sum_{P \in \mathrm{Pl}(\mathcal{X})} n'_P \cdot P \quad \Longleftrightarrow \quad n_P \leq n'_P \text{ for all } P.$$

If $f \in K(\mathcal{X})$, then one can define the *principal divisor*:

$$(f) = \sum_{P \in \mathrm{Pl}(\mathcal{X})} v_P(f) \cdot P \, .$$

A divisor D is said to be *defined* over K if it is stable under the natural action of $\mathrm{Gal}(\overline{\mathrm{K}}/\mathrm{K})$.

If $(P_1, \ldots, P_n)$ is a tuple of places of degree 1, then for a function $f \in K(\mathcal{X})$, $f(P_i)$ can be seen as an element of K. If D is a divisor defined over K, then the *Riemann–Roch* space $\mathcal{L}(D)$ of D is the K-vector space of dimension k:

$$\mathcal{L}(D) = \{f \in K(\mathcal{X})^* \mid (f) + D \geq 0\} \cup \{0\}.$$

Now provided D has support disjoint from $S = \{P_1, \ldots, P_n\}$, we can define the *algebraic geometric code* to be the $[n, k]_q$-code:

$$C = C(S, D) = \{(f(P_1), \ldots, f(P_n)) \, , f \in \mathcal{L}(D)\}.$$

If $\langle f_1, \ldots, f_k \rangle$ is a base of $\mathcal{L}_K(D)$ as a K-vector space, then a generator matrix for C is:

$$G = \begin{pmatrix} f_1(P_1) & \cdots & f_1(P_n) \\ \vdots & \ddots & \vdots \\ f_k(P_1) & \cdots & f_k(P_n) \end{pmatrix}.$$

Standard references are [Sti93] and [TV91].

There are two different implementations of the construction of AG–Codes in MAGMA. The first was implemented by Lancelot Pecquet and is based on the work of Hache [HLB95, Hac96]. The second approach exploits the divisor machinery for function fields implemented by Florian Hess. In MAGMA V2.8, only the second implementation is exported. It is intended to rework the Pecquet version to take advantage of the new curve machinery before releasing it.

153.2 Creation of an Algebraic Geometric Code

`AlgebraicGeometricCode(S, D)`

`AGCode(S, D)`

Suppose X is an irreducible plane curve. Let S be a sequence of places of X having degree 1 and let D be a divisor of X whose support is disjoint from the support of S. The function returns the (weakly) algebraic–geometric code obtained by evaluating functions of the Riemann–Roch space of D at the points of S. The degree of D need not be bounded by the cardinality of S.

`AlgebraicGeometricDualCode(S, D)`

`AGDualCode(S, D)`

Construct the dual of the algebraic geometric code constructed from the sequence of places S and the divisor D, which corresponds to a differential code. In order to take advantage of the algebraic geometric structure, the dual must be constructed in this way, and not by directly calling the function `Dual`.

`HermitianCode(q, r)`

Given the prime power q and a positive integer r, construct a Hermitian code C with respect to the Hermitian curve

$$X = x^{(q+1)} + y^{(q+1)} + z^{(q+1)}$$

defined over $\mathbf{F}_{q^2}$. The support of C consists of all places of degree one of X over $\mathbf{F}_{q^2}$, with the exception of the place over $P = (1 : 1 : 0)$. The divisor used to define a Riemann–Roch space is $r * P$.

Example H153E1

We construct a $[25, 9, 16]$ code over F_{16} using the genus 1 curve $x^3 + x^2z + y^3 + y^2z + z^3$.

```
> F<w> := GF(16);
> P2<x,y,z> := ProjectiveSpace(F, 2);
> f := x^3+x^2*z+y^3+y^2*z+z^3;
> X := Curve(P2, f);
> g := Genus(X);
> g;
1
> places1 := Places(X, 1);
> #places1;
25
```

We now need to find an appropriate divisor D. Since we require a code of dimension $k = 9$ we take the divisor corresponding to a place of degree $k + g - 1 = 9$ (g is the genus of the curve).

```
> found, place_k := HasPlace(X, 9+g-1);
> D := DivisorGroup(X) ! place_k;
> C := AlgebraicGeometricCode(places1, D);
> C;
[25, 9] Linear Code over GF(2^4)
Generator matrix:
[1 0 0 0 0 0 0 0 0 w^9 w w^8 0 w^5 1 1 w^5 w^7 w^8 w^10 w^4 w^11 w^5
   w^10 w^8]
[0 1 0 0 0 0 0 0 0 w^4 w^5 1 w^10 w^4 w^11 w^12 0 1 w^2 w^4 w^6 w^4 w^5
   w^14 w^4]
[0 0 1 0 0 0 0 0 0 w^5 w^13 w^7 w^10 w^7 w^5 w^14 w^14 w 1 w^10 w^9 1 0
   w^5 w]
[0 0 0 1 0 0 0 0 0 w^8 w^3 w^3 w^12 w^7 w^10 w w^6 0 w^7 w^10 w^4 w^9
   w^14 w^8 w^12]
[0 0 0 0 1 0 0 0 0 w^8 1 w^4 w^7 w^5 w w^8 w w^5 w w^13 0 w^14 w^14 w^14
   w^6]
[0 0 0 0 0 1 0 0 0 1 1 w^12 w^14 w^9 w^10 w^6 w^6 w^7 w^10 w^4 w^3 w^13
   w^13 w^3 w^4]
[0 0 0 0 0 0 1 0 0 w w w^10 w^4 w^12 w w^13 w^4 w w^2 w^3 w^3 w^12 w^10
   w^5 w^13]
[0 0 0 0 0 0 0 1 0 w^13 w^6 w^12 w^2 w^3 w^7 w^3 w^4 w^14 w^4 w^11 w^4 w
   w^6 w^4 w^14]
[0 0 0 0 0 0 0 0 1 0 w^11 w w^7 w^12 w^4 w^3 w^6 w^12 w^3 w^13 w^2 w^11
   w^10 w^3 1]
> MinimumDistance(C);
16
```

Example H153E2

We construct a $[44, 12, 29]$ code over F_{16} using the genus 4 curve $(y^2 + xy + x^2)z^3 + y^3z^2 + (xy^3 + x^2y^2 + x^3y + x^4)z + x^3y^2 + x^4y + x^5$ and taking as the divisor a multiple of a degree 1 place.

```
> k<w> := GF(16);
> P2<x,y,z> := ProjectiveSpace(k, 2);
> f := (y^2+x*y+x^2)*z^3+y^3*z^2+(x*y^3+x^2*y^2+x^3*y+x^4)*z+x^3*y^2+x^4*y+x^5;
> X := Curve(P2, f);
> g := Genus(X);
> g;
4
```

We find all the places of degree 1.

```
> places1 := Places(X, 1);
> #places1;
45
```

We choose as our divisor $15 * P1$, where $P1$ is a place of degree 1. Before applying the AG-Code construction we must remove $P1$ from the set of places of degree 1.

```
> P1 := Random(places1);
> Exclude(~places1, P1);
> #places1;
44
> D := 15 * (DivisorGroup(X) ! P1);
> C := AlgebraicGeometricCode(places1, D);
> C:Minimal;
[44, 12] Linear Code over GF(2^4)
> MinimumWeight(C);
29
```

153.3 Properties of AG–Codes

`IsWeaklyAG(C)`

Return `true` if and only if the code C is a weakly algebraic–geometric code, i.e. C has been constructed as an algebraic–geometric code with respect to a divisor of any degree.

`IsWeaklyAGDual(C)`

Return `true` if and only if the code C was constructed as the dual of a weakly algebraic–geometric code.

`IsAlgebraicGeometric(C)`

Return `true` if and only if the code C is of algebraic–geometric construction of length n, built from a divisor D with $deg(D) < n$.

`IsStronglyAG(C)`

Return `true` if and only if C is an algebraic–geometric code of length n constructed from a divisor D satisfying $2g - 2 < deg(D) < n$, where g is the genus of the curve.

153.4 Access Functions

At the time an AG–Code is constructed a number of attributes describing its construction are stored along with the code. The functions in this section give the user access to these attributes.

`Curve(C)`

Given an algebraic–geometric code C, returns the curve from which C was defined.

`GeometricSupport(C)`

Given an algebraic–geometric code C, return the sequence of places which forms the support for C.

`Divisor(C)`

Given an algebraic–geometric code C, return the divisor from which C was constructed.

`GoppaDesignedDistance(C)`

Given an algebraic–geometric code C constructed from a divisor D, return the Goppa designed distance $n - deg(D)$.

153.5 Decoding AG Codes

Specialized decoding algorithms exist for differential code, those which are the duals of the standard algebraic-geometric codes. These algorithms generally require as input another divisor on the curve whose support is disjoint from the divisor defining the code.

`AGDecode(C, v, Fd)`

Decode the received vector v of the dual algebraic geometric code C using the divisor Fd.

Example H153E3

An algebraic-geometric code with Goppa designated distance of 3 is used to correct one error.

```
> q := 8;
> F<a> := GF(q);
> PS<x,y,z> := ProjectiveSpace(F, 2);
> W := x^3*y + y^3*z + x*z^3;
> Cv := Curve(PS, W);
> FF<X,Y> := FunctionField(Cv);
> Pl := Places(Cv, 1);
> plc := Place(Cv ! [0,1,0]);
> P := [ Pl[i] : i in [1..#Pl] | Pl[i] ne plc ];
> G := 11*plc;
> C := AGDualCode(P, G);
>
> v := Random(C);
> rec_vec := v;
> rec_vec[Random(1,Length(C))] +:= Random(F);
> res := AGDecode(C, v, 4*plc);
> res eq v;
true
```

153.6 Toric Codes

`ToricCode(P, q)`

The linear code C over the finite field $\mathbf{F}_q$ associated with the lattice points of the polygon P.

To achieve this, after a translation so that the lattice points of P lie in the first quadrant, as close to the origin as possible, these points must lie in the box $[0, q-2] \times [0, q-2]$. Then the code is the monomial evaluation code where each point (a, b) corresponds to the monomial $x^a y^b$, and these monomials are evaluated at the points of the torus $(\mathbf{F}_q^*)^2$.

`ToricCode(S, q)`

`ToricCode(S, q)`

The linear code C over the finite field $\mathbf{F}_q$ associated with the lattice points in S. (Note that the points will be translated to lie within a box at the origin of the first quadrant, as is usual.)

Example H153E4

We construct the toric code based on the lattice points in the polygon with vertices $(3,0)$, $(5,0)$, $(3,3)$, $(1,5)$, $(0,3)$, $(0,1)$.

```
> P := Polytope( [[3,0], [5,0], [3,3], [1,5], [0,3], [0,1]] );
> C := ToricCode(P, 7);
> [ Length(C), Dimension(C), MinimumDistance(C) ];
[ 36, 19, 12 ]
```

We can compare this with the current database of best known linear codes.

```
> BKLCLowerBound(Field(C), Length(C), Dimension(C));
11
```

153.7 Bibliography

[**Gop81a**] V. D. Goppa. Codes on algebraic curves. *Dokl. Akad. Nauk SSSR*, 259(6): 1289–1290, 1981.

[**Gop81b**] V. D. Goppa. Codes on algebraic curves. *Soviet Math. Dokl.*, 24(1):170–172, 1981.

[**Hac96**] Gaétan Haché. *Construction effective des codes géométriques.* PhD thesis, l'Université Paris 6, 1996.

[**HLB95**] Gaétan Haché and Dominique Le Brigand. Effective construction of algebraic geometry codes. *IEEE Trans. Inform. Theory*, 41(6, part 1):1615–1628, 1995. Special issue on algebraic geometry codes.

[**Sti93**] Henning Stichtenoth. *Algebraic function fields and codes.* Springer-Verlag, Berlin, 1993.

[**TV91**] M. A. Tsfasman and S. G. Vlăduţ. *Algebraic-geometric codes.* Kluwer Academic Publishers Group, Dordrecht, 1991. Translated from the Russian by the authors.

154 LOW DENSITY PARITY CHECK CODES

Chapter 154
LOW DENSITY PARITY CHECK CODES

154.1 Introduction

Low density parity check (LDPC) codes are among the best performing codes in practice, being capable of correcting errors close to the Shannon limit. MAGMA provides facilities for the construction, decoding, simulation and analysis of LDPC codes.

154.1.1 Constructing LDPC Codes

LDPC codes come in two main varieties, *regular* and *irregular*, defined by the row and column weights of the sparse parity check matrix. If all columns in the parity check matrix have some constant weight a, and all rows have some constant weight b, then the LDPC code is said to be (a, b)-regular. When either the columns or the rows have a distribution of weights, the LDPC code is said to be irregular.

Currently, there do not exist many techniques for the explicit construction of LDPC codes. More commonly, these codes are selected at random from an ensemble, and their properties determined through simulation.

`LDPCCode(H)`

Given a sparse binary matrix H, return the LDPC code which has H as its parity check matrix.

`GallagerCode(n, a, b)`

Return a random (a, b)-regular LDPC code of length n, using Gallager's original method of construction. The row weight a must dividethe length n.

`RegularLDPCEnsemble(n, a, b)`

Return a random code from the ensemble of (a, b)-regular binary LDPC codes.

`IrregularLDPCEnsemble(n, Sv, Sc)`

Given (unnormalized) distributions for the variable and check weights, return length n irregular LDPC codes whose degree distributions match the given distribution. The arguments Sv and Sc are sequences of real numbers, where the i-th entry indicates what percentage of the variable (resp. check) nodes should have weight i.

Note that the distributions will not be matched perfectly unless everything is in complete balance.

`MargulisCode(p)`

Return the (3,6)-regular binary LDPC code of length $2(p^3 - p)$ using the group-based construction of Margulis.

Example H154E1

Most LDPC constructions are generated pseudo-randomly from an ensemble, so the same function will return a different code each time. To be able to re-use an LDPC code, the sparse parity check matrix which must be saved.

```
> C1 := RegularLDPCEnsemble(10, 2, 4);
> C2 := RegularLDPCEnsemble(10, 2, 4);
> C1 eq C2;
false
> LDPCMatrix(C1):Magma;
SparseMatrix(GF(2), 5, 10, [
    4, 2,1, 3,1, 4,1, 6,1,
    4, 1,1, 7,1, 9,1, 10,1,
    4, 1,1, 2,1, 3,1, 7,1,
    4, 5,1, 6,1, 8,1, 9,1,
    4, 4,1, 5,1, 8,1, 10,1
])
> H := SparseMatrix(GF(2), 5, 10, [
>     4, 2,1, 3,1, 4,1, 6,1,
>     4, 1,1, 7,1, 9,1, 10,1,
>     4, 1,1, 2,1, 3,1, 7,1,
>     4, 5,1, 6,1, 8,1, 9,1,
>     4, 4,1, 5,1, 8,1, 10,1 ]);
> C3 := LDPCCode(H);
> C3 eq C1;
true
```

154.1.2 Access Functions

Since a code can have many different parity check matrices, the matrix which defines a code as being LDPC must be assigned specifically. Any parity check matrix can be assigned for this purpose, and once a code is assigned an LDPC matrix it is considered by MAGMA to be an LDPC code (regardless of the density or other properties of the matrix). The matrix must be of sparse type (`MtrxSprs`).

`IsLDPC(C)`

Return true if C is an LDPC code (which is true if it has been assigned an LDPC matrix).

`AssignLDPCMatrix(~C, H)`

Given a sparse matrix H which is a parity check matrix of the code C, assign H as the LDPC matrix of C.

`LDPCMatrix(C)`

Given an LDPC code C, return the sparse matrix which has been assigned as its low density parity check matrix.

`LDPCDensity(C)`

Given an LDPC code C, return the density of the sparse matrix which has been assigned as its low density parity check matrix.

`IsRegularLDPC(C)`

Returns true if C is an LDPC code and has regular column and row weights. If true, the row and column weights are also returned.

`TannerGraph(C)`

For an LDPC code C, return its Tanner graph. If there are n variables and m checks, then the graph has $n+m$ nodes, the first n of which are the variable nodes.

`LDPCGirth(C)`

For an LDPC code C, return the girth of its Tanner graph.

`LDPCEnsembleRate(v, c)`

`LDPCEnsembleRate(Sv, Sc)`

Return the theoretical rate of LDPC codes from the ensemble described by the given inputs.

Example H154E2

In MAGMA, whether or not a code is considered LDPC is based solely on whether or not an LDPC matrix has been assigned. This example shows that any code can be made to be considered LDPC, although a random parity check matrix without low density will perform very badly using LDPC decoding.

```
> C := RandomLinearCode(GF(2),100,50);
> IsLDPC(C);
false
> H := SparseMatrix(ParityCheckMatrix(C));
> H;
Sparse matrix with 50 rows and 100 columns over GF(2)
> AssignLDPCMatrix(~C, H);
> IsLDPC(C);
true
> LDPCDensity(C);
0.25340000000000014122036873232
```

The density of the parity check matrices of LDPC codes is much lower than that of randomly generated codes.

```
> C1 := RegularLDPCEnsemble(100,3,6);
```

```
> C1:Minimal;
[100, 50] Linear Code over GF(2)
> LDPCDensity(C1);
0.059999999999999977795539507497
```

154.1.3 LDPC Decoding and Simulation

The impressive performance of LDPC codes lies in their iterative decoding algorithm. MAGMA provides facilities to decode using LDPC codes, as well as simulating transmission over a binary symmetric or white Gaussian noise channels.

The binary symmetric channel transmits binary values and is defined by $p < 0.5$. Each individual bit independently sustains a "bit-flip" error with probability p.

The Gaussian channel is analog, transmitting real-values, and is defined by a standard deviation σ. Binary values are mapped to -1 and 1 before being transmitted. Each value independently sustains an errors which are normally distributed about 0 with standard deviation σ.

`LDPCDecode(C, v)`

`Channel`	MONSTGELT	*Default :* "*BinarySymmetric*"
`p`	RNGRESUBELT	*Default :* 0.1
`StdDev`	RNGRESUBELT	*Default :* 0.25
`Iterations`	RNGINTELT	*Default :* `Dimension(C)`

For an LDPC code C and a received vector v, decode v to a codeword of C using the LDPC iterative decoding algorithm.

The nature of the channel from which v is received is described by the variable argument `Channel`, which can either be the `BinarySymmetric` channel or the `Gaussian` channel. Errors on the binary symmetric channel is described by the argument `p`, while on the Gaussian channel they are described by `StdDev`.

The vector v must be over a ring corresponding to the channel which is selected. For the binary symmetric channel v must be a binary vector over F_2, while for the Gaussian channel it must be real-valued.

Since the decoding algorithm is iterative and does not necessarily terminate on its own, a maximum number of iterations needs to be specified using the argument `Iterations`. The default value is much larger than would normally be used in practice, giving maximum error-correcting performance (at possibly some cost to efficiency).

Example H154E3

Errors in the binary symmetric channel are just bit flips.

```
> n := 500;
> C := RegularLDPCEnsemble(n, 4, 8);
> e := 5;
```

```
> Errs := {};
> repeat Include(~Errs, Random(1,n)); until #Errs eq e;
> v := Random(C);
> ev := AmbientSpace(C)![ (i in Errs) select 1 else 0 : i in [1..n]];
> rec_vec := v + ev;
> time res := LDPCDecode(C, rec_vec : Channel:="BinarySymmetric", p:=0.2);
Time: 0.000
> res eq v;
true
```

Example H154E4

For the Gaussian channel binary vectors are considered to be transmitted as sequences of the values 1 and -1. Errors are normally distributed with a standard deviation defined by the channel. To simulate a Gaussian channel requires obtaining normally distributed errors. This can be done (discretely) by generating a multiset of possible errors.

```
> sigma := 0.5;
> MaxE := 3.0;
> N := 100;
> V := [ MaxE*(i/N)     : i in [-N div 2..N div 2]];
> E := [ 0.5*(1+Erf(x/(sigma*Sqrt(2)))) : x in V ];
> Dist := {* V[i]^^Round(1000*(E[i]-E[i-1])) : i in [2..#V]*};
```

A codeword of an LDPC code needs to be mapped into the real domain.

```
> n := 500;
> C := RegularLDPCEnsemble(n, 4, 8);
> v := Random(C);
> R := RealField();
> RS := RSpace(R, n);
> vR := RS ! [ IsOne(v[i]) select 1 else -1 : i in [1..n]];
```

Normally distributed errors are then introduced, and the received vector decoded.

```
> for i in [1..n] do
>    vR[i] +:= Random(Dist);
> end for;
> time res := LDPCDecode(C, vR : Channel:="Gaussian", StdDev:=sigma);
Time: 0.000
> res eq v;
true
```

`LDPCSimulate(C, N)`

`Channel`	MonStgElt	*Default :* "*BinarySymmetric*"
`p`	RngReSubElt	*Default :* 0.1
`StdDev`	RngReSubElt	*Default :* 0.25
`Iterations`	RngIntElt	*Default :* `Dimension(C)`

For an LDPC code C, simulate N transmissions across the given channel and return the accumulated bit error rate and word error rate.

The variable arguments are as described for the function `LDPCDecode`. The channel which is described controls not only the way the decoding algorithm functions, but also the nature of the errors introduced during the simulation.

Example H154E5

The more noise that is introduced into the channel the error rate increases. Note that the bit error rate (the first return value) is always lower than the word error rate (the second return value).

```
> C := RegularLDPCEnsemble(200, 3, 6);
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.7);
0.0011824999999999999739519143001 0.006199999999999999978211873141731
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.75);
0.0074949999999999999292617016886243 0.041000000000000000172084568816 90
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.8);
0.033722000000000001973532448 5737 0.15970000000000000086153306710 91
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.85);
0.085617499999999999916067139338 34 0.37080000000000001847411112976 3
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.9);
0.162790999999999991265653420669 0.64049999999999999582556142740 94
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 0.95);
0.237657499999999993756105709508 0.84009999999999999579003429062 14
> LDPCSimulate(C, 10000 : Channel := "Gaussian", StdDev := 1.0);
0.296526500000000026169288958044 0.94479999999999999731770117250 56
```

154.1.4 Density Evolution

The asymptotic performance of ensembles of LDPC codes can be determined using *density evolution*. An ensemble of LDPC codes (either regular or irregular) is defined by a pair of degree distributions, corresponding to the degrees at the variable and check nodes of the Tanner graph.

Over a specific channel, the critical parameter which defines the asymptotic performance of a given ensemble is its *threshold*, which is a value of the channel parameter (i.e., the probability of error p for the binary symmetric channel, and the standard deviation σ for the Gaussian channel). When the channel parameter is less than the threshold, asymptotically a code from the ensemble will decode with an error probability tending to zero. However,

at any channel parameter above the threshold there will be a non-vanishing finite error probability.

Determining the threshold of an ensemble over the binary symmetric channel is relatively trivial, however over the real-valued Gaussian channel it can involve extensive computations. The speed depends heavily on the granularity of the discretization which is used, though this also affects the accuracy of the result.

The default settings of the MAGMA implementation use a reasonably coarse discretization, emphasizing speed over accuracy. These (still quite accurate) approximate results can then be used to help reduce the workload of calculations over finer discretizations if more accuracy is required.

```
LDPCBinarySymmetricThreshold(v, c)
```

```
LDPCBinarySymmetricThreshold(Sv, Sc)
```

`Precision` RngReSubElt *Default :* 0.00005

Determines the threshold of the described ensemble of LDPC codes over the binary symmetric channel, which is the critical value of the channel parameter above which there is a non-vanishing error probability (asymptotically). The ensemble can either be defined by two integers for (v, c)-regular LDPC codes, or by two density distributions Sv and Sc, which are sequences of non-negative real numbers. The density distributions do not ned to be normalized, though the first entry (corresponding to weight 1 nodes in the Tanner graph) should always be zero.

The computation proceeds by establishing lower and upper bounds on the threshold, then narrowing this range by repeatedly performing density evolution on the midpoint. The argument `Precision` controls the precision to which the threshold is desired.

```
DensityEvolutionBinarySymmetric(v, c, p)
```

```
DensityEvolutionBinarySymmetric(Sv, Sc, p)
```

Perform density evolution on the binary symmetric channel using channel parameter p and determine the asymptotic behaviour for the given LDPC ensemble. The return value is boolean, where `true` indicates that p is below the threshold and the ensemble has error probability asymptotically tending to zero.

Example H154E6

Density evolution on the binary symmetric channel is not computationally intensive.

```
> time LDPCBinarySymmetricThreshold(3, 6);
0.039414062500000000000000000000000
Time: 0.010
> time LDPCBinarySymmetricThreshold(4, 8);
0.047392578125000000000000000000001
Time: 0.110
> time LDPCBinarySymmetricThreshold(4, 10);
0.036835937500000000000000000000000
```

```
Time: 0.090
```

`LDPCGaussianThreshold(v, c)`

`LDPCGaussianThreshold(Sv, Sc)`

`Lower`	RngReSubElt	*Default :* 0
`Upper`	RngReSubElt	*Default :* ∞
`Points`	RngIntElt	*Default :* 500
`MaxLLR`	RngReSubElt	*Default :* 25
`MaxIterations`	RngIntElt	*Default :* ∞
`QuickCheck`	BoolElt	*Default :* `true`
`Precision`	RngReSubElt	*Default :* 0.00005

Determines the threshold of the described ensemble of LDPC codes over the Gaussian channel, which is the critical value of the standard deviation above which there is a non-vanishing error probability (asymptotically). The ensemble can either be defined by two integers for (v, c)-regular LDPC codes, or by two density distributions Sv and Sc, which are sequences of non-negative real numbers. The density distributions do not ned to be normalized, though the first entry (corresponding to weight 1 nodes in the Tanner graph) should always be zero.

The computation proceeds by establishing lower and upper bounds on the threshold, then narrowing this range by repeatedly performing density evolution on the midpoint. If the threshold is approximately known then manually setting tight `Lower` and `Upper` bounds can reduce the length of the calculation.

The speed with which these evolutions are computed depends on how fine the discretization is, controlled by the variable argument `Points`. If the threshold is needed to high levels of accuracy then an initial computation with fewer points is recommended to get a reduced searched range. The specific meaning of eachvariable argument is described below.

`Lower` and `Upper` are real-valued bounds on the threshold, which (if tight) can help to reduce the search range and speed up the threshold determination. The validity of an input bound is verified before the search begins, and an error is returned if it is incorrect.

`Points` and `MaxLLR` define the discretized basis of log likelihood ratios on which density evolution is performed, and have integer and real values resp. Specifically, the probability mass function is defined on the range [$-$`MaxLLR`, . . . ,`MaxLLR`] on $2 * Points + 1$ discretized points.

`MaxIterations` allows the user to set a finite limit of iterations that a density evolution should perform in determining the asymptotic behaviour at each channel parameter. Although this may help reduce the time of a computation, it should be kept in mind that the result may not be valid.

`QuickCheck` defines the method by which the asymptotic behaviour at each channel parameter is identified. If set to `false`, then the probability density must evolve all the way to within an infinitesimal value of unity. When set to `true`, if the rate of change of the probability density is seen to be successively increasing then the asymptotic behaviour is assumed to go to unity. Empirically this method seems to give accurate results and so the default behaviour is `true`, however it has no theoretical justification.

`Precision` is a real-valued parameter defining the precision to which the threshold should be determined.

Setting the verbose mode `Code` prints out the bounds on the threshold as subsequent density evolutions narrow the search range.

`DensityEvolutionGaussian(v, c, σ)`

`DensityEvolutionGaussian(Sv, Sc, σ)`

`Points`	RngIntElt	*Default :* 500
`MaxLLR`	RngReSubElt	*Default :* 25
`MaxIterations`	RngIntElt	*Default :* ∞
`QuickCheck`	BoolElt	*Default :* `true`

Perform density evolution on the Gaussian channel using standard deviation σ and determine the asymptotic behaviour for the given LDPC ensemble. The return value is boolean, where `true` indicates that σ is below the threshold and the ensemble has error probability asymptotically tending to zero.

See the description of `LDPCGaussianThreshold` for a description of the variable arguments.

`GoodLDPCEnsemble(i)`

Access a small database of density distributions defining good irregular LDPC ensembles. Returned is the published threshold of the ensemble over the Gaussian channel, along with the variable and check degree distributions. The input i is a non-negative integer, which indexes the database (in no particular order).

Example H154E7

Since performing density evolution on a large number of discrete points is time consuming, it is normally better to first get an estimate with an easier computation.
In this example a published value of the threshold of an ensemble (obtained using a different implementation) can be compared to the outputs from different levels of discretization.

```
> thresh, Sv, Sc := GoodLDPCEnsemble(5);
> R4 := RealField(4);
> [R4| x : x in Sv];
[ 0.0000, 0.3001, 0.2839, 0.0000, 0.0000, 0.0000, 0.0000, 0.4159 ]
> [R4| x : x in Sc];
[ 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.2292, 0.7708 ]
> thresh;
```

```
0.94970000000000000000000000000
> time approx1 := LDPCGaussianThreshold(Sv, Sc:
>         Points := 500, Precision := 0.001);
Time: 19.400
> approx1;
0.960976562499999964472863211995
> time approx2 := LDPCGaussianThreshold(Sv, Sc:
>       Points := 3000,
>       Lower := approx1-0.02, Upper := approx1+0.02);
Time: 873.560
> approx2;
0.951210937499999964472863211995
```

This estimate can now be used to narrow the search range of accurate density evolution. For the very long calculation the verbose mode is used to keep track of the progress of the calculation.

```
> SetVerbose("Code", true);
> time approx3 := LDPCGaussianThreshold(Sv, Sc:
>        Points := 5000,
>        Lower := approx2-0.005, Upper := approx2+0.0005);
Determining the mapping matrix...
                        ...mapping matrix obtained  19.10s
Threshold Determination for LDPC code ensemble:
c:  (6):0.229190 (7):0.770810
v:  (2):0.300130 (3):0.283950 (8):0.415920
will be found to precision 0.000050
Max LLR is 25.000000 distributed across 5000 points
Asymptotic behaviour determination is: fast
Beginning search with lb = 0.946211, ub = 0.951711
New Bounds: lb = 0.948961, ub = 0.951711  114.41s
New Bounds: lb = 0.950336, ub = 0.951711  367.19s
New Bounds: lb = 0.950336, ub = 0.951023  553.95s
New Bounds: lb = 0.950336, ub = 0.950680  814.35s
New Bounds: lb = 0.950508, ub = 0.950680  1261.46s
New Bounds: lb = 0.950508, ub = 0.950594  1557.46s
New Bounds: lb = 0.950508, ub = 0.950551  1891.52s
Time: 2136.150
```

The thresholds given in the database are published values taken from other implementations, and so are not guaranteed to match up exactly with the values obtained using MAGMA.

155 LINEAR CODES OVER FINITE RINGS

Chapter 155
LINEAR CODES OVER FINITE RINGS

155.1 Introduction

This chapter describes those functions which are applicable to linear codes over finite rings. MAGMA currently supports the basic facilities for codes over integer residue rings and galois rings, including cyclic codes, and the complete weight enumerator calculation. Additional functionality is available for the special case of codes over Z_4, the integers modulo 4.

For modules defined over rings with zero divisors, it is of course not possible to talk about the concept of dimension (the modules are not free). But in MAGMA each code over such a ring has a *unique* generator matrix corresponding to the *Howell form*. The number of rows k in this unique generator matrix will be called the *pseudo-dimension* of the code. It should be noted that this pseudo-dimension is not invariant between equivalent codes, and so does not provide structural information like the dimension of a code over a finite field.

Note that the rank of the generator matrix is always well-defined and unique (based on the Smith form which is well-defined over PIRs), but k may sometimes be larger than the rank.

Without a concept of dimension, codes over finite rings are referenced by their cardinality. A code C is called an (n, M, d) *code* if it has length n, cardinality M and minimum Hamming weight d.

In this chapter, as for codes over finite fields, the term "code" will refer to a linear code, unless otherwise specified.

The reader is referred to [Wan97] as a general reference on $\mathbf{Z}_4$-codes.

155.2 Construction of Codes

155.2.1 Construction of General Linear Codes

```
LinearCode< R, n | L >
```

Create a code as a subspace of the R-space $V = R^{(n)}$ which is generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of V.

(b) A set or sequence of elements of V.

(c) A sequence of n elements of R, defining an element of V.

(d) A set or sequence of sequences of type (c).

(e) A subspace of V.

(f) A set or sequence of subspaces of V.

`LinearCode(U)`

Let V be the R-space $R^{(n)}$ and suppose that U is a subspace of V. The effect of this function is to define the linear code C corresponding to the subspace U.

`LinearCode(A)`

Given a $k \times n$ matrix A over the ring R, construct the linear code generated by the rows of A. Note that it is not assumed that the rank of A is k. The effect of this constructor is otherwise identical to that described above.

`PermutationCode(u, G)`

Given a finite permutation group G of degree n, and a vector u belonging to the n-dimensional vector space V over the ring R, construct the code C corresponding to the subspace of V spanned by the set of vectors obtained by applying the permutations of G to the vector u.

Example H155E1

The octacode O_8 over $\mathbf{Z}_4$ [Wan97, Ex. 1.3] can be defined as follows:

```
> Z4 := IntegerRing(4);
> O8 := LinearCode<Z4, 8 |
>     [1,0,0,0,3,1,2,1],
>     [0,1,0,0,1,2,3,1],
>     [0,0,1,0,3,3,3,2],
>     [0,0,0,1,2,3,1,1]>;
> O8;
[8, 4, 4] Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2 1]
[0 1 0 0 1 2 3 1]
[0 0 1 0 3 3 3 2]
[0 0 0 1 2 3 1 1]
```

Alternatively, if we want to see the code as a subspace of $R^{(8)}$, where $R = \mathbf{Z}_4$, we could proceed as follows:

```
> O8 := LinearCode(sub<RSpace(Z4, 8) |
>     [1,0,0,0,3,1,2,1],
>     [0,1,0,0,1,2,3,1],
>     [0,0,1,0,3,3,3,2],
>     [0,0,0,1,2,3,1,1]>);
```

Example H155E2

We define a code by constructing a matrix over GR$(4, 3)$, and using its rowspace to generate the code:

```
> R<w> := GaloisRing(4,3);
> S := [1, 1, 0, w^2, w, w + 2, 2*w^2, 2*w^2 + w + 3];
> G := Matrix(R, 2, 4, S);
> G;
[            1             1             0           w^2]
[            w         w + 2         2*w^2 2*w^2 + w + 3]
> C := LinearCode(G);
> C;
(4, 512, 3) Linear Code over GaloisRing(2, 2, 3)
Generator matrix:
[          1           1           0         w^2]
[          0           2       2*w^2 2*w^2 + 2*w]
> #C;
512
```

Example H155E3

We define G to be a permutation group of degree 7 and construct the code C as the $\mathbf{Z}_4$-code generated by applying the permutations of G to a certain vector:

```
> G := PSL(3, 2);
> G;
Permutation group G of degree 7
    (1, 4)(6, 7)
    (1, 3, 2)(4, 7, 5)
> Z4 := IntegerRing(4);
> V := RSpace(Z4, 7);
> u := V ! [1, 0, 0, 1, 0, 1, 1];
> C := PermutationCode(u, G);
> C;
[7, 6, 2] Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 1 0 1 1]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
[0 0 0 2 0 0 2]
[0 0 0 0 2 0 2]
[0 0 0 0 0 2 2]
```

155.2.2 Construction of Simple Linear Codes

`ZeroCode(R, n)`

Given a ring R and positive integer n, return the $(n, 0, n)$ code consisting of only the zero code word, (where the minimum weight is by convention equal to n).

`RepetitionCode(R, n)`

Given a ring R and positive integer n, return the length n code with minimum Hamming weight n, generated by the all-ones vector.

`ZeroSumCode(R, n)`

Given a ring R and positive integer n, return the length n code over R such that for all codewords $(c_1, c_2, \ldots, c_n)$ we have $\sum_i c_i = 0$.

`UniverseCode(R, n)`

Given a ring R and positive integer n, return the length n code with minimum Hamming weight 1, consisting of all possible codewords.

`RandomLinearCode(R, n, k)`

Given a finite ring R and positive integers n and k, such that $0 < k \leq n$, the function returns a random linear code of length n over R with k generators.

Example H155E4

The repetition and zero sum codes are dual over all rings.

```
> R := Integers(9);
> C1 := RepetitionCode(R, 5);
> C1;
(5, 9, 5) Linear Code over IntegerRing(9)
Generator matrix:
[1 1 1 1 1]
> C2 := ZeroSumCode(R, 5);
> C2;
(5, 6561, 2) Linear Code over IntegerRing(9)
Generator matrix:
[1 0 0 0 8]
[0 1 0 0 8]
[0 0 1 0 8]
[0 0 0 1 8]
> C1 eq Dual(C2);
true
```

155.2.3 Construction of General Cyclic Codes

Cyclic codes form an important family of linear codes over all rings. A cyclic code is one which is generated by all of the cyclic shifts of a given codeword:

$$(c_0, c_1, \ldots, c_{n-1}, c_n), (c_n, c_0, \ldots, c_{n-2}, c_{n-1}), \ldots, (c_1, c_2, \ldots, c_n, c_0)$$

Using the correspondence $(c_0, c_1, \ldots, c_n) \iff c_0 + c_1 x + \cdots + c_n x^n$, the cyclic codes of length n over the ring R are in one-to-one correspondence with the principal ideals of $R[x]/(x^n - 1)R[x]$.

`CyclicCode(u)`

Given a vector u belonging to the R-space $R^{(n)}$, construct the length n cyclic code generated by the right cyclic shifts of the vector u.

`CyclicCode(n, g)`

Let R be a ring. Given a positive integer n and a univariate polynomial $g(x) \in R[x]$, construct the length n cyclic code generated by $g(x)$.

`CyclotomicFactors(R, n)`

Given a Galois ring R (which is possibly an integer residue ring with a prime power modulus), and a positive integer n which is coprime to the characteristic of R, return a factorisation of $x^n - 1$ over R.

Note that since factorisation is not necessarily unique over R, the factorisation returned is the one obtained by first factoring over the residue field of R and then performing Hensel lifting.

Example H155E5

We construct some cyclic codes over $\mathbf{Z}_4$ by factorizing $x^n - 1$ over $\mathbf{Z}_4$ for $n = 7, 23$ and using some of the irreducible factors found.

```
> Z4 := IntegerRing(4);
> P<x> := PolynomialRing(Z4);
> n := 7; L := CyclotomicFactors(Z4, n); L;
[
    x + 3,
    x^3 + 2*x^2 + x + 3,
    x^3 + 3*x^2 + 2*x + 3
]
> CyclicCode(n, L[1]);
[7, 6, 2] Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 3]
[0 1 0 0 0 0 3]
[0 0 1 0 0 0 3]
[0 0 0 1 0 0 3]
```

```
[0 0 0 0 1 0 3]
[0 0 0 0 0 1 3]
> CyclicCode(n, L[2]);
[7, 4, 3] Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2]
[0 1 0 0 2 1 1]
[0 0 1 0 1 1 3]
[0 0 0 1 3 2 3]
> CyclicCode(n, L[3]);
[7, 4, 3] Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 2 3]
[0 1 0 0 3 1 1]
[0 0 1 0 1 1 2]
[0 0 0 1 2 1 3]
> n := 23; L := CyclotomicFactors(Z4, n); L;
[
    x + 3,
    x^11 + 2*x^10 + 3*x^9 + 3*x^7 + 3*x^6 + 3*x^5 + 2*x^4 + x + 3,
    x^11 + 3*x^10 + 2*x^7 + x^6 + x^5 + x^4 + x^2 + 2*x + 3
]
> CyclicCode(n, L[2]);
[23, 12] Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 3 1 0 0 2 3 3 3 0 3 2]
[0 1 0 0 0 0 0 0 0 0 0 0 2 1 1 0 0 0 1 1 3 2 3]
[0 0 1 0 0 0 0 0 0 0 0 0 3 3 1 1 2 3 3 0 1 2 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 3 3 1 1 2 3 3 0 1 2]
[0 0 0 0 1 0 0 0 0 0 0 0 2 2 3 3 1 3 0 1 3 2 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 1 3 0 1 3 3 0 2 2 1 3]
[0 0 0 0 0 0 0 0 0 1 0 0 3 2 3 0 3 2 2 3 2 1 3]
[0 0 0 0 0 0 0 0 0 0 1 0 3 0 2 3 2 2 1 1 3 1 3]
[0 0 0 0 0 0 0 0 0 0 0 1 3 0 0 2 1 1 1 0 1 2 3]
```

Example H155E6

We create a cyclic code of length 5 over $\mathtt{GR}(4, 2)$.

```
> R<w> := GR(4,2);
> P<x> := PolynomialRing(R);
> g := CyclotomicFactors(R, 5)[2];
> g;
x^2 + (3*w + 2)*x + 1
> C := CyclicCode(5, g);
```

```
> C;
(5, 4096, 3) Cyclic Code over GaloisRing(2, 2, 2)
Generator matrix:
[      1       0       0       1 3*w + 2]
[      0       1       0   w + 2   w + 2]
[      0       0       1 3*w + 2       1]
```

155.3 Invariants of Codes

`#C`

Given a code C, return the number of codewords belonging to C.

`C . i`
`Name(C, i)`

Given a code C and a positive integer i, return the i-th generator of C.

`Alphabet(C)`

The underlying ring (or alphabet) R of the code C.

`AmbientSpace(C)`

The ambient space of the code C, i.e., the generic R-space V in which C is contained.

`Basis(C)`

The basis of the linear code C, returned as a sequence of elements of C.

`Generators(C)`

The generators for the linear code C, returned as a set.

`GeneratorMatrix(C)`

The generator matrix for the linear code C. This gives a unique canonical generating set for the code.

`Generic(C)`

Given a length n code C over a ring R, return the generic $(n, \#R^n, 1)$ code in which C is contained.

`Length(C)`

Given an code C, return the block length n of C.

`PseudoDimension(C)`

`NumberOfGenerators(C)`

`Ngens(C)`

The number of generators (which equals the pseudo-dimension k) of the linear code C.

`ParityCheckMatrix(C)`

The parity check matrix for the code C, which can be defined as the canonical generator matrix of the dual of C.

`Random(C)`

A random codeword of the code C.

`RSpace(C)`

Given a length n linear code C, defined as a subspace U of the n-dimensional space V, return U as a subspace of V with basis corresponding to the rows of the generator matrix for C.

`InformationRate(C)`

Given a code C over a ring with cardinality q, return the information rate of C, that is, the ratio $Log_q(\#C)/n$.

155.4 Codes over $\mathbf{Z}_4$

The ring $\mathbf{Z}_4$, the ring of integers modulo 4, is a special case for which extra functionality is available. Error correcting codes over $\mathbf{Z}_4$ are often referred to as *quaternary* codes.

Important concepts when discussing quaternary codes are *Lee weight* and the *Gray map*, which maps linear codes over $\mathbf{Z}_4$ to (possibly non-linear) codes over $\mathbf{Z}_2$. Many good non-linear binary codes can be defined as the images of simple linear quaternary codes.

155.4.1 The Gray Map

For an element $x \in \mathbf{Z}_4$, the *Gray map* $\phi : \mathbf{Z}_4 \to \mathbf{Z}_2^2$ is defined by:

$$0 \mapsto 00, \quad 1 \mapsto 01, \quad 2 \mapsto 11, \quad 3 \mapsto 10.$$

This map is extended to a map from $\mathbf{Z}_4^n$ onto $\mathbf{Z}_2^{2n}$ in the obvious way (by concatenating the images of each component). The resulting map is a weight- and distance-preserving map from $\mathbf{Z}_4^n$ (with Lee weight metric) to $\mathbf{Z}_2^{2n}$ (with Hamming weight metric). See [Wan97, Chapter 3] for more information (but note that that author has a different order for the components of the image of a vector).

`GrayMap(C)`

Given a $\mathbf{Z}_4$-linear code C, this function returns the Gray map for C. This is the map ϕ from C to $\mathbf{F}_2^{2n}$, as defined above.

`GrayMapImage(C)`

Given a $\mathbf{Z}_4$-linear code C, this function returns the image of C under the Gray map as a sequence of vectors in $\mathbf{F}_2^{2n}$. As the resulting image may not be a $\mathbf{F}_2$-linear code, a sequence of vectors is returned rather than a code.

`HasLinearGrayMapImage(C)`

Given a $\mathbf{Z}_4$-linear code C, this function returns true if and only if the image of C under the Gray map is a $\mathbf{F}_2$-linear code. If so, the function also returns the image B as a $\mathbf{F}_2$-linear code, together with the bijection $\phi : C \to B$.

Example H155E7

Let $\phi(O_8)$ be the image of the octacode O_8 under the Gray map. This image is not a $\mathbf{F}_2$-linear code, but it is the non-linear $(8, 256, 6)$ Nordstrom-Robinson code [Wan97, Ex.3.4]. We demonstrate that the Hamming weight distribution of the $\mathbf{F}_2$ image is identical to the Lee weight distribution of the linear $\mathbf{Z}_4$ code.

```
> Z4 := IntegerRing(4);
> O8 := LinearCode<Z4, 8 |
>     [1,0,0,0,3,1,2,1],
>     [0,1,0,0,1,2,3,1],
>     [0,0,1,0,3,3,3,2],
>     [0,0,0,1,2,3,1,1]>;
> HasLinearGrayMapImage(O8);
false
> NR := GrayMapImage(O8);
> #NR;
256
> LeeWeightDistribution(O8);
[ <0, 1>, <6, 112>, <8, 30>, <10, 112>, <16, 1> ]
> {* Weight(v): v in NR *};
{* 0, 16, 6^^112, 8^^30, 10^^112 *}
```

For the code K_8, we first note the image of some of the vectors under the Gray map.

```
> Z4 := IntegerRing(4);
> K8 := LinearCode< Z4, 8 |
>     [1,1,1,1,1,1,1,1],
>     [0,2,0,0,0,0,0,2],
>     [0,0,2,0,0,0,0,2],
>     [0,0,0,2,0,0,0,2],
>     [0,0,0,0,2,0,0,2],
>     [0,0,0,0,0,2,0,2],
>     [0,0,0,0,0,0,2,2]>;
> f := GrayMap(K8);
> K8.1;
(1 1 1 1 1 1 1 1)
> f(K8.1);
```

```
(0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1)
> K8.2;
(0 2 0 0 0 0 0 2)
> f(K8.2);
(0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1)
```

Finally, we see that the image of K_8 is linear over $\mathbf{F}_2$.

```
> l, B, g := HasLinearGrayMapImage(K8);
> l;
true
> B;
[16, 8, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 0]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1]
[0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1]
[0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
> g(K8.1) in B;
true
```

155.4.2 Families of Codes over $\mathbf{Z}_4$

This section gives some standard constructions for $\mathbf{Z}_4$-linear codes. Further constructions will be available in the future.

`KerdockCode(m)`

Given an integer $m \geq 2$, return the quaternary Kerdock code $K(m)$ of length $2^m - 1$ defined by a default primitive polynomial $h \in \mathbf{Z}_4[x]$ of degree m.

`PreparataCode(m)`

Given an integer $m \geq 2$, return the quaternary Preparata code $P(m)$ of length $2^m - 1$ defined by a default primitive polynomial $h \in \mathbf{Z}_4[x]$ of degree m.

`ReedMullerCodeZ4(r, m)`

Given an integer $m \geq 2$ and an integer r such that $0 \leq r \leq m$ this function returns the r-th order Reed-Muller code over Z_4 of length 2^m.

`GoethalsCode(m)`

Given a positive integer m, where m must be an odd and greater than or equal to 3, return the Goethals code of length 2^m.

`DelsarteGoethalsCode(m, delta)`

Return the Delsarte-Goethals Code of length 2^m.

`GoethalsDelsarteCode(m, delta)`

Return the Goethals-Delsarte code of length 2^m

`QRCodeZ4(p)`

Given a prime number p such that 2 is a quadratic residue modulo p, return the quadratic residue code of length p over Z_4.

`GolayCodeZ4(e)`

Return the Golay Code over Z_4. If e is `true` then return the extended Golay Code

`SimplexAlphaCodeZ4(k)`

Return the simplex alpha code over Z_4 of degree k.

`SimplexBetaCodeZ4(k)`

Return the simplex beta code over Z_4 of degree k.

Example H155E8

We compute some default Kerdock and Preparata codes and their minimum Lee weights.

```
> PreparataCode(3);
(8, 256, 4) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2 1]
[0 1 0 0 2 1 1 3]
[0 0 1 0 1 1 3 2]
[0 0 0 1 3 2 3 3]
> MinimumLeeWeight($1);
6
> KerdockCode(4);
[16, 5, 8] Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 1 1 3 0 3 3 0 2 1 2 3]
[0 1 0 0 0 2 3 3 3 2 1 3 0 0 1 1]
[0 0 1 0 0 3 1 0 3 0 3 1 1 3 2 2]
[0 0 0 1 0 2 1 3 0 1 2 3 1 3 3 0]
[0 0 0 0 1 1 3 0 3 3 0 2 1 2 1 3]
> MinimumLeeWeight($1);
12
> KerdockCode(5);
(32, 4096, 16) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 3 3 3 2 0 3 2 2 0 3 0 1 0 1 3 1 1 0 3 1 2 3 2 2 3 3]
[0 1 0 0 0 0 3 2 2 1 2 3 1 0 2 3 3 1 1 1 0 0 2 1 3 0 3 1 1 0 1 2]
```

```
[0 0 1 0 0 0 1 0 3 0 1 3 1 3 0 3 3 2 1 0 2 3 3 2 2 2 2 0 3 3 1 3]
[0 0 0 1 0 0 1 2 1 1 0 2 1 3 3 1 3 2 2 0 1 1 2 3 3 1 0 3 2 1 0 0]
[0 0 0 0 1 0 0 1 2 1 1 0 2 1 3 3 1 3 2 2 0 1 1 2 3 3 1 0 3 2 1 0]
[0 0 0 0 0 1 1 1 2 0 1 2 2 0 1 0 3 0 3 1 3 3 0 1 3 2 1 2 2 1 3 1]
> MinimumLeeWeight($1);
28
```

`HadamardCodeZ4(δ, m)`

Given an integer $m \geq 2$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, return a Hadamard code over $\mathbf{Z}_4$ of length 2^{m-1} and type $2^\gamma 4^\delta$, where $\gamma = m + 1 - 2\delta$. Moreover, return a generator matrix with $\gamma + \delta$ rows constructed in a recursive way from the `Plotkin` and `BQPlotkin` constructions defined in Section 155.4.5.

A Hadamard code over $\mathbf{Z}_4$ of length 2^{m-1} is a code over $\mathbf{Z}_4$ such that, after the Gray map, give a binary (not necessarily linear) code with the same parameters as the binary Hadamard code of length 2^m.

`ExtendedPerfectCodeZ4(δ, m)`

Given an integer $m \geq 2$ and an integer δ such that $1 \leq \delta \leq \lfloor (m+1)/2 \rfloor$, return an extended perfect code over $\mathbf{Z}_4$ of length 2^{m-1}, such that its dual code is of type $2^\gamma 4^\delta$, where $\gamma = m + 1 - 2\delta$. Moreover, return a generator matrix constructed in a recursive way from the `Plotkin` and `BQPlotkin` constructions defined in Section 155.4.5.

An extended perfect code over $\mathbf{Z}_4$ of length 2^{m-1} is a code over $\mathbf{Z}_4$ such that, after the Gray map, give a binary (not necessarily linear) code with the same parameters as the binary extended perfect code of length 2^m.

Example H155E9

We compute codes over $\mathbf{Z}_4$ such that, after the Gray map, they are binary codes with the same parameters as some well-known families of binary linear codes.
First, we define a Hadamard code C over $\mathbf{Z}_4$ of length 8 and type $2^1 4^2$. The matrix Gc is the quaternary matrix used to generate C and obtained in a recursive way from `Plotkin` and `BQPlotkin` constructions.

```
> C, Gc := HadamardCodeZ4(2,4);
> C;
((8, 4^2 2^1)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 3 2 1 0 3 2]
[0 1 2 3 0 1 2 3]
[0 0 0 0 2 2 2 2]
> Gc;
[1 1 1 1 1 1 1 1]
[0 1 2 3 0 1 2 3]
[0 0 0 0 2 2 2 2]
> HasLinearGrayMapImage(C);
```

```
true [16, 5, 8] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 1 1 0 1 1 1 1 0 0 0]
[0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1]
[0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
Mapping from: CodeLinRng: C to [16, 5, 8] Linear Code over GF(2) given by a rule
```

Then, we define an extended perfect code D over $\mathbf{Z}_4$ of length 8, such that its dual code is of type $2^1 4^2$. The matrix Gd is the quaternary matrix which is used to generate D and obtained in a recursive way from `Plotkin` and `BQPlotkin` constructions. Note that the code D is the Kronecker dual code of C.

```
> D, Gd := ExtendedPerfectCodeZ4(2,4);
> D;
((8, 4^5 2^1)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 1 0 0 1 3]
[0 1 0 1 0 0 2 2]
[0 0 1 1 0 0 1 1]
[0 0 0 2 0 0 0 2]
[0 0 0 0 1 0 3 2]
[0 0 0 0 0 1 2 3]
> Gd;
[1 1 1 1 1 1 1 1]
[0 1 2 3 0 1 2 3]
[0 0 1 1 0 0 1 1]
[0 0 0 2 0 0 0 2]
[0 0 0 0 1 1 1 1]
[0 0 0 0 0 1 2 3]
> DualKroneckerZ4(C) eq D;
true
```

`ReedMullerCodeZ4(r, m)`

`ReedMullerCodeQRMZ4(r, m)`

Given an integer $m \geq 2$ and an integer r such that $0 \leq r \leq m$, return the r-th order Reed-Muller code over $\mathbf{Z}_4$ of length 2^m.

The binary image under the modulo 2 map is the binary linear r-th order Reed-Muller code of length 2^m. For $r = 1$ and $r = m - 2$, the function returns the quaternary linear Kerdock and Preparata code, respectively.

`ReedMullerCodesLRMZ4(r, m)`

Given an integer $m \geq 1$ and an integer r such that $0 \leq r \leq m$, return a set of r-th order Reed-Muller codes over $\mathbf{Z}_4$ of length 2^{m-1}.

The binary image under the Gray map of any of these codes is a binary (not necessarily linear) code with the same parameters as the binary linear r-th order Reed-Muller code of length 2^m. Note that for these codes neither the usual inclusion nor duality properties of the binary linear Reed-Muller family are satisfied.

`ReedMullerCodeRMZ4(s, r, m)`

Given an integer $m \geq 1$, an integer r such that $0 \leq r \leq m$, and an integer s such that $0 \leq s \leq \lfloor (m-1)/2 \rfloor$, return a r-th order Reed-Muller code over $\mathbf{Z}_4$ of length 2^{m-1}, denoted by $RM_s(r, m)$.

The binary image under the Gray map is a binary (not necessarily linear) code with the same parameters as the binary linear r-th order Reed-Muller code of length 2^m. Note that the inclusion and duality properties are also satisfied, that is, the code $RM_s(r-1, m)$ is a subcode of $RM_s(r, m)$, $r > 0$, and the code $RM_s(r, m)$ is the Kronecker dual code of $RM_s(m-r-1, m)$, $r < m$.

Example H155E10

We define $RM_1(1, 4)$ and $RM_1(2, 4)$. We can see that the former is a subcode of the latter. Note that $RM_1(1, 4)$ and $RM_1(2, 4)$ are the same as the ones given in Example H155E9 by `HadamardCodeZ4(2,4)` and `ExtendedPerfectCodeZ4(2,4)`, respectively.

```
> C1,G1 := ReedMullerCodeRMZ4(1,1,4);
> C2,G2 := ReedMullerCodeRMZ4(1,2,4);
> C1;
((8, 4^2 2^1)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 3 2 1 0 3 2]
[0 1 2 3 0 1 2 3]
[0 0 0 0 2 2 2 2]
> C2;
((8, 4^5 2^1)) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 1 0 0 1 3]
[0 1 0 1 0 0 2 2]
[0 0 1 1 0 0 1 1]
[0 0 0 2 0 0 0 2]
[0 0 0 0 1 0 3 2]
[0 0 0 0 0 1 2 3]
> C1 subset C2;
true
> DualKroneckerZ4(C2) eq C1;
true
```

`ReedMullerCodesRMZ4(s, m)`

Given an integer $m \geq 1$, and an integer s such that $0 \leq s \leq \lfloor (m-1)/2 \rfloor$, return the family of Reed-Muller codes over $\mathbf{Z}_4$ of length 2^{m-1}, that is, the codes $RM_s(r, m)$, for all $0 \leq r \leq m$.

The binary image of these codes under the Gray map gives a family of binary (not necessarily linear) codes with the same parameters as the binary linear Reed-Muller family of codes of length 2^m. Note that $RM_s(0, m) \subset RM_s(1, m) \subset \ldots \subset RM_s(m, m)$.

Example H155E11

We construct the family of Reed-Muller codes over $\mathbf{Z}_4$ of length 2^2 given by $s = 0$.

```
> F := ReedMullerCodesRMZ4(0,3);
> F;
[((4, 4^0 2^1)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[2 2 2 2],
((4, 4^1 2^2)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[1 1 1 1]
[0 2 0 2]
[0 0 2 2],
((4, 4^3 2^1)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 2],
((4, 4^4 2^0)) Cyclic Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]]
> F[1] subset F[2] and F[2] subset F[3] and F[3] subset F[4];
true
```

155.4.3 Derived Binary Codes

As well as the binary image of a quaternary code under the Gray map (see section 155.4.1), there are also two other associated canonical binary codes. They are known the *residue* and *torsion* codes, the former being a subcode of the latter.

From any binary code-subcode pair $C_1 \subset C_2$, a quaternary code C can be constructed such that the residue and torsion codes of C will be C_1 and C_2 respectively. Note that this quaternary code is not unique.

`BinaryResidueCode(C)`

Given a quaternary code C, return the binary code formed by taking each codeword in C modulo 2. This is known as the *binary residue code* of C.

`BinaryTorsionCode(C)`

Given a quaternary code C, return the binary code formed by the support of each codeword in C which is zero modulo 2. This is known as the *binary torsion code* of C.

`Z4CodeFromBinaryChain(C1, C2)`

Given binary code C_1 and C_2 such that $C_1 \subset C_2$, return a quaternary code such that its binary residue code is C_1 and its binary torsion code is C_2.

Example H155E12

We look at the derived binary codes of the $\mathbf{Z}_4$ Golay code, which are in fact equal to the binary Golay code.

```
> C := GolayCodeZ4(false);
> C;
(23, 16777216) Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 3 1 0 0 2 3 3 3 0 3 2]
[0 1 0 0 0 0 0 0 0 0 0 0 2 1 1 0 0 0 1 1 3 2 3]
[0 0 1 0 0 0 0 0 0 0 0 0 3 3 1 1 2 3 3 0 1 2 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 3 3 1 1 2 3 3 0 1 2]
[0 0 0 0 1 0 0 0 0 0 0 0 2 2 3 3 1 3 0 1 3 2 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 2 3 1 2 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 1 3 0 1 3 3 0 2 2 1 3]
[0 0 0 0 0 0 0 0 0 1 0 0 3 2 3 0 3 2 2 3 2 1 3]
[0 0 0 0 0 0 0 0 0 0 1 0 3 0 2 3 2 2 1 1 3 1 3]
[0 0 0 0 0 0 0 0 0 0 0 1 3 0 0 2 1 1 1 0 1 2 3]
>
> CRes := BinaryResidueCode(C);
> CTor := BinaryResidueCode(C);
> CRes eq CTor;
true
```

```
> CRes:Minimal;
[23, 12, 7] Linear Code over GF(2)
> AreEq, _ := IsEquivalent( CRes, GolayCode(GF(2), false) );
> AreEq;
true
```

Note that the canonical code over $\mathbf{Z}_4$ corresponding to the derived binary codes $CRes$ and $CTor$ is not the same as the one we started from.

```
> C1 := Z4CodeFromBinaryChain(CRes, CTor);
> C1:Minimal;
(23, 16777216) Linear Code over IntegerRing(4)
> C eq C1;
false
```

155.4.4 The Standard Form

A $\mathbf{Z}_4$-linear code is in *standard form* if its generator matrix is of the form:

$$\begin{pmatrix} I_{k_1} & A & B \\ 0 & 2I_{k_2} & 2C \end{pmatrix}$$

where I_{k_1} and I_{k_2} are the $k_1 \times k_1$ and $k_2 \times k_2$ identity matrices, respectively, A and C are $\mathbf{Z}_2$-matrices, and B is a $\mathbf{Z}_4$-matrix. Any $\mathbf{Z}_4$-linear code C is permutation-equivalent to a code S which is in standard form. Furthermore, the integers k_1 and k_2, defined above, are unique [Wan97, Prop. 1.1].

`StandardForm(C)`

This function, given any $\mathbf{Z}_4$-linear code C, returns a permutation-equivalent code S in standard form, together with the corresponding isomorphism from C onto S.

Example H155E13

We compute the standard form of a certain code. Note that the number of rows in the generator matrix of the standard code may be less than that of the original code.

```
> Z4 := IntegerRing(4);
> C := LinearCode<Z4, 4 | [2,2,1,1], [0,2,0,2]>;
> C;
[4, 3, 2] Linear Code over IntegerRing(4)
Generator matrix:
[2 0 1 3]
[0 2 0 2]
[0 0 2 2]
> S, f := StandardForm(C);
> S;
[4, 2, 2] Linear Code over IntegerRing(4)
Generator matrix:
[1 1 2 2]
```

```
[0 2 2 0]
> #S;
8
> #C;
8
> f(C.1);
(1 3 0 2)
> f(C.2);
(0 2 2 0)
> f(C.3);
(2 2 0 0)
> S.1@@f;
(2 2 1 1)
> S.2@@f;
(0 2 0 2)
```

155.4.5 Constructing New Codes from Old

The functions described here produce a new code over $\mathbf{Z}_4$ by modifying in some way the codewords of some given codes over $\mathbf{Z}_4$.

`PlotkinSum(A, B)`

Given matrices A and B both over the same ring and with the same number of columns, return the P_{AB} matrix over the same ring of A and B, where

$$P_{AB} = \begin{pmatrix} A & A \\ 0 & B \end{pmatrix}.$$

`PlotkinSum(C, D)`

Given codes C and D both over the same ring and of the same length, construct the Plotkin sum of C and D. The Plotkin sum consists of all vectors of the form $(u|u+v)$, where $u \in C$ and $v \in D$.

Note that the Plotkin sum is computed using generator matrices of C and D and the `PlotkinSum` function for matrices, that is, this function returns the code over $\mathbf{Z}_4$ generated by the matrix P_{AB} defined above, where A and B are generators matrices of C and D, respectively.

`QuaternaryPlotkinSum(A, B)`

Given two matrices A and B over $\mathbf{Z}_4$, both with the same number of columns, return the QP_{AB} matrix over $\mathbf{Z}_4$, where

$$QP_{AB} = \begin{pmatrix} A & A & A & A \\ 0 & B & 2B & 3B \end{pmatrix}.$$

`QuaternaryPlotkinSum(C, D)`

Given two codes C and D over $\mathbf{Z}_4$, both of the same length, construct the Quaternary Plotkin sum of C and D. The Quaternary Plotkin sum is a code over $\mathbf{Z}_4$ that consists of all vectors of the form $(u, u+v, u+2v, u+3v)$, where $u \in C$ and $v \in D$.

Note that the Quaternary Plotkin sum is computed using generator matrices of C and D and the `QuaternaryPlotkinSum` function for matrices, that is, this function returns the code over $\mathbf{Z}_4$ generated by the matrix QP_{AB} defined above, where A and B are generators matrices of C and D, respectively.

`BQPlotkinSum(A, B, C)`

Given three matrices A, B, and C over $\mathbf{Z}_4$, all with the same number of columns, return the BQP_{ABC} matrix over $\mathbf{Z}_4$, where

$$BQP_{ABC} = \begin{pmatrix} A & A & A & A \\ 0 & B' & 2B' & 3B' \\ 0 & 0 & \hat{B} & \hat{B} \\ 0 & 0 & 0 & C \end{pmatrix},$$

B' is obtained from B replacing the twos with ones in the rows of order two, and $\hat{B}$ is obtained from B removing the rows of order two.

`BQPlotkinSum(D, E, F)`

Given three codes D, E and F over $\mathbf{Z}_4$, all of the same length, construct the BQ Plotkin sum of D, E and F. Let Ge be a generator matrix of the code E of type $2^\gamma 4^\delta$. The code E' over $\mathbf{Z}_4$ is obtained from E replacing the twos with ones in the γ rows of order two of Ge, and the code $\hat{E}$ over $\mathbf{Z}_4$ is obtained from E removing the γ rows of order two of Ge." The BQ Plotkin sum is a code over $\mathbf{Z}_4$ that consists of all vectors of the form $(u, u+v', u+2v'+\hat{v}, u+3v'+\hat{v}+z)$, where $u \in Gd$, $v' \in Ge'$ $\hat{v} \in \hat{G}e$, and $z \in Gf$, where Gd, Ge', $\hat{G}e$ and Gf are generators matrices of D, E', $\hat{E}$ and F, respectively.

Note that the BQPlotkin sum is computed using generator matrices of D, E and F and the `BQPlotkinSum` function for matrices. However, this function does not necessarily return the same code over $\mathbf{Z}_4$ generated by the matrix QP_{ABC} defined above, where A, B and C are generators matrices of D, E and F, respectively, as shown in Example H2E4.

`DoublePlotkinSum(A, B, C, D)`

Given four matrices A, B, C, and D over $\mathbf{Z}_4$, all with the same number of columns, return the DP_{ABC} matrix over $\mathbf{Z}_4$, where

$$DP_{ABCD} = \begin{pmatrix} A & A & A & A \\ 0 & B & 2B & 3B \\ 0 & 0 & C & C \\ 0 & 0 & 0 & D \end{pmatrix}.$$

`DoublePlotkinSum(E, F, G, H)`

Given four codes E, F, G and H over $\mathbf{Z}_4$, all of the same length, construct the Double Plotkin sum of E, F, G and H. The Double Plotkin sum is a code over $\mathbf{Z}_4$ that consists of all vectors of the form $(u, u+v, u+2v+z, u+3v+z+t)$, where $u \in E$, $v \in F$, $z \in G$ and $t \in H$.

Note that the Double Plotkin sum is computed using generator matrices of E, F, G and H and the `DoublePlotkinSum` function for matrices, that is, this function returns the code over $\mathbf{Z}_4$ generated by the matrix DP_{ABCD} defined above, where A, B, C and D are generators matrices of E, F, G and H, respectively.

`DualKroneckerZ4(C)`

Given a code C over $\mathbf{Z}_4$ of length 2^m, return its Kronecker dual code. The Kronecker dual code of C is $C^{\perp}_{\otimes} = \{x \in \mathbf{Z}_4^{2^m} : x \cdot K_{2^m} \cdot y^t = 0, \forall y \in C\}$, where $K_{2^m} = \otimes_{j=1}^{m} K_2$, $K_2 = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix}$ and $\otimes$ denotes the Kronecker product of matrices. Equivalently, K_{2^m} is a quaternary matrix of length 2^m with the vector $(1, 3, 3, 1, 3, 1, 1, 3, \ldots)$ in the main diagonal and zeros elsewhere.

Example H155E14

In this example, we show that the codes over $\mathbf{Z}_4$ constructed from the `BQPlotkinSum` function for matrices are not necessarily the same as the ones constructed from the `BQPlotkinSum` function for codes.

```
> Z4:=IntegerRing(4);
> Ga:=Matrix(Z4,1,2,[1,1]);
> Gb:=Matrix(Z4,2,2,[1,2,0,2]);
> Gc:=Matrix(Z4,1,2,[2,2]);
> Ca:=LinearCode(Ga);
> Cb:=LinearCode(Gb);
> Cc:=LinearCode(Gc);
> C:=LinearCode(BQPlotkinSum(Ga,Gb,Gc));
> D:=BQPlotkinSum(Ca,Cb,Cc);
> C eq D;
false
```

Example H155E15

```
> Ga := GeneratorMatrix(ReedMullerCodeRMZ4(1,2,3));
> Gb := GeneratorMatrix(ReedMullerCodeRMZ4(1,1,3));
> Gc := GeneratorMatrix(ReedMullerCodeRMZ4(1,0,3));
> C := ReedMullerCodeRMZ4(1,2,4);
> Cp := LinearCode(PlotkinSum(Ga, Gb));
> C eq Cp;
true
> D := ReedMullerCodeRMZ4(2,2,5);
> Dp := LinearCode(BQPlotkinSum(Ga, Gb, Gc));
> D eq Dp;
true
```

155.4.6 Invariants of Codes over $\mathbf{Z}_4$

`SpanZ2CodeZ4(C)`

Given a code C over $\mathbf{Z}_4$ of length n, return $S_C = \Phi^{-1}(S_{bin})$ as a code over $\mathbf{Z}_4$, and the linear span of C_{bin}, $S_{bin} = \langle C_{bin} \rangle$, as a binary linear code of length $2n$, where $C_{bin} = \Phi(C)$ and Φ is the Gray map.

`KernelZ2CodeZ4(C)`

Given a code C over $\mathbf{Z}_4$ of length n, return its kernel K_C as a subcode over $\mathbf{Z}_4$ of C, and $K_{bin} = \Phi(K_C)$ as a binary linear subcode of C_{bin} of length $2n$, where $C_{bin} = \Phi(C)$ and Φ is the Gray map.

The kernel K_C contains the codewords v such that $2v * u \in C$ for all $u \in C$, where $*$ denotes the component-wise product. Equivalently, the kernel $K_{bin} = \Phi(K_C)$ contains the codewords $c \in C_{bin}$ such that $c + C_{bin} = C_{bin}$, where $C_{bin} = \Phi(C)$ and Φ is the Gray map.

`DimensionOfSpanZ2(C)`

`RankZ2(C)`

Given a code C over $\mathbf{Z}_4$, return the dimension of the linear span of C_{bin}, that is, the dimension of $\langle C_{bin} \rangle$, where $C_{bin} = \Phi(C)$ and Φ is the Gray map.

`DimensionOfKernelZ2(C)`

Given a code C over $\mathbf{Z}_4$, return the dimension of the Gray map image of its kernel K_C over $\mathbf{Z}_4$, that is the dimension of $K_{bin} = \Phi(K_C)$, where Φ is the Gray map. Note that K_{bin} is always a binary linear code.

Example H155E16

```
> C := ReedMullerCodeRMZ4(0,3,5);
> DimensionOfKernelZ2(C);
20
> DimensionOfSpanZ2(C);
27
> K, Kb := KernelZ2CodeZ4(C);
> S, Sb := SpanZ2CodeZ4(C);
> K subset C;
true
> C subset S;
true
> Dimension(Kb) eq DimensionOfKernelZ2(C);
true
> Dimension(Sb) eq DimensionOfSpanZ2(C);
true
```

155.4.7 Other $\mathbf{Z}_4$ functions

`Correlation(v)`

Let v be a codeword over $\mathbf{Z}_4$. Define $w_j = \#\{k : v[k] = j\}$ for $j = 0, \ldots, 3$. Then the *correlation* of v is the Gaussian integer $(w_0 - w_2) + i * (w_1 - w_3)$.

155.5 Construction of Subcodes of Linear Codes

155.5.1 The Subcode Constructor

`sub< C | L >`

Given a length n linear code C over R, construct the subcode of C, generated by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of C;

(b) A set or sequence of elements of C;

(c) A sequence of n elements of R, defining an element of C;

(d) A set or sequence of sequences of type (c);

(e) A subcode of C;

(f) A set or sequence of subcodes of C.

`Subcode(C, t)`

Given a length n linear code C with k generators and an integer t, $1 \leq t < k$, return a subcode of C of pseudo-dimension t.

`Subcode(C, S)`

Given a length n linear code C with k generators and a set S of integers, each of which lies in the range $[1, k]$, return the subcode of C generated by the basis elements whose positions appear in S.

Example H155E17

We construct a subcode of a code over a Galois ring by multiplying each of its generators by a zero divisor.

```
> R<w> := GR(4,2);
> C := RandomLinearCode(R, 4, 2);
> C;
(4, 256, 3) Linear Code over GaloisRing(2, 2, 2)
Generator matrix:
[      1       0   w + 1 3*w + 2]
[      0       1 3*w + 1       1]
> #C;
256
>
> C1 := sub< C | 2*C.1, 2*C.2 >;
> C1;
(4, 16, 3) Linear Code over GaloisRing(2, 2, 2)
Generator matrix:
[      2       0 2*w + 2     2*w]
[      0       2 2*w + 2       2]
> #C1;
16
```

155.6 Weight Distributions

In the case of a linear code, weight and distance distributions are equivalent (in particular minimum weight and minimum distance are equivalent).

155.6.1 Hamming Weight

For an element $x \in \mathbf{R}$ for any finite ring R, the *Hamming weight* $w_H(x)$ is defined by:

$$w_H(x) = 0 \iff x = 0, \qquad w_H(x) = 1 \iff x \neq 0$$

The *Hamming weight* $w_H(v)$ of a vector $v \in R^n$ is defined to be the sum (in $\mathbf{Z}$) of the Hamming weights of its components.

The *Hamming weight* is often referred to as simply the *weight*.

`MinimumWeight(C)`

`MinimumDistance(C)`

Determine the minimum (Hamming) weight of the words belonging to the code C, which is also the minimum distance between any two codewords.

`WeightDistribution(C)`

Determine the (Hamming) weight distribution for the code C. The distribution is returned in the form of a sequence of tuples, where the i-th tuple contains the i-th weight, w_i say, and the number of codewords having weight w_i.

`DualWeightDistribution(C)`

The (Hamming) weight distribution of the dual code of C. For more explanation, see `WeightDistribution`.

Example H155E18

We calculate the weight distribution of a cyclic code over the Galois ring of size 81.

```
> R<w> := GR(9,2);
> P<x> := PolynomialRing(R);
> L := CyclotomicFactors(R, 4);
> g := L[3] * L[4];
> g;
x^2 + (8*w + 7)*x + w + 1
> C := CyclicCode(4, g);
> C;
(4, 6561, 3) Cyclic Code over GaloisRing(3, 2, 2)
Generator matrix:
[      1       0   w + 1 8*w + 7]
[      0       1       w 8*w + 8]
> WeightDistribution(C);
[ <0, 1>, <3, 320>, <4, 6240> ]
```

155.6.2 Lee Weight

For an element $x \in \mathbf{Z}_4$, the *Lee weight* $w_L(x)$ is defined by:

$$w_L(0) = 0, \quad w_L(1) = w_L(3) = 1, \quad w_L(2) = 2.$$

The *Lee weight* $w_L(v)$ of a vector $v \in \mathbf{Z}_4^n$ is defined to be the sum (in $\mathbf{Z}$) of the Lee weights of its components. See [Wan97, p. 16].

`LeeWeight(a)`

The Lee weight of the element $a \in \mathbf{Z}_4$.

`LeeWeight(v)`

The Lee weight of the codeword v.

`LeeDistance(u, v)`

The Lee distance between the codewords u and v, where u and v belong to the same code C. This is defined to be the Lee weight of $(u - v)$.

`MinimumLeeWeight(C)`

`MinimumLeeDistance(C)`

The minimum Lee weight of the code C.

`LeeWeightDistribution(C)`

The Lee weight distribution of the code C.

`DualLeeWeightDistribution(C)`

The Lee weight distribution of the dual of the code C (see `LeeWeightDistribution`)

`WordsOfLeeWeight(C, w)`

`Cutoff` RngIntElt *Default :* ∞

Given a linear code C, return the set of all words of C having Lee weight w. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found.

`WordsOfBoundedLeeWeight(C, l, u)`

`Cutoff` RngIntElt *Default :* ∞

Given a linear code C, return the set of all words of C having Lee weight between l and u, inclusive. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found.

Example H155E19

We calculate the Lee weight distribution of a Reed Muller code over $\mathbf{Z}_4$ and enumerate all words of Lee weight 8.

```
> C := ReedMullerCodeZ4(1, 3);
> C;
(8, 256, 4) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2 1]
[0 1 0 0 2 1 1 3]
[0 0 1 0 1 1 3 2]
[0 0 0 1 3 2 3 3]
> LeeWeightDistribution(C);
[ <0, 1>, <6, 112>, <8, 30>, <10, 112>, <16, 1> ]
> W := WordsOfLeeWeight(C, 8);
```

```
> #W;
30
```

155.6.3 Euclidean Weight

For an element $x \in \mathbf{Z}_4$, the *Euclidean weight* $w_E(x)$ is defined by:

$$w_E(0) = 0, \quad w_E(1) = w_E(3) = 1, \quad w_E(2) = 4.$$

The *Euclidean weight* $w_E(v)$ of a vector $v \in \mathbf{Z}_4^n$ is defined to be the sum (in $\mathbf{Z}$) of the Euclidean weights of its components. See [Wan97, p. 16].

`EuclideanWeight(a)`

The Euclidean weight of the element $a \in \mathbf{Z}4$.

`EuclideanWeight(v)`

The Euclidean weight of the $\mathbf{Z}_4$-codeword v.

`EuclideanDistance(u, v)`

The Euclidean distance between the $\mathbf{Z}_4$-codewords u and v, where u and v belong to the same code C. This is defined to be the Euclidean weight of $(u - v)$.

`MinimumEuclideanWeight(C)`

`MinimumEuclideanDistance(C)`

The minimum Euclidean weight of the $\mathbf{Z}_4$-code C.

`EuclideanWeightDistribution(C)`

The Euclidean weight distribution of the $\mathbf{Z}_4$-code C.

`DualEuclideanWeightDistribution(C)`

The Euclidean weight distribution of the dual of the $\mathbf{Z}_4$-code C.

Example H155E20

We calculate the Euclidean weight distribution of quadratic residue code over $\mathbf{Z}_4$

```
> C := QRCodeZ4(17);
> C;
(17, 262144) Cyclic Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 0 0 0 0 0 1 1 3 0 3 0 3 1]
[0 1 0 0 0 0 0 0 0 3 0 2 3 1 3 1 2]
[0 0 1 0 0 0 0 0 0 2 1 2 2 1 1 1 3]
[0 0 0 1 0 0 0 0 0 1 3 0 2 1 1 0 2]
[0 0 0 0 1 0 0 0 0 2 3 1 0 0 1 3 2]
[0 0 0 0 0 1 0 0 0 2 0 1 1 2 0 3 1]
[0 0 0 0 0 0 1 0 0 3 1 1 1 2 2 1 2]
[0 0 0 0 0 0 0 1 0 2 1 3 1 3 2 0 3]
[0 0 0 0 0 0 0 0 1 1 3 0 3 0 3 1 1]
> EuclideanWeightDistribution(C);
[ <0, 1>, <7, 136>, <8, 170>, <9, 170>, <10, 408>, <11, 544>, <12, 986>, <13,
1768>, <14, 3128>, <15, 5032>, <16, 6120>, <17, 6360>, <18, 8432>, <19, 12512>,
<20, 12682>, <21, 11152>, <22, 14416>, <23, 17680>, <24, 16048>, <25, 15164>,
<26, 17952>, <27, 16864>, <28, 13328>, <29, 14144>, <30, 14144>, <31, 10064>,
<32, 7837>, <33, 8024>, <34, 6800>, <35, 4896>, <36, 3485>, <37, 2992>, <38,
2992>, <39, 1768>, <40, 510>, <41, 1258>, <42, 1224>, <44, 238>, <45, 408>, <46,
136>, <47, 136>, <48, 34>, <68, 1> ]
```

155.7 Weight Enumerators

`CompleteWeightEnumerator(C)`

Let C be a code over a finite ring R of cardinality q, and suppose that the elements of R are ordered in some way. Then for a codeword $v \in C$ and the i-th element $a \in R$, let $s_i(v)$ denote the number of components of v equal to a.

This function returns the complete weight enumerator $\mathcal{W}_C(X_0, X_1, \ldots, X_{q-1})$ of C, which is defined by:

$$\mathcal{W}_C(X_0, X_1, \ldots, X_{q-1}) = \sum_{v \in C} {X_0}^{s_0(v)} {X_1}^{s_1(v)} \cdots {X_{q-1}}^{s_{q-1}(v)}.$$

See [Wan97, p. 9] for more information. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with q variables. The angle-bracket notation may be used to assign names to the indeterminates.

`SymmetricWeightEnumerator(C)`

Suppose C is a $\mathbf{Z}_4$-code. This function returns the symmetric weight enumerator $\mathrm{swe}_C(X_0, X_1, X_2)$ of C, which is defined by:

$$\mathrm{swe}_C(X_0, X_1, X_2) = \mathcal{W}_C(X_0, X_1, X_2, X_1),$$

where $\mathcal{W}_C$ is the complete weight enumerator, defined above. See [Wan97, p. 14] for more information. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with three variables. The angle-bracket notation may be used to assign names to the indeterminates.

`WeightEnumerator(C)`

`HammingWeightEnumerator(C)`

Suppose C is a code over some finite ring R. This function returns the Hamming weight enumerator $\mathrm{Ham}_C(X, Y)$ of C, which is defined by:

$$\mathrm{Ham}_C(X, Y) = \sum_{v \in C} X^{n - w_H(v)} Y^{w_H(v)},$$

where $w_H(v)$ is the Hamming weight function. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with two variables. The angle-bracket notation may be used to assign names to the indeterminates.

`LeeWeightEnumerator(C)`

Suppose C is a $\mathbf{Z}_4$-code. This function returns the Lee weight enumerator $\mathrm{Lee}_C(X, Y)$ of C, which is defined by:

$$\mathrm{Lee}_C(X, Y) = \sum_{v \in C} X^{2*n - w_L(v)} Y^{w_L(v)},$$

where $w_L(v)$ is the Lee weight function, defined in Section 155.6.2. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with two variables. The angle-bracket notation may be used to assign names to the indeterminates.

`EuclideanWeightEnumerator(C)`

Suppose C is a $\mathbf{Z}_4$-code. This function returns the Euclidean weight enumerator $\mathrm{Euclidean}_C(X, Y)$ of C, which is defined by:

$$\mathrm{Euclidean}_C(X, Y) = \sum_{v \in C} X^{4*n - w_E(v)} Y^{w_E(v)},$$

where $w_E(v)$ is the Euclidean weight function, defined in Section 155.6.3. The result will lie in a global multivariate polynomial ring over $\mathbf{Z}$ with two variables. The angle-bracket notation may be used to assign names to the indeterminates.

Example H155E21

We compute the complete weight enumerator of a cyclic code over the Galois ring GR$(4, 2)$.

```
> R<w> := GR(4,2);
> P<x> := PolynomialRing(R);
> L := CyclotomicFactors(R, 3);
> g := L[1];
> g;
x + 3
> C := CyclicCode(3, g);
> C;
(3, 256, 2) Cyclic Code over GaloisRing(2, 2, 2)
Generator matrix:
[1 0 3]
[0 1 3]
> CWE<[X]> := CompleteWeightEnumerator(C);
> CWE;
X[1]^3 + 6*X[1]*X[2]*X[4] + 3*X[1]*X[3]^2 + 6*X[1]*X[5]*X[13] +
    6*X[1]*X[6]*X[16] + 6*X[1]*X[7]*X[15] + 6*X[1]*X[8]*X[14] +
    3*X[1]*X[9]^2 + 6*X[1]*X[10]*X[12] + 3*X[1]*X[11]^2 +
    3*X[2]^2*X[3] + 6*X[2]*X[5]*X[16] + 6*X[2]*X[6]*X[15] +
    6*X[2]*X[7]*X[14] + 6*X[2]*X[8]*X[13] + 6*X[2]*X[9]*X[12] +
    6*X[2]*X[10]*X[11] + 3*X[3]*X[4]^2 + 6*X[3]*X[5]*X[15] +
    6*X[3]*X[6]*X[14] + 6*X[3]*X[7]*X[13] + 6*X[3]*X[8]*X[16] +
    6*X[3]*X[9]*X[11] + 3*X[3]*X[10]^2 + 3*X[3]*X[12]^2 +
    6*X[4]*X[5]*X[14] + 6*X[4]*X[6]*X[13] + 6*X[4]*X[7]*X[16] +
    6*X[4]*X[8]*X[15] + 6*X[4]*X[9]*X[10] + 6*X[4]*X[11]*X[12] +
    3*X[5]^2*X[9] + 6*X[5]*X[6]*X[12] + 6*X[5]*X[7]*X[11] +
    6*X[5]*X[8]*X[10] + 3*X[6]^2*X[11] + 6*X[6]*X[7]*X[10] +
    6*X[6]*X[8]*X[9] + 3*X[7]^2*X[9] + 6*X[7]*X[8]*X[12] +
    3*X[8]^2*X[11] + 3*X[9]*X[13]^2 + 6*X[9]*X[14]*X[16] +
    3*X[9]*X[15]^2 + 6*X[10]*X[13]*X[16] + 6*X[10]*X[14]*X[15] +
    6*X[11]*X[13]*X[15] + 3*X[11]*X[14]^2 + 3*X[11]*X[16]^2 +
    6*X[12]*X[13]*X[14] + 6*X[12]*X[15]*X[16]
```

Example H155E22

We compute the various weight enumerators of the octacode. To ensure the polynomials print out nicely, we assign names to the polynomial ring indeterminates in each case. These names will persist if further calls to these functions (over $\mathbf{Z}_4$) are made.

```
> Z4 := IntegerRing(4);
> O8 := LinearCode<Z4, 8 |
>     [1,0,0,0,3,1,2,1],
>     [0,1,0,0,1,2,3,1],
>     [0,0,1,0,3,3,3,2],
>     [0,0,0,1,2,3,1,1]>;
> #O8;
```

```
256
> CWE<X0,X1,X2,X3> := CompleteWeightEnumerator(O8);
> CWE;
X0^8 + 14*X0^4*X2^4 + 56*X0^3*X1^3*X2*X3 + 56*X0^3*X1*X2*X3^3 +
    56*X0*X1^3*X2^3*X3 + 56*X0*X1*X2^3*X3^3 + X1^8 + 14*X1^4*X3^4 +
    X2^8 + X3^8
> SWE<X0,X1,X2> := SymmetricWeightEnumerator(O8);
> SWE;
X0^8 + 14*X0^4*X2^4 + 112*X0^3*X1^4*X2 + 112*X0*X1^4*X2^3 + 16*X1^8 +
    X2^8
> HWE<X,Y> := HammingWeightEnumerator(O8);
> HWE;
X^8 + 14*X^4*Y^4 + 112*X^3*Y^5 + 112*X*Y^7 + 17*Y^8
> LeeWeightEnumerator(O8);
X^16 + 112*X^10*Y^6 + 30*X^8*Y^8 + 112*X^6*Y^10 + Y^16
> EuclideanWeightEnumerator(O8);
X^32 + 128*X^24*Y^8 + 126*X^16*Y^16 + Y^32
```

155.8 Constructing New Codes from Old

The operations described here produce a new code by modifying in some way the code words of a given code.

155.8.1 Sum, Intersection and Dual

For the following operators, C and D are codes defined as subsets (or subspaces) of the same R-space V.

`C + D`

The (vector space) sum of the linear codes C and D, where C and D are contained in the same R-space V.

`C meet D`

The intersection of the linear codes C and D, where C and D are contained in the same R-space V.

`Dual(C)`

The dual D of the linear code C. The dual consists of all codewords in the R-space V which are orthogonal to all codewords of C.

Example H155E23

Verify some simple results from the sum and intersection of subcodes.

```
> R<w> := GR(9,2);
> P<x> := PolynomialRing(R);
> g := x^2 + 7*w*x + 1;
> C := CyclicCode(5, g);
> C;
(5, 43046721) Cyclic Code over GaloisRing(3, 2, 2)
Generator matrix:
[  1   0   0   1   w]
[  0   1   0 2*w 2*w]
[  0   0   1   w   1]
[  0   0   0   3   0]
[  0   0   0   0   3]
>
> C1 := sub< C | C.1 >;
> C1;
(5, 81, 3) Linear Code over GaloisRing(3, 2, 2)
Generator matrix:
[1 0 0 1 w]
> C2 := sub< C | C.4 >;
> C2;
(5, 9, 1) Linear Code over GaloisRing(3, 2, 2)
Generator matrix:
[0 0 0 3 0]
> C3 := sub< C | { C.1 , C.4} >;
> C3;
(5, 729, 1) Linear Code over GaloisRing(3, 2, 2)
Generator matrix:
[1 0 0 1 w]
[0 0 0 3 0]
> (C1 + C2) eq C3;
true
> (C1 meet C3) eq C1;
true
```

155.8.2 Standard Constructions

`DirectSum(C, D)`

Given a length n_1 code C and a length n_2 code D, both over the same ring R, construct the direct sum of C and D. The direct sum consists of all length $n_1 + n_2$ vectors $u|v$, where $u \in C$ and $v \in D$.

`DirectProduct(C, D)`

Given a length n_1 code C and a length n_2 code D, both over the same ring R, construct the direct product of C and D. The direct product has length $n_1 \cdot n_2$ and its generator matrix is the Kronecker product of the basis matrices of C and D.

`C1 cat C2`

Given codes $C1$ and $C2$, both defined over the same ring R, return the concatenation C of $C1$ and $C2$. If A and B are the generator matrices of $C1$ and $C2$, respectively, the concatenation of $C1$ and $C2$ is the code with generator matrix whose rows consist of each row of A concatenated with each row of B.

`ExtendCode(C)`

Given a length n code C form a new code C' from C by adding the appropriate extra coordinate to each vector of C such that the sum of the coordinates of the extended vector is zero.

`ExtendCode(C, n)`

Return the code C extended n times.

`PadCode(C, n)`

Add n zeros to the end of each codeword of C.

`PlotkinSum(C, D)`

Given codes C and D both over the same ring R and of the same length n, construct the Plotkin sum of C and D. The Plotkin sum consists of all vectors $u|u+v$, $u \in C$ and $v \in D$.

`PunctureCode(C, i)`

Given an length n code C, and an integer i, $1 \leq i \leq n$, construct a new code C' by deleting the i-th coordinate from each code word of C.

`PunctureCode(C, S)`

Given a length n code C and a set S of distinct integers $\{i_1, \cdots, \mathrm{i}_r\}$ each of which lies in the range $[1, n]$, construct a new code C' by deleting the components $i_1, \cdots, i_r$ from each code word of C.

`ShortenCode(C, i)`

Given a length n code C and an integer i, $1 \leq i \leq n$, construct a new code from C by selecting only those codewords of C having a zero as their i-th component and deleting the i-th component from these codewords. Thus, the resulting code will have length $n - 1$.

`ShortenCode(C, S)`

Given a length n code C and a set S of distinct integers $\{i_1, \cdots, \mathrm{i}_r\}$, each of which lies in the range $[1, n]$, construct a new code from C by selecting only those codewords of C having zeros in each of the coordinate positions $i_1, \cdots, i_r$, and deleting these components. Thus, the resulting code will have length $n - r$.

Example H155E24

We combine codes in various ways and look at the length of the new code.

```
> R<w> := GR(8,2);
> C1 := RandomLinearCode(R, 4, 2);
> C2 := RandomLinearCode(R, 5, 3);
> Length(C1);
4
> Length(C2);
5
> C3 := DirectSum(C1, C2);
> Length(C3);
9
> C4 := DirectProduct(C1, C2);
> Length(C4);
20
> C5 := C1 cat C2;
> Length(C5);
9
```

Example H155E25

We note that, in general, puncturing a code over $\mathbf{Z}_4$ reduces the minimum Lee distance by 2.

```
> C := PreparataCode(3);
> C;
(8, 256, 4) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2 1]
[0 1 0 0 2 1 1 3]
[0 0 1 0 1 1 3 2]
[0 0 0 1 3 2 3 3]
> MinimumLeeWeight(C);
6
> C1 := PunctureCode(C,8);
> C1;
(7, 256, 3) Linear Code over IntegerRing(4)
Generator matrix:
[1 0 0 0 3 1 2]
[0 1 0 0 2 1 1]
[0 0 1 0 1 1 3]
[0 0 0 1 3 2 3]
> MinimumLeeWeight(C1);
4
```

155.9 Operations on Codewords

155.9.1 Construction of a Codeword

`C ! [a1, ..., an]`

`elt< C | a1, ..., an >`

Given a code C which is defined as a subset of the R-space $R^{(n)}$, and elements $a_1, \ldots, a_n$ belonging to R, construct the codeword $(a_1, \ldots, a_n)$ of C. It is checked that the vector $(a_1, \ldots, a_n)$ is an element of C.

`C ! u`

Given a code C which is defined as a subset of the R-space $V = R^{(n)}$, and an element u belonging to V, create the codeword of C corresponding to u. The function will fail if u does not belong to C.

`C ! 0`

The zero word of the code C.

Example H155E26

We create some elements of a code over a finite ring.

```
> R<w> := GR(16,2);
> P<x> := PolynomialRing(R);
> L := CyclotomicFactors(R, 7);
> C := CyclicCode(7, L[2]);
> C ! [1, 2*w, 0, w+3, 7*w, 12*w+3, w+3];
(       1      2*w        0    w + 3      7*w 12*w + 3    w + 3)
> elt< C | 0, 3, 0, 2*w + 5, 6*w + 9, 4*w + 5, 14*w + 14 >;
(        0         3         0   2*w + 5   6*w + 9   4*w + 5 14*w + 14)
```

If the given vector does not lie in the given code then an error will result.

```
>  C ! [0,0,0,0,0,0,1];
>> C ! [0,0,0,0,0,0,1];
     ^
Runtime error in '!': Result is not in the given structure
>  elt< C | 1, 0, 1, 0, 1, 0, 1>;
>> elt< C | 1, 0, 1, 0, 1, 0, 1>;
      ^
Runtime error in elt< ... >: Result is not in the lhs of the constructor
```

155.9.2 Operations on Codewords and Vectors

`u + v`

Sum of the codewords u and v, where u and v belong to the same linear code C.

`-u`

Additive inverse of the codeword u belonging to the linear code C.

`u - v`

Difference of the codewords u and v, where u and v belong to the same linear code C.

`a * u`

Given an element a belonging to the ring R, and a codeword u belonging to the linear code C, return the codeword $a * u$.

`Weight(v)`

The Hamming weight of the codeword v, i.e., the number of non-zero components of v.

`Distance(u, v)`

The Hamming distance between the codewords u and v, where u and v belong to the same code C.

`Support(w)`

Given a word w belonging to the length n code C, return its support as a subset of the integer set $\{1..n\}$. The support of w consists of the coordinates at which w has non-zero entries.

`(u, v)`

`InnerProduct(u, v)`

Inner product of the vectors u and v with respect to the Euclidean norm, where u and v belong to the parent vector space of the code C.

`Coordinates(C, u)`

Given a length n linear code C and a codeword u of C return the coordinates of u with respect to C. The coordinates of u are returned as a sequence $Q = [a_1, \ldots, a_k]$ of elements from the alphabet of C so that $u = a_1 * C.1 + \ldots + a_k * C.k$.

`Normalize(u)`

Given an element u of a code defined over the ring R, return the normalization of u, which is the unique vector v such that $v = a \cdot u$ for some scalar $a \in R$ such that the first non-zero entry of v is the canonical associate in R of the first non-zero entry of u (v is zero if u is zero).

`Rotate(u, k)`

Given a vector u, return the vector obtained from u by cyclically shifting its components to the right by k coordinate positions.

`Rotate(~u, k)`

Given a vector u, destructively rotate u by k coordinate positions.

`Parent(w)`

Given a word w belonging to the code C, return the ambient space V of C.

Example H155E27

Given a code over a finite ring, we explore various operations on its code words.

```
> R<w> := GR(4, 4);
> P<x> := PolynomialRing(R);
> g := x + 2*w^3 + 3*w^2 + w + 2;
> C := CyclicCode(3, g);
> C;
(3, 1048576) Cyclic Code over GaloisRing(2, 2, 4)
Generator matrix:
[          1           0     w^2 + w]
[          0           1 w^2 + w + 1]
[          0           0           2]
> u := C.1;
> v := C.2;
> u;
(      1       0 w^2 + w)
> v;
(          0           1 w^2 + w + 1)
> u + v;
(              1               1 2*w^2 + 2*w + 1)
> 2*u;
(          2           0 2*w^2 + 2*w)
> 4*u;
(0 0 0)
> Weight(u);
2
> Support(u);
{ 1, 3 }
```

155.9.3 Accessing Components of a Codeword

`u[i]`

Given a codeword u belonging to the code C defined over the ring R, return the i-th component of u (as an element of R).

`u[i] := x;`

Given an element u belonging to a subcode C of the full R-space $V = R^n$, a positive integer i, $1 \leq i \leq n$, and an element x of R, this function returns a vector in V which is u with its i-th component redefined to be x.

155.10 Boolean Predicates

For the following operators, C and D are codes defined as a subset (or subspace) of the R-space V.

`u in C`

Return `true` if and only if the vector u of V belongs to the code C.

`u notin C`

Return `true` if and only if the vector u of V does not belong to the code C.

`C subset D`

Return `true` if and only if the code C is a subcode of the code D.

`C notsubset D`

Return `true` if and only if the code C is not a subcode of the code D.

`C eq D`

Return `true` if and only if the codes C and D are equal.

`C ne D`

Return `true` if and only if the codes C and D are not equal.

`IsCyclic(C)`

Return `true` if and only if the linear code C is a cyclic code.

`IsSelfDual(C)`

Return `true` if and only if the linear code C is self-dual (or self-orthogonal) (i.e., C equals the dual of C).

`IsSelfOrthogonal(C)`

Return `true` if and only if the linear code C is self-orthogonal; that is, return whether C is contained in the dual of C.

`IsProjective(C)`

Returns `true` if and only if the (non-quantum) code C is projective.

`IsZero(u)`

Return `true` if and only if the codeword u is the zero vector.

Example H155E28

We consider an [8, 7] linear code K_8 over Z_4 and examine some of its properties.

```
> Z4 := IntegerRing(4);
> K8 := LinearCode< Z4, 8 |
>     [1,1,1,1,1,1,1,1],
>     [0,2,0,0,0,0,0,2],
>     [0,0,2,0,0,0,0,2],
>     [0,0,0,2,0,0,0,2],
>     [0,0,0,0,2,0,0,2],
>     [0,0,0,0,0,2,0,2],
>     [0,0,0,0,0,0,2,2]>;
> K8;
[8, 7, 2] Linear Code over IntegerRing(4)
Generator matrix:
[1 1 1 1 1 1 1 1]
[0 2 0 0 0 0 0 2]
[0 0 2 0 0 0 0 2]
[0 0 0 2 0 0 0 2]
[0 0 0 0 2 0 0 2]
[0 0 0 0 0 2 0 2]
[0 0 0 0 0 0 2 2]
> IsCyclic(K8);
true
> IsSelfDual(K8);
true
> K8 eq Dual(K8);
true
```

155.11 Bibliography

[**Wan97**] Zhe-Xian Wan. *Quaternary Codes*, volume 8 of *Series on Applied Mathematics*. World Scientific, Singapore, 1997.

156 ADDITIVE CODES

Chapter 156
ADDITIVE CODES

156.1 Introduction

The concept of a linear codes over a finite field (see Chapter 152) can be generalized to the notion of an *additive* code. Given a finite field F and the space of all n-tuples of F, an additive code is a subset of $F^{(n)}$ which is a K-linear subspace for some subfield $K \subseteq F$.

Additive codes have become increasingly important recently due to their application to the construction of quantum error-correcting codes, though they are also of interest in their own right. A MAGMA package for quantum error-correcting codes is built on the machinery for additive codes.

In MAGMA an additive code has both an *alphabet* F and a *coefficient field* K, which is a subfield of F. An error-correcting code is considered to be defined by its wordset, so there may be several different ways of presenting a given code using different coefficient fields.

Since a given code may be presented over different coefficient rings, the *dimension* k of an additive code is defined relative to the alphabet of the code, $\#C = (\#F)^k$, leading to possibility of *fractional* dimensions. Consequently, the number of generators of an additive code will not equal its dimension, there being $[F : K]$ times as many generators. So a length n K-additive code over F has between zero and $n * [F : K]$ generators.

For example, consider the two length 3 vectors over F_4: $(1, 0, \omega^2), (0, \omega, 0)$. The linear code generated by these vectors consists of all scalar multiples and sums, resulting in a total of $4^2 = 16$ vectors. But the F_2-additive code generated by these two vectors contains *only their sums*, resulting in a total of $2^2 = 4$ vectors. These vectors are $(0, 0, 0), (1, 0, \omega^2), (0, \omega, 0), (1, \omega, \omega^2)$. The alphabet of this code is F_4, its coefficient field is F_2, it has 2 generators and is of dimension 1.

A length n, dimension k K-additive code over F with k_g generators is represented in MAGMA as an $[n, k : k_g]$ K-additive code over F. The concepts of weight, distance and their respective distributions and enumerators transfer directly from linear codes. An $[n, k : k_g, d]$ K-additive code over F is a K-linear subset of $F^{(n)}$ which has fractional dimension k, k_g generators and a minimum weight of d.

As a general rule, additive and linear codes may be used interchangeably. Indeed any linear code can be expressed as an additive code, using either its alphabet or any subfield as its coefficient field. So any linear code over a finite field, of type `CodeLinFld`, is in fact also an additive code, of type `CodeAdd`. The theory of purely linear codes is more general than that of additive codes so unfortunately not all operations are transferable.

156.2 Construction of Additive Codes

156.2.1 Construction of General Additive Codes

```
AdditiveCode< F, K, n | L >
```

Create the K-additive code in $F^{(n)}$ of length n which is generated by the elements specified by the list L, where K is a subfield of F and L is one or more items of the following types:

(a) An element of $F^{(n)}$;

(b) A set or sequence of elements of $F^{(n)}$;

(c) A sequence of n elements of F, defining an element of $F^{(n)}$;

(d) A set or sequence of sequences of type (c);

(e) A subcode of $F^{(n)}$;

```
AdditiveCode(G)
```

```
AdditiveCode(K, G)
```

Given a matrix G over a field F and a subfield K of F, return the K-additive code over F generated by the rows of G. If no coefficient field K is specified, then the prime field of F is used.

Example H156E1

Starting with two linearly independent vectors in $\mathbf{F}_4^{(3)}$, we compare the linear code over F_4 they generate with the corresponding F_2-additive code.

```
> F<w> := GF(4);
> G := Matrix(F, 2, 3, [1,0,w^2,0,w,0]);
> G;
[  1   0 w^2]
[  0   w   0]
> C1 := LinearCode(G);
> C2 := AdditiveCode(GF(2), G);
> #C1;
16
> #C2;
4
> C2 subset C1;
true
```

The codewords of C_2 are arise only through addition of the generators: scalar multiplication is not permitted.

```
> { v : v in C2 };
{
    (  1   w w^2),
    (  0   0   0),
    (  1   0 w^2),
```

```
    (  0   w   0)
}
```

Example H156E2

We define an $\mathbf{F}_2$-additive code over $\mathbf{F}_8$ by constructing a random matrix and considering the code generated by its rows. Note that the number of generators exceeds the length of the code.

```
> K<w> := GF(8);
> M := KMatrixSpace(K, 5, 4);
> C := AdditiveCode(GF(2), Random(M));
> C;
[4, 1 2/3 : 5] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   1 w^2   0]
[  w w^2   w   1]
[w^2 w^2 w^2   1]
[  0 w^4 w^4 w^5]
[  0   0   1   0]
> WeightDistribution(C);
[ <0, 1>, <1, 1>, <2, 2>, <3, 9>, <4, 19> ]
> C;
[4, 1 2/3 : 5, 1] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   1 w^2   0]
[  w w^2   w   1]
[w^2 w^2 w^2   1]
[  0 w^4 w^4 w^5]
[  0   0   1   0]
```

`AdditiveCode(K, C)`

Given a code (linear or additive) C over some finite field F, and a subfield K of F such that the wordset of C forms a K-linear subspace, then return C as a K-additive code.

Example H156E3

Any linear code can be regarded as an additive code with respect to a subfield of its alphabet.

```
> C := RandomLinearCode(GF(4), 8, 3);
> C:Minimal;
[8, 3, 4] Linear Code over GF(2^2)
> A1 := AdditiveCode(GF(4), C);
> A1:Minimal;
[8, 3 : 3, 4] GF(2^2)-Additive Code over GF(2^2)
> { v : v in C } eq {v : v in A1 };
true
```

```
>
> A2 := AdditiveCode(GF(2), C);
> A2:Minimal;
[8, 3 : 6, 4] GF(2)-Additive Code over GF(2^2)
> { v : v in C } eq {v : v in A2 };
true
```

Example H156E4

A K-additive code over F can be viewed as an E-additive code for any subfield $E \subseteq K$.

```
> C4 := RandomAdditiveCode(GF(16), GF(4), 8, 5);
> C4:Minimal;
[8, 2 1/2 : 5] GF(2^2)-Additive Code over GF(2^4)
>
> C2 := AdditiveCode(GF(2), C4);
> C2:Minimal;
[8, 2 1/2 : 10] GF(2)-Additive Code over GF(2^4)
> { v : v in C2 } eq {v : v in C4 };
true
```

But for any E such that $K \subset E \subseteq F$ we can create an E-additive code if and only if the wordset is in fact an E-linear subspace.

```
> C2:Minimal;
[8, 2 1/2 : 10] GF(2)-Additive Code over GF(2^4)
> A1 := AdditiveCode(GF(4), C2);
> A1 eq C4;
true
>  A2 := AdditiveCode(GF(16), C2);
>> A2 := AdditiveCode(GF(16), C2);
                      ^
Runtime error in 'AdditiveCode': Code is not additive over given field
```

156.2.2 Some Trivial Additive Codes

`AdditiveZeroCode(F, K, n)`

Given a field F and subfield $K \subseteq F$ along with a positive integer n, return the $[n, 0, n]$ code consisting of only the zero code word, (where the minimum weight is by convention equal to n).

`AdditiveRepetitionCode(F, K, n)`

Given a field F and subfield $K \subseteq F$ along with a positive integer n, return the $[n, 1, n]$ code consisting of all repeating codewords.

`AdditiveZeroSumCode(F, K, n)`

Given a field F and subfield $K \subseteq F$ along with a positive integer n, return the $[n, n-1, 2]$ K-additive code over F such that for all codewords $(c_1, c_2, \ldots, c_n)$, we have $\sum_i c_i = 0$.

`AdditiveUniverseCode(F, K, n)`

Given a field F and subfield $K \subseteq F$ along with a positive integer n, return the $[n, n, 1]$ K-additive code over F consisting of all possible codewords.

`RandomAdditiveCode(F, K, n, k)`

Given a field F and subfield $K \subseteq F$ along with positive integers n and k, such that $0 < k \leq n * [F : K]$, and k, return a random K-additive code of length n and k generators over the field F.

Example H156E5

Over any finite field chain $K \subseteq F$, the zero code of length n is contained in every code of length n, and similarly every code of length n is contained in the universe code of length n.

```
> F := GF(9);
> K := GF(3);
> U := AdditiveUniverseCode(F, K, 5);
> Z := AdditiveZeroCode(F, K, 5);
> R := RandomAdditiveCode(F, K, 5, 2);
> (Z subset R) and (R subset U);
true
```

156.3 Invariants of an Additive Code

156.3.1 The Ambient Space and Alphabet

A length n additive code has an alphabet F and coefficient field $K \subseteq F$. The code consists of codewords which are a K-linear subspace of $F^{(n)}$.

`Alphabet(C)`

`Field(C)`

The underlying field (or alphabet) of the codewords of the additive code C. A length n additive code with alphabet F consists of codewords from $F^{(n)}$.

`CoefficientField(C)`

The field over which the codewords of the additive code C are considered linear. This will be a subfield of the alphabet of C.

`AmbientSpace(C)`

The ambient space of the additive code C, i.e. the generic R-space V in which C is contained.

`Generic(C)`

Given a length n additive code C, return the generic $[n, n, 1]$ code in which C is contained.

Example H156E6

A code can often be represented using several different coefficient fields.

```
> F<w> := GF(5,4);
> K := GF(5,2);
> C := RandomAdditiveCode(F, K, 12, 5);
> C:Minimal;
[12, 2 1/2 : 5] GF(5^2)-Additive Code over GF(5^4)
> #C;
9765625
> Alphabet(C);
Finite field of size 5^4
> CoefficientField(C);
Finite field of size 5^2
>
> C1 := AdditiveCode(GF(5), C);
> C1:Minimal;
[12, 2 1/2 : 10] GF(5)-Additive Code over GF(5^4)
> #C1;
9765625
> Alphabet(C1);
Finite field of size 5^4
> CoefficientField(C1);
Finite field of size 5
```

156.3.2 Basic Numerical Invariants

`Length(C)`

Return the block length n of an additive code C.

`Dimension(C)`

The (rational) dimension k of C. If the alphabet of C is F, then the dimension is defined by the equation $\#C = (\#F)^k$.

Note that since any basis of the additive code C is relative to the coefficient field K, this dimension is not necessarily equal to the number of generators of C and is not even necessarily integral.

`NumberOfGenerators(C)`

`Ngens(C)`

The number of generators of the additive code C. Note that if the coefficient ring of C is not the same as its alphabet then this will be different from the dimension of C.

`#C`

Given an additive code C, return the number of codewords belonging to C.

`InformationRate(C)`

The information rate of the $[n, k]$ code C. This is the ratio k/n.

156.3.3 The Code Space

`GeneratorMatrix(C)`

`BasisMatrix(C)`

The generator matrix for an $[n, k(k_g)]$ K-additive code C over F is a $k_g \times n$ matrix over F, whose k_g rows form a basis for C when considered as vectors over K.

`Basis(C)`

`Generators(C)`

A basis for the K-additive code C, returned as a sequence of codewords over the alphabet of C, which generate the code over K.

`C . i`

Given an $[n, k(k_g)]$ K-additive code C and a positive integer i, $1 \leq i \leq k_g$, return the i-th element of the current basis of C over K.

156.3.4 The Dual Space

`Dual(C)`

The code that is dual to the code C. For an additive code C, this is the nullspace with respect to the trace inner product of the coefficient field.

`ParityCheckMatrix(C)`

The parity check matrix for the code C, returned as an element of $\mathrm{Hom}(V, U)$.

156.4 Operations on Codewords

156.4.1 Construction of a Codeword

`C ! [a1, ..., an]`

`elt< C | a1, ..., an >`

Given a length n additive code C with alphabet F, then the codewords of C lie in $F^{(n)}$. Given elements $a_1, \ldots, a_n$ belonging to F, construct the codeword $(a_1, \ldots, a_n)$ of C. A check is made that the vector $(a_1, \ldots, a_n)$ is an element of C.

`C ! u`

Given an additive code C which is defined as a subset of the F-space $V = F^{(n)}$, and an element u belonging to V, create the codeword of C corresponding to u. The function will fail if u does not belong to C.

`C ! 0`

The zero word of the additive code C.

`Random(C)`

A random codeword of the additive code C.

156.4.2 Arithmetic Operations on Codewords

`u + v`

Sum of the codewords u and v, where u and v belong to the same linear code C.

`-u`

Additive inverse of the codeword u belonging to the linear code C.

`u - v`

Difference of the codewords u and v, where u and v belong to the same linear code C.

`a * u`

Given an element a belonging to the alphabet F, and a codeword u belonging to the additive code C, return the codeword $a * u$.

`Normalize(u)`

Normalize a codeword u of an additive code C, returning a scalar multiple of u such that its first non-zero entry is 1.

156.4.3 Distance and Weight

`Distance(u, v)`

The Hamming distance between the codewords u and v, where u and v belong to the same additive code C.

`Weight(u)`

The Hamming weight of the codeword u, i.e., the number of non-zero components of u.

156.4.4 Vector Space and Related Operations

`(u, v)`

`InnerProduct(u, v)`

Inner product of the vectors u and v with respect to the Euclidean norm, where u and v belong to the parent vector space of the code C.

`TraceInnerProduct(K, u, v)`

Given vectors u and v defined over a finite field L and a subfield K of L, this function returns the trace of the inner product of the vectors u and v with respect to K.

`Support(w)`

Given a word w belonging to the $[n, k]$ code C, return its support as a subset of the integer set $\{1..n\}$. The support of w consists of the coordinates at which w has non-zero entries.

`Coordinates(C, u)`

Given an $[n, k : k_g]$ K-additive code C and a codeword u of C return the coordinates of u with respect to the current basis of C. The coordinates of u are returned as a sequence $Q = [a_1, \ldots, a_{k_g}]$ of elements from K such that $u = a_1 * C.1 + \ldots + a_{k_g} * C.k_g$.

`Parent(w)`

Given a word w belonging to the code C, return the ambient space V of C.

`Rotate(u, k)`

Given a vector u, return the vector obtained from u by cyclically shifting its components to the right by k coordinate positions.

`Rotate(~u, k)`

Given a vector u, destructively rotate u by k coordinate positions.

`Trace(u, S)`

`Trace(u)`

Given a vector u with components in K, and a subfield S of K, construct the vector with components in S obtained from u by taking the trace of each component with respect to S. If S is omitted, it is taken to be the prime field of K.

156.4.5 Predicates for Codewords

`u eq v`

The function returns `true` if and only if the codewords u and v belonging to the same additive code are equal.

`u ne v`

The function returns `true` if and only if the codewords u and v belonging to the same additive code are not equal.

`IsZero(u)`

The function returns `true` if and only if the codeword u is the zero vector.

156.4.6 Accessing Components of a Codeword

`u[i]`

Given a codeword u belonging to the code C defined over the ring R, return the i-th component of u (as an element of R).

`u[i] := x;`

Given an element u belonging to a subcode C of the full R-space $V = R^n$, a positive integer i, $1 \leq i \leq n$, and an element x of R, this function returns a vector in V which is u with its i-th component redefined to be x.

156.5 Subcodes

156.5.1 The Subcode Constructor

`sub< C | L >`

Given a K-additive linear code C over F, construct the subcode of C, generated (over K) by the elements specified by the list L, where L is a list of one or more items of the following types:

(a) An element of C;

(b) A set or sequence of elements of C;

(c) A sequence of n elements of F, defining an element of C;

(d) A set or sequence of sequences of type (c);

(e) A subcode of C;

`Subcode(C, k)`

Given an additive code C and an integer k, where k is less than the number of generators of C, then return a subcode of C with k generators.

`Subcode(C, S)`

Suppose C is an additive code and S is a set of positive integers, each of which is less than the number of generators of C. The function returns the subcode of C generated by the generators of C indexed by S.

`SubcodeBetweenCode(C1, C2, k)`

Given an additive code C_1 and a subcode C_2 of C_1, return a subcode of C_1 with k generators containing C_2.

`SubcodeWordsOfWeight(C, w)`

Given a length n additive code C and an integer which lies in the range $[1, n]$, return the subcode of C generated by those words of C of weight w.

`SubcodeWordsOfWeight(C, S)`

Given a length n additive code C and a set S of integers, each of which lies in the range $[1, n]$, return the subcode of C generated by those words of C whose weights lie in S.

Example H156E7

We give an example of how `SubcodeBetweenCode` may be used to create a code nested in between a subcode pair.

```
> F<w> := GF(8);
> C1 := AdditiveRepetitionCode(F, GF(2), 6);
> C1;
[6, 1 : 3, 6] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   1   1   1   1   1]
[  w   w   w   w   w   w]
[w^2 w^2 w^2 w^2 w^2 w^2]
> C3 := AdditiveZeroSumCode(F, GF(2), 6);
> C3;
[6, 5 : 15, 2] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   0   0   0   0   1]
[  w   0   0   0   0   w]
[w^2   0   0   0   0 w^2]
[  0   1   0   0   0   1]
[  0   w   0   0   0   w]
[  0 w^2   0   0   0 w^2]
[  0   0   1   0   0   1]
[  0   0   w   0   0   w]
[  0   0 w^2   0   0 w^2]
[  0   0   0   1   0   1]
[  0   0   0   w   0   w]
[  0   0   0 w^2   0 w^2]
```

```
[  0   0   0   0   1   1]
[  0   0   0   0   w   w]
[  0   0   0   0 w^2 w^2]
> C1 subset C3;
true
> C2 := SubcodeBetweenCode(C3, C1, 11);
> C2;
[6, 3 2/3 : 11] GF(2)-Additive Code over GF(2^3)
Generator matrix:
[  1   0   0   0   1   0]
[  w   0   0   0   w   0]
[w^2   0   0 w^2 w^2 w^2]
[  0   1   0   0   0   1]
[  0   w   0   0   0   w]
[  0 w^2   0   0   0 w^2]
[  0   0   1   0   0   1]
[  0   0   w   0   0   w]
[  0   0 w^2   0   0 w^2]
[  0   0   0   1   0   1]
[  0   0   0   w   0   w]
> (C1 subset C2) and (C2 subset C3);
true
```

156.5.2 Sum, Intersection and Dual

For the following operators, C and D are additive codes defined as subsets (or subspaces) of the same R-space F^n.

`C + D`

Given two additive codes which have the same length, which are defined over the same alphabet, and which have the same coefficient ring F, return the sum of these two codes with respect to F.

`C meet D`

The intersection of the additive codes C and D.

`Dual(C)`

The code that is dual to the code C. For an additive code C, this is the code generated by the nullspace of C, relative to the trace inner product.

156.5.3 Membership and Equality

`u in C`

Return `true` if and only if the vector u of V belongs to the additive code C, where V is the generic vector space containing C.

`u notin C`

Return `true` if and only if the vector u does not belong to the additive code C, where V is the generic vector space containing C.

`C subset D`

Return `true` if and only if the wordset of the code C is a subset of the wordset of the code D. (Either code may possibly be additive).

`C notsubset D`

Return `true` if and only if the wordset of the code C is not a subset of the wordset of the code D. (Either code may possibly be additive).

`C eq D`

Return `true` if and only if the codes C and D have the same wordsets. (Either code may possibly be additive).

`C ne D`

Return `true` if and only if the codes C and D have different wordsets. (Either code may possibly be additive).

156.6 Properties of Codes

For the following operators, C and D are codes defined as a subset (or subspace) of the vector space V.

`IsSelfDual(C)`

Return `true` if and only if the linear code C is self-dual (or self-orthogonal) (i.e. C equals the dual of C).

`IsSelfOrthogonal(C)`

Return `true` if and only if the linear code C is self-orthogonal (i.e. C is contained in the dual of C).

`IsPerfect(C)`

Return `true` if and only if the linear code C is perfect; that is, if and only if the cardinality of C is equal to the size of the sphere packing bound of C.

`IsProjective(C)`

Returns `true` if and only if the (non-quantum) code C is projective over its alphabet.

`IsAdditiveProjective(C)`

Returns `true` if and only if the additive code C is projective over its coefficient field. It is possible that some of the columns may not be independent with respect to the alphabet of the code.

156.7 The Weight Distribution

156.7.1 The Minimum Weight

An adaptation of the minimum weight algorithm for linear codes (see Section 152.8.1) has been developed for additive codes by Markus Grassl and Greg White.

From a user's perspective, the description given in 152.8.1 is sufficient to understand the additive case. The algorithm is still new, and has yet to be optimised to its full potential.

`MinimumWeight(C: parameters)`

`MinimumDistance(C: parameters)`

`RankLowerBound`	RngIntElt	*Default :* 0
`MaximumTime`	RngReSubElt	*Default :* ∞

Determine the minimum weight of the words belonging to the code C, which is also the minimum distance between any two codewords. The parameter `RankLowerBound` sets a minimum rank on the informations sets used in the calculation, while the parameter `MaximumTime` sets a time limit (in seconds of "user time") after which the calculation is aborted.

By setting the verbose flag `"Code"`, information about the progress of the computation can be printed. An example to demonstrate the interpretation of the verbose output follows:

```
> SetVerbose("Code", true);
> SetSeed(1);
> MinimumWeight(RandomAdditiveCode(GF(4),GF(2),82,39));
GF(2)-Additive code over GF(4) of length 82 with 39 generators.
Is not cyclic
Lower Bound: 1, Upper Bound: 64
Constructed 5 distinct generator matrices
Total Ranks:     21  20  20  20  20
Relative Ranks:  21  20  20  20   1
Starting search for low weight codewords... 0.020
     Discarding non-contributing rank 1 matrix
Enumerating using 1 generator at a time:
     New codeword identified of weight 48, time 0.020
     New codeword identified of weight 46, time 0.020
     New codeword identified of weight 44, time 0.020
     New codeword identified of weight 42, time 0.020
```

```
   Completed Matrix  1:  lower =  5, upper = 42. Time so far: 0.030
     New codeword identified of weight 41, time 0.030
   Completed Matrix  2:  lower =  6, upper = 41. Time so far: 0.030
   Completed Matrix  3:  lower =  7, upper = 41. Time so far: 0.040
     New codeword identified of weight 40, time 0.040
   Completed Matrix  4:  lower =  8, upper = 40. Time so far: 0.040
Enumerating using 2 generators at a time:
     New codeword identified of weight 37, time 0.050
   Completed Matrix  1:  lower =  9, upper = 37. Time so far: 0.050
   Completed Matrix  2:  lower = 10, upper = 37. Time so far: 0.050
   Completed Matrix  3:  lower = 11, upper = 37. Time so far: 0.060
     New codeword identified of weight 36, time 0.060
   Completed Matrix  4:  lower = 12, upper = 36. Time so far: 0.060
Enumerating using 3 generators at a time:
     New codeword identified of weight 34, time 0.070
   Completed Matrix  1:  lower = 13, upper = 34. Time so far: 0.070
     New codeword identified of weight 33, time 0.080
   Completed Matrix  2:  lower = 14, upper = 33. Time so far: 0.090
   Completed Matrix  3:  lower = 15, upper = 33. Time so far: 0.100
   Completed Matrix  4:  lower = 16, upper = 33. Time so far: 0.110
Enumerating using 4 generators at a time:
     New codeword identified of weight 32, time 0.120
   Completed Matrix  1:  lower = 17, upper = 32. Time so far: 0.170
   Completed Matrix  2:  lower = 18, upper = 32. Time so far: 0.250
   Completed Matrix  3:  lower = 19, upper = 32. Time so far: 0.320
   Completed Matrix  4:  lower = 20, upper = 32. Time so far: 0.390
Termination predicted with 7 generators at matrix 4
Enumerating using 5 generators at a time:
   Completed Matrix  1:  lower = 21, upper = 32. Time so far: 0.960
   Completed Matrix  2:  lower = 22, upper = 32. Time so far: 1.570
   Completed Matrix  3:  lower = 23, upper = 32. Time so far: 2.160
   Completed Matrix  4:  lower = 24, upper = 32. Time so far: 2.750
Termination predicted at 118 s (1 m 57 s) with 7 generators at matrix 4
Enumerating using 6 generators at a time:
   Completed Matrix  1:  lower = 25, upper = 32. Time so far: 6.680
   Completed Matrix  2:  lower = 26, upper = 32. Time so far: 10.969
   Completed Matrix  3:  lower = 27, upper = 32. Time so far: 15.149
   Completed Matrix  4:  lower = 28, upper = 32. Time so far: 19.440
Termination predicted at 114 s (1 m 54 s) with 7 generators at matrix 4
Enumerating using 7 generators at a time:
   Completed Matrix  1:  lower = 29, upper = 32. Time so far: 41.739
   Completed Matrix  2:  lower = 30, upper = 32. Time so far: 66.239
   Completed Matrix  3:  lower = 31, upper = 32. Time so far: 90.780
   Completed Matrix  4:  lower = 32, upper = 32. Time so far: 115.510
```

```
Final Results: lower = 32, upper = 32, Total time: 115.510
32
```

Verbose output can be invaluable in the case of lengthy minimum weight calculations.

The algorithm constructs different (equivalent) generator matrices, each of which has pivots in different column positions of the code, called its *information set*. The *relative rank* of a generator matrix is the size of its information set independent of the previously constructed matrices.

When enumerating all generators taken r at a time, once r exceeds the difference between the total rank of a matrix, and its relative rank, the lower bound on the minimum weight will be incremented by 1 for that step.

The upper bound on the minimum weight is determined by the minimum weight of codewords that are enumerated. As soon as these bounds become equal, the computation is complete.

Example H156E8

We illustrate the much greater efficiency of the minimum weight algorithm compared to computing the full weight distribution.

```
> SetVerbose("Code",true);
> C := RandomAdditiveCode(GF(9),GF(3),39,25);
> MinimumWeight(C);
GF(3)-Additive code over GF(9) of length 39 with 25 generators. Is not cyclic
Lower Bound: 1, Upper Bound: 28
Constructed 4 distinct generator matrices
Total Ranks:     13  13  13  13
Relative Ranks:  13  13  12   1
Starting search for low weight codewords... 0.009
     Discarding non-contributing rank 1 matrix
Enumerating using 1 generator at a time:
     New codeword identified of weight 25, time 0.009
     New codeword identified of weight 23, time 0.009
     New codeword identified of weight 21, time 0.009
   Completed Matrix  1:  lower =  3, upper = 21. Time so far: 0.009
   Completed Matrix  2:  lower =  4, upper = 21. Time so far: 0.009
   Completed Matrix  3:  lower =  5, upper = 21. Time so far: 0.009
Enumerating using 2 generators at a time:
     New codeword identified of weight 20, time 0.009
     New codeword identified of weight 19, time 0.009
     New codeword identified of weight 18, time 0.009
   Completed Matrix  1:  lower =  6, upper = 18. Time so far: 0.009
   Completed Matrix  2:  lower =  7, upper = 18. Time so far: 0.019
   Completed Matrix  3:  lower =  8, upper = 18. Time so far: 0.019
Enumerating using 3 generators at a time:
     New codeword identified of weight 17, time 0.070
   Completed Matrix  1:  lower =  9, upper = 17. Time so far: 0.089
   Completed Matrix  2:  lower = 10, upper = 17. Time so far: 0.149
```

```
   Completed Matrix  3:  lower = 11, upper = 17. Time so far: 0.210
Enumerating using 4 generators at a time:
   Completed Matrix  1:  lower = 12, upper = 17. Time so far: 1.379
   Completed Matrix  2:  lower = 13, upper = 17. Time so far: 2.539
   Completed Matrix  3:  lower = 14, upper = 17. Time so far: 3.719
Termination predicted at 49 s with 5 generators at matrix 3
Enumerating using 5 generators at a time:
   Completed Matrix  1:  lower = 15, upper = 17. Time so far: 19.409
   Completed Matrix  2:  lower = 16, upper = 17. Time so far: 35.019
   Completed Matrix  3:  lower = 17, upper = 17. Time so far: 50.649
Final Results: lower = 17, upper = 17, Total time: 50.649
17
> time WeightDistribution(C);
[ <0, 1>, <17, 2>, <18, 58>, <19, 496>, <20, 4000>, <21, 29608>, <22, 194760>,
<23, 1146680>, <24, 6126884>, <25, 29400612>, <26, 126624092>, <27, 487889854>,
<28, 1672552654>, <29, 5075315756>, <30, 13534236754>, <31, 31434430104>, <32,
62869109200>, <33, 106686382216>, <34, 150616653852>, <35, 172132748756>, <36,
153007413552>, <37, 99247655566>, <38, 41788710876>, <39, 8571983110> ]
Time: 224142.820
```

156.7.2 The Weight Distribution

`WeightDistribution(C)`

This function determines the weight distribution for the code C. The distribution is returned in the form of a sequence of tuples, where the i-th tuple contains the i-th weight, w_i say, and the number of codewords having weight w_i.

`DualWeightDistribution(C)`

The weight distribution of the code which is dual to the additive code C (see `WeightDistribution`).

156.7.3 The Weight Enumerator

`WeightEnumerator(C)`

The (Hamming) weight enumerator $W_C(x, y)$ for the additive code C. The weight enumerator is defined by

$$W_C(x, y) = \sum_{u \in C} x^{n-wt(u)} y^{wt(u)}.$$

`CompleteWeightEnumerator(C)`

The complete weight enumerator $\mathcal{W}_C(z_0, \ldots, z_{q-1})$ for the additive code C where q is the size of the alphabet E of C. Let the q elements of E be denoted by $\omega_0, \ldots, \omega_{q-1}$.

If E is a prime field, we let ω_i be i (i.e. take the natural representation of each number). If E is a non-prime field, we let ω_0 be the zero element of E and let ω_i be α^{i-1} for $i = 1 \ldots q-1$ where α is the primitive element of E. Now for a codeword u of C, let $s_i(u)$ be the number of components of u equal to ω_i. The complete weight enumerator is defined by

$$\mathcal{W}_C(z_0, \ldots, z_{q-1}) = \sum_{u \in C} {z_0}^{s_0(u)} \ldots {z_{q-1}}^{s_{q-1}(u)}.$$

`CompleteWeightEnumerator(C, u)`

The complete weight enumerator $\mathcal{W}_{C+u}(z_0, \ldots, z_{q-1})$ for the coset $C + u$, where u is an element of the generic vector space for the code C.

156.7.4 The MacWilliams Transform

`MacWilliamsTransform(n, k, q, W)`

Let C be a hypothetical $[n, k]$ linear code over a finite field of cardinality q. Let W be the weight distribution of C (in the form as returned by the function `WeightDistribution`). This function applies the MacWilliams transform to W to obtain the weight distribution W' of the dual code of C. The transform is a combinatorial algorithm based on n, k, q and W alone. Thus C itself need not exist—the function simply works with the sequence of integer pairs supplied by the user. Furthermore, if W is not the weight distribution of an actual code, the result W' will be meaningless and even negative weights may be returned.

156.7.5 Words

The functions in this section only apply to codes over finite fields.

`Words(C, w: parameters)`

`Cutoff`	RngIntElt	*Default :* ∞
`StoreWords`	BoolElt	*Default :* `true`

Given a linear code C defined over a finite field, return the set of all words of C having weight w. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found. If `StoreWords` is `true` then the words generated will be stored internally.

`NumberOfWords(C, w)`

Given a linear code C defined over a finite field, return the number of words of C having weight w.

`WordsOfBoundedWeight(C, l, u: parameters)`

`Cutoff`	RngIntElt	*Default :* ∞
`StoreWords`	BoolElt	*Default :* `true`

Given a linear code C defined over a finite field, return the set of all words of C having weight between l and u, inclusive. If `Cutoff` is set to a non-negative integer c, then the algorithm will terminate after a total of c words have been found. If `StoreWords` is `true` then any words of a *single weight* generated will be stored internally.

156.8 Families of Linear Codes

156.8.1 Cyclic Codes

While cyclic linear codes are always generated by a single generating polynomial (vector), this is not the case for additive codes. Cyclic additive codes may be created in Magma using either a single generator, or a sequence of generators.

In the important case of $GF(2)$-additive vectors over $GF4$, all cyclic codes can be described in terms of two generators with one generator taken over $GF(4)$ and the other over $GF(2)$. A special function is provided for this construction.

`AdditiveCyclicCode(v)`
`AdditiveCyclicCode(K, v)`
`AdditiveCyclicCode(Q)`
`AdditiveCyclicCode(K, Q)`

Given either a single vector v or sequence of vectors Q over some finite field F, return the K-additive code over F generated by all shifts of the inputs. The field K must be a subfield of F and if it is the prime subfield of F, it may be omitted.

`AdditiveCyclicCode(n, f)`
`AdditiveCyclicCode(K, n, f)`
`AdditiveCyclicCode(n, Q)`
`AdditiveCyclicCode(K, n, Q)`

Given either a single polynomial f or sequence Q of polynomials over some finite field F, return the K-additive code of length n over F generated by all shifts of the inputs. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

```
AdditiveCyclicCode(v4, v2)
```

Given two vectors of equal length n, where v_4 is over $GF(4)$ and v_2 is over $GF(2)$, return the F_2–additive code generated by all of their cyclic shifts. Note that for the case of $GF(2)$-additive codes over $GF(4)$, two generators suffice to generate any such code.

```
AdditiveCyclicCode(n, f4, f2)
```

Given two polynomials f_4 and f_2, where f_4 is over $GF(4)$ and f_2 is over $GF(2)$, return the F_2–additive code of length n generated by all of their cyclic shifts. The degree of the polynomials f_4 and f_2 must not exceed $n-1$. Note that for the case of $GF(2)$-additive codes over $GF(4)$, two generators suffice to generate any such code.

156.8.2 Quasicyclic Codes

Quasicyclic codes are a generalisation of cyclic codes. In MAGMA quasicyclic codes consist of horizontally joined cyclic blocks.

```
AdditiveQuasiCyclicCode(n, Q)
```

```
AdditiveQuasiCyclicCode(K, n, Q)
```

Given an integer n, and a sequence Q of polynomials over some finite field F, return the K–additive quasicyclic code, whose cyclic blocks are generated by the polynomials in Q. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

```
AdditiveQuasiCyclicCode(n, Q, h)
```

```
AdditiveQuasiCyclicCode(K, n, Q, h)
```

Given an integer n, and a sequence Q of polynomials over some finite field F, and an integer h, then return the K–additive quasicyclic code, whose cyclic blocks are generated by the polynomials in Q and stacked 2-dimensionally of height h. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

```
AdditiveQuasiCyclicCode(Q)
```

```
AdditiveQuasiCyclicCode(K, Q)
```

Given a sequence Q of vectors over some finite field F, then return the K–additive quasicyclic code, whose cyclic blocks are generated by the vectors in Q. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

```
AdditiveQuasiCyclicCode(Q, h)
```

```
AdditiveQuasiCyclicCode(K, Q, h)
```

Given a sequence Q of vectors over some finite field F, and an integer h, then return the K–additive quasicyclic code, whose cyclic blocks are generated by the vectors in Q and stacked 2-dimensionally of height h. The field K must be a subfield of F, and if it is the prime subfield of F, it may be omitted.

156.9 New Codes from Old

The operations described here produce a new code by modifying in some way the codewords of a given code.

156.9.1 Standard Constructions

`AugmentCode(C)`

Construct a new additive code by including the all-ones vector with the words of the additive code C.

`CodeComplement(C, S)`

Given a subcode S of the code C, return a code C' such that $C = S + C'$. Both C and S must be defined over the same field.

`DirectSum(C, D)`

Given codes C and D, form the code that is direct sum of C and D. The direct sum consists of all vectors $u|v$, where $u \in C$ and $v \in D$.

`DirectSum(Q)`

Given a sequence of codes $Q = [C_1, \ldots, C_r]$, all defined over the same field F, construct the direct sum of the C_i.

`DirectProduct(C, D)`

Given an $[n_1, k_1]$ code C and an $[n_2, k_2]$ code D, both over the same ring R, construct the direct product of C and D. The direct product has length $n_1 \cdot n_2$ and its generator matrix is the Kronecker product of the basis matrices of C and D.

`ExtendCode(C)`

Given an $[n, k, d]$ additive code C, form a new code C' from C by adding the appropriate extra coordinate to each vector of C such that the sum of the coordinates of the extended vector is zero.

`ExtendCode(C, n)`

Return the code obtained by extending the code C extended n times.

`PadCode(C, n)`

Add n zeros to the end of each codeword of the code C.

`PlotkinSum(C1, C2)`

Given codes C_1 and C_2 defined over the same alphabet, return the code consisting of all vectors of the form $u|u + v$, where $u \in C1$ and $v \in C2$. Zeros are appended where needed to make up any length differences in the two codes.

`PlotkinSum(C1, C2, C3: parameters)`

`a` FldFinElt *Default :* -1

Given three codes C_1, C_2 and C_3 defined over the same alphabet K, return the code consisting of all vectors of the form $u|u+a*v|u+v+w$, where $u \in C1$, $v \in C2$ and $w \in C3$. The default value of the multiplier a is a primitive element of K. Zeros are appended where needed to ensure that every codeword has the same length.

`PunctureCode(C, i)`

Given an $[n, k]$ code C, and an integer i, $1 \leq i \leq n$, construct a new code C' by deleting the i-th coordinate from each code word of C.

`PunctureCode(C, S)`

Given an $[n, k]$ code C and a set S of distinct integers $\{i_1, \cdots, \mathrm{i}_r\}$ each of which lies in the range $[1, n]$, construct a new code C' by deleting the components $i_1, \cdots, i_r$ from each code word of C.

`ShortenCode(C, i)`

Given an $[n, k]$ code C and an integer i, $1 \leq i \leq n$, construct a new code from C by selecting only those codewords of C having a zero as their i-th component and deleting the i-th component from these codewords. Thus, the resulting code will have length $n - 1$.

`ShortenCode(C, S)`

Given an $[n, k]$ code C and a set S of distinct integers $\{i_1, \cdots, \mathrm{i}_r\}$, each of which lies in the range $[1, n]$, construct a new code from C by selecting only those codewords of C having zeros in each of the coordinate positions $i_1, \cdots, i_r$, and deleting these components. Thus, the resulting code will have length $n - r$.

156.9.2 Combining Codes

`C1 cat C2`

Given codes $C1$ and $C2$, both defined over the same field K, return the concatenation C of $C1$ and $C2$. The generators of the resultant code are the concatenations of the generators of $C1$ and $C2$.

`Juxtaposition(C1, C2)`

Given an $[n_1, k, d_1]$ code $C1$ and an $[n_2, k, d_2]$ code $C2$ of the same dimension, where both codes are defined over the same field K, this function returns a $[n_1 + n_2, k, \geq d_1 + d_2]$ code whose generator matrix is `HorizontalJoin(A, B)`, where A and B are the generator matrices for codes $C1$ and $C2$, respectively.

156.10 Automorphism Group

`AutomorphismGroup(C)`

The automorphism group of the additive code C. Currently, this function is only available for additive codes over $GF(4)$.

`PermutationGroup(C)`

The subgroup of the automorphism group of the additive code C consisting of permutations of the coordinates. Currently, this function is only available for additive codes over $GF(4)$.

157 QUANTUM CODES

Chapter 157
QUANTUM CODES

157.1 Introduction

Interest in quantum computing has grown rapidly following the discovery by Peter Shor in 1994 of a polynomial-time algorithm for integer factorization [Sho94]. In a classical computer a sequence of N binary digits defines one specific configuration among the 2^N possible values. However, in a quantum computer a collection of N "qubits" has a state function (in 'ket' notation) $|\psi\rangle$ in a *Hilbert space*, which can be in a superposition of all 2^N possible values

$$|\psi\rangle = \sum_{\mathbf{v}\in\mathbf{Z}_2^N} \alpha_{\mathbf{v}}|\mathbf{v}\rangle, \qquad \alpha_{\mathbf{v}} \in \mathbf{C}, \quad \sum_{\mathbf{v}} |\alpha_{\mathbf{v}}|^2 = 1.$$

A basic theorem in quantum information theory states that it is impossible to clone a quantum state. Since this implies that it is not possible to copy quantum information, it was initially believed that error-correction would be impossible on a quantum computer. However, in 1995 Shor showed that it *was* possible to encode quantum information in such a way that errors can be corrected, assuming an error model in which errors occur independently in distinct qubits [Sho95].

Following this discovery, an class of quantum error-correcting codes known as *stabilizer codes* were developed. In [CRSS98] (which is the major reference for this chapter of the MAGMA Handbook), it was shown that stabilizer codes can be represented in terms of additive codes over finite fields (see chapter 156 for a description of additive codes). This remarkable result reduces the problem of constructing fault-tolerant encodings on a continuous Hilbert space to that of constructing certain discrete codes, allowing the use of many of the tools developed in classical coding theory.

The current MAGMA package for quantum codes deals exclusively with finite field representations of stabilizer codes. It is important to keep in mind that, although a quantum code is *represented* by a code over a finite field, an actual quantum code is in fact a totally different object. The full theory behind quantum stabilizer codes will not be described here, for that the reader should consult the main reference [CRSS98]. A brief synopsis will outline how the finite field representation of a stabilizer code is to be interpreted, and the specifics of this representation in MAGMA.

Many of the conventions and functions for classical error-correcting code types in MAGMA can be ambiguous in the context of quantum codes. For this reason the handbook should be carefully consulted before assuming that any particular aspect of a quantum code follows naturally from classical coding theory definitions.

The reduction of the problem of continuous errors on a Hilbert space to a problem employing a discrete finite field representation is achieved by confining attention to a

finite *error group*. An element of the error group, acting on the N qubits, is expressed as a combination of bit flip errors, designated by the operator X, and phase shift errors, designated by the operator Z (as well as an overall phase factor that will be ignored here):

$$X(\mathbf{a})Z(\mathbf{b})|\mathbf{v}\rangle = (-1)^{\mathbf{v}\cdot\mathbf{b}}|\mathbf{v}+\mathbf{a}\rangle$$

The error group is given by the set $\{X(\mathbf{a})Z(\mathbf{b}) : \mathbf{a},\mathbf{b} \in \mathbf{Z}_2^n\}$ and its elements can be written as length $2N$ binary vectors $(\mathbf{a}|\mathbf{b})$. An error represented by such a vector in MAGMA is said to be in *extended format* which is distinct from the default representation. A more common (and practical) representation is as the element $\mathbf{w}$ of $F_4^{(N)}$ given by $\mathbf{w} = \mathbf{a} + \omega\mathbf{b}$, where ω is a primitive element of $GF(4)$. This representation is referred to as the *compact format*, and is the default format used in MAGMA for quantum codes. Note that this is slightly different to the representation $\widehat{\mathbf{w}} = \omega\mathbf{a} + \overline{\omega}\mathbf{b}$ used in [CRSS98] for binary quantum codes, but they are equivalent: $\mathbf{w} = \overline{\omega} * \widehat{\mathbf{w}}$.

The MAGMA package also supports non-binary quantum codes, which are obtained by generalizing from the binary case in a natural way. For quantum codes based on qubits over $GF(q)$, the compact format in $GF(q^2)$ will be $\mathbf{w} = \mathbf{a} + \lambda\mathbf{b}$, where λ is a fixed element returned by the function `QuantumBasisElement`$(GF(q^2))$.

A symplectic inner product is defined on the group of errors, in its representation as a set of $GF(q)$-vectors. For vectors in extended format this is defined by

$$(\mathbf{a}_1|\mathbf{b}_1) * (\mathbf{a}_2|\mathbf{b}_2) = \mathbf{a}_1 \cdot \mathbf{b}_2 - \mathbf{a}_2 \cdot \mathbf{b}_1$$

In compact format (over $GF(4)$) the equivalent inner product is defined by

$$\mathbf{w}_1 * \mathbf{w}_2 = \mathrm{Trace}(\mathbf{w}_1 \cdot \overline{\mathbf{w}}_2).$$

Since the commutator of two errors is given by

$$\begin{aligned}
&\Big[\big(X(\mathbf{a}_1)Z(\mathbf{b}_1)\big)\big(X(\mathbf{a}_2)Z(\mathbf{b}_2)\big) - \big(X(\mathbf{a}_2)Z(\mathbf{b}_2)\big)\big(X(\mathbf{a}_1)Z(\mathbf{b}_1)\big)\Big]|\mathbf{v}\rangle \\
&\qquad = (-1)^{\mathbf{v}\cdot\mathbf{b}_2+(\mathbf{v}+\mathbf{a}_2)\cdot\mathbf{b}_1}|\mathbf{v}+\mathbf{a}_1+\mathbf{a}_2\rangle + (-1)^{\mathbf{v}\cdot\mathbf{b}_1+(\mathbf{v}+\mathbf{a}_1)\cdot\mathbf{b}_2}|\mathbf{v}+\mathbf{a}_1+\mathbf{a}_2\rangle \\
&\qquad = \Big[(-1)^{\mathbf{a}_2\cdot\mathbf{b}_1} - (-1)^{\mathbf{a}_1\cdot\mathbf{b}_2}\Big](-1)^{\mathbf{v}\cdot(\mathbf{b}_1-\mathbf{b}_2)}|\mathbf{v}+\mathbf{a}_1+\mathbf{a}_2\rangle \\
&\qquad = \Big[1 - \delta_{\mathbf{a}_1\cdot\mathbf{b}_2,\,\mathbf{a}_2\cdot\mathbf{b}_1}\Big]\cdots
\end{aligned}$$

then clearly errors will commute if and only if their finite field representations are orthogonal with respect to the symplectic inner product.

A quantum stabilizer code is defined by an abelian subgroup of the error group. In the context of its finite field representation this translates to a self-orthogonal additive code under the symplectic inner product. So a quantum stabilizer code Q is defined by a symplectic self-orthogonal additive code S, which is (with some redundancy) termed the stabilizer code of Q.

The error-correcting capability of a code is determined by the set of errors which can not be detected. For classical linear codes these undetectable errors are precisely the non-zero codewords of the code, while for a quantum code, the undetectable errors are given by the set $S^{\perp}\backslash S$, where $S^{\perp}$ is the symplectic dual of S.

The most important measure of the ability of a quantum code to correct errors is its *minimum weight*, that is, is the minimum of the weights of the words of $S^{\perp}\backslash S$. An exception to this definition occurs in the case of quantum codes having dimension zero, which are defined by symplectic self-dual stabilizer codes. These are termed "self-dual quantum codes" and are defined to have a minimum weight equal to the (classical) minimum weight of their stabilizer code.

157.2 Constructing Quantum Codes

A quantum code of length n over $GF(q)$ is defined in terms of a symplectic self-orthogonal stabilizer code, which is given either as a length n additive code over $GF(q^2)$ (compact format) or as a length $2n$ additive code over $GF(q)$ (extended format). If Q is a quantum code with generator matrix G_1 in compact format, and generator matrix $G_2 = (A|B)$ in extended format, then

$$G_1 = A + \lambda B,$$

where λ is a fixed element returned by the function `QuantumBasisElement`$(GF(q^2))$. By default MAGMA assumes the compact format. However, the extended format can be flagged by setting the variable argument `ExtendedFormat` to `true`.

An $[n, k]$ symplectic self-orthogonal linear code over $GF(q^2)$ will generate an $[[n, n/2 - k]]$ quantum stabilizer code. A (compact format) additive symplectic self-orthogonal code C over $GF(q^2)$ will give a quantum code of the same length and "dimension" $\log_q(N)$, where N is the number of code words in C.

157.2.1 Construction of General Quantum Codes

`QuantumCode(S)`

`ExtendedFormat` BOOLELT *Default :* `false`

Given an additive code S which is self-orthogonal with respect to the symplectic inner product, return the quantum code defined by S. By default, S is interpreted as being in compact format, that is, a length n additive code over $GF(q^2)$. If `ExtendedFormat` is set *true*, then S is interpreted as being in extended format, that is, a length $2n$ additive code over $GF(q)$.

Example H157E1

A linear code over $GF(4)$ that is *even* is symplectic self-orthogonal. Note that when a quantum code is printed in MAGMA, an *additive* stabilizer matrix over $GF(q^2)$ is displayed.

```
> F<w> := GF(4);
> M := Matrix(F, 2, 6, [1,0,0,1,w,w, 0,1,0,w,w,1]);
```

```
> C := LinearCode(M);
> C;
[6, 2, 4] Linear Code over GF(2^2)
Generator matrix:
[  1   0   0   1   w   w]
[  0   1   0   w   w   1]
> IsEven(C);
true
> IsSymplecticSelfOrthogonal(C);
true
> Q := QuantumCode(C);
> Q;
[[6, 2]] Quantum code over GF(2^2), stabilized by:
[  1   0   0   1   w   w]
[  w   0   0   w w^2 w^2]
[  0   1   0   w   w   1]
[  0   w   0 w^2 w^2   w]
> MinimumWeight(Q);
1
> Q;
[[6, 2, 1]] Quantum code over GF(2^2), stabilized by:
[  1   0   0   1   w   w]
[  w   0   0   w w^2 w^2]
[  0   1   0   w   w   1]
[  0   w   0 w^2 w^2   w]
```

Example H157E2

Any stabilizer code used to construct a quantum code, may be expressed in either compact or extended format. The length 6 quaternary additive code S in the previous example (H157E1) is equivalent to a length 12 binary additive code in extended format.
Note that the code will still be displayed in compact format.

```
> F<w> := GF(4);
> C := LinearCode<GF(2), 12 |
>      [ 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1 ],
>      [ 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0 ],
>      [ 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0 ],
>      [ 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1 ] >;
> C;
[12, 4, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 1 1 0 0 1 1]
[0 1 0 0 0 1 0 0 0 1 1 0]
[0 0 0 1 0 1 1 1 0 0 0 0]
[0 0 0 0 1 1 1 0 0 1 1 1]
> Q := QuantumCode(C : ExtendedFormat := true);
> Q;
```

```
[[6, 2]] Quantum code over GF(2^2), stabilized by:
[  1   0   0   1   w   w]
[  w   0   0   w w^2 w^2]
[  0   1   0   w   w   1]
[  0   w   0 w^2 w^2   w]
```

Example H157E3

Any self-orthogonal code which has a rate of 1/2 must be self-dual, and gives rise to a dimension zero quantum code (which is also termed self-dual). In this example we construct the hexacode, which is the unique extremal length 6 self-dual quantum code of minimum weight 4.

```
> F<w> := GF(4);
> M := Matrix(F, 3, 6, [0,0,1,1,1,1, 0,1,0,1,w,w^2, 1,0,0,1,w^2,w]);
> C := LinearCode(M);
> C;
[6, 3, 4] Linear Code over GF(2^2)
Generator matrix:
[  1   0   0   1 w^2   w]
[  0   1   0   1   w w^2]
[  0   0   1   1   1   1]
> IsSymplecticSelfOrthogonal(C);
true
> Q := QuantumCode(C);
> MinimumWeight(Q);
4
> Q;
[[6, 0, 4]] self-dual Quantum code over GF(2^2), stabilized by:
[  1   0   0   1 w^2   w]
[  w   0   0   w   1 w^2]
[  0   1   0   1   w w^2]
[  0   w   0   w w^2   1]
[  0   0   1   1   1   1]
[  0   0   w   w   w   w]
```

Example H157E4

Stabilizer codes neither have to be linear nor even and indeed any additive code which is symplectic self-orthogonal will generate a quantum code. The following code was randomly generated.

```
> F<w> := GF(4);
> M := Matrix(F, 3, 7, [1,w,w,w,0,0,1, w,0,1,0,w^2,0,1, 0,w^2,w,w^2,w,0,0]);
> C := AdditiveCode(GF(2),M);
> C;
[7, 1 1/2 : 3, 4] GF(2)-Additive Code over GF(2^2)
Generator matrix:
[  1   w   w   w   0   0   1]
[  w   0   1   0 w^2   0   1]
```

```
[  0 w^2   w w^2   w   0   0]
> IsSymplecticSelfOrthogonal(C);
true
```

The code C can be shown to be neither linear nor even: in fact it has the same number of even and odd codewords.

```
> IsLinear(C);
false
> {* Weight(v) mod 2 : v in C *};
{* 0^^4, 1^^4 *}
>
> Q := QuantumCode(C);
> MinimumWeight(Q);
1
> Q;
[[7, 4, 1]] Quantum code over GF(2^2), stabilized by:
[  1   w   w   w   0   0   1]
[  w   0   1   0 w^2   0   1]
[  0 w^2   w w^2   w   0   0]
```

`QuantumCode(M)`

`ExtendedFormat` BoolElt *Default :* `false`

Given a matrix M over $GF(q^2)$ for which the $GF(q)$ additive span of its rows is a self-orthogonal code S with respect to the symplectic inner product, return the quantum code defined by S. By default, M is interpreted as being in compact format, that is, a matrix whose rows are length n vectors over $GF(q^2)$. However, if `ExtendedFormat` is set `true`, then M will be interpreted as being in extended format, that is, a matrix whose rows are length $2n$ vectors over $GF(q)$.

Example H157E5

A quantum code can be constructed directly from an additive stabilizer matrix, thereby avoiding creation of the stabilizer code. The quantum code given in example H157E4 could have also been constructed as follows:

```
> F<w> := GF(4);
> M := Matrix(F, 3, 7, [1,w,w,w,0,0,1, w,0,1,0,w^2,0,1, 0,w^2,w,w^2,w,0,0]);
> Q := QuantumCode(M);
> Q;
[[7, 4]] Quantum code over GF(2^2), stabilized by:
[  1   w   w   w   0   0   1]
[  w   0   1   0 w^2   0   1]
[  0 w^2   w w^2   w   0   0]
```

`QuantumCode(G)`

Given a graph G, return the self-dual (dimension 0) quantum code defined by the adjacency matrix of G.

Example H157E6

The unique extremal $[[6, 0, 4]]$ hexacode can be defined in terms of a graph representing a 5–spoked wheel. The graph is specified by listing the edges comprising its circumference, followed by the spokes radiating out from the center.

```
> G := Graph<6 | {1,2},{2,3},{3,4},{4,5},{5,1}, <6, {1,2,3,4,5}> >;
> Q := QuantumCode(G);
> Q:Minimal;
[[6, 0]] self-dual Quantum code over GF(2^2)
> MinimumWeight(Q);
4
> Q:Minimal;
[[6, 0, 4]] self-dual Quantum code over GF(2^2)
```

Example H157E7

The unique extremal $[[12, 0, 6]]$ dodecacode can also be described by a graph with a nice mathematical structure. The graph construction is derived from the diagram given by Danielson in [Dan05]. In order to employ modular arithmetic, the graph vertices are numbered from 0 to 11.

```
> S := {@ i : i in [0 .. 11] @};
> G := Graph<S |
>     { {4*k+i,4*k+i+2}                    : i in [0..1], k in [0..2] },
>     { {4*k+i,4*k+(i+1) mod 4}            : i in [0..3], k in [0..2] },
>     { {4*k+i,4*((k+1) mod 3)+(i+1) mod 4} : i in [0..3], k in [0..2] } >;
> Q := QuantumCode(G);
> MinimumWeight(Q);
6
> Q:Minimal;
[[12, 0, 6]] self-dual Quantum code over GF(2^2)
```

`RandomQuantumCode(F, n, k)`

Let F be a degree 2 extension of a finite field $GF(q)$. Given positive integers n and k such that $n \geq k$, this function returns a random $[[n, k]]$ quantum stabilizer code over F. The field F is assumed to be given in compact format.

Example H157E8

We construct a random $[[10, 6]]$ quantum code over $GF(4)$.

```
> F<w> := GF(4);
> Q := RandomQuantumCode(F, 10, 6);
> Q;
[[10, 6]] Quantum code over GF(2^2), stabilized by:
[  w   0   0   w   1   1   w w^2   w w^2]
[  0   1   0   w   1 w^2 w^2   1   w   w]
[  0   w   1   1   1   0 w^2   0   0   0]
[  0   0   w   1   1   0   1 w^2   1   w]
```

`Subcode(Q, k)`

Given a quantum code Q of dimension $k_Q \geq k$ then return a subcode of Q of dimension k.

157.2.2 Construction of Special Quantum Codes

`Hexacode()`

Return the $[[6, 0, 4]]$ self-dual quantum hexacode.

`Dodecacode()`

Return the $[[12, 0, 6]]$ self-dual quantum dodecacode.

157.2.3 CSS Codes

`CSSCode(C1, C2)`

`CalderbankShorSteaneCode(C1, C2)`

Given two classical linear binary codes C_1 and C_2 of length n such that C_2 is a subcode of C_1, form a quantum code using the construction of Calderbank, Shor and Steane [CS96, Ste96a, Ste96b].

Example H157E9

Let C_1 denote the $[7, 4, 3]$ Hamming code and C_2 denote its dual. Observing that C_1 contains C_2, we may apply the CSS construction using C_1 and C_2 to obtain a $[[7, 1, 3]]$ code.

```
> F<w> := GF(4);
> C1 := HammingCode(GF(2), 3);
> C1;
[7, 4, 3] "Hamming code (r = 3)" Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 1 0]
[0 1 0 0 0 1 1]
[0 0 1 0 1 1 1]
[0 0 0 1 1 0 1]
> C2 := Dual(C1);
> C2;
[7, 3, 4] Cyclic Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1 1]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
> C2 subset C1;
true
> Q := CSSCode(C1, C2);
> MinimumWeight(Q);
3
> Q;
[[7, 1, 3]] CSS Quantum code over GF(2^2), stabilised by:
[  1   0   0   1   0   1   1]
[  w   0   0   w   0   w   w]
[  0   1   0   1   1   1   0]
[  0   w   0   w   w   w   0]
[  0   0   1   0   1   1   1]
[  0   0   w   0   w   w   w]
```

157.2.4 Cyclic Quantum Codes

Cyclic quantum codes are those having cyclic stabilizer codes. Conditions are listed in [CRSS98] for generating polynomials which give rise to symplectic self-orthogonal stabilizer codes.

`QuantumCyclicCode(v)`

`QuantumCyclicCode(Q)`

`LinearSpan` BoolElt *Default :* `false`

Given either a single vector v or sequence of vectors Q defined over a finite field F, return the quantum code generated by the span of the cyclic shifts of the supplied

vectors. The span must be self-orthogonal with respect to the symplectic inner product. By default, the additive span is taken over the prime field, but if the variable argument `LinearSpan` is set to `true`, then the linear span will be taken.

Example H157E10

A large number of good cyclic quantum codes exist. For example, the best known binary self-dual quantum code of length 15 is cyclic.

```
> F<w> := GF(4);
> v := VectorSpace(F, 15) ! [w,1,1,0,1,0,1,0,0,1,0,1,0,1,1];
> Q := QuantumCyclicCode(v);
> MinimumWeight(Q);
6
> Q:Minimal;
[[15, 0, 6]] self-dual Quantum code over GF(2^2)
```

`QuantumCyclicCode(n, f)`

`QuantumCyclicCode(n, Q)`

`LinearSpan` BoolElt *Default :* `false`

Let n be a positive integer. Given either a single polynomial f or a sequence of polynomials Q over some finite field F, return the quantum code of length n generated by the additive span of their cyclic shifts. The additive span must be symplectic self-orthogonal. By default, the additive span is taken over the prime field, but if the variable argument `LinearSpan` is set to `true`, then the linear span will be taken.

Example H157E11

Since classical cyclic codes correspond to factors of cyclotomic polynomials it is frequently convenient to specify a cyclic code in terms of polynomials. Here we construct the best known binary quantum codes with parameters $[[23, 12, 4]]$ and $[[25, 0, 8]]$.

```
> F<w> := GF(4);
> P<x> := PolynomialRing(F);
> f := x^16 + x^15 + x^13 + w*x^12 + x^11 + w*x^10 + x^9 + x^8 + w^2*x^7 +
>      x^6 + x^5 + w*x^4 + w^2*x^3 + w*x^2 + w^2*x + w^2;
> Q := QuantumCyclicCode(23, f);
> MinimumWeight(Q);
4
> Q:Minimal;
[[23, 12, 4]] Quantum code over GF(2^2)
>
> f := x^12 + x^11 + x^10 + x^8 + w*x^6 + x^4 + x^2 + x + 1;
> Q := QuantumCyclicCode(25, f);
> MinimumWeight(Q);
```

```
8
> Q:Minimal;
[[25, 0, 8]] self-dual Quantum code over GF(2^2)
```

`QuantumCyclicCode(v4, v2)`

In the important case of $GF(2)$-additive codes over $GF(4)$, any cyclic code can be specified by two generators. Given vectors $v4$ and $v2$ both of length n, where v_4 is over $GF(4)$ and v_2 is over $GF(2)$, this function returns the length n quantum code generated by the additive span of their cyclic shifts. This span must be self-orthogonal with respect to the symplectic inner product.

Example H157E12

Any cyclic binary quantum code of length n is determined by a cyclic stabilizer code, which can be defined uniquely in terms of an n-dimensional vector over $GF(4)$ together with an n-dimensional vector over $GF(2)$. We construct the best known $[[21, 0, 8]]$ and $[[21, 5, 6]]$ cyclic binary quantum codes.

```
> F<w> := GF(4);
> v4 := RSpace(F, 21) ! [w^2,w^2,1,w,0,0,1,1,1,1,0,1,0,1,1,0,1,1,0,0,0];
> v2 := RSpace(GF(2),21) ! [1,0,1,1,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,0];
> Q := QuantumCyclicCode(v4,v2);
> MinimumWeight(Q);
8
> Q:Minimal;
[[21, 0, 8]] self-dual Quantum code over GF(2^2)
>
> v4 := RSpace(F, 21) ! [w,0,w^2,w^2,w,w^2,w^2,0,w,1,0,0,1,0,0,0,0,1,0,0,1];
> v2 := RSpace(GF(2), 21) ! [1,0,1,1,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,0];
> Q := QuantumCyclicCode(v4,v2);
> MinimumWeight(Q);
6
> Q:Minimal;
[[21, 5, 6]] Quantum code over GF(2^2)
```

157.2.5 Quasi-Cyclic Quantum Codes

Quasi-cyclic quantum codes are those having quasi-cyclic stabilizer codes.

`QuantumQuasiCyclicCode(n, Q)`

`LinearSpan` BoolElt *Default :* `false`

Given an integer n, and a sequence Q of polynomials over some finite field F, let S be the quasi-cyclic classical code generated by the span of the set of vectors formed by concatenating cyclic blocks generated by the polynomials in Q. Assuming that S is self-orthogonal with respect to the symplectic inner product, this function returns the quasi-cyclic quantum code with stabiliser code S. If the span of the vectors is not symplectic self-orthogonal, an error will be flagged.

By default the additive span is taken over the prime field, but if the variable argument `LinearSpan` is set to `true`, then the linear span will be taken.

`QuantumQuasiCyclicCode(Q)`

`LinearSpan` BoolElt *Default :* `false`

Given a sequence Q of vectors, return the quantum code whose additive stabilizer matrix is constructed from the length n cyclic blocks generated by the cyclic shifts of the vectors in Q. If the variable argument `LinearSpan` is set to `true`, then the linear span of the shifts will be used, else the additive span will be used (default).

Example H157E13

Most quasi-cyclic quantum codes currently known are linear, since this is where most research on quasi-cyclic codes has been focused. In this example we construct the best known quasi-cyclic binary quantum codes with parameters $[[14, 0, 6]]$ and $[[18, 6, 5]]$.

```
> F<w> := GF(4);
> V7 := VectorSpace(F, 7);
> v1 := V7 ! [1,0,0,0,0,0,0];
> v2 := V7 ! [w^2,1,w^2,w,0,0,w];
> Q := QuantumQuasiCyclicCode([v1, v2] : LinearSpan := true);
> MinimumWeight(Q);
6
> Q:Minimal;
[[14, 0, 6]] self-dual Quantum code over GF(2^2)
>
> V6 := VectorSpace(F, 6);
> v1 := V6 ! [1,1,0,0,0,0];
> v2 := V6 ! [1,0,1,w^2,0,0];
> v3 := V6 ! [1,1,w,1,w,0];
> Q := QuantumQuasiCyclicCode([v1, v2, v3] : LinearSpan := true);
> MinimumWeight(Q);
5
> Q:Minimal;
[[18, 6, 5]] Quantum code over GF(2^2)
```

157.3 Access Functions

`QuantumBasisElement(F)`

Given a degree 2 extension field $F = GF(q^2)$, return the element $\lambda \in F$ which acts to connect the extended and compact formats. For a vector $(\mathbf{a}|\mathbf{b})$ in extended format, the corresponding compact format of this vector will be $\mathbf{w} = \mathbf{a} + \lambda\mathbf{b}$.

`StabilizerCode(Q)`

`StabiliserCode(Q)`

`ExtendedFormat` BoolElt *Default :* `false`

The additive stabiliser code S which defines the quantum code Q. By default S is returned in the compact format of a length n code over $GF(q^2)$, but if `ExtendedFormat` is set to `true`, then it will be returned in extended format as a length $2n$ code over $GF(q)$.

`StabilizerMatrix(Q)`

`StabiliserMatrix(Q)`

`ExtendedFormat` BoolElt *Default :* `false`

Given a quantum code Q return the additive stabiliser matrix M defining Q. By default M is returned in the compact format of a length n code over $GF(q^2)$, but if `ExtendedFormat` is set to `true`, then it will be returned in the extended format as a length $2n$ code over $GF(q)$.

`NormalizerCode(Q)`

`NormaliserCode(Q)`

`ExtendedFormat` BoolElt *Default :* `false`

The additive normalizer code N which defines the quantum code Q. By default N is returned in the compact format of a length n code over $GF(q^2)$, but if `ExtendedFormat` is set to `true`, then it will be returned in extended format as a length $2n$ code over $GF(q)$.

`NormalizerMatrix(Q)`

`NormaliserMatrix(Q)`

`ExtendedFormat` BoolElt *Default :* `false`

Given a quantum code Q return the additive normalizer matrix M defining Q. By default M is returned in the compact format of a length n code over $GF(q^2)$, but if `ExtendedFormat` is set to `true`, then it will be returned in the extended format as a length $2n$ code over $GF(q)$.

157.3.1 Quantum Error Group

As described in the introduction to this chapter, vectors over a finite field used to describe a quantum stabilizer code actually represent elements of the corresponding quantum error group. For a p-ary N qubit system (where p is prime) this error group is the extra-special group with order 2^{2N+1} consisting of combinations of N bit-flip errors, N phase flip errors, and an overall phase shift. All groups in this section use a polycyclic group representation.

`QuantumErrorGroup(p, n)`

Return the abelian group representing all possible errors for a length n p-ary qubit system, which is an extra-special group of order p^{2n+1} with $2n+1$ generators. The generators correspond to the qubit-flip operators $X(i)$, the phase-flip operators $Z(i)$, and an overall phase multiplication W by the p-th root of unity. The generators appear in the order $X(1), Z(1), \ldots, X(n), Z(n), W$.

`QuantumBinaryErrorGroup(n)`

Return the abelian group representing all possible errors on a length n binary qubit system, which is an extra special group of order 2^{2n-1}.

Example H157E14

The image of a vector in the error group is easily obtained from its extended format representation. We illustrate the connection between symplectic orthogonality as a vector, and commutativity as an element of the error group.

```
> n := 5;
> VSn  := VectorSpace(GF(2), n);
> VS2n := VectorSpace(GF(2), 2*n);
> E := QuantumBinaryErrorGroup(n);
> BitFlips   := [E.i : i in [1..2*n] | IsOdd(i)  ];
> PhaseFlips := [E.i : i in [1..2*n] | IsEven(i) ];
```

We first take two vectors which are not orthogonal and show their images in the error group do not commute.

```
> v1a := VSn ! [0,1,1,0,1]; v1b := VSn ! [0,1,1,0,1];
> v1  := VS2n ! HorizontalJoin(v1a, v1b);
> v2a := VSn ! [1,0,1,1,0]; v2b := VSn ! [0,1,0,1,1];
> v2  := VS2n ! HorizontalJoin(v2a, v2b);
> SymplecticInnerProduct(v1,v2 : ExtendedFormat := true);
1
>
> e1 := &*[ BitFlips[i]   : i in Support(v1a) ] *
>       &*[ PhaseFlips[i] : i in Support(v1b) ];
> e2 := &*[ BitFlips[i]   : i in Support(v2a) ] *
>       &*[ PhaseFlips[i] : i in Support(v2b) ];
> e1*e2 eq e2*e1;
```

```
false
```

Next a pair of orthogonal vectors is shown to commute.

```
> v1a := VSn ! [1,1,0,1,0]; v1b := VSn ! [0,0,1,1,0];
> v1  := VS2n ! HorizontalJoin(v1a, v1b);
> v2a := VSn ! [0,1,1,1,0]; v2b := VSn ! [0,1,1,1,0];
> v2  := VS2n ! HorizontalJoin(v2a, v2b);
> SymplecticInnerProduct(v1,v2 : ExtendedFormat := true);
0
>
> e1 := &*[ BitFlips[i]   : i in Support(v1a) ] *
>       &*[ PhaseFlips[i] : i in Support(v1b) ];
> e2 := &*[ BitFlips[i]   : i in Support(v2a) ] *
>       &*[ PhaseFlips[i] : i in Support(v2b) ];
> e1*e2 eq e2*e1;
true
```

`QuantumErrorGroup(Q)`

For a quantum code Q of length n, return the group of all errors on n qubits. This is the full error group, the ambient space containing all possible errors.

`StabilizerGroup(Q)`

`StabiliserGroup(Q)`

Return the abelian group of errors that defines the quantum code Q, which is a subgroup of the group returned by `QuantumErrorGroup(Q)`.

`StabilizerGroup(Q, G)`

`StabiliserGroup(Q, G)`

Given a quantum code Q with error group G (an extra-special group), return the abelian group of errors of Q as a subgroup of G.

Example H157E15

The stabilizer group of any quantum stabilizer code over $GF(4)$ will be abelian.

```
> F<w> := GF(4);
> Q := RandomQuantumCode(F, 10, 6);
> G := StabilizerGroup(Q);
> IsAbelian(G);
true
```

Example H157E16

In order to make stabilizer groups from distinct codes compatible with one another, the groups must be created within the same super-structure. This is done by first creating a copy of the full error group, and then generating each instance of a stabilizer group as a subgroup.
In this example, the intersection of the stabilizer groups of two random codes is formed. An error group E which will be a common over group for the two stabilizer groups is first created.

```
> F<w> := GF(4);
> Q1 := RandomQuantumCode(F, 15, 8);
> Q2 := RandomQuantumCode(F, 15, 8);
>
> E := QuantumErrorGroup(Q1);
> S1 := StabilizerGroup(Q1, E);
> S2 := StabilizerGroup(Q2, E);
> #(S1 meet S2);
2
```

157.4 Inner Products and Duals

The functions described in this section use the symplectic inner product defined for quantum codes.

`SymplecticInnerProduct(v1, v2)`

`ExtendedFormat` BoolElt *Default :* `false`

Let $v1$ and $v2$ be two vectors belonging to the vector space $K^{(n)}$, where K is a finite field. This function returns the inner product of $v1$ and $v2$ with respect to the symplectic inner product. The symplectic inner product in extended format is defined by $(a|b)*(c|d) = ad-bc$, and its definition transfers naturally to the compact format.

For binary quantum codes whose compact format is over $GF(4)$, the symplectic inner product is given by `Trace`$(v_1 \cdot \overline{v}_2)$.

`SymplecticDual(C)`

`ExtendedFormat` BoolElt *Default :* `false`

The dual of the additive (or possibly linear) code C with respect to the symplectic inner product. By default, C is interpreted as being in the compact format (a length n code over $GF(q^2)$), but if `ExtendedFormat` is set to `true`, then it will be interpreted as being in extended format (a code of length $2n$ over $GF(q)$).

`IsSymplecticSelfDual(C)`

`ExtendedFormat` BoolElt *Default :* `false`

Return `true` if the code C is equal to its symplectic dual and `false` otherwise. By default, C is interpreted as being in the compact format (a length n code over $GF(q^2)$), but if `ExtendedFormat` is set to `true`, then it will be interpreted as being in extended format (a code of length $2n$ over $GF(q)$).

`IsSymplecticSelfOrthogonal(C)`

`ExtendedFormat` BoolElt *Default :* `false`

Return `true` if the code C is contained in its symplectic dual. By default, C is interpreted as being in the compact format (a length n code over $GF(q^2)$), but if `ExtendedFormat` is set to `true`, then it will be interpreted as being in extended format (a code of length $2n$ over $GF(q)$).

Example H157E17

Vectors which are symplectically orthogonal to one another can be used to construct symplectic self-orthogonal codes.

```
> F<w> := GF(4);
> V5 := VectorSpace(F, 5);
> v := V5 ! [1,0,w,0,1];
> w := V5 ! [w,1,0,w,w];
> SymplecticInnerProduct(v,w);
0
> C := AdditiveCode<F, GF(2), 5 | v, w>;
> C;
[5, 1 : 2] GF(2)-Additive Code over GF(2^2)
Generator matrix:
[  1   0   w   0   1]
[  w   1   0   w   w]
> D := SymplecticDual(C);
> D;
[5, 4 : 8] GF(2)-Additive Code over GF(2^2)
Generator matrix:
[  1   0   0   0   1]
[  w   0   0   0   w]
[  0   1   0   0   0]
[  0   w   0   0   1]
[  0   0   1   0   w]
[  0   0   w   0   0]
[  0   0   0   1   1]
[  0   0   0   w   0]
> C subset D;
true
> Q := QuantumCode(C);
> Q;
```

```
[[5, 3]] Quantum code over GF(2^2), stabilised by:
[  1   0   w   0   1]
[  w   1   0   w   w]
```

Example H157E18

Any vector over $GF(4)$ will be symplectically orthogonal to itself.

```
> V5 := VectorSpace(GF(4), 5);
> { SymplecticInnerProduct(v, v) : v in V5 };
{ 0 }
```

157.5 Weight Distribution and Minimum Weight

The weight distribution of a quantum code Q consists of three separate distributions:

- The weight distribution of the stabilizer code S.
- The weight distribution of the symplectic dual $S^\perp$ of S.
- The weight distribution of the codewords in $S^\perp \backslash S$. Note that this set is not a linear space.

For a quantum code Q with stabilizer code S, the weights of the undetectable errors are the weights of the codewords in $S^\perp \backslash S$.

For a quantum code to be considered *pure*, its minimum weight must be less than or equal to the weight of its stabilizer code.

`WeightDistribution(Q)`

Given a quantum code Q with stabiliser code S, return its weight distribution. Recall that the quantum weight distribution comprises the weight distributions of S, $S^\perp$ and $S^\perp \backslash S$. The function returns each distribution as a separate value. Each weight distribution is returned in the form of a sequence of tuples consisting of a weight and the number of code words of that weight.

Example H157E19

Looking at a small quantum code from the database of best known codes. Its first weight distribution is of its stabilizer code S, the second of its normalizer code $S^\perp$, and the final weight distribution is of those non-zero codewords in $S^\perp \backslash S$.

```
> F<w> := GF(4);
> Q := QECC(GF(4),6,3);
> Q;
[[6, 3, 2]] Quantum code over GF(2^2), stabilised by:
[  1   0   1   1   1   0]
[  w   w   w   w   w   w]
[  0   1   0   0   0   1]
```

```
> WD_S, WD_N, WD := WeightDistribution(Q);
> WD_S eq WeightDistribution(StabiliserCode(Q));
true
> WD_N eq WeightDistribution(NormaliserCode(Q));
true
> WD;
[ <2, 28>, <3, 56>, <4, 154>, <5, 168>, <6, 98> ]
```

`MinimumWeight(Q)`

`Method`	MonStgElt	*Default :* "*Auto*"
`RankLowerBound`	RngIntElt	*Default :* 0
`MaximumTime`	RngReSubElt	*Default :* ∞

For the quantum code Q with stabilizer code S, return the minimum weight of Q, which is the minimum weight of the codewords in $S^{\perp}\backslash S$. For self-dual quantum codes (those of dimension 0), the minimum weight is defined to be the minimum weight of S. The default algorithm is based on the minimum weight algorithm for classical linear codes which is described in detail in section 152.8.1. For a description of the algorithm and its variable argument parameters please consult the full description provided there. The minimum weight may alternatively be calculated by finding the complete weight distribution. This algorithm may be selected by setting the value of the variable argument `Method` to "`Distribution`".

Example H157E20

The verbose output can be used in long minimum weight calculations to estimate the remaining running time. The algorithm terminates once the lower bound reaches the upper bound. The example below finishes in a very short period of time.

```
> F<w> := GF(4);
> V5 := VectorSpace(F, 5);
> gens := [V5| [0,0,1,w,w], [0,1,1,w,1], [0,0,1,0,w^2],
>              [0,0,1,w,1], [0,0,1,0,1], [1,w,1,w,w^2],
>              [1,1,1,w^2,w], [0,1,w,1,w^2] ];
> Q := QuantumQuasiCyclicCode(gens : LinearSpan := true);
> Q:Minimal;
[[40, 30]] Quantum code over GF(2^2)
> SetVerbose("Code",true);
> MinimumWeight(Q);
Quantum GF(2)-Additive code over GF(4) of length 40 with 70 generators.
Lower Bound: 1, Upper Bound: 40
Constructed 2 distinct generator matrices
Total Ranks:     35  35
Relative Ranks:  35   5
Time Taken: 0.14
Starting search for low weight codewords... (reset timings)
```

```
Enumerating using 1 generator at a time:
     New codeword identified of weight 6, time 0.00
     New codeword identified of weight 4, time 0.00
     Discarding non-contributing rank 5 matrix
     New Total Ranks:     35
     New Relative Ranks:  35
   Completed Matrix  1:  lower =  2, upper =  4. Elapsed: 0.00s
Termination predicted with 3 generators at matrix 1
Enumerating using 2 generators at a time:
   Completed Matrix  1:  lower =  3, upper =  4. Elapsed: 0.00s
Termination predicted with 3 generators at matrix 1
predicting enumerating (1820) 60725 vectors (0.000000% of 40 40 code)
Enumerating using 3 generators at a time:
   Completed Matrix  1:  lower =  4, upper =  4. Elapsed: 0.03s
Final Results: lower = 4, upper = 4, Total time: 0.03
4
```

`IsPure(Q)`

Return `true` if Q is a *pure* quantum code. That is, if the minimum weight of Q is less than or equal to the minimum weight of its stabiliser code.

Example H157E21

Many good codes are impure, the purity of best known quantum codes of length 15 are investigated.

```
> F<w> := GF(4);
> n := 15;
> time {* IsPure(QECC(F, n, k)) : k in [1..n] *};
{* false^^10, true^^5 *}
Time: 0.410
```

157.6 New Codes From Old

`DirectSum(Q1, Q2)`

Given an $[[n_1, k_1, d_1]]$ quantum code Q_1, and an $[[n_2, k_2, d_2]]$ quantum code Q_2, return the $[[n_1+n_n, k_1+k_2, \min\{d_1, d_2\}]]$ quantum code which is their direct product.

`ExtendCode(Q)`

Given an $[[n, k, d]]$ quantum code Q, return the extended $[[n+1, k, d]]$ quantum code.

`ExtendCode(Q, m)`

Perform m extensions on the $[[n, k, d]]$ quantum code Q, returning the extended $[[n + m, k, d]]$ quantum code.

`PunctureCode(Q, i)`

Given a $[[n, k, d]]$ quantum code Q, and a coordinate position i, return the $[[n - 1, k, d' >= d - 1]]$ quantum code produced by puncturing at position i.

`PunctureCode(Q, I)`

Given a $[[n, k, d]]$ quantum code Q, and a set of coordinate positions I of size s, return the $[[n - s, k, d' >= d - s]]$ quantum code produced by puncturing at the positions in I.

`ShortenCode(Q, i)`

Given a $[[n, k, d]]$ quantum code Q, and a coordinate position i, return the $[[n - 1, k' >= k - 1, d' >= d]]$ quantum code produced by shortening at position i.

This process will not necessarily result in a valid (symplectic self-orthogonal) quantum code, and an error will be given if it fails.

`ShortenCode(Q, I)`

Given a $[[n, k, d]]$ quantum code Q, and a set of coordinate positions I of size s, return the $[[n - s, k' >= k - s, d' >= d]]$ quantum code produced by shortening at the positions in I.

This process will not necessarily result in a valid (symplectic self-orthogonal) quantum code, and an error will be given if it fails.

Example H157E22

Good quantum codes can be created by combining stabilizer codes, using methods which are not general enough to warrant a specific quantum code function. This example creates a $[[28, 8, 6]]$ quantum code from $[[14, 8, 3]]$ and $[[14, 0, 6]]$ quantum codes using a Plotkin sum. It relies on the stabilizer codes forming a subcode chain, as described in Theorem 12 in [CRSS98].

```
> F<w> := GF(4);
> V7 := VectorSpace(F, 7);
> v1 := V7 ! [1,0,0,0,0,0,0];
> v2 := V7 ! [w^2,1,w^2,w,0,0,w];
> Q1 := QuantumQuasiCyclicCode([v1, v2] : LinearSpan := true);
> _ := MinimumWeight(Q1);
> Q1:Minimal;
[[14, 0, 6]] self-dual Quantum code over GF(2^2)
>
> v1 := V7 ! [1,0,1,1,1,0,0];
> v2 := V7 ! [1,w^2,w,w,1,0,w^2];
> Q2 := QuantumQuasiCyclicCode([v1, v2] : LinearSpan := true);
> _ := MinimumWeight(Q2);
> Q2:Minimal;
[[14, 8, 3]] Quantum code over GF(2^2)
>
> S1 := StabilizerCode(Q1);
> S2 := StabilizerCode(Q2);
> S2 subset S1;
true
>
> S3 := PlotkinSum(SymplecticDual(S1), S2);
> Q3 := QuantumCode(S3);
> _ := MinimumWeight(Q3);
> Q3:Minimal;
[[28, 8, 6]] Quantum code over GF(2^2)
```

157.7 Best Known Quantum Codes

An $[[n, k]]$ quantum stabiliser code Q is said to be a *best known* $[[n, k]]$ *quantum code* (BKQC) if C has the highest minimum weight among all known $[[n, k]]$ quantum codes. The acronym QECC (Quantum Error Correcting Code) will be used to more easily distinguish from the best known linear codes database (BKLC).

MAGMA currently has a database for binary quantum codes, though it should be noted that these codes are considered to be over the alphabet $GF(4)$, not $GF(2)$. The database for codes over $GF(4)$ currently contains constructions of all best known quantum codes of length 35. This includes self-dual quantum codes up to length 35, which are stored in the database as dimension 0 quantum codes.

Quantum codes of length up to 12 are optimal, in the sense that their minimum weights meet the upper bound. Thus the user has access to 665 best-known binary quantum codes.

The MAGMA QECC database uses the tables of bounds and constructions compiled by Markus Grassl (Karlsruhe), available online at [Gra], which are based on the results in [CRSS98]. Good codes have also been contributed by Eric Rains and Zlatko Varbanov.

The user can display the method used to construct a particular QECC code through use of a verbose mode, triggered by the verbose flag `BestCode`. When it is set to `true`, all of the functions in this section will output the steps involved in each code they construct.

`QECC(F, n, k)`

`BKQC(F, n, k)`

`BestKnownQuantumCode(F, n, k)`

Given a finite field F, and positive integers n and k such that $k \leq n$, return an $[[n, k]]$ quantum code over F which has the largest minimum weight among all known $[[n, k]]$ quantum codes. A second boolean return value signals whether or not the desired code exists in the database.

The database currently exists for $GF(4)$ (which are in fact binary quantum codes) up to length 35.

Example H157E23

The weight distribution of a small best known quantum code is calculated, verifying its minimum weight. Note that the *larger* the dimension of a quantum code, the easier it is to calculate its weight distribution.

```
> F<w> := GF(4);
> Q := QECC(F,25,16);
> Q:Minimal;
[[25, 16, 3]] Quantum code over GF(2^2)
> time WD_S, WD_N, WD := WeightDistribution(Q);
Time: 0.010
> WD_S;
[ <0, 1>, <1, 2>, <2, 1>, <14, 4>, <15, 16>, <16, 38>, <17, 79>, <18, 126>, <19,
129>, <20, 77>, <21, 27>, <22, 9>, <23, 3> ]
> WD_N;
[ <0, 1>, <1, 2>, <2, 1>, <3, 399>, <4, 6527>, <5, 75363>, <6, 707543>, <7,
5404369>, <8, 34084490>, <9, 180107319>, <10, 804255370>, <11, 3052443894>, <12,
9883860222>, <13, 27348684334>, <14, 64649758926>, <15, 130286413858>, <16,
222912028997>, <17, 321704696752>, <18, 387985433701>, <19, 385943417035>, <20,
310898936275>, <21, 197566276671>, <22, 95232787563>, <23, 32688613821>, <24,
7109768160>, <25, 735493959> ]
> WD;
[ <3, 399>, <4, 6527>, <5, 75363>, <6, 707543>, <7, 5404369>, <8, 34084490>, <9,
180107319>, <10, 804255370>, <11, 3052443894>, <12, 9883860222>, <13,
27348684334>, <14, 64649758922>, <15, 130286413842>, <16, 222912028959>, <17,
321704696673>, <18, 387985433575>, <19, 385943416906>, <20, 310898936198>, <21,
```

```
197566276644>, <22, 95232787554>, <23, 32688613818>, <24, 7109768160>, <25,
735493959> ]
```

So the $[[25, 16]]$ code is impure, and has a minimum distance of 3.

Example H157E24

Unlike linear codes, dimension 0 quantum codes are non-trivial and are the subject of much study. These are the *self-dual* quantum codes, which form a special subclass of quantum stabilizer codes. It can be seen that a length n self-dual code is described by a $n \times n$ generator matrix, an indication of the non-triviality of its structure.

```
> F<w> := GF(4);
> C := QECC(GF(4),8, 0);
> C;
[[8, 0, 4]] self-dual Quantum code over GF(2^2), stabilised by:
[  1   0   0   1   0   1   1   0]
[  w   0   0   w   0   w   w   0]
[  0   1   0   1   0   1   0   1]
[  0   w   0   w   0   w   0   w]
[  0   0   1   1   0   0   1   1]
[  0   0   w   w   0   0   w   w]
[  0   0   0   0   1   1   1   1]
[  0   0   0   0   w   w   w   w]
```

Example H157E25

The verbose flag `BestCode` will show the method by which the best code is constructed in the database. In this example the construction of a $[[25, 11, 4]]$ quantum code is described.

```
> SetVerbose("BestCode",true);
> F<w> := GF(4);
> Q := QECC(F,25,11);
Construction of a [[ 25 , 11 , 4 ]] Quantum Code:
[1]:  [[40, 30, 4]] Quantum code over GF(2^2)
        QuasiCyclicCode of length 40 stacked to height 2 with generating
        polynomials: 1,  w^2*x^4 + w*x^3 + w^2*x^2 + w*x + w^2,  x^4 + w^2*x^2 +
        w^2*x + w^2,  x^4 + w*x^3 + x^2,  w*x^4 + x^3 + w^2*x,  w*x^4 + x^3 +
        x^2 + x,  w^2*x^4 + x^2 + w*x,  x^4 + w^2*x^3 + w^2*x^2 + x + 1,  w,
        x^4 + w^2*x^3 + x^2 + w^2*x + 1,  w*x^4 + x^2 + x + 1,  w*x^4 + w^2*x^3
        + w*x^2,  w^2*x^4 + w*x^3 + x,  w^2*x^4 + w*x^3 + w*x^2 + w*x,  x^4 +
        w*x^2 + w^2*x,  w*x^4 + x^3 + x^2 + w*x + w
[2]:  [[21, 11, 4]] Quantum code over GF(2^2)
        Shortening of [1] at { 2, 3, 4, 6, 8, 9, 10, 12, 13, 14, 15, 16, 18, 19,
        21, 24, 28, 34, 37 }
[3]:  [[25, 11, 4]] Quantum code over GF(2^2)
        ExtendCode [2] by 4
> Q:Minimal;
```

```
[[25, 11, 4]] Quantum code over GF(2^2)
```

157.8 Best Known Bounds

Along with the database of best known quantum codes in the previous section, there is also a database of best known upper and lower bounds on the maximal possible minimum weights of quantum codes. The upper bounds are not currently known with much accuracy, while the lower bounds match the minimum weights of the best known quantum codes database.

`QECCLowerBound(F, n, k)`

Return the best known lower bound on the maximal minimum distance of $[[n, k]]$ quantum codes over F. The bounds are currently available for binary quantum codes (which corresponds to $F = GF(4)$) up to length 35.

`QECCUpperBound(F, n, k)`

Return the best known upper bound on the minimum distance of $[[n, k]]$ quantum codes over F. The bounds are currently available for binary quantum codes (which corresponds to $F = GF(4)$) up to length 35.

Example H157E26

The best known lower bound on the minimum weight will always correspond to the best known quantum code from the MAGMA database. In this example the first code is in fact optimal, while the second one does not meet the upper bound, and so there is a theoretical possibility of an improvement.

```
> F<w> := GF(4);
> Q1 := QECC(F, 20, 10);
> Q1:Minimal;
[[20, 10, 4]] Quantum code over GF(2^2)
> QECCLowerBound(F, 20, 10);
4
> QECCUpperBound(F, 20, 10);
4
>
> Q2 := QECC(F, 25, 13);
> Q2:Minimal;
[[25, 13, 4]] Quantum code over GF(2^2)
> QECCLowerBound(F, 25, 13);
4
> QECCUpperBound(F, 25, 13);
5
```

157.9 Automorphism Group

Automorphisms acting on a quantum code are a slight generalization of those which act on the underlying additive stabilizer code. Automorphisms consists of both a permutation action on the columns of a stabilizer code, combined with a monomial action on the individual columns which permute the values.

The automorphism group of a length n additive stabilizer code over $\mathbf{F}_4$ is a subgroup of $Z_3 \wr Sym(n)$ of order $3 * n!$. However the automorphism group of the quantum code it generates is a subgroup of $Sym(3) \wr Sym(n)$ of order $3! * n!$ because of the more general action on the values in the columns.

In MAGMA automorphisms are returned as permutations, either as length $3n$ permutations for the full monomial action on a code, or as length n permutations when the automorphism is restricted to only the permutation action on the columns.

`AutomorphismGroup(Q)`

The automorphism group of the quantum code Q. Currently this function only applies to binary quantum codes.

`PermutationGroup(Q)`

The subgroup of the automorphism group of the quantum code Q consisting of those automorphisms which permute the coordinates of codewords. Currently this function only applies to binary quantum codes.

Example H157E27

The full automorphism group and its subgroup of coordinate permutations are calculated for the dodecacode.

```
> F<w> := GF(4);
> Q := Dodecacode();
> Q;
[[12, 0, 6]] self-dual Quantum code over GF(2^2), stabilised by:
[  1   0   0   0   0   0 w^2 w^2   0   w   1   w]
[  w   0   0   0   0   0   w   0   w   w   w   1]
[  0   1   0   0   0   0   1   0   1 w^2 w^2   1]
[  0   w   0   0   0   0   0   w   1   w   w   w]
[  0   0   1   0   0   0   0   1   1   1 w^2 w^2]
[  0   0   w   0   0   0 w^2   1 w^2   w   w   0]
[  0   0   0   1   0   0   w   w   1   1   0   1]
[  0   0   0   w   0   0   w   1   w w^2 w^2   0]
[  0   0   0   0   1   0   w   w   w   0   1   w]
[  0   0   0   0   w   0 w^2   1   1 w^2   0   w]
[  0   0   0   0   0   1 w^2 w^2 w^2   1   0 w^2]
[  0   0   0   0   0   w   w   1   0   1   w   w]
>
> AutomorphismGroup(Q);
Permutation group acting on a set of cardinality 36
Order = 648 = 2^3 * 3^4
```

```
    (1, 4, 32)(2, 5, 33)(3, 6, 31)(7, 13, 29)(8, 14, 30)(9, 15, 28)(10, 35, 22)
        (11, 36, 23)(12, 34, 24)(16, 19, 26)(17, 20, 27)(18, 21, 25)
    (4, 23, 8, 29, 10, 20, 18, 36, 32)(5, 24, 9, 30, 11, 21, 16, 34, 33)
        (6, 22, 7, 28, 12, 19, 17, 35, 31)(13, 14, 15)(25, 27, 26)
    (7, 35)(8, 36)(9, 34)(10, 20)(11, 21)(12, 19)(13, 26)(14, 27)(15, 25)
        (16, 30)(17, 28)(18, 29)(22, 31)(23, 32)(24, 33)
    (4, 29, 18)(5, 30, 16)(6, 28, 17)(7, 19, 31)(8, 20, 32)(9, 21, 33)
        (10, 36, 23)(11, 34, 24)(12, 35, 22)
> PermutationGroup(Q);
Permutation group acting on a set of cardinality 12
    (1, 7, 9, 3, 5, 11)(2, 8, 10, 4, 6, 12)
    (1, 2)(3, 4)(5, 10)(6, 9)(7, 12)(8, 11)
    (2, 4)(5, 9)(6, 12)(7, 11)(8, 10)
```

Example H157E28

The automorphism group for a quantum code is larger than that of its stabilizer code. In this example that is shown for the Hexacode.

```
> F<w> := GF(4);
> Q := Hexacode();
> Q:Minimal;
[[6, 0, 4]] self-dual Quantum code over GF(2^2)
> A_Q := AutomorphismGroup(Q);
> A_Q;
Permutation group A_Q acting on a set of cardinality 18
Order = 2160 = 2^4 * 3^3 * 5
    (1, 4)(2, 6)(3, 5)(7, 8)(10, 12)(13, 14)(17, 18)
    (2, 3)(5, 6)(7, 8)(10, 18)(11, 16)(12, 17)(13, 14)
    (4, 7)(5, 8)(6, 9)(13, 17)(14, 16)(15, 18)
    (7, 13)(8, 14)(9, 15)(10, 17)(11, 16)(12, 18)
    (7, 12)(8, 10)(9, 11)(13, 18)(14, 17)(15, 16)
> S := StabilizerCode(Q);
> A_S := AutomorphismGroup(S);
> A_S;
Permutation group A_S acting on a set of cardinality 18
Order = 180 = 2^2 * 3^2 * 5
    (1, 4)(2, 5)(3, 6)(7, 13)(8, 14)(9, 15)
    (4, 7, 12)(5, 8, 10)(6, 9, 11)(13, 15, 14)(16, 18, 17)
    (4, 6, 5)(7, 14, 11)(8, 15, 12)(9, 13, 10)(16, 18, 17)
> A_S subset A_Q;
true
```

157.10 Hilbert Spaces

In this first release, MAGMA offers a basic package for creating and computing with quantum Hilbert spaces. A Hilbert space in MAGMA can either be *densely* or *sparsely* represented, depending on how many qubits are required and how dense the desired quantum states will be. While a dense representation has a speed advantage in computations, the sparse representation uses less memory. Currently there are capabilities for doing basic unitary transformations and manipulations of quantum states.

In future versions, functionality will be added for more complex unitary transformations and measurements, allowing for a more general simulation of quantum computations. There will also be machinery for encoding quantum states using quantum error correcting codes, and testing their effectiveness by simulating a noisy quantum channel and decoding the results.

`HilbertSpace(F, n)`

`IsDense` BOOLELT *Default :*

Given a complex field F and a positive integer n, return then quantum Hilbert Space on n qubits over F.

If the variable argument `IsDense` is set to either `true` or `false` then return a densely or sparsely represented quantum space respectively. If no value is set for `IsDense` then MAGMA will decide automatically.

`Field(H)`

Given a Hilbert space H, return the complex field over which the coefficients of states of H are defined.

`NumberOfQubits(H)`

`Nqubits(H)`

Given a Hilbert space H, return the number of qubits which comprises the space.

`Dimension(H)`

Given a Hilbert space H, return its dimension. This is 2^n, where n is the number of qubits of H.

`IsDenselyRepresented(H)`

Return `true` if the quantum Hilbert space H uses a dense representation.

`H1 eq H2`

Return `true` if the Hilbert spaces are equal.

`H1 ne H2`

Return `true` if the Hilbert spaces are not equal.

Example H157E29

A Hilbert space over 5 qubits will by default be a densely represented quantum space. It can however be manually chosen to use a sparse representation, it can be seen that these two space are not considered equal.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 5);
> H;
A densely represented Hilbert Space on 5 qubits to precision 4
> Dimension(H);
32
> IsDenselyRepresented(H);
true
>
> H1 := HilbertSpace(F, 5 : IsDense := false);
> H1;
A sparely represented Hilbert Space on 5 qubits to precision 4
> IsDenselyRepresented(H1);
false
> H eq H1;
false
```

157.10.1 Creation of Quantum States

`QuantumState(H, v)`

`QuantumState(H, v)`

Given a Hilbert space H and coefficients v (which can be either a dense or a sparse vector), of length equal to the dimension of H, then return the quantum state in H defined by v.

`H ! i`

Return the i-th quantum basis state of the Hilbert space H. This corresponds to the basis state whose qubits giving a binary representation of i.

`H ! s`

Given a sequence s of binary values, whose length is equal to the number of qubits of the Hilbert space H, return the quantum basis state corresponding to s.

`SetPrintKetsInteger(b)`

Input is a boolean value b, which controls a global variable determining the way quantum states are printed. If set to `false` (which is the default) then values in basis kets will be printed as binary sequences such as $|1010\rangle$. If set to `true` then basis kets will be printed using integer values to represent the binary sequences, the previous example becoming $|5\rangle$.

Example H157E30

One way to create a quantum state is to specify each coefficient of the state with a vector of length equal to the dimension of the Hilbert space.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 4);
> KS := KSpace(F, Dimension(H));
> v := KS! [F| i,    1,    0, -i,
>             2,    0,    0, 1+i,
>             -i-1, -3*i, 7, 0.5,
>             2.5*i, 0,   0, 1.2];
> v;
(1.000*i 1.000 0.0000 -1.000*i 2.000 0.0000 0.0000 1.000 + 1.000*i
    -1.000 - 1.000*i -3.000*i 7.000 0.5000 2.500*i 0.0000 0.0000
    1.200)
> e := QuantumState(H, v);
> e;
1.000*i|0000> + |1000> - 1.000*i|1100> + 2.000|0010> + (1.000 +
1.000*i)|1110> - (1.000 + 1.000*i)|0001> - 3.000*i|1001> + 7.000|0101>
+ 0.5000|1101> + 2.500*i|0011> + 1.200|1111>
```

Example H157E31

Quantum states can be created by combining basis states, input as either integer values or binary sequences.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 12);
> Dimension(H);
4096
> e1 := H!1 + (1+i)*(H!76) - H!3000;
> e1;
|100000000000> + (1.000 + 1.000*i)|001100100000> - |000111011101>
> e2 := H![1,0,1,1,1,0,0,0,1,1,0,0] - H![1,1,0,1,0,0,0,0,1,1,0,1];
> e2;
|101110001100> - |110100001101>
```

By using the function `SetPrintKetsInteger` basis states can also be printed as either integer values of binary sequences.

```
> SetPrintKetsInteger(true);
> e1;
|1> + (1.000 + 1.000*i)|76> - |3000>
> e2;
|797> - |2827>
```

157.10.2 Manipulation of Quantum States

`a * e`

Given a complex scalar value a, multiply the coefficients of the quantum state e by a.

`-e`

Negate all coefficients of the quantum state e.

`e1 + e2`
`e1 - e2`

Addition and subtraction of the quantum states e_1 and e_2.

`Normalisation(e)`
`Normalisation(~e)`
`Normalization(e)`
`Normalization(~e)`

Normalize the coefficients of the quantum state e, giving an equivalent state whose normalization coefficient is equal to one. Available either as a procedure or a function.

`NormalisationCoefficient(e)`
`NormalizationCoefficient(e)`

Return the normalisation coefficient of the quantum state e

`e1 eq e2`

Return `true` if and only if the quantum states e_1 and e_2 are equal. States are still considered equal if they have different normalizations.

`e1 ne e2`

Return `true` if and only if the quantum states e_1 and e_2 are not equal. States are still considered equal if they have different normalizations.

Example H157E32

Although a quantum state can be expressed with any normalisation, in reality a quantum state occupies a ray in a Hilbert space. So two quantum states are still considered equal if they lie on the same ray.

```
> F<i> := ComplexField(8);
> H := HilbertSpace(F, 1);
> e := H!0 + H!1;
> e;
|0> + |1>
> NormalisationCoefficient(e);
2.0000000
> e1 := Normalisation(e);
> e1;
0.70710678|0> + 0.70710678|1>
> NormalisationCoefficient(e1);
0.99999999
> e eq e1;
true
```

157.10.3 Inner Product and Probabilities of Quantum States

`InnerProduct(e1, e2)`

Return the inner product of the quantum states e_1 and e_2.

`ProbabilityDistribution(e)`

Return the probability distribution of the quantum state as a vector over the reals.

`Probability(e, i)`

Return the probability of basis state i being returned as the result of a measurement on the quantum state e.

`Probability(e, v)`

Given a binary vector v of length equal to the number of qubits in the quantum state e, return the probability of basis state corresponding to v being returned as the result of a measurement on e.

`PrintProbabilityDistribution(e)`

Print the probability distribution of the quantum state.

`PrintSortedProbabilityDistribution(e)`

`Max`	RngIntElt	*Default :* ∞
`MinProbability`	RngIntElt	*Default :* 0

Print the probability distribution of the quantum state in sorted order, with the most probable states printed first.

If the variable argument `Max` is set to a positive integer, then it will denote the maximum number of basis states to be printed.

If the variable argument `MinProbability` is set to some integer between 1 and 100, then it will denote the minimum probability of any basis state to be printed. This is useful for investigating those basis states which will are the likely results of any measurement.

Example H157E33

From a quantum state it is possible to either access the full probability distribution, or the probabilities of individual basis states.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 3);
> e := -0.5*H!0 + 6*i*H!3 + 7*H!4 - (1+i)*H!7;
> ProbabilityDistribution(e);
(0.002865 0.0000 0.0000 0.4126 0.5616 0.0000 0.0000 0.02292)
> Probability(e, 0);
0.002865
> Probability(e, 1);
0.0000
```

It is also possible to print out the full probability distribution.

```
> PrintProbabilityDistribution(e);
Non-zero probabilities:
|000>:  0.2865%
|110>:  41.26%
|001>:  56.16%
|111>:  2.292%
```

Example H157E34

It is usually only those basis states with large probabilities that are of interest. With the function `PrintSortedProbabilitydistribution` these basis states can be identified.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 4);
> KS := KSpace(F, 2^4);
> v := KS! [F| i,   11,   0, -3*i,
>              2,   0,    0, 6+i,
>             -i-1, -3*i, 7, -0.5,
>             2.5*i, 0,   0, 9.2];
```

```
> e := QuantumState(H, v);
> e;
1.000*i|0000> + 11.00|1000> - 3.000*i|1100> + 2.000|0010> + (6.000 +
1.000*i)|1110> - (1.000 + 1.000*i)|0001> - 3.000*i|1001> + 7.000|0101>
- 0.5000|1101> + 2.500*i|0011> + 9.200|1111>
> PrintSortedProbabilityDistribution(e);
Non-zero probabilities:
|1000>:         37.45%
|1111>:         26.19%
|0101>:         15.16%
|1110>:         11.45%
|1100>:         2.785%
|1001>:         2.785%
|0011>:         1.934%
|0010>:         1.238%
|0001>:         0.6190%
|0000>:         0.3095%
|1101>:         0.07737%
```

A useful way to isolate the important basis states is to provide a minimum cutoff probability.

```
> PrintSortedProbabilityDistribution(e: MinProbability := 15);
Non-zero probabilities:
|1000>:         37.45%
|1111>:         26.19%
|0101>:         15.16%
Reached Minimum Percentage
```

Another way is to supply the maximum number basis states that should be printed. A combination of these methods can also be used

```
> PrintSortedProbabilityDistribution(e: Max := 6);
Non-zero probabilities:
|1000>:         37.45%
|1111>:         26.19%
|0101>:         15.16%
|1110>:         11.45%
|1100>:         2.785%
|1001>:         2.785%
Reached Maximum count
```

157.10.4 Unitary Transformations on Quantum States

In this first release MAGMA offers a small selection of unitary transformations on quantum states. In future versions this list will be expanded to include more complex operations.

`BitFlip(e, k)`

`BitFlip(~e, k)`

Flip the value of the k-th qubit of the quantum state e.

`BitFlip(e, B)`

`BitFlip(~e, B)`

Given a set of positive integers B, flip the value of the qubits of the quantum state e indexed by the entries in B.

`PhaseFlip(e, k)`

`PhaseFlip(~e, k)`

Flip the phase on the k-th qubit of the quantum state e.

`PhaseFlip(e, B)`

`PhaseFlip(~e, B)`

Given a set of positive integers B, flip the phase on the qubits of the quantum state e indexed by the entries in B.

`ControlledNot(e, B, k)`

`ControlledNot(~e, B, k)`

Flip the k-th bit of the quantum state e if all bits contained in B are set to 1.

`HadamardTrasformation(e)`

`HadamardTrasformation(~e)`

Perform a Hadamard transformation on the quantum state e, which must be densely represented.

Example H157E35

The behaviours of several of the available unitary transformations are displayed on a quantum state.

```
> F<i> := ComplexField(4);
> H := HilbertSpace(F, 4);
> e := H!0 + H!3 + H!6 + H!15;
> PhaseFlip(~e, 4); e;
|0000> + |1100> + |0110> - |1111>
> ControlledNot(~e, {1,2}, 4); e;
|0000> + |0110> - |1110> + |1101>
> BitFlip(~e, 2); e;
```

```
|0100> + |0010> - |1010> + |1001>
> ControlledNot(~e, {2}, 3); e;
|0010> - |1010> + |0110> + |1001>
```

157.11 Bibliography

[**CRSS98**] A. Robert Calderbank, Eric M. Rains, P. W. Shor, and Neil J. A. Sloane. Quantum error correction via codes over GF(4). *IEEE Trans. Inform. Theory*, 44(4): 1369–1387, 1998.

[**CS96**] A. R. Calderbank and P. W. Shor. Good quantum error-correcting codes exist. *Phys. Rev. A*, 54:2551–2577, 1996.

[**Dan05**] D. E. Danielsen. On-self dual quantum codes, graphs, and Boolean functions. Master's thesis, University of Bergen, 2005.

[**Gra**] Markus Grassl. Bounds on the minimum distance of quantum codes. URL:http://iaks-www.ira.uka.de/home/grassl/QECC/.

[**Sho94**] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science (Santa Fe, NM, 1994)*, pages 124–134. IEEE Comput. Soc. Press, Los Alamitos, CA, 1994.

[**Sho95**] P. W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:2493–2496, 1995.

[**Ste96a**] A. M. Steane. Error correcting codes in quantum theory. *Phys. Rev. Lett.*, 77(5):793–797, 1996.

[**Ste96b**] Andrew Steane. Multiple-particle interference and quantum error correction. *Proc. Roy. Soc. London Ser. A*, 452(1954):2551–2577, 1996.

PART XXII
CRYPTOGRAPHY

158 PSEUDO-RANDOM BIT SEQUENCES

Chapter 158
PSEUDO-RANDOM BIT SEQUENCES

158.1 Introduction

MAGMA provides some tools for the creation and analysis of pseudo-random bit sequences. The universe of these sequences is generally $\mathbf{F}_2$. However, some functions, such as `BerlekampMassey`, may be applied to sequences defined over arbitrary finite fields.

158.2 Linear Feedback Shift Registers

For a linear feedback shift register (LFSR) of length L, initial state $s_0, \ldots, s_{L-1} \in \mathbf{F}_q$, and connection polynomial $C(D) = 1 + c_1 D + c_2 D^2 + \ldots + c_L D^L$ (also over $\mathbf{F}_q$), the j'th element of the sequence is computed as $s_j = -\sum_{i=1}^{L} c_i s_{j-i}$ for $j \geq L$.

`LFSRSequence(C, S, t)`

Computes the first t sequence elements of the LFSR with connection polynomial C and initial state the sequence S (thus, the length of the LFSR is assumed to be the length of S). C must be at least degree 1, its coefficients must come from the same finite field as the universe of S, and its constant coefficient must be 1. Also, the sequence S must have at least as many terms as the degree of C.

`LFSRStep(C, S)`

Computes the next state of the LFSR having connection polynomial C and current state the sequence S (thus, the length of the LFSR is assumed to be the length of S). C must be at least degree 1, its coefficients must come from the same finite field as the universe of S, and its constant coefficient must be 1. Also, the sequence S must have at least as many terms as the degree of C.

`BerlekampMassey(S)`

`ConnectionPolynomial(S)`

`CharacteristicPolynomial(S)`

Given a sequence S of elements from $\mathbf{F}_q$, return the connection polynomial $C(D)$ and the length L of a LFSR that generates the sequence S.

Note that it is possible that the `BerlekampMassey` will return a singular LFSR (i.e. the degree of $C(D)$ is less than L), and therefore one must be sure to use the first L elements of S to regenerate the sequence.

Example H158E1

We first create a sequence and then use `BerlekampMassey` to get the connection polynomial and its length:

```
> S:= [GF(2)| 1,1,0,1,0,1,1,1,0,0,1,0];
> C<D>, L := BerlekampMassey(S);
> C;
D^3 + D^2 + 1
> L;
5
```

Now create a new sequence T containing the first L elements of S, and reconstruct the sequence from $C(D)$ and T.

```
> T := S[1..L];
> LFSRSequence(C, T, #S);
[ 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0 ]
```

`ShrinkingGenerator(C1, S1, C2, S2, t)`

Outputs a sequence of t bits from the shrinking generator having connection polynomials C_1 and C_2 and initial states sequences S_1 and S_2 (thus, the lengths of the LFSRs are assumed to be the lengths of S_1 and S_2). Bits are represented as elements from $\mathbf{F}_2$. Polynomial coefficients and sequence elements must be from $\mathbf{F}_2$. The degrees of the connection polynomials must be at least 1 and their trailing coefficients must be 1. The number of elements in the initial states must be at least as large as the degrees of the corresponding connection polynomials.

158.3 Number Theoretic Bit Generators

`RandomSequenceRSA(b, t)`

Generates a sequence of t bits using the RSA pseudo-random bit generator with an RSA modulus of approximately b bits in length. The modulus n is computed by finding (pseudo-)random primes with the `RandomPrime` function. If $\gcd(\phi(n),\ 3)$ is 1, then the exponent 3 will be used. Otherwise, a (pseudo-)random exponent e is chosen so that $\gcd(\phi(n),\ e) = 1$. The seed is also chosen as a (pseudo-)random number modulo n. Bits are represented as elements of $\mathbf{F}_2$.

Example H158E2

The code below counts the number of 1's that appear in a sequence of 1000 bits generated from a 100-bit RSA modulus.

```
> Z := Integers();
> &+[ Z | b : b in RandomSequenceRSA(100, 1000) ];
497
```

`RandomSequenceRSA(n, e, s, t)`

Generates a sequence of t bits using the RSA pseudo-random bit generator with modulus n, exponent e, and seed value s. Bits are represented as elements from $\mathbf{F}_2$. The integer n must be larger than 1.

`RSAModulus(b)`

Returns an RSA Modulus n of b bits in length, and an exponent e such that `Gcd(EulerPhi(`n`),`e`)=1`. The resulting values can be used to generate random bits with the function `RandomSequenceRSA`. The argument b must be at least 16. *Warning*: RSA Moduli generated by MAGMA should not be used for real world cryptographic applications. Such applications require a "true random" source to seed the random number generator. MAGMA's method of seeding may not be sufficiently random to meet the requirements of cryptographic standards.

`RSAModulus(b, e)`

Returns an RSA Modulus n of b bits in length such that `Gcd(EulerPhi(`n`),`e`)=1`. The resulting value can be used with e for the exponent to generate random bits with the function `RandomSequenceRSA`. The argument b must be at least 16. The argument e must be odd and must also be in the range $1 < e < 2^b$. *Warning*: RSA Moduli generated by MAGMA should not be used for real world cryptographic applications. Such applications require a "true random" source to seed the random number generator. MAGMA's method of seeding may not be sufficiently random to meet the requirements of cryptographic standards.

`RandomSequenceBlumBlumShub(b, t)`

`BlumBlumShub(b, t)`

Generates a sequence of t bits using the Blum-Blum-Shub pseudo-random bit generator with a Blum-Blum-Shub modulus of approximately b bits in length. The modulus n is computed within MAGMA by finding (pseudo-)random primes with the `RandomPrime` function (the condition being that the primes are congruent to 3 mod 4). The seed is chosen as a (pseudo-)random number modulo n. Bits are represented as elements from $\mathbf{F}_2$. b must be at least 16.

`RandomSequenceBlumBlumShub(n, s, t)`

`BlumBlumShub(n, s, t)`

Generates a sequence of t bits using the Blum-Blum-Shub pseudo-random bit generator with modulus n and seed value s. Bits are represented as elements from $\mathbf{F}_2$. The argument n must be larger than 1 and $\gcd(s, n)$ must be 1.

`BBSModulus(b)`

`BlumBlumShubModulus(b)`

Returns a Blum-Blum-Shub Modulus b bits in length. The resulting value can be used to generate random bits with the function `RandomSequenceBlumBlumShub`. The argument b must be at least 16. *Warning*: Blum-Blum-Shub Moduli generated by MAGMA should not be used for real world cryptographic applications. Such applications require a "true random" source to seed the random number generator. MAGMA's method of seeding may not be sufficiently random to meet the requirements of cryptographic standards.

158.4 Correlation Functions

`AutoCorrelation(S, t)`

Computes the autocorrelation of a sequence S, where S must have universe $\mathbf{F}_2$. The autocorrelation is defined to be

$$C(t) = \sum_{i=1}^{L} (-1)^{S[i]+S[i+t]}$$

where L is the length of the sequence, and the values of $S[i+t]$ wrap around to the beginning of the sequence when $i + t > L$.

Example H158E3

It is well known that the LFSR's with maximal periods have nice autocorrelation properties. This is illustrated below.

```
> C<D> := PrimitivePolynomial (GF(2), 5);
> C;
D^5 + D^2 + 1
> s := [GF(2)|1,1,1,1,1];
> t := LFSRSequence(C, s, 31);
> t;
[ 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0,
1, 1, 0, 0, 0 ]
> AutoCorrelation (t, 2);
-1
```

`CrossCorrelation(S1, S2, t)`

Computes the crosscorrelation of two binary sequences S_1 and S_2, where S_1 and S_2 must each have universe $\mathbf{F}_2$, and they must have the same length L. The crosscorrelation is defined to be:

$$C(t) = \sum_{i=1}^{L} (-1)^{S_1[i]+S_2[i+t]}$$

and the values of $S_2[i+t]$ wrap around to the beginning of the sequence when $i + t > L$.

158.5 Decimation

`Decimation(S, f, d)`

Given a binary sequence S, and integers f and d, return the decimation of S. This is the sequence containing elements $S[f]$, $S[f+d]$, $S[f+2d]$, ... where the indices in S are interpreted with wrap-around as integers between 1 and $\#S$.

`Decimation(S, f, d, t)`

Decimation of the sequence S. Returns a new sequence containing the first t elements of $S[f]$, $S[f+d]$, $S[f+2d]$, ... where the indices in S are interpreted with wrap-around as integers between 1 and $\#S$.

Example H158E4

Given a primitive polynomial over $\mathbf{F}_q$, one can obtain another primitive polynomial by decimating an LFSR sequence obtained from the initial polynomial. This is demonstrated in the code below.

```
> K := GF(7);
> C<D> := PrimitivePolynomial(K, 2);
> C;
D^2 + 6*D + 3
```

In order to generate an LFSR sequence, we must first multiply this polynomial by a suitable constant so that the trailing coefficient becomes 1.

```
> C := C * Coefficient(C,0)^-1;
> C;
5*D^2 + 2*D + 1
```

We are now able to generate an LFSR sequence of length $7^2 - 1$. The initial state can be anything other than $[0, 0]$.

```
> t := LFSRSequence (C, [K| 1,1], 48);
> t;
[ 1, 1, 0, 2, 3, 5, 3, 4, 5, 5, 0, 3, 1, 4, 1, 6, 4, 4, 0, 1, 5, 6, 5, 2, 6, 6,
```

```
0, 5, 4, 2, 4, 3, 2, 2, 0, 4, 6, 3, 6, 1, 3, 3, 0, 6, 2, 1, 2, 5 ]
```

We decimate the sequence by a value d having the property $\gcd(d, 48) = 1$.

```
> t := Decimation(t, 1, 5);
> t;
[ 1, 5, 0, 6, 5, 6, 4, 4, 3, 1, 0, 4, 1, 4, 5, 5, 2, 3, 0, 5, 3, 5, 1, 1, 6, 2,
0, 1, 2, 1, 3, 3, 4, 6, 0, 3, 6, 3, 2, 2, 5, 4, 0, 2, 4, 2, 6, 6 ]
> B := BerlekampMassey(t);
> B;
3*D^2 + 5*D + 1
```

To get the corresponding primitive polynomial, we multiply by a constant to make it monic.

```
> B := B * Coefficient(B, 2)^-1;
> B;
D^2 + 4*D + 5
> IsPrimitive(B);
true
```

PART XXIII
OPTIMIZATION

159 LINEAR PROGRAMMING

Chapter 159
LINEAR PROGRAMMING

159.1 Introduction

A Linear Program in n variables $x_1, \cdots, x_n$ with m constraints of the form

$$\sum_{j=1}^{n} a_j x_j \ \leq \ c$$

(the relations in any of the constraints may also be $=$ or $\geq$) may be represented in matrix form as:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad (\,REL\,) \quad \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$$

where $(\,REL\,)$ represents a componentwise relation between vectors, with each element $=$, $\leq$, or $\geq$.

Note that there is an additional implicit constraint, wherein all variables are assumed to be nonnegative.

We wish to find a solution $(\,x_i\,)$ that maximises (or minimises) the objective function:

$$\sum_{i=1}^{n} o_i x_i$$

Magma provides two methods for solving LP problems. The first is to set up suitable constraint matrices and then use an explicit LP solving function to solve the problem. The second involves creating an instance of the `LP` process, which is of category `LP`. Constraints are added and options set before calling `Solution` to get a solution to the problem.

All functions that actually solve an LP problem return a solution vector together with an integer code representing the state of the solution, provided by the lp_solve library. The codes are:

0 Optimal Solution

1 Failure

2 Infeasible problem

3 Unbounded problem

4 Failure

Magma supports LP problems over Integer, Rational, and Real rings. For Integer and Real problems, the solutions will be provided as Integer and Real vectors respectively. For LP problems provided in Rationals, the solution is a Real vector.

Linear programming in Magma is implemented using the lp_solve library written by Michel Berkelaar (michel@ics.ele.tue.nl). The library source may be found at `ftp://ftp.ics.ele.tue.nl/pub/lp_solve/`.

For further reference see [Naz87], [Chv83], [OH68] and [NW88].

159.2 Explicit LP Solving Functions

Each explicit LP solving function takes four arguments to represent an LP problem in n variables with m constraints:

1 `LHS` : $m \times n$ matrix, representing the left-hand-side coefficients of the m constraints.

2 `relations` : $m \times 1$ matrix over the same ring as `LHS`, representing the relations for each constraint, with a positive entry representing $\geq$, a zero entry representing $=$, and a negative entry representing $\leq$.

3 `RHS` : $m \times 1$ matrix over the same ring as `LHS`, representing the right-hand-side values of the m constraints.

4 `objective` : $1 \times n$ matrix over the same ring as `LHS`, representing the coefficients of the objective function to be optimised.

Each function returns a vector representing an optimal solution to the problem, and an integer indicating the state of the solution, as described in the introduction.

```
MaximalSolution(LHS, relations, RHS, objective)
```

The vector maximising the LP problem, with an integer describing the state of the solution.

```
MinimalSolution(LHS, relations, RHS, objective)
```

The vector minimising the LP problem, with an integer describing the state of the solution.

```
MaximalIntegerSolution(LHS, relations, RHS, objective)
```

The integer vector maximising the LP problem, with an integer describing the state of the solution.

```
MinimalIntegerSolution(LHS, relations, RHS, objective)
```

The integer vector minimising the LP problem, with an integer describing the state of the solution.

```
MaximalZeroOneSolution(LHS, relations, RHS, objective)
```

The vector with each entry either zero or one maximising the LP problem, with an integer describing the state of the solution.

```
MinimalZeroOneSolution(LHS, relations, RHS, objective)
```

The vector with each entry either zero or one minimising the LP problem, with an integer describing the state of the solution.

Example H159E1

We solve the LP maximising

$$F(x, y) = 8x + 2y \quad x, y \in \mathbf{R}$$

subject to the constraints

$$10x + 21y \leq 156$$

$$2x + y \leq 22$$

```
> R := RealField( );
> lhs := Matrix(R, 2, 2, [10, 21, 2, 1]);
> rhs := Matrix(R, 2, 1, [156, 22]);
> rel := Matrix(R, 2, 1, [-1, -1]); // negative values - less-or-equal relation
> obj := Matrix(R, 1, 2, [8, 15]);
> MaximalSolution(lhs, rel, rhs, obj);
[9.562500000000000000 2.875000000000000888]
0
```

Example H159E2

We find solutions to the LP maximising

$$F(x_1, \cdots, x_7) = 592x_1 + 381x_2 + 273x_3 + 55x_4 + 48x_5 + 37x_6 + 23x_7$$

subject to the constraint

$$3534x_1 + 2356x_2 + 2767x_3 + 589x_4 + 528x_5 + 451x_6 + 304x_7 \leq 119567$$

with $(x_1, \cdots, x_7)$ taking real values, integer values, and zero/one values.

```
> R := RealField( );
> lhs := Matrix(R, 1, 7, [3534, 2356, 2767, 589, 528, 451, 304]);
> rhs := Matrix(R, 1, 1, [119567]);
> rel := Matrix(R, 1, 1, [-1]);
> obj := Matrix(R, 1, 7, [592, 381, 273, 55, 48, 37, 23]);
> MaximalSolution(lhs, rel, rhs, obj);
[33.83333333333333570 0.E-92 0.E-92 0.E-92 0.E-92 0.E-92 0.E-92]
0
> MaximalIntegerSolution(lhs, rel, rhs, obj);
[33.00000000000000000 1.000000000000000000 0.E-92 1.000000000000000000 0.E-92
    0.E-92 0.E-92]
0
> MaximalZeroOneSolution(lhs, rel, rhs, obj);
[1.000000000000000000 1.000000000000000000 1.000000000000000000
    1.000000000000000000 1.000000000000000000 1.000000000000000000
    1.000000000000000000]
0
```

159.3 Creation of LP objects

`LPProcess(R, n)`

A Linear Program over the ring R in n variables.

Example H159E3

We create an LP representing a problem in 2 real variables:

```
> R := RealField( );
> L := LPProcess(R, 2);
> L;
LP <Real Field, 2 variables>
Minimising objective function: [0 0]
Subject to constraints:
Variables bounded above by: [ ]
Variables bounded below by: [ ]
Solving in integers for variables [ ]
```

159.4 Operations on LP objects

`AddConstraints(L, lhs, rhs)`

`Rel` MonStgElt *Default : "eq"*

Add some constraints to the LP problem L. All constraints will have the same relation, given by `Rel`, which may be set to `"eq"` for strict equality (the default), `"le"` for less-or-equal constraints, or `"ge"` for greater-or-equal constraints.

Constraints are of the form

$$\sum_{j=1}^{n} \texttt{lhs}_{ij} \quad \texttt{Rel} \quad \texttt{rhs}_{i1}$$

where *lhs* and *rhs* are described in Section 159.2.

`NumberOfConstraints(L)`

The number of constraints in the LP problem L.

`NumberOfVariables(L)`

The number of variables in the LP problem L.

`EvaluateAt(L, p)`

Evaluate the objective function of the LP problem L at the point p given by a matrix.

`Constraint(L, n)`

The LHS, RHS and relation (-1 for $\leq$, 0 for $=$, 1 for $\geq$) of the n-th constraint of the LP problem L.

`IntegerSolutionVariables(L)`

Sequence of indices of the variables in the LP problem L to be solved in integers.

`ObjectiveFunction(L)`

The objective function of the LP problem L.

`IsMaximisingFunction(L)`

Returns `true` if the LP problem L is set to maximise its objective function, `false` if set to minimise.

`RemoveConstraint(L, n)`

Remove the n-th constraint from the LP problem L.

`SetIntegerSolutionVariables(L, I, m)`

Set the variables of the LP problem L indexed by elements of the sequence I to be solved in integers if m is `true`, or in the usual ring if `false`.

`SetLowerBound(L, n, b)`

Set the lower bound on the n-th variable in the LP problem L to b.

Note that for all LP problems in MAGMA there is an implicit constraint that all variables are ≥ 0. This constraint is overridden if a lower bound is specified by using this function (e.g., specifying a lower bound of -5 works as expected), but the lower bound can currently not be completely removed.

`SetMaximiseFunction(L, m)`

Set the LP problem L to maximise its objective function if m is `true`, or to minimise the objective function if m is false.

`SetObjectiveFunction(L, F)`

Set the objective function of the LP problem L to the matrix F.

`SetUpperBound(L, n, b)`

Set the upper bound on the n-th variable in the LP problem L to b.

`Solution(L)`

Solve the LP problem L; returns a point representing an optimal solution, and an integer representing the state of the solution.

`UnsetBounds(L)`

Remove any bounds on all variables in the LP problem L.

Note that this reactivates the implicit constraint that all variables are ≥ 0.

Example H159E4

We use an LP object to solve the LP maximising

$$F(x, y) = 3x + 13y$$

subject to constraints

$$2x + 9y <= 40$$

$$11x - 8y <= 82$$

```
> R := RealField( );
> L := LPProcess(R, 2);
> SetObjectiveFunction(L, Matrix(R, 1, 2, [3,13]));
> lhs := Matrix(R, 2, 2, [2, 9, 11, -8]);
> rhs := Matrix(R, 2, 1, [40, 82]);
> AddConstraints(L, lhs, rhs : Rel := "le");
> SetMaximiseFunction(L, true);
> L;
LP <Real Field, 2 variables>
Maximising objective function: [ 3 13]
Subject to constraints:
1 : [2 9] <= [40]
2 : [11 -8] <= [82]
Variables bounded above by: [ ]
Variables bounded below by: [ ]
Solving in integers for variables [ ]
> Solution(L);
[9.199999999999999289 2.400000000000000355]
0
```

Now, we place some bounds on y:

```
> SetUpperBound(L, 2, R!2);
> SetLowerBound(L, 2, R!1);
> Solution(L);
[8.909090909090908283 2.000000000000000000]
0
```

And find integer solutions:

```
> SetIntegerSolutionVariables(L, [1,2], true);
> Solution(L);
[8.000000000000000000 2.000000000000000000]
0
```

Now, removing the 2nd constraint:

```
> RemoveConstraint(L, 2);
> L;
LP <Real Field, 2 variables>
```

```
Maximising objective function: [ 3 13]
Subject to constraints:
1 : [2 9] <= [40]
Variables bounded above by: [ 2:2 ]
Variables bounded below by: [ 2:1 ]
Solving in integers for variables [ 1, 2 ]
> Solution(L);
[11.00000000000000000 2.000000000000000000]
0
```

And removing the restriction to Integer values for y,

```
> SetIntegerSolutionVariables(L, [2], false);
> Solution(L);
[15.00000000000000000 1.111111111111111160]
0
```

159.5 Bibliography

[**Chv83**] V. Chvatal. *Linear Programming.* W.H. Freeman and Company, 1983.

[**Naz87**] John Lawrence Nazareth. *Computer Solution of Linear Programs.* Oxford University Press, 1987.

[**NW88**] G.L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, Inc., 1988.

[**OH68**] W. Orchard-Hays. *Advanced linear–programming computing techniques.* McGraw–Hill, 1968.

INDEX OF INTRINSICS